Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2006-037 May 22, 2006

First Class Copy & Paste

Jonathan Edwards

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

First Class Copy & Paste

Jonathan Edwards

May 22, 2006

MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar St. Cambridge, MA 02139

edwards@csail.mit.edu
http://subtextual.org

ABSTRACT

The Subtext project seeks to make programming fundamentally easier by altering the nature of programming languages and tools. This paper defines an operational semantics for an essential subset of the Subtext language. It also presents a fresh approach to the problems of mutable state, I/O, and concurrency.

Inclusions reify copy & paste edits into persistent relationships that propagate changes from their source into their destination. Inclusions formulate a programming language in which there is no distinction between a program's representation and its execution. Like spreadsheets, programs are live executions within a persistent runtime, and programming is direct manipulation of these executions via a graphical user interface. There is no need to encode programs into source text.

Mutation of state is effected by the computation of *hypothetical* recursive variants of the state, which can then be lifted into new versions of the state. *Transactional concurrency* is based upon queued single-threaded execution. Speculative execution of queued hypotheticals provides concurrency as a semantically transparent implementation optimization.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *Concurrent programming structures, Input/output, Procedures, functions, and subroutines, Recursion*; D.3.2 [**Programming Languages**]: Language Classifications – *Applicative (functional) languages, Concurrent, distributed, and parallel languages, Data-flow languages*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *Semantics*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems.

General Terms

Languages, Theory, Human Factors

Keywords

prototypes, copy and paste, modularity, reactivity, transactions

1. INTRODUCTION

The Subtext project [http://subtextual.org] seeks to make programming fundamentally easier by altering the nature of programming languages and tools. A previous paper [10] introduced the motivation and philosophy of the project, and informally presented the first prototype implementation. The current paper has two purposes: first, to precisely define the semantics of Subtext so that it can be evaluated and critiqued by others; second, to extend the previously reported capabilities to encompass features required by interactive systems: mutable state, I/O, and concurrency.

Subtext focuses on improving the experience of programming, and thus sees usability [33] as the driving issue in programming language design. Three key observations inform this effort:

1. Programmers naturally tend to construct programs by copy & paste [24].
2. Software is all about abstractions, but people best learn and understand abstractions through concrete examples [9].
3. Encoding programs as text strings creates a large conceptual gap between their representation and their meaning. Spreadsheets, which use a more direct representation, dramatically simplify programming within their domain [6].

In accordance with these observations, Subtext embraces copy & paste as the primary means of program construction. *Inclusions* reify copy & paste edits into persistent relationships, with the crucial enhancement that changes to the source of the inclusion are propagated into its destination. The expressive power of inclusions stems from the fact that they are higher-order: inclusions include inclusions. As a result, inclusions are computationally complete (as will be shown with a translation from lambda calculus).

Inclusions allow a spreadsheet-like form of programming that dispenses with source text. Programs are living executions in a persistent runtime environment, and programming is the direct-manipulation [37] of such executions via a Graphical User Interface (GUI). Programs are seen as living concrete examples rather than dead textual abstractions.

The previous paper [10] described in depth the justification and benefits of this approach to programming, and the design of a prototype user interface. A web video [11] demonstrates the user interface in action. The current paper offers a formal definition of an essential subset of Subtext. An operational semantics is

developed to make the definitions precise, and a diagrammatic notation is presented to make them clear.

This paper concentrates on semantic issues, and defers two major areas for future work: the user interface and performance. One benefit of Subtext is to allow such a decoupling of semantic issues from the "syntactic" ones of the user interface. On the other hand, performance issues have been deferred in order to focus on the simplicity and coherence of the semantics without premature optimization.

The next section, REIFYING COPY & PASTE, develops the formal theory of inclusions, and discusses the connections to standard modularity mechanisms. Then §3, FUNCTIONAL PROGRAMMING, uses inclusions to construct a small functional programming language. Functional programming languages have simple semantics, but are difficult to apply in reactive systems that require such features as mutable state, I/O, and concurrency. A new approach to this problem using inclusions is presented in §4, REACTIVE PROGRAMMING. The conclusion is that the simple idea of copy & paste offers a novel unification of a wide range of programming language features, and fresh perspectives on some old problems.

2. REIFYING COPY & PASTE

Inclusions are an abstraction of the common practice of copy and paste editing. Copy and paste operations typically operate on text strings, for that is what we primarily edit. In the case of programs, these linear strings are merely encodings of far richer structures: syntax trees. The theory of inclusions will be developed on a domain richer than strings but simpler than syntax trees: edge-labeled trees[1].

Inclusions are assumed to occur within a single global tree (inclusions across trees can be handled simply by wrapping them in a larger tree). An inclusion is thus a specification that one subtree is a pasted copy of another subtree. Subtrees are denoted by a path of labels walking down the tree from its root. An inclusion can be specified as an ordered pair of paths, written as destination : source. The source of an inclusion can also be the special value ∅ (pronounced "null"), representing an empty tree. A set of inclusions is a mapping from destination paths to source paths. The process of generating the tree specified by a set of inclusions is called *integration*. Figure 1 shows an example of a set of inclusions and the tree integrated from them. The formula on the left specifies the following inclusions:

1. a : ∅ creates an empty subtree of the root labeled a.
2. a.x : ∅ creates an empty subtree of a labeled x.
3. b : a creates a subtree of the root labeled b which is a copy of a. The copy will cause x to be *inherited* by b from a.

The tree integrated from these inclusions is diagramed in two ways: in the middle as a standard edge-labeled graph, and on the right with labeled nested regions. Each boundary of a region corresponds to a node of the tree, and the points correspond to leaves. Arrows denote inclusions with non-∅ sources. Note that the inclusion arrow points in the direction of the copy, from a to b, which is the reverse of the order in the inclusion mapping b:a. Nested regions show the similarities between copies more clearly than graphs, and so are used throughout the rest of this paper.

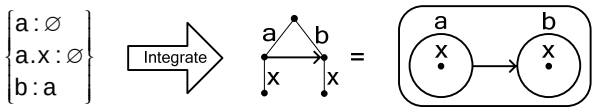

Figure 1. Integration

$$\begin{aligned} \mathit{Tree} &= \mathit{Inclusions} \times (\mathit{Branches} \cup \{?\}) \\ \mathit{Branches} &= \mathit{Label} \rightharpoonup \mathit{Tree} \\ \mathit{Inclusions} &= \mathit{Path} \rightharpoonup \mathit{Site} \\ \mathit{Path} &= \mathit{Label}^* \\ \mathit{Site} &= \mathit{Path} \cup \mathit{Constant} \end{aligned}$$

Figure 2. Definition of Integral Trees

2.1 Pure Integral Trees

Integral trees are edge-labeled trees that incorporate into each subtree the inclusions from which it was (or will be) integrated. An operational semantics of integration can be defined as a rewrite system on trees that expands pending inclusions into subtrees. As in the classical pattern of lambda calculus [2], the semantics will be enriched in stages, starting with a minimal kernel, by adding constants and rewrite rules. The starting point is *pure integral trees*, which are built solely out of inclusions of the empty tree, analogously to the pure lambda calculus.

Integral trees are formally defined in Figure 2. An integral tree, henceforth just a tree, pairs a set of inclusions with a set of branches to subtrees. When a subtree is still awaiting integration, the marker "?" (pronounced "unintegrated") takes the place of the branches, serving as a flag for the integration rewrite rule. A set of branches is a finite partial function from labels to trees. An infinite set of labels is assumed to be given in *Label*. A set of inclusions is a finite partial function from paths to *sites*, indicating that the subtree found by walking down the path (starting at the location of the inclusion) is to be overlaid with a copy of the tree at the specified site. Sites are either paths or constants, reflecting the fact that constants are read-only and thus may be the source but not the destination of an inclusion. The constants are fixed in the set *Constant*, which for pure trees contains only ∅, representing an empty tree.

A path is a finite sequence of labels. The empty path is written as "•". The singleton path containing just the label l is written "•l". Path concatenation is written "$p*q$", and "$p.l$" appends label l to the end of path p. A partial function $F : A \rightharpoonup B$ is a set of pairs each written as $a : b$, with all the left hand components distinct. The set of all left hand components is the domain of the function, written *dom*(F). The equation $F(a) = b$ will be interpreted to mean $(a : b) \in F$. Partial function override is defined as:

$$F \oplus G = G \cup \{ x : y \mid (x : y) \in F \wedge (\nexists z \mid (x : z) \in G) \}$$

[1] Subtext orders the edges from a node, but they are left unordered here as a simplification.

$$T[\cdot] = T$$
$$\langle I,B\rangle[\cdot l] = \begin{cases} V & \text{if } B \neq ? \ \wedge \ B(l) = V \\ \Omega & \text{otherwise} \end{cases}$$
$$\Omega = \langle \{\}, ? \rangle$$
$$T[\cdot l * p] = T[\cdot l][p]$$
$$T[\varnothing] = \langle \{\}, \{\} \rangle$$

Figure 3. Subtree access

$$T[\cdot := V] = V$$
$$\langle I,B\rangle[\cdot l := V] = \begin{cases} \langle I,\ B \oplus \{l : V\} \rangle & \text{if } B \neq ? \\ \Omega & \text{otherwise} \end{cases}$$
$$T[\cdot l * p := V] = T[\cdot l := T[\cdot l][p := V]]$$

Figure 4. Subtree replacement

The subtree at site p within tree T is written as $T[p]$ and determined by following the path down the tree, as defined in Figure 3. Accessing a non-existent path returns the special tree Ω, indicating an error. Constants are treated as if they were empty trees[2]. The tree obtained by replacing the subtree at path p within tree T with the tree V is written as $T[p := V]$ and defined in Figure 4. All of the previous notation (and some yet to be defined) is summarized in Figure 5.

Figure 6 defines the primary (and at this point, the only) rewrite rule for integration. It establishes a binary relation $\rightarrow_\int$ on trees that incrementally integrates a tree from a set of inclusions. A starting set of inclusions I is specified in an *initial tree* of the form $\langle I, ? \rangle$. The integration rule looks for the ? unintegrated marker and replaces it with a set of subtrees integrated from the inclusions I. Each of the new subtrees will be marked unintegrated, causing integration to proceed top-down through the tree.

Certain error situations (for example, cyclic inclusions) will cause integration to halt with unintegrated inclusions left over[3]. Integration can also proceed forever because of recursive inclusions, to be discussed in §2.3.

Integration rewrites subtrees at each step, but is dependent upon the contextual state of the tree. Therefore the rewrite relation is defined between complete trees, and is not a Term Rewriting System [26], except trivially. Integration is non-deterministic: a tree can be integrated in more than one way. However such divergences can always reconverge: integration is confluent. Confluence follows from the fact that integration changes only unintegrated nodes into integrated nodes, and that these changes

$a : b$	partial function map $a \mapsto b$
$F \oplus G$	partial function override of F by G
$p * q$	path concatenation
$\cdot = \{\}$	empty path
$\cdot l = \{l\}$	singleton path
$p.l = p * \{l\}$	label append
?	unintegrated marker
$\varnothing$	literal for empty tree
$\langle I, B \rangle$	tree with inclusions I and branches B
$\Omega = \langle \{\}, ? \rangle$	undefined tree
$T[p]$	subtree access
$T[p := V]$	subtree replacement
$T\|p\|$	subtree value
$T \rightarrow_\int V$	integration rewrite

Figure 5. Notation key

$$\frac{T[p] = \langle I, ? \rangle \quad I(\cdot) = x \quad T[x] = \langle J, B \rangle \quad B \neq ?}{T \rightarrow_\int T[p := \langle K,C \rangle], \text{ where}}$$
$$K = J \oplus \{r : p{*}q \mid \exists q,r \mid (r : x{*}q) \in J\} \oplus I \qquad \text{(Inherit)}$$
$$C = \{l : \langle K/l, ? \rangle \mid K/l \neq \{\}\} \qquad \text{(Divide)}$$
$$K/l(s) = y \text{ iff } K(\cdot l * s) = y$$

Figure 6. Primary integration rule

are determined by the immutable state of the previously integrated nodes.

2.2 Integration by Example

The first rewrite of the integration pictured in Figure 1 will be worked through in detail. We start with the initial tree T:

$$T = \langle \{\cdot : \varnothing, \cdot \mathsf{a} : \varnothing, \cdot \mathsf{a}.\mathsf{x} : \varnothing, \cdot \mathsf{b} : \cdot \mathsf{a}\}, ? \rangle$$

The antecedents (formulas above the line) of the rewrite rule are satisfied as follows:

$T[p] = \langle I, ? \rangle$ where $p = \cdot$ and $I = \{\cdot : \varnothing, \cdot \mathsf{a} : \varnothing, \cdot \mathsf{a}.\mathsf{x} : \varnothing, \cdot \mathsf{b} : \cdot \mathsf{a}\}$
$I(\cdot) = x$ where $x = \varnothing$
$T[x] = \langle J, B \rangle$ where $J = \{\}$ and $B = \{\}$ since $T[\varnothing] = \langle \{\}, \{\} \rangle$
$B \neq ?$ since $\{\} \neq ?$

The integration rule calculates the following rewrite:

$$T \rightarrow_\int T[p := \langle K,C \rangle] = T[\cdot := \langle K,C \rangle] = \langle K,C \rangle$$

The *source* of a subtree is the mapping of the empty path by its inclusions. In this case the source is $I(\cdot) = \varnothing$. The source plays a central role in integration: it is the "copy" in "copy & paste". The (Inherit) formula overrides the inclusions of the source (J) with the current inclusions of the subtree (I) to calculate the rewritten inclusions (K). The set comprehension in the (Inherit) formula

[2] In Subtext, constants are located in a special section of the tree, to provide a uniform user interface. It is simpler in the formal theory to treat constants as a special case.

[3] In Subtext errors are handled more informatively with special constants.

maintains a property called *monomorphism*, explained in §2.3, which is irrelevant in this example. For this example, the source is the empty tree ∅, which offers nothing to inherit, and the inclusions are unchanged. The actual computation is:

$$\begin{aligned} K &= J \oplus \{r : p{*}q \mid \exists q,r \mid (r : x{*}q) \in J\} \oplus I \\ &= \{\} \oplus \{r : p{*}q \mid \exists q,r \mid (r : x{*}q) \in \{\}\} \oplus I \\ &= I \end{aligned}$$

After the new set of inclusions is calculated by (Inherit), the new set of subtrees is calculated by (Divide). Division partitions the set of inclusions by the labels prefixing their destinations. A new subtree is created for each such prefix label, containing the inclusions minus the prefix. The two divisions are:

$$\begin{aligned} K/a &= \{\cdot : \varnothing,\ \cdot\mathsf{a} : \varnothing,\ \cdot\mathsf{a}.\mathsf{x} : \varnothing,\ \cdot\mathsf{b} : \cdot\mathsf{a}\}/\mathsf{a} \\ &= \{\cdot : \varnothing,\ \cdot\mathsf{x} : \varnothing\} \\ K/b &= \{\cdot : \varnothing,\ \cdot\mathsf{a} : \varnothing,\ \cdot\mathsf{a}.\mathsf{x} : \varnothing,\ \cdot\mathsf{b} : \cdot\mathsf{a}\}/b \\ &= \{\cdot : \cdot\mathsf{a}\} \end{aligned}$$

The branches a and b for these two new subtrees are calculated as:

$$\begin{aligned} C &= \{l : \langle K/l, ? \rangle \mid K/l \neq \{\}\} \\ &= \{a : \langle \{\cdot : \varnothing,\ \cdot\mathsf{x} : \varnothing\}, ? \rangle,\ b : \langle \{\cdot : \cdot\mathsf{a}\}, ? \rangle\} \end{aligned}$$

The final value of the rewrite thus becomes:

$$T \rightarrow_{\int} \left\langle \begin{array}{l} \{\cdot : \varnothing, \cdot\mathsf{a} : \varnothing, \cdot\mathsf{a}.\mathsf{x} : \varnothing, \cdot\mathsf{b} : \cdot\mathsf{a}\}, \\ \{a : \langle \{\cdot : \varnothing,\ \cdot\mathsf{x} : \varnothing\}, ? \rangle, b : \langle \{\cdot : \cdot\mathsf{a}\}, ? \rangle\} \end{array} \right\rangle$$

This is a tree with two subtrees a and b, each of which is marked unintegrated, each with a set of inclusions divided out of the initial set. Division essentially "unrolls" inclusions down the tree as it is integrated, "peeling off" the prefix of their destination paths as they flow down, until they end up as inclusions onto the empty path, establishing the source of that subtree[4]. The source of subtree a is ∅, and b has source a. Note that division does not unroll the sources of the inclusions, which are passed down unchanged. This means that the destination of an inclusion is interpreted as a path downward from its location, but the source of an inclusion is always an absolute path down from the root of the tree.

Figure 7 details the entire five-step integration of Figure 1 using both formulas and diagrams. The first integration step that was just worked through is shown by the dashed arrow between the formulas in (1) and (2). The arrow is drawn from the unintegrated marker to the branches that replace it (as an abbreviation because there was no inheritance to alter the inclusions).

The process of integration can be seen more clearly in the diagrams on the right side of Figure 7. As in Figure 1, an integrated tree is diagrammed with labeled nested regions, using

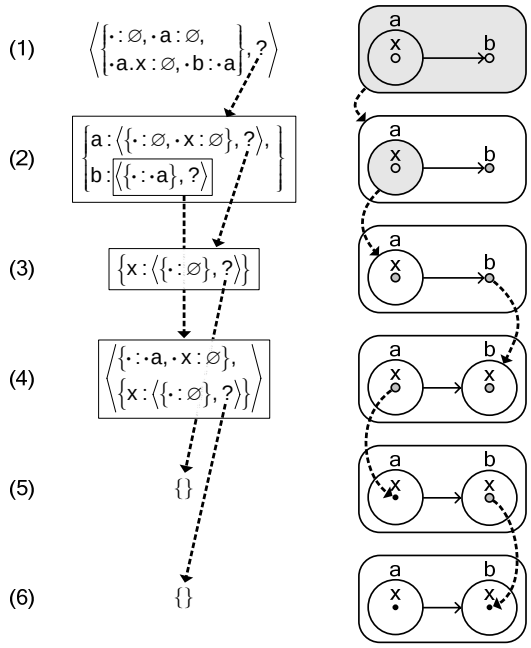

Figure 7. Integration example

arrows to indicate the non-∅ source of each subtree. In addition, unintegrated subtrees are shown as shaded regions. Within each shaded region, the set of pending inclusions is shown using a "diagrammatic pun". A set of inclusions can be seen as an edge-labeled tree (of the destination paths under prefix containment) decorated with arrows (from the non-∅ sources)[5]. The leaves in this tree of inclusions are drawn as hollow points to retain their shading.

Each integration step is shown by a dashed arrow drawn from the unintegrated tree to its corresponding integrated state in the next step. Notice how in the step from (1) to (2) the shaded region "flows down" the diagram, moving from the entire tree down to a and b. This movement corresponds to the fact that integration proceeds top-down in the tree. Also notice that there is an arrow from a to b in both diagrams (1) and (2). In (1) this arrow represents the root inclusion $\cdot\mathsf{b} : \cdot\mathsf{a}$, which is unrolled by division into the inclusion $\cdot : \cdot\mathsf{a}$ on b, establishing a as the source of b, which is represented by the arrow in (2). All that changes between diagrams (1) and (2) is the shading: the boundaries and arrows remain intact through integration, justifying the diagrammatic pun.

Up until (3), the sources are all ∅, and there is no inheritance, so division just unrolls the inclusions onto an identical tree. However in (3), b has source a, which contains x. The (Inherit) formula takes care of including the contents of a into b. It does this by overriding the inclusions of the source a with the inclusions of the destination b (ignoring the central comprehension of the formula,

[4] This property enables an alternative representation for integral trees used in Subtext. Subtrees record only their source and its *inheritance depth*: the height in the tree from which the source inclusion was rolled down.

[5] It is possible for there to be "gaps" in the tree of inclusions, which are indicated with dashed boundaries. An example of this will be seen in Figure 11.

which still has no effect in this case). The inclusions of b are calculated as:

$$K = \{\cdot : \varnothing, \cdot\,\mathsf{x} : \varnothing\} \oplus \{\} \oplus \{\cdot : \cdot\,\mathsf{a}\} = \{\cdot : \cdot\,\mathsf{a}, \cdot\,\mathsf{x} : \varnothing\}$$

The override operator in this formula allows the contents of the source to be inherited, overridden, or extended by the destination[6]. Inheritance is pictured on the formula side as a dashed arrow from the boxed unintegrated tree expression in (2) to its replacement in (4). On the diagram side, inheritance causes b to expand from a shaded hollow point in (3) to an unshaded circle containing x in (4). Inheritance has the effect of laying down new structure at the frontier of integration as it sweeps down into the tree.

Integration halts at a.x in (5) and b.x in (6) because there are no inclusions left. Diagrammatically, the hollow shaded points collapse into unshaded points, which are leaves of the tree.

2.3 Higher-order Copy & Paste

The expressive power of integral trees stems from the fact that inclusions are higher-order: inclusions include inclusions. Every subtree records the inclusions that it was integrated from, and they get copied along whenever it is the source of an inclusion. In other words, what is being copied and pasted is the reification of prior copies and pastes. It's copy and paste "all the way down". Several examples will help to illustrate the implications of higher-order inclusion.

Figure 8 shows *disinheritance*. The double-headed dashed arrow between the diagrams represents the composition of zero or more integration steps. In this example, b.f includes a. Since a contains x, b.f inherits x alongside y when it is integrated in the right-hand diagram. Now c includes b, which would normally cause it to inherit x and y into c.f. But the inclusions in the left-hand diagram "override" the source of c.f to be d. As a result c.f inherits y from b.f, but not x. This happens because b.f inherited x from its source a, but c.f has a different source d, which *disinherits* x, and bequeaths z instead. Since b.f.y was "grown" inside b.f rather than being inherited, it gets bequeathed to c.f regardless.

Disinheritance is surprising only if an inclusion is thought of as copying the state of the source. Rather, it is copying the inclusions of the source that record how it was integrated, and then replaying them in a new context. It is often the case that this replay results in the same end state, encouraging the metaphor of copying state. But in this example, the inherited inclusions get overridden by the context, altering the way in which they unfold. Therefore a more accurate metaphor is that inclusions copy the *design* of the source, not its state.

Figure 9 shows another example of a higher-order effect: *monomorphism*[7]. Subtree b contains x and y, with b.x the source of b.y. Subtree c includes b, inheriting both x and y. But the source of c.y is c.x, not b.x. The internal structure of inclusions within a tree is reproduced in any inclusions of that tree. Note however, that the source of both b.x and c.x is a, because a is located

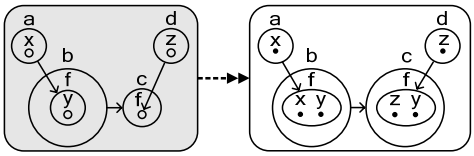

Figure 8. Disinheritance

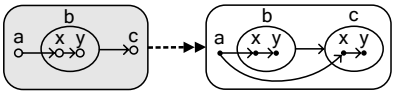

Figure 9. Monomorphism

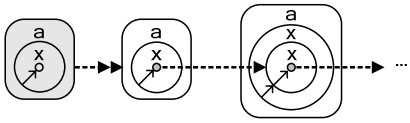

Figure 10. Recursion

outside b, which is the source of c. Monomorphic inclusion is calculated by the set comprehension in the (Inherit) formula. It maps sources within the contextual source to the same relative path within the destination.

Figure 10 shows *recursion*, wherein a tree is included into itself. The dashed arrow shows how the unintegrated inclusion expands out to a copy of its containing source, and reproduces itself inside endlessly. Recursion is a direct consequence of monomorphism: the source of x is mapped to its prior self, which now contains it, perpetuating the recursion.

These properties of higher-order inclusions are used in the rest of the paper to reformulate standard programming language mechanisms.

2.4 Temporal Integral Trees

Up to this point, inclusions have been presented as a set of predefined instructions that unfold via integration into some final result. To fully capture the process of copy and paste editing, incremental modification of existing trees is needed. Incremental editing is supported by adding a model of time to integral trees, forming *temporal integral trees*.

The basic idea is to record a history of versions, with each version including the previous one, plus an incremental change. Each version will be a top-level branch of the tree labeled t_0, t_1, and so forth. Each such timestamped version will hold its *state* in a subtree labeled out (to leave room for code that computes the state, as in §4.3). Each edit to the tree, called an *edit command*, will overlay a single inclusion onto a copy of the previous version.

[6] Deletion has not yet been implemented.

[7] A monomorphism is an injective homomorphism, that is, a structure-preserving embedding.

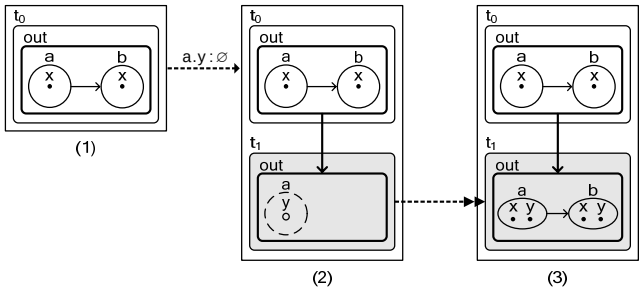

Figure 11. Edit command[8]

$$\frac{T = \langle I, B \rangle \quad p \in Path \quad x \in Site \quad t = Cur(T)}{T \xrightarrow{p:x} \langle I \cup D, C \rangle \text{ where}}$$

$$D = \{ \cdot t' : \varnothing,\ \cdot t'.\text{out} : t,\ \cdot t'.\text{out} * p : y \}$$

$$y = \begin{cases} x & \text{if } x \in Constant \\ \cdot t'.\text{out} * x & \text{otherwise} \end{cases}$$

$$C = \begin{cases} B \cup \{ t' : \langle D / t', ? \rangle \} & \text{if } B \neq ? \\ ? & \text{otherwise} \end{cases}$$

Figure 12. Edit command rule

Figure 11 shows an example of an edit command. Starting in (1) with the result from Figure 7, a new leaf a.y is created by adding the inclusion $a.y : \varnothing$. The state of each version (labeled out) is highlighted with a bold boundary. The initial state has timestamp t_0. In (2), the edit adds a new state t_1 that is a copy of t_0 with the addition of the inclusion $a.y : \varnothing$. The eventual result of integrating the edit is (3).

Note that in (3), both a.y and b.y have been created, because the integration of b gets replayed in the new version. The insertion of y into a can be seen as having been propagated through the inclusion from a into b. From this perspective, inclusions are constraints that propagate changes from their source into their destination. This property is essential to many uses of inclusions.

The individual versions of a temporal tree can be seen as the frames of a movie; when the movie is "played", edits appear as changes that propagate through containing inclusions. The Subtext user interface adopts this "character-in-the-movie" perspective by continually refreshing a window with the state of the current version of the tree. The user directly edits within this window through drag-and-drop or copy-and-paste operations. These operations execute edit commands which cause a new version to be created and in turn displayed in the window. Temporal trees thus project the illusion of a mutable state with internal constraints while the reality is incremental inclusions creating immutable versions. This approach provides a simple and deterministic semantics, as well as a complete record of history.

In the interest of simplicity, a number of useful features have been omitted. One is the ability to undo and redo edits. Another is a localized form of undo called a *revert*, which removes all overlaying inclusions (that is, edits) from a subtree, reverting it back to the state of its source. It is convenient to be able to write-protect subtrees. It is also useful to declare that a subtree is always an exact copy of its source (called a *reference*), automatically forwarding any edits within the reference into the source. All of these features are implemented in Subtext.

Henceforth, plain atemporal trees will often be treated as if they were incrementally editable, with the understanding that the discussion could be recast in terms of temporal trees as a justification.

2.4.1 Formalizing Temporal Trees

An infinite set of labels used as timestamps is assumed: $Time = \{ t_i \}_{i \in \mathbb{N}} \subseteq Label$. The path to the current version's state is determined by the upper bound of the existing timestamps:

$$Cur(\langle I, B \rangle) = \begin{cases} \cdot t_n.\text{out} & \text{if } \exists\, t_n \in dom(I) \mid t_m \in dom(I) \Rightarrow n \geq m \\ \varnothing & \text{otherwise} \end{cases}$$

The next version's timestamp is determined by:

$$t' = \begin{cases} t_0 & \text{if } t = \varnothing \\ t_{n+1} & \text{if } t = \cdot t_n.\text{out} \end{cases}$$

An *edit command* is a rewrite relation $\xrightarrow{p:x}$ that creates a tree with a new version incorporating the inclusion $p : x$. The path *p* (and *x*, if it isn't a constant) is interpreted as relative to the current version's state, not the global tree. The rewrite rule is defined in Figure 12. The formula for *D* calculates the inclusion for the new version, mapping *p* into the new state. The formula for *y* maps *x* into the new state if is not a constant. The formula for *C* integrates a branch for the new version if the root of the tree is already integrated.

2.5 Performance

Performance has been deferred to future work, but a few comments may be in order here. Temporal trees imply that the entire state of the current version is copied on every change, which would be a performance problem if done literally (as in the current implementation). A basic optimization technique is *virtual integration*. It is not necessary to actually make every copy and permanently store them: it is only necessary to present that illusion. Version control systems [13][39] typically keep only one version and record deltas from it so as to dynamically recreate other versions on demand. Inclusions are like deltas: they can be used to recreate their entire subtree from its source. A sophisticated implementation could cache needed portions of the tree, and reintegrate others on demand.

Another optimization technique is *incremental integration*, which finds opportunities to avoid needless replay of inclusions by inferring that regions of the source tree can be copied wholesale. In fact such constant regions could be shared rather than being physically copied. There is a wealth of related work on copy-

[8] The dashed circle in (2) indicates that a is needed to diagram the inclusion for a.y, but has no inclusion itself: it is a *gap*. Gaps are errors that cause integration to halt, unless they are filled in from a higher source, in this case t_0. Gaps can also occur when the source of an inclusion does not exist.

elimination in functional languages [14][18][34]. Effect systems [28] may also help to localize changes.

A different performance issue is that the set of root inclusions grows monotonically, journaling all changes. It will be necessary to gradually forget the details of history by summarizing them into coarser versions, based on some configurable policy. In an end-user production environment, the policy might be to immediately forget history[9], enabling special optimizations.

2.6 Decriminalizing Copy & Paste

Programmers are routinely enjoined to avoid copy and paste in favor of modularity mechanisms that avoid duplication, such as functions, methods, classes, and templates [12][19]. Yet practicing programmers continually violate this injunction [24]. The primary disadvantage of copy and paste is the need to manually coordinate changes afterwards. But the prescribed alternatives have some disadvantages of their own:

1. They require extra programming effort.
2. They can be too coarse to usefully capture sharing between small fragments of code.
3. They favor changes oriented in certain directions, and are disrupted by changes that cross-cut this "grain", leading to a "whipsaw effect" of repeated code restructuring.
4. They add levels of indirection, making the code harder to understand.

Inclusions bring fresh options to this old conflict. They combine the flexibility and generality of copy & paste with the declarative and self-maintaining nature of modularity mechanisms. Inclusions are a kind of *lightweight semi-modularity*. They could supplant traditional mechanisms in some situations, although this would depend upon tools to manage large webs of inclusions. Traditional modularity mechanisms can be seen as special-case patterns of inclusions. These patterns could be refactored out of ad hoc webs of inclusions, yielding *emergent modularity*. Future work will explore inclusions as a novel modularity mechanism, while this paper returns to the agenda of using them to reformulate conventional programming language constructs.

3. FUNCTIONAL PROGRAMMING

In this section, integral trees will be developed into a small functional programming language while constructing a factorial function. In fact pure integral trees are already computationally complete. This is demonstrated in Appendix A by a translation of pure lambda calculus[10]. But like pure lambda calculus, pure integral trees are not a very appealing programming language. As in the classical treatment of lambda calculus [2], a more realistic language can be created by enriching the pure kernel with constants and primitives.

[9] However it is the author's experience that end-users find historical information indispensable, once they learn it is available.

[10] This is a one-way translation, which leaves open the question of what could be a denotational semantics of inclusions.

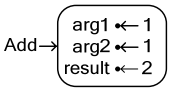

Figure 13. Call of addition function

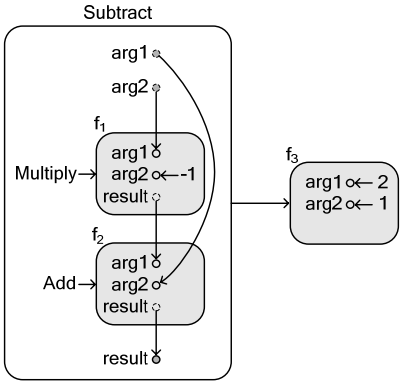

Figure 14. Definition and call of Subtract function

The set of constants is enriched with the integers, the Booleans True and False, and a set of constants representing primitive functions: Add, Multiply, Equal, If. A set of labels are designated for the arguments and results of these primitive functions: arg1, arg2, result, test, then, else.

Figure 13 shows an example of a call to the addition function. The function is specified as the source of the tree (Add), and the arguments are specified as the sources of the arg1 and arg2 leaves. The addition function sets the source of the result leaf to be the sum of the arguments. This diagram adds the convention that constant sources (other than ∅) are indicated with an arrow from the adjacently placed literal name of the constant. A thin arrow is used for computed results.

Functions add reactivity to integral trees. By including a function into a tree, a constraint is established between the input arguments and the output result (or results). Editing one of the inputs causes the results to change in reaction (although as explained in §2.4, the function is actually re-executing with the changed inputs in a new temporal version). Functions are one-way: editing the result will not change the inputs (and is an error).

3.1 Defining Functions

Figure 14 shows a function defined from primitive functions. Subtraction is defined in terms of addition and multiplication by negative one. The function uses the standard arg1, arg2, and result labels, wiring them up with inclusions to calls on multiplication and addition. The internal calls to Multiply and Add are labeled as f_1 and f_2. Subtext automatically allocates such internal labels and hides them from the programmer[11].

Much of the function definition is shaded as unintegrated because no sources have been specified for the input arguments (indicated by dashed circles). This situation corresponds to the standard notion of a function as an abstract description of a computation

[11] The previous paper [10] explores the issue of names in depth.

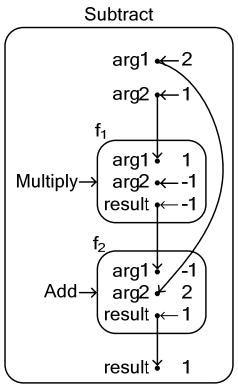

Figure 15. Definition as example

using symbolic variables. The subtraction function is called at f_3 with arguments of 2 and 1. The internals of the subtraction function will be integrated into f_3, which when combined with the actual argument values, will produce a complete execution like that in Figure 15. The execution of the function depends on the fact that constants "flow" through inclusions, so for example, Subtract.f1.arg1 gets its value from Subtract.arg2, which is 1. Such flowing values are formalized in §3.4 below.

Note how calling by inclusion depends upon the principle of monomorphic inclusion (§2.3), mapping the internal inclusion "wiring" of a called function into the call. Monomorphism fills the role that scoped name binding plays in conventional programming languages. The fact that monomorphism only applies within the boundaries of the inclusion's source is equivalent to lexical scoping. References outside the context of the source act as if captured by a closure.

In a conventional language, functions are called by name so as to defer determining the function's definition (which may change). But deferred binding adds a level of indirection that complicates understanding the program. Calling functions by including them is essentially "edit-time" inlining, making the effect of the call immediately clear. Any subsequent changes to the definition of the function will be propagated through the inclusions to all its call sites. Inclusions allow functional abstraction without indirection.

Inclusions also support higher-order functions, as functions are just subtrees that flow through inclusions the same as constant values. The lambda calculus translation in Appendix A gives an example of a higher-order function. Disinheritance (§2.3) is crucial to higher-order functions: it allows a call whose function value changes to replace the internals of the old function with that of the new, while preserving the argument bindings.

What is missing from this model of functions is the usual notion of an interface: a hard boundary between the arguments/results of a function and its internal machinery. In Subtext, function interfaces reappear as a UI feature, hiding the internals of a function unless they are explicitly expanded, and protecting them from accidental modification in calls. Intrusive modification of called functions could be allowed to support techniques like Aspect Oriented Programming [22].

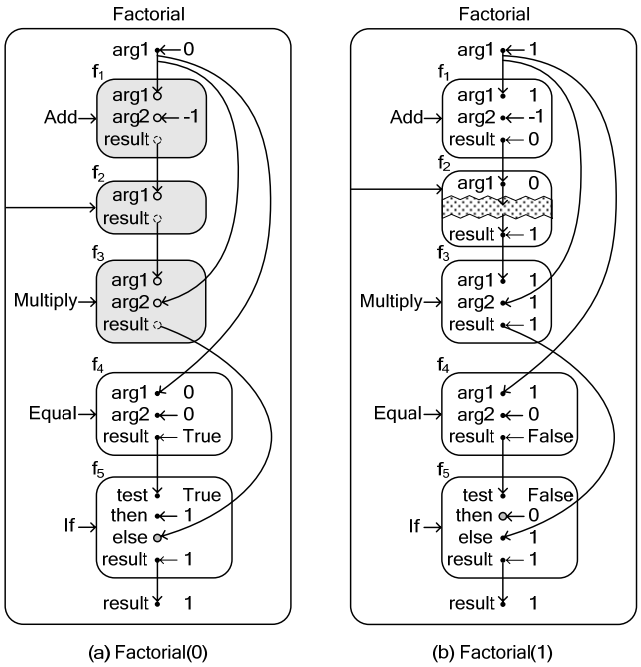

Figure 16. Factorial

Function definitions do not need to be abstract as in Figure 14, and in fact shouldn't be. Figure 15 shows the definition of the subtraction function altered to incorporate an example execution. To help visualize execution, subtrees that do not have a constant source (that is, variables) are annotated with their non-∅ values. With the addition of these annotations, the function definition serves as a worked-out example. The function takes specified example input values, and every calculated intermediate value is fully visible. Calls to other functions are inlined, revealing the full detail of the execution down to the primitives. The Subtext UI exploits this property to display all code as living example executions that are edited by direct manipulation. The result is What You See Is What You Get (WYSIWYG) programming: there is no longer any difference between a program's static representation and its dynamic execution. This unification is the key innovation of Subtext, and the main source of its promised benefits [9][10].

3.2 Conditionals and Recursion

Conditionals and recursion are needed to complete the development of a minimal functional programming language. Figure 16(a) diagrams a factorial function executing the base case of zero. The calls at f_1 through f_3 are only used in recursive cases, and are left unintegrated. The Equal function called at f_4 compares the argument to zero, producing the Boolean value True in result. This result feeds into the test argument of the If function, which selects whether to map the value of the then or else arguments into the result. Since the test is true, the value of then, which is set to the constant one, is passed through to result.

The If function is non-strict: it does not depend upon the evaluation of the then and else arguments – it merely includes at most one of them into the result. This allows a strategy of *lazy integration* to be adopted, as in lazy functional languages [41].

Lazy integration is demand-driven, with outputs triggering the integration of their inputs. The initial demands are generated by the external world, via the user interface and I/O devices. Lazy integration is not formalized in this paper. Instead it is simulated by choosing to integrate only those portions of the function that are needed, with the result that in (a) the else argument of the If is left unintegrated, as are the calls in f_1 through f_3. The shaded unintegrated regions can be interpreted as a visualization of dead code[12].

Figure 16(b) shows the factorial of one, which utilizes recursion. As described in §2.3, recursion is just self-inclusion, done by the call at f_2. The internals of f_2 will integrate into the same diagram as in (a). To save space, this inclusion has been elided with the wavy shaded region. Subtext provides selective expansion and zooming to cope with the explosion of nested detail. Subtext handles infinite recursion by limiting the maximum length of inclusion chains, passing back a special error constant when that limit is exceeded.

3.3 Pragmatics of Functions

It must be emphasized that the diagrams presented here are not meant as a means of actually writing programs. The Subtext UI [10] displays integral trees as an indented outline that is largely textual. These diagrams are better thought of as picturing the semantic data model underneath the UI. The connections with work on visual programming languages are discussed in the Related Work section.

An efficient implementation of this language will depend upon the techniques of virtual and incremental integration discussed in §2.5. Virtual integration avoids permanently allocating storage for the complete trace of every execution. Incremental execution provides automatic memoization [29] of function execution. However functions also complicate incremental integration, as they add new pathways for changes to propagate across the tree. Incremental integration will also need to do garbage collection, which would become more a matter of *garbage neglection*.

3.4 Formalizing Functions

Functions are defined by "wiring up" other functions with inclusions. This requires a concept of values that flow through these wires. Accordingly, the *value* of a subtree is defined by searching backward through its chain of sources until a constant is found. (This definition will be extended later to encompass other sorts of values.) The value can be thought of as an *identity* that is inherited through inclusions (and is what the Equal function compares).

To define values formally, we restrict ourselves to *well-formed* integral trees, in which every integrated subtree has an ancestral source of $\varnothing$. To simplify the treatment, non-$\varnothing$ constants are given a source of $\varnothing$ by adding the following clause to the definition of the $[\,]$ operator:

$$T[c] = \langle \{\cdot : \varnothing\}, \{\} \rangle \quad \text{if } c \in \mathit{Constant} - \{\varnothing\}$$

[12] In Subtext, merely observing a subtree in the user interface triggers its integration. Dead code is a separate analysis fed by counterfactual conditionals.

Define the *source relation* $\prec_T$ over *Site*:

$$x \prec_T y \;\text{ iff }\; I(\cdot) = y \wedge B \neq ? \;\text{ where }\; \langle I, B \rangle = T[x]$$

Note that $x \prec_T y$ corresponds in a diagram to an inclusion arrow from y to an integrated (non-shaded) subtree x. The transitive reflexive closure of the source relation $\prec_T$ is the *ancestor relation* $\preceq_T^*$. A well-formed tree is one in which $\preceq_T^*$ is a partial order over all integrated subtrees and has upper bound $\varnothing$. An initial tree is well-formed, and it is easy to show that integration preserves well-formedness, so we will henceforth assume well-formedness.

The value of an integrated subtree is found by tracing back its sources until a constant is found. The partial function $\mathit{Tree}\|\mathit{Path}\| \rightharpoonup \mathit{Site}$ determines the value of a subtree and is defined as:

$$T\|p\| = \begin{Bmatrix} x & \text{if } x \in \mathit{Constant} \\ T\|x\| & \text{otherwise} \end{Bmatrix} \text{ iff } \; p \prec_T x$$

Note that the value of a subtree is not defined until it is integrated.

Figure 17 shows the integration rules that define the semantics of the primitive functions. For example, the (Add) rule will fire when the value of a subtree is Add, there is not already a result leaf, and the values of its arg1 and arg2 leaves are both integers. When fired, the rule fills in the result leaf with the sum.

$$\frac{T\|p\| = \mathsf{Add} \quad T[p.\mathsf{result}] = \Omega \qquad T\|p.\mathsf{arg1}\| = n \in \mathbb{Z} \quad T\|p.\mathsf{arg2}\| = m \in \mathbb{Z}}{T \rightarrow_{\int} T\left[p.\mathsf{result} := \left\langle \{\cdot : (n+m)\}, \{\} \right\rangle\right]} \qquad \text{(Add)}$$

$$\frac{T\|p\| = \mathsf{Multiply} \quad T[p.\mathsf{result}] = \Omega \qquad T\|p.\mathsf{arg1}\| = n \in \mathbb{Z} \quad T\|p.\mathsf{arg2}\| = m \in \mathbb{Z}}{T \rightarrow_{\int} T\left[p.\mathsf{result} := \left\langle \{\cdot : (n \times m)\}, \{\} \right\rangle\right]} \qquad \text{(Multiply)}$$

$$\frac{T\|p\| = \mathsf{Equal} \quad T[p.\mathsf{result}] = \Omega \qquad T\|p.\mathsf{arg1}\| = x \quad T\|p.\mathsf{arg2}\| = y}{T \rightarrow_{\int} T\left[p.\mathsf{result} := \left\langle \{\cdot : b\}, \{\} \right\rangle\right] \text{ where } b = \begin{cases} \mathsf{True} & \text{if } x = y \\ \mathsf{False} & \text{otherwise} \end{cases}} \qquad \text{(Equal)}$$

$$\frac{T\|p\| = \mathsf{If} \quad T[p.\mathsf{result}] = \Omega \qquad T\|p.\mathsf{test}\| = b \in \{\mathsf{True}, \mathsf{False}\}}{T \rightarrow_{\int} T\left[p.\mathsf{result} := \left\langle \{\cdot : x\}, \{\} \right\rangle\right] \text{ where } x = \begin{cases} p.\mathsf{then} & \text{if b} = \mathsf{True} \\ p.\mathsf{else} & \text{otherwise} \end{cases}} \qquad \text{(If)}$$

Figure 17. Function integration rules

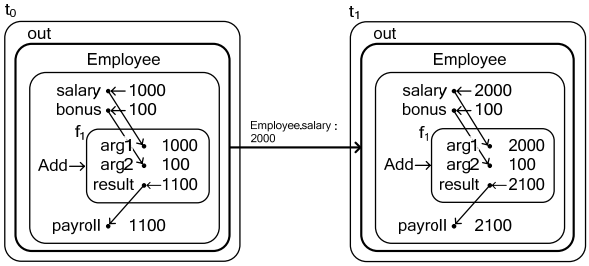

Figure 18. Data-flow

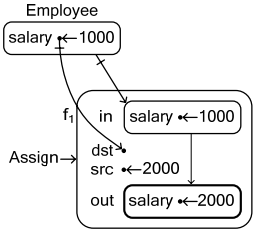

Figure 19. Assignment action

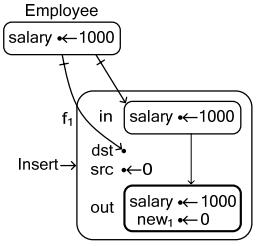

Figure 20. Insertion action

4. REACTIVE PROGRAMMING

The previous section developed integral trees into a pure lazy functional language. The allure of such languages is the simplifying absence of control-flow and side-effects. But the whole point of many systems is to have side-effects: to interact with the world and maintain an internal model of it. Such systems will be called *reactive*. Functional programs live in a Platonic realm of timeless immutable values which do not easily interact with a changing world. There has been much work on supporting features such as mutable state, I/O, and concurrency in functional languages, leading to the current popularity of monads [44][21]. Unfortunately some programmers find monads baffling, as evidenced by the number of tutorials.

This section will introduce reactive features on top of the functional language of the previous section. The approach taken avoids descending to the hardware level of control-flow and side-effects, preserving the simplicity of functional programming. But the approach also offers a common-sense model of mutable state, avoids higher-order abstractions, and supports a transactional form of concurrency. Full WYSIWYG visibility of program execution is maintained.

The model of time introduced in §2.4 already contains the germ of the solution: that mutable state can be modeled with inclusions as a timeline of incrementally altered versions. This model will be generalized in several stages to support reactive systems. In fact a weak form of reactivity is already present: *data-flow* computation. Figure 18 shows an example of an Employee data structure with embedded logic, in the form of an Add function that sums salary and bonus to compute payroll. This logic will be reproduced in all instances of the Employee structure. Recall from §2.4 that temporal integral trees simulate in-place change with a series of incrementally edited versions. Here the Employee.salary field is edited to 2000, creating version t_1 in which the change has propagated through the Add function into the payroll field. Note that this diagram simplifies matters by showing the edit command as a labeled bold arrow between versions. Strictly, the edit is a rewrite between entire trees, the first of which contains only t_0, and the second of which contains both versions as pictured.

4.1 Actions

Simple data-flow of the sort described above is constrained to pre-defined channels. In general, functions need the ability to make arbitrary changes to the current state of the tree, just as external edit commands can do. As a first step, we introduce primitive functions that can edit locally contained state, and then later explain how they can be used to change global state.

Figure 19 shows an example of the Assign primitive function, which allows a tree to be edited by changing the source of any subtree within it. It works by creating a tree within the out argument which is a modified copy of the tree supplied by the in argument. The modification is determined by the dst and src arguments, which, just like an edit command, specify the destination and source of an inclusion to be overlaid on the tree. The bars on the tails of the arrows will be explained shortly. This example performs the same edit as the previous example: it assigns the salary field of an Employee structure to 2000. If the extra fields and payroll calculation logic of Figure 18 had been included, they would have all been inherited into the output, where the payroll would have been recalculated just as in the previous example.

Functions like Assign that produce an out result that is a modification of their in argument are called *actions*. While the examples in this section show very small structures being transformed by actions, the intention is that these can scale up to the entire mutable state of a system. Efficient performance depends upon the techniques of virtual and incremental integration discussed in §2.5 to avoid physically copying the entire state.

The assignment action can edit existing structures, but can not create new ones. Figure 20 shows an example of the Insert action, which operates like Assign, except that it creates a new subtree of the destination argument (which in this case is the input tree itself). A fresh label is created for this new subtree, shown as new_1.[13]

[13] Subtext actually hides labels behind user-editable textual tags, which start out as empty strings on insertions [10].

$$\frac{T\|p\| = \mathsf{Assign} \quad T\|p.\mathsf{in}\| = i \quad T\|p.\mathsf{dst}\| = i*d \quad d \neq \cdot \quad T\|p.\mathsf{src}\| = s \quad T[p.\mathsf{out}] = \Omega}{T \rightarrow_{\int} T\left[p.\mathsf{out} := \left\langle \{\cdot : p.\mathsf{in},\ d : x\},\ ?\right\rangle\right] \text{ where } x = \begin{cases} p.\mathsf{out} * q & \text{if } \exists q \mid s = i*q \\ s & \text{otherwise} \end{cases}}$$

Figure 21. Assignment rule

$$\frac{T\|p\| = \mathsf{Insert} \quad T\|p.\mathsf{in}\| = i \quad T\|p.\mathsf{dst}\| = i*d \quad T\|p.\mathsf{src}\| = s \quad T[p.\mathsf{out}] = \Omega \quad l \text{ fresh in } T}{T \rightarrow_{\int} T\left[p.\mathsf{out} := \left\langle \{\cdot : p.\mathsf{in},\ d.l : x\},\ ?\right\rangle\right] \text{ where } x = \begin{cases} p.\mathsf{out} * q & \text{if } \exists q \mid s = i*q \\ s & \text{otherwise} \end{cases}}$$

Figure 22. Insertion rule

4.1.1 Formalizing Actions

The in and dst arguments of the assignment and insertion actions take a new kind of value: a *quoted location*, which is created by a *quoted inclusion*, shown in the diagrams with a bar at the tail of the arrow. The value that a quoted inclusion conveys to its destination is the location of its source, which is a first-class values that will flow through normal inclusions.

To formalize the assignment action, the labels in, dst, src, out, and quote are defined, and Assign is added to *Constant*. Quoted inclusions are specified by appending the special label quote to the end of their source path. The value function is modified to recognize quotes:

$$T\|p\| = \left.\begin{cases} x & \text{if } x \in \mathit{Constant} \\ q & \text{if } x = q.\mathsf{quote} \\ T\|x\| & \text{otherwise} \end{cases}\right\} \text{ iff } \ p \prec_T x$$

Recall that previously the value function traced back sources to an ancestral constant. Now this trace will halt when a quoted inclusion is hit, and the location of the source will be used as the value. Quoting does not affect inheritance: the contents of the quoted location are still included into the destination. Accordingly, the path access function is modified to make quoting transparent:

$$\langle I, B\rangle[\cdot l] = \begin{cases} \langle I, B\rangle & \text{if } l = \mathsf{quote} \\ V & \text{else if } B \neq ? \ \wedge\ B(l) = V \\ \Omega & \text{otherwise} \end{cases}$$

The integration rule of the assignment action is defined in Figure 21. It creates the output argument as an inclusion of the input argument plus the specified edit. Note that the destination location must be a proper subtree of the input location for the rule to fire. Also note that if the source is a subtree of the input then it will be monomorphically mapped into the output.

Figure 22 defines the integration rule for the insertion action. It assumes some method for generating a fresh label not mentioned anywhere in the current tree.

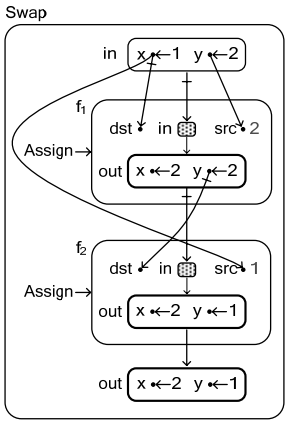

Figure 23. Sequential swap action

4.2 Composing Actions

Actions can be composed to form more complex actions, as they are just functions that transform states. The most familiar way to compose actions is in a sequential chain. Figure 23 shows an example of an action that swaps the x and y subtrees of a structure. Swapping is done by chaining two assignments sequentially, first setting x to y, then y to x.

Sequential actions mimic the way that consecutive statements execute in imperative languages. Each statement starts with the state left over from the previous, and alters the state seen by the next statement. But instead of imposing an order of execution on statements, sequential actions explicitly hook up the input and output states of each action. This extra "plumbing" would add complexity and work for the programmer, if it had to be dealt with explicitly. Subtext offers a specialized UI presentation in which sequential actions look and feel much like statements: they are presented in a sequential list; and the in-out chaining is automatically maintained when actions are inserted, deleted, or moved [11].

The example of a swap action demonstrates some of the advantages of sequential actions over conventional imperative statements. Firstly, all intermediate states are visible, so that debugging is just inspection, not the crude practice of stepping through execution. Secondly, "time-traveling" cross-state access is possible, as when the original value of x is needed in the second assignment statement. Imperative languages force one to carefully copy aside values that will be needed later and may be altered in the meantime, while offering no way in general to know what those alterations may be.

A major disadvantage of sequential programming is that it enforces a strictly linear ordering of events when that may not be the precise intention. The example of swapping shows this in the choice to do one assignment before the other: the two assignments are symmetrical, neither depending upon the result of the other. The sequentialization of the assignments obscures these facts. Worse, if the assignment primitives were instead complex compound actions, the first might have non-obvious side-effects upon the second. We really want to do both actions independently and in parallel, and then combine their results.

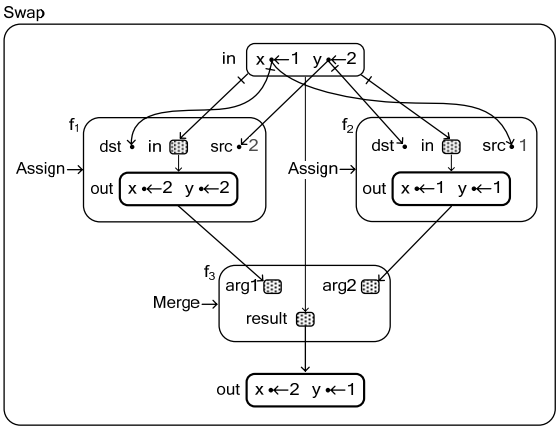

Figure 24. Parallel swap action

Figure 24 shows how to do a parallel swap with the Merge primitive defined in Appendix B. Merging combines non-interfering changes in two descendants of a common ancestor, which is precisely what we want from parallel actions. If parallel actions make different changes to the same location, the merge will produce a conflict error, indicating a bug. Such conflicts may arise during merge, but there is no possibility of accidental side-effects between parallel actions, as in conventional approaches to parallelism through interleaving: each parallel action sees the same immutable input state.

Merging supports an arbitrary (directed acyclic) graph of actions, combined both sequentially and in parallel. The UI presentation of sequential actions mentioned above will be correspondingly generalized to support graphs, subsuming the merges into the topology.

Graphs of actions allow programs to more accurately express their meaning. Actions (and imperative statements) are *causally ordered*: an action must occur after some others and before yet others. Causality is a partial order on actions, which can be directly represented as a graph. When we code a linear sequence of actions/statements we must mentally construct a total order which satisfies the casual order, so that nothing happens too early or too late. What we are doing is a topological sort on the causal order, which is not only mentally laborious and error-prone, but also obscures the true meaning of the program. The programmer must constantly recall what the hidden causal structure is while maintaining the topological sort of it. Topological sorting should be hidden as an internal optimization technique.

4.3 Hypotheticals

The assignment and insertion actions permit arbitrary edits to be performed to a tree, but in a functional manner that only creates altered versions of the tree. These versions are similar to the time-stamped versions introduced in temporal integral trees. What is needed is some way to "lift" actions into the versions of a temporal tree. A flawed attempt is pictured in Figure 25. Version t_0 contains the initial state of the Employee structure. Version t_1 consists of an assignment action whose input is the state of the previous version. The out result of the action specifies the state of the second version, which contains the modified Employee structure. This example explains why the state of a version is

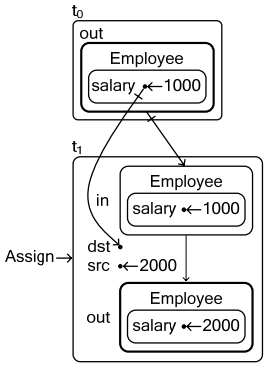

Figure 25. Actions as versions

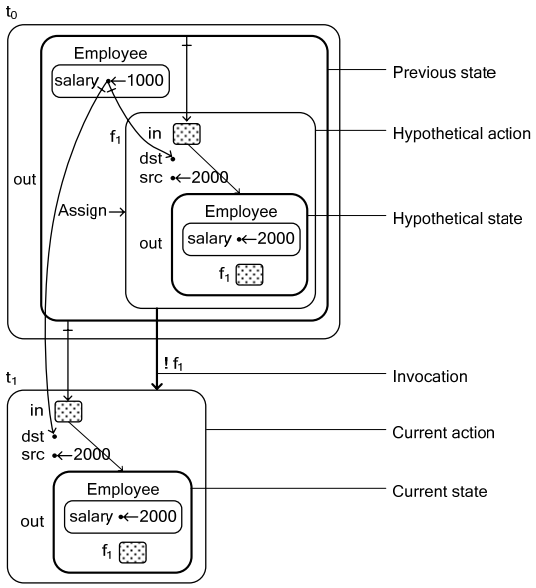

Figure 26. Action Invocation

wrapped in an out subtree: to allow actions to be versions that calculate the state of the version.

But where do these actions come from? It is as if they magically appear from the external environment. We want programs to be part of the computational world, so that they are always living example executions. How can we maintain this goal while also allowing programs to make global changes to state? The key observation is that if actions are part of the current state, and actions also produce future states, then the future must be recursive.

Figure 26 shows the previous attempt recast in a recursive model of time. The initial state is t_0.out, which includes both an Employee structure and an Assign action that modifies it. The input to the assignment is the entire enclosing state. The state flows through the in argument (elided by a dotted background) into the out argument, where the Employee structure is modified. The output of the assignment action is a *hypothetical state*: it shows the state as it would appear after the modification specified by the assignment's arguments. Note that a recursive copy of the

$$\frac{T = \langle I, B \rangle \quad p \in Path \quad t = Cur(T)}{T \xrightarrow{!p} \langle I \cup \{\cdot t' : t * p\}, C \rangle \text{ where}}$$

$$C = \begin{cases} B \cup \{t' : \langle \{\cdot : t * p\}, ? \rangle\} & \text{if } B \neq ? \\ ? & \text{otherwise} \end{cases}$$

Figure 27. Invocation rule

assignment action itself is also present in this hypothetical future state.

An *invocation* is a command that specifies an action within the current state of the tree, and copies it out to become the next version. In so doing, the hypothetical output of the action becomes the new *actual* state. Figure 26 shows the invocation of the action, which includes it into the new version. Note how the in and dst arguments of the action get automatically set as a consequence of this inclusion as non-monomorphic copies. Invocation is formalized in Figure 27 as the rewrite relation $\xrightarrow{!p}$ that invokes the path *p* within the current state.

Invocation answers the question of where actions come from: they are part of the current state, and recursively modify it. The stimulus to invoke an action comes from the external world. In the Subtext UI, invocation is a double-click on an action. Hardware device events also trigger invocations, as will be seen in §4.4. A realistic system will contain numerous possible actions, each calculating its own hypothetical future. Compound actions essentially posit entire hypothetical histories. Invocation chooses which of these possible future worlds is to become actual. Note that all of these possible actions get recursively copied into the next state, and so recalculate a new range of possible worlds. In the current example the action $t_1.out.f_1$ will recalculate the (uninteresting) hypothetical effect of setting the salary to 2000 yet again.

It is the job of the user interface to make the structure of hypothetical actions clear and manageable to the programmer. The current prototype [11] uses background color and special decorations to illuminate the "state-flow" of a program. States are sparsely expanded to show deltas relative to a previous state.

Hypothetical actions reconcile functional and imperative programming in a novel way. Actions are still purely functional, in that they depend only on their inputs, which are immutable. But actions are also imperative in that they can freely modify anything in the global state, even themselves. It is just that these modifications take place in a hypothetical world distinct from the current one in which they are running. Within each version actions are atomic and free of conflicts through side-effects. But because actions are themselves part of the state that they are modifying, they "see" the changes they have made when they "reincarnate" in the next version.

Hypothetical actions resemble monadic computation in that they are functions consuming and producing states. The difference is that actions are composed like normal functions, rather than using higher-order monadic combinators. Monads allow only sequential execution, and segregate "pure" from "impure" code. Haskell's IO Monad [21] exploits these restrictions to provide a thin, efficient implementation on conventional hardware. Hypotheticals

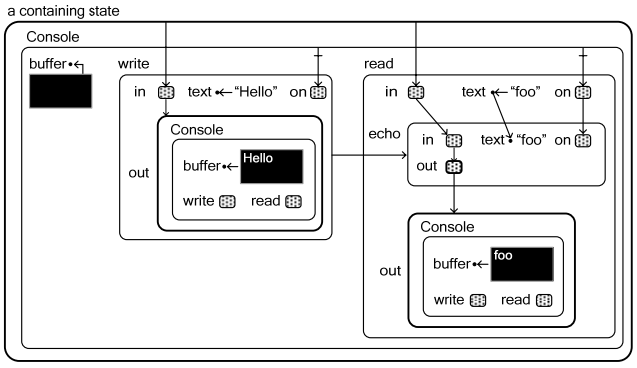

Figure 28. Console agent

provide a simple and flexible programming model, but presume sophisticated optimization techniques.

It is being boldly assumed that the profligate copying of entire system states can be virtualized and optimized to the point of acceptable performance. If the implementation challenges can be overcome, hypothetical actions offer an appealing way to program reactive systems.

Hypothetical actions retain the common-sense idea that you can change anything you can see, but do so without descending to the hardware level of control-flow and global side-effects as in imperative languages. All computation is still pure and lazy, but without the need for higher-order combinators. Perhaps most importantly, the full details of hypothetical executions are visible to the programmer: What You See is What You *Would* Get. Debugging becomes inspection, and testing reactive code becomes as easy as testing pure functions.

4.4 Hypothetical Input/Output

This section will informally sketch how input and output can be incorporated into a model of hypothetical computation. The basic idea is that I/O is simulated and queued within hypothetical states, only physically occurring from top-level actual states. Hypothetical I/O is mediated by *agents*, which combine a representation of the internal status of the device along with actions to do input and output to the device. Agents reprise some familiar patterns of object-oriented programming.

Figure 28 shows a *console agent* that interfaces to a simple terminal emulation. The buffer field of the agent records its internal status, which is the text displayed in the terminal emulation window. Buffer values will be treated as primitive constants with a graphical display in the diagram (Subtext uses a similar specialized UI presentation).

The agent contains two actions, which respectively write and read from the console. The write action takes as input a system state containing the console, and a text string to write to the console. It outputs a modified state in which the console has been written to. The write action also has an internal variable on, which quotes its containing agent, serving much like "self-variables" in an object-oriented language. The prototype of the write action uses the example of writing "Hello", and thus its hypothetical output shows a terminal window containing "Hello". If and when the

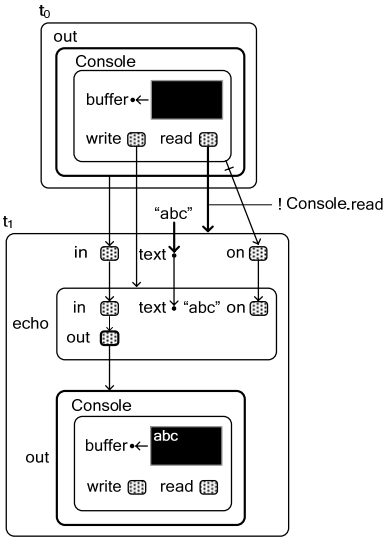

Figure 29. Input invocation

write action is invoked, "Hello" will actually be output to the terminal.

If the user types into the console window, the read action will be invoked with the text argument set to the typed text. Figure 29 shows a user input of "abc". The two bold arrows represent the invocation; the others are their consequences. The read action by default contains an internal call to the write action to echo the typed text back to the same console. The output of the echo flows out to the output of the entire invoked action, making it the next actual state of the system, and causing the echo to physically occur. To respond to user input in other ways beyond just echoing, additional actions would be inserted into the read action, threaded into the chain of states leading to its output. A more general solution is a separate action to register such "callbacks" as in the Observer pattern [12].

The model of I/O proposed here is purely reactive [3][4][32]. The system is driven by input events (including clock ticks), and can respond to them with output events. There is no way to stall execution to wait for an input event. Reactive programming imposes a discipline similar to "event-loop" programming in GUI frameworks.

Output is *hypothetical*: it only physically occurs when a modified agent flows up to an actual version of the system. Until then, the agent's actions only simulate and queue output, exemplified by the screenshots in the diagrams. When a modified agent finally surfaces in an actual state of the system, it must execute all the output actions that have been queued since the last actual state. These requirements make I/O interfaces more complicated to implement than in traditional languages, but have compensating benefits. I/O programming becomes concrete and visible: you can prototype I/O and see exactly what it would do. Testing I/O becomes as straightforward as testing pure functions. Output also becomes transactional: hypothetical outputs will be discarded by a canceled transaction (see §4.5).

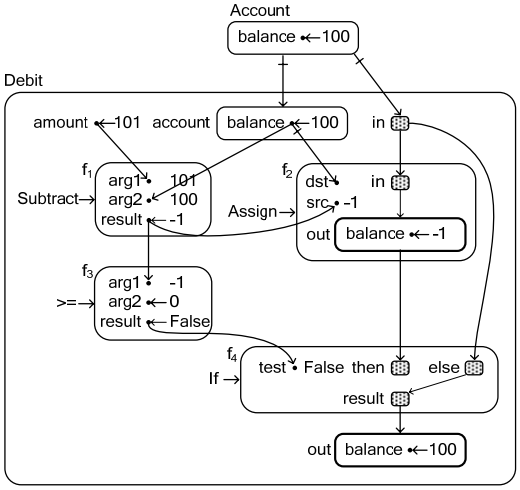

Figure 30. Debit transaction

4.5 Transactional Concurrency

The model of reactive systems developed so far is single-threaded: it reacts to a single input at a time, and completes its reaction to that input before reacting to the next. Realistic systems must be concurrent: reacting to multiple inputs at the same time. The standard approach to concurrency is preemptive multi-tasking: instruction-level interleaving of multiple imperative programs with global side-effects. The ensuing chaos is left to the programmer to manage, typically with complex and error-prone locking schemes. Databases have long used *transactions* [4][45] to provide a simpler and more reliable method of controlling concurrency. There has been recent interest in integrating transactions into programming languages [36][17]. However these proposals add transactions as special-purpose features on top of traditional concurrency, or as a special-purpose sublanguage. This section presents a transactional form of concurrency that is hidden from the semantics of the language as an implementation optimization technique[14].

An important property of transactions is the ability to rollback to the beginning state of the transaction. This requires complex run-time mechanisms when doing in-place updates of a single global state, but with hypothetical states requires only an if function. Figure 30 shows the classic bank-account debit transaction that subtracts an amount from the balance in an account[15]. If the account is overdrawn, the transaction is cancelled. Rolling back to the input state is done by an If function gating the output of the action, choosing whether to pass out the modified state or the original input state. Transactional rollback is achieved without adding any new semantics. A disadvantage is that the transactional logic is made explicit and visible, rather than being an implicit property of the runtime environment. As with similar

[14] Subtext has not yet implemented transactional concurrency.

[15] The debit transaction depends on a detail omitted from the definitions of the Assign and Insert primitives. The dst argument of the assignment will be *dereferenced* from Debit.account.balance to Account.balance because it is nested inside another quote.

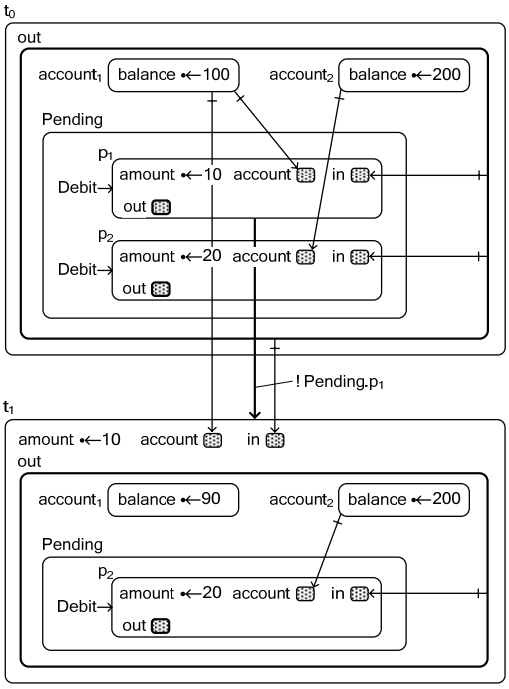

Figure 31. Transaction commit

issues, this problem can be passed to the user interface, where transactional plumbing could be hidden to make it seem implicit.

Concurrency is all about efficiency. Single-threaded systems are simpler and more reliable, but they leave both hardware and users sitting idle. Concurrency allows throughput to be increased, and response time to be decreased. Conventional transaction mechanisms simulate single-threaded semantics (called *serialization*) on top of concurrent interleaved execution. We will take the opposite approach: start with a simple single-threaded semantics and treat concurrency as an implementation optimization technique. After all, since concurrency is purely a performance concern, it belongs in the implementation, not the semantics.

Figure 31 pictures two transactions. Transactions are initiated by placing actions in the specially designated Pending subtree, in this case the two debit transactions p_1 and p_2, which respectively debit the accounts $account_1$ and $account_2$. A pending transaction is *committed* by invoking it and deleting it from Pending. Transaction p_1 is committed at t_1, debiting $account_1$. The set of pending transactions is essentially a run-queue, and can be used as a general mechanism for spawning actions.

Transactions are committed serially, and each executes as an atomic change after the result of the previous transaction. There is no possibility of side-effects or inconsistencies. The implementation is free to choose the order in which to commit pending transactions, and it is from this freedom that concurrency arises. An efficient implementation will evaluate pending transactions in parallel in order to maximize throughput, and then commit them as they complete in order to minimize response-time. Evaluating pending transactions is a generalized form of speculative execution – it could be said that hypotheticals give free reign to speculation.

When a pending transaction is committed, the other pending transactions are reinstantiated in the new version with a changed input state, possibly changing their results (for example if both transactions were updating the same account). Incremental integration (§2.5) is essential for an efficient implementation of this approach. It allows speculative computations to be reused when they won't change, effectively doing *partial retries* of only the portions of pending transactions that can change. Such optimizations are possible because computations are first-class data structures in integral trees.

In contrast, traditional concurrency control must cope with the whims of the process scheduler as it blindly maximizes utilization without regard to the consequences. Techniques such as locking, versioning, and logging are used to construct a logically serialized execution out of the chaos. The opposite approach is being proposed: instead of *enforcing* serialization on top of concurrency, concurrency is *discovered* inside serialization. Concurrency becomes a semantically transparent implementation optimization, affecting only the speed and order in which transactions commit. There are many technical challenges to an efficient implementation of this idea, but it offers the opportunity to make concurrent programming fundamentally simpler.

5. RELATED WORK

This work is directly inspired by Self [42], both technically and philosophically. Self first proposed that copying (in the guise of *prototypes*) could provide a unifying framework for both a programming language and its development environment. Self sought to improve the experience of programming through simplicity, concreteness, uniformity, and flexibility [38][43].

Inclusions go beyond prototype-based languages [27][31] in two ways. Firstly, inclusions persistently propagate changes, while prototype clones are one-shot copies. Delegation or indirection must be used to coordinate dynamic state between clones. Secondly, inclusions extend "all the way down" into the syntax and semantics of the language. Prototypes gather methods together, but the methods themselves are atomic hunks of syntax with no prototypical structure, and their execution occurs behind the veil of a virtual machine. Inclusions are used both to write code and to execute it. Inclusions unify code and data even more intimately than in LISP. Not only is the static representation of a program a first-class value, so is its dynamic execution, and the two are one and the same.

Proposals related to inclusions are *similarity inheritance* [8] for sharing formulas between spreadsheets; *linked editing* [39] as a replacement for functional abstraction; and *clone genealogy extraction* [25] for reverse-engineering copy relationships out of version control histories.

There is a long history of related work in Visual Programming Languages [7][20][30] that sought to replace text with diagrams. Unfortunately, diagrams turned out not to work as well in practice as text [35][15]. Diagrams are used in this paper to represent integral trees, but are not intended for writing programs. The Subtext UI is in fact largely textual [10]. Tree diagrams should not be seen as a kind of syntax, but rather as a data model of program semantics. The emergence of diagrams at the semantic

level comports with the original intuitions behind visual programming.

Forms/3 [6] extended the spreadsheet into a first-order functional programming language. General purpose programming concepts were cleverly, but intricately, simulated within the spreadsheet metaphor. Rather than make a spreadsheet work like a programming language, Subtext makes a programming language work like a spreadsheet. Pictorial Janus [23] had a unified representation of programs and their execution, supported recursion as infinite containment, and replaced names with topological properties. However its imperative semantics meant that while you could animate program execution, you could not see a program and its entire execution at once. Vital [16] is a visual execution environment for Haskell. Vital is most similar to Subtext in that copy & paste operations on data structure instances correspondingly alter their source definitions.

Subtext is in spirit a functional programming language, harking back to Backus' call to liberate programming from its hardware roots [1]. Modern functional programming languages demonstrate that sophisticated higher-order abstractions like monads [44][21] can be both powerful and efficient. Subtext instead prioritizes ease of use over power, and simplicity over ease of optimization.

6. CONCLUSIONS

Inclusions generalize copy & paste editing into a model of persistent higher-order copying within trees. Inclusions can be used to formulate a programming language in which there is no distinction between a program's representation and its execution. This property enables new kinds of development environments, and fresh approaches to some of the classic problems of programming language design. The simple idea of copy & paste turns out to be surprisingly fundamental.

The goal of this paper has been to set out these ideas with sufficient clarity and precision to enable others to evaluate and critique them. A formal operational semantics of integral trees has been defined. Informal diagrams have been used to present these constructions intuitively. Most of the ideas have been validated for basic sanity in the current implementation of Subtext, but much further work is needed. Before integral trees can be useful in practice, at least two major further results must be achieved:

1. Demonstration of a programming user interface that matches the fluidity and scalability of textual tools, along with an empirically measurable increase in programmer productivity. The current prototype UI is a only a first step in this direction.
2. Proof that, in an end-user environment with development-specific features unused, the space and time complexity of integral trees with respect to conventional languages is bounded by a constant factor.

With those two provisos, inclusions offer a number of benefits:

1. Integral trees are a medium in which programming is the direct manipulation of running programs within a persistent runtime, without the need for a separate source text encoding. User interfaces for programming other than text editing can be explored.
2. Every program is a living example of its execution, demonstrating its meaning even as it is edited, much like a spreadsheet.
3. Persistent higher-order copy & paste opens a middle-ground between traditional forms of modularity and ad hoc copying.
4. Recursive hypothetical states are a concrete formulation of reactive software that avoids higher-order abstractions like monads and yet remains uncontaminated by the hardware-level mechanisms of control flow and globally writeable state. Mutation and I/O are fully visible and testable. The causal structure of stateful programs is made explicit, not encoded into a linear schedule.
5. Transactional concurrency presents the programmer with a semantics of queued single-threaded execution free of side-effects, race conditions, and locking. Speculative execution of queued hypotheticals provides concurrency as a semantically transparent implementation optimization, affecting only the speed and order in which transactions commit.

These results are only a conceptual stepping-stone. The long-term vision of this research is to fundamentally alter programming languages and tools so as to radically simplify programming, and liberate the creativity of programmers.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] Backus, J. Can Programming be Liberated From the von Neumann Style?: A Functional Style and its Algebra of Programs. *Commun. ACM 21, 8* (1978).

[2] Barendregt, H. P. *The Lambda Calculus.* Elsevier, 1984.

[3] Benveniste, A., Berry, G. *The synchronous approach to reactive and real-time systems* Proceedings of the IEEE 79(9), September, 1991.

[4] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[5] Black, A.P., Carlsson, M., Jones, M.P., Kieburtz, R., Nordlander, J. *Timber: a Programming Language for Real-Time Embedded Systems*. Oregon Graduate Institute School of Science & Engineering Technical Report CSE-02-002, 2002.

[6] Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J., Yang, S., Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. In *Journal of Functional Programming 11, 2* (March 2001).

[7] Burnett, M., Goldberg, A., Lewis, T. *Visual Object-Oriented Programming: Concepts and Environments.* Manning, Greenwich, CT, 1995.

[8] Djang, R., Burnett, M. Similarity Inheritance: A New Model of Inheritance for Spreadsheet VPLs. In 1998 IEEE Symp. Visual Languages (Halifax, Canada, Sep. 1998).

[9] Edwards, J. Example Centric Programming. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '04 Onward)* (Vancouver, BC, CANADA) *SIGPLAN Not.* 39, 12 (Dec. 2004). http://subtextual.org/OOPSLA04.pdf

[10] Edwards, J. Subtext: Uncovering the Simplicity of Programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications(OOPSLA '05)* (San Diego, CA, USA, October 16 - 20, 2005). http://subtextual.org/OOPSLA05.pdf

[11] Edwards, J. Subtext demonstration. http://subtextual.org/demo1.html

[12] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] Glasser, A. L. The evolution of a Source Code Control System. In *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues* S. Jackson and J. A. Lockett, Eds. 1978.

[14] Gopinath, K. and Hennessy, J. L. Copy elimination in functional languages. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, United States, January 11 - 13, 1989).

[15] Green, T. R. G., Petre, M. Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing 7, 2*, 131-174.

[16] Hanna, K. Interactive Visual Functional Programming. In *Proceedings of the Seventh ACM SIGPLAN international Conference on Functional Programming* (Pittsburgh, PA, USA, October, 2002).

[17] Harris, T., Marlow, S., Jones, S.P., and Herlihy, M. Composable Memory Transactions. In *ACM Conference on Principles and Practice of Parallel Programming 2005*.

[18] Hudak, P. and Bloss, A. 1985. The aggregate update problem in functional programming systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, Louisiana, United States, January 14 - 16, 1985).

[19] Hunt, A., Thomas, D. *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley, 1999.

[20] Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., Doyle, K. Fabrik: A Visual Programming Environment. In *Conference proceedings on Object-oriented programming systems, languages and applications(OOPSLA '88)* (San Diego, California, United States, Sep. 1988).

[21] Jones, S.P. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, IOS Press, 2001.

[22] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. In *Proc. of European Conference on Object-Oriented Programming (ECOOP 1997)* (Jyväskylä, Finland, 1997).

[23] Kahn, K. M., Saraswat, V. A. Complete Visualization of Concurrent Programs and Their Executions. In *Proceedings of the ICLP 1990 Workshop on Logic Programming Environments*, (Eilat, Israel, June 1990).

[24] Kim, M., Bergman, L., Lau, T., Notkin, D. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proc. of the 2004 ACM-IEEE International Symposium on Empirical Software Engineering* (Redondo Beach, CA, Aug. 2004).

[25] Kim, M. and Notkin, D. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *Proceedings of the 2005 international Workshop on Mining Software Repositories* (St. Louis, Missouri, May 17, 2005).

[26] Klop, J. W. Term Rewriting Systems. Chapter 1 in *Handbook of Logic in Computer Science*, vol. 2, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, eds. Oxford University Press, 1992.

[27] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA'86)* (Portland, Oregon, United States, 1986).

[28] Lucassen, J. M. and Gifford, D. K. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, United States, January 10 - 13, 1988).

[29] Michie, D. 'Memo' functions and Machine Learning. *Nature* 218 (1968).

[30] Myers, B. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing 1, 1* (March 1990), 97-123.

[31] Noble, J., Taivalsaari, A., Moore, I. *Prototype-Based Programming: Concepts, Languages and Applications.* Springer-Verlag Telos, January, 1999.

[32] Nordlander, J., Jones, M., Carlsson, M., Dick Kieburtz, and Black, A. Reactive objects. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2002.

[33] Norman, D. A. *The Design of Everyday Things.* Basic Books, New York, 1988.

[34] Okasaki, C. *Purely Functional Data Structures.* Cambridge University Press (1998).

[35] Petre, M. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Comm. ACM 38, 6* (June 1995) 33-44.

[36] Shavit, N., and Touitou, D. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, August 1995.

[37] Shneiderman, B. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer 16, 8* (August, 1983).

[38] Smith, R. B., Maloney, J., Ungar, D. The Self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *Proc. of the tenth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA '95)* (Austin, Texas, United States, 1995).

[39] Tichy, W. F. Design, implementation, and evaluation of a Revision Control System. In *Proceedings of the 6th international Conference on Software Engineering* (Tokyo, Japan, September 13 - 16, 1982).

[40] Toomim, M. Begel, A., Graham, S. L. Managing Duplicated Code with Linked Editing. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*(Rome, Italy, Sep. 2004).

[41] Turner, D. A. Miranda: A non-strict functional language with ploymorphic types. In *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture*, Nancy France, September 1985.

[42] Ungar, D., Smith, R. B. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA '87)* (Orlando, Florida, United States, Oct. 1987), 227-242.

[43] Ungar, D., Smith, R. B. Programming as an Experience: The Inspiration for Self. In *ECOOP '95 Conference Proceedings*, LNCS 952. Springer Verlag, 1995

[44] Wadler, P. The essence of functional programming. In *19th Symposium on Principles of Programming Languages* (Albuquerque, NM, January, 1992).

[45] Weikum, G., Vossen, G., *Transactional Information Systems*. Morgan Kaufmann, 2002.

APPENDIX A. TRANSLATION FROM LAMBDA CALCULUS

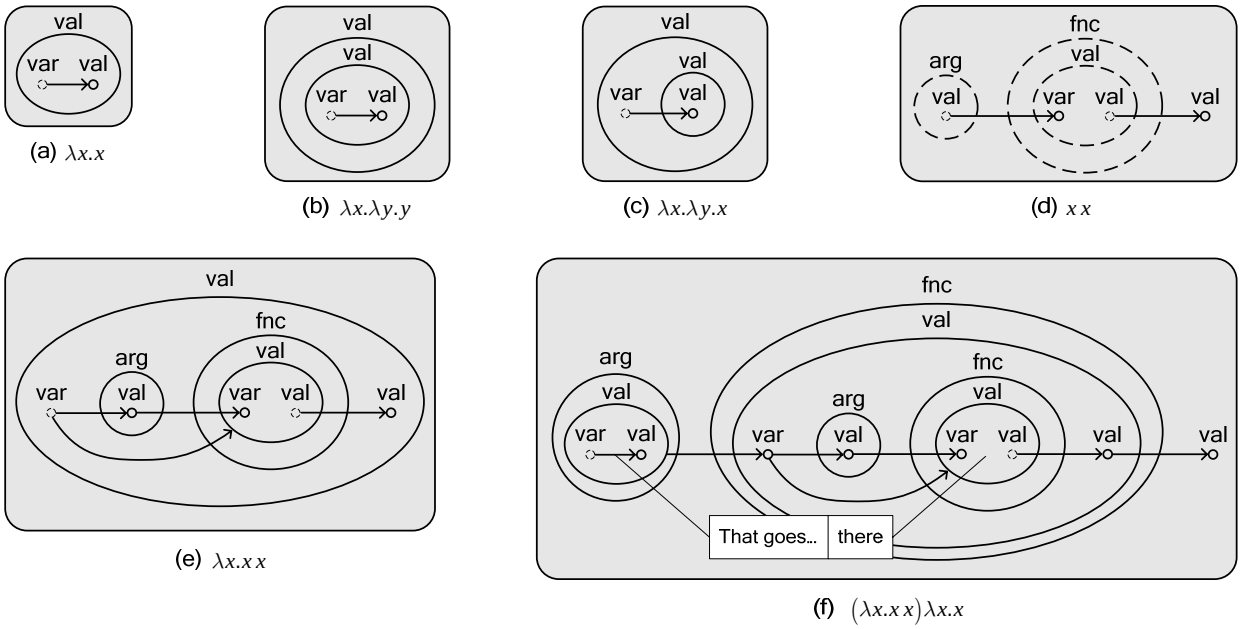

Figure 32. Translation examples

Pure lambda calculus can be translated into pure integral trees, demonstrating that the latter are computationally complete. Recall that given an infinite set of variables *V*, the set Λ of λ-terms is defined by the abstract syntax:

$$\Lambda = V \mid \lambda V.\Lambda \mid \Lambda\,\Lambda$$

The translation is defined in Figure 33. Four labels are used: val, var, fnc, and arg. Environments record variable bindings, defined as finite partial functions from variables to paths: $Env = V \rightharpoonup Path$. The translation of a subterm *M* is $p[\![M]\!]\eta$, where *p* is the contextual location in the translated tree, and η is the binding environment in effect. $[\![M]\!]$ is shorthand for $\cdot[\![M]\!]\{\}$. A term *M* is translated into the initial tree $\langle [\![M]\!], ? \rangle$.

$$\text{With } p \in Path \quad x \in V \quad M,N \in \Lambda \quad \eta \in Env$$

$$p[\![x]\!]\eta = \begin{cases} \{p.\mathsf{val} : \eta(x),\ p : \varnothing\} & \text{if } x \in dom(\eta) \\ \{\} & \text{otherwise} \end{cases}$$

$$p[\![\lambda x.M]\!]\eta = p.\mathsf{val}[\![M]\!](\eta \oplus \{x : p.\mathsf{val}.\mathsf{var}\}) \cup \{p : \varnothing\}$$

$$p[\![M\ N]\!]\eta = p.\mathsf{arg}[\![N]\!]\eta \cup p.\mathsf{fnc}[\![M]\!]\eta \cup \begin{Bmatrix} p.\mathsf{val} : p.\mathsf{fnc}.\mathsf{val}.\mathsf{val}, \\ p.\mathsf{fnc}.\mathsf{val}.\mathsf{var} : p.\mathsf{arg}.\mathsf{val}, \\ p : \varnothing \end{Bmatrix}$$

Figure 33. Translation of lambda-calculus

Figure 32 diagrams some translation examples. The identity function is translated in (a). Unlike lambda calculus reduction, terms are not consumed in the process of evaluation – the translation of every term contains a val subtree which integrates into the value of the term. Lambda abstractions are wrapped inside a val because they serve as literal values in the lambda calculus. The bound variable of a lambda abstraction is translated into a var leaf. When a lambda term is applied, its var field will include the argument value. But the lambda term in (a) is not being applied, so var has no inclusion, and is not actually present in the tree, even though it is referenced as the source of val.val. Such gaps in the tree are diagrammed with dashed lines.

The identity function in (a) simply copies the variable into the value of the function. (b) and (c) show the effects of variable binding on nested expressions. Application is sketched in (d), using unbound variables so as to reveal the basic pattern. The function term is translated into the fnc subtree, and the argument term is translated into the arg subtree. Note that what is to be applied is the value of the function term to the value of the argument term. Accordingly, the value of the argument is copied into the variable of the value of the function. The value of the value of the function is then copied to the value of the entire term.

(e) shows the result of wrapping (d) in a lambda abstraction to create a closed term, and (f) shows the result of applying this abstraction to the identity function. The key point to observe is the way in which inclusion merging simulates higher-order function application. The identity function flows through fnc.val.var into fnc.val.fnc.val. There the var-to-val arrow of the identity function gets merged in, completing the gap in the horizontal pipeline of arrows. The completed pipeline allows the identity function to flow through to the value of the entire term.

The soundness of the translation is established by the fact that if a lambda term *M* reduces to a normal form *N*, then the translation of *M* can integrate into a tree whose val branch equals that of the translation of *N*. Soundness is stated in the following theorem, where the *eval* function is β-reduction to normal form.

Theorem : If $eval(M) = N$ then

$\exists T \mid \langle [\![M]\!], ? \rangle \twoheadrightarrow_{\int} T \ \wedge \ Inc(T[\cdot \mathsf{val}]) \oplus \{\cdot : \varnothing\} = [\![N]\!]/\mathsf{val}$

where $Inc(\langle I, B \rangle) = I$

APPENDIX B. MERGING

This appendix defines the Merge primitive function, that merges entire trees based on how they have changed since they diverged from a common ancestor. Merging has not yet been implemented in Subtext. Figure 34 shows the merging of two trees, m1 and m2, which are included into the two arguments arg1 and arg2, with the merger computed in result. The merge function determines the closest common ancestral source of the two arguments (which is guaranteed to exist in a well-formed tree, even if it is only ∅). The source of the result is set to that ancestor. The double-headed arrows in the diagram indicate the possibility of a chain of inclusions in between the ancestor and the arguments. The example illustrates each of the possible cases:

1. x is unchanged in each argument, and remains unchanged in the merge.
2. y is changed in only one argument, and that change is inherited by the merge
3. z is changed in both arguments, resulting in a conflict, indicated with the special constant ⁇.There is no conflict if z is changed to the same source in both arguments. An alternative design would treat this as a conflict.
4. a and b are distinct insertions into the arguments, and remain distinct in the merge.

Merging is formalized as follows. Merge and ⁇ are added to *Constant*. The *closest common ancestor* $\mathcal{A}_T(x,y)$ of two integrated subtrees *x* and *y* is defined as the least upper bound on the ancestor relation $\preceq_T^*$:

$$\mathcal{A}_T(x,y) = a \text{ iff } \begin{cases} x \preceq_T^* a \quad y \preceq_T^* a \\ \forall z \left(x \preceq_T^* z \wedge y \preceq_T^* z \right) \Rightarrow a \preceq_T^* z \end{cases}$$

The differences between the arguments are determined modulo monomorphism. The function $\Phi_T(x,y)$ monomorphically maps the inclusions of subtree x to the location y in the same way that the primary integration rule does.

$$\Phi_T(x,y) = I \oplus \{ p : y{*}q \mid \exists p,q \mid (p : x{*}q) \in I \}$$
$$\text{where } \langle I, B \rangle = T[x]$$

The integration rule of the merge function is detailed in Figure 35. The monomorphic projections of the two arguments and their closest common ancestor are calculated. Differences between all three of them become conflicts, whereas changes occurring in only one of the arguments are copied to the result. The source of the result is set to the common ancestor, from which the unchanged state gets subsequently inherited.

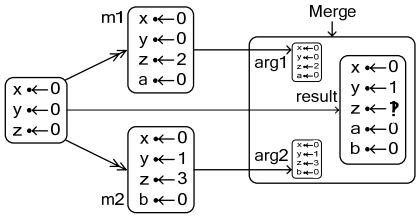

Figure 34. Merge function

$$\frac{T\|p\| = \mathsf{Merge} \quad \mathcal{A}_T(p.\mathsf{arg1},\, p.\mathsf{arg2}) = a \quad T[p.\mathsf{result}] = \Omega}{T \twoheadrightarrow_{\int} T[p.\mathsf{result} := \langle I, ? \rangle] \ \text{ where}}$$

$$I_1 = \Phi_T(p.\mathsf{arg1},\, p.\mathsf{result}) \quad I_2 = \Phi_T(p.\mathsf{arg2},\, p.\mathsf{result})$$

$$I_a = \Phi_T(a,\, p.\mathsf{result})$$

$$C = \{ x : ? \mid \exists y,z,w \mid I_a(x) = y \neq I_1(x) = z \neq I_2(x) = w \neq y \}$$

$$I = (I_1 - I_a) \oplus (I_2 - I_a) \oplus C \oplus \{\cdot : a\}$$

Figure 35. Merge rule

Note the similarity of the logic of merging with that of version control systems [13][39]. The previous paper [10] proposed to use merging as a version control mechanism as well as a form of multiple inheritance, and even as a kind of runtime conditional. It is used in this paper to model parallelism (§4.2).