Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2008-061 October 10, 2008

Modular Generation and Customization

Jonathan Edwards

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

Modular Generation and Customization

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

Abstract

Modularity and flexibility can conflict in multi-language systems. For example, the templates commonly used to generate web pages must be manually updated when the database schema changes. Modularity can be improved by generating web pages automatically from the database schema, but it is hard for such a generator to produce the same variety of outputs that are easily achieved by ad hoc edits to a template. Ideally, such ad hoc edits would be abstracted into transformations that compose with the generator, offering both modularity and flexibility. However common customizations cannot be abstracted using the standard techniques of textual identifiers and ordinal positions.

These difficulties are distilled into a challenge problem to evaluate potential solutions. A solution is proposed based on *field trees*, a new data model for software artifacts that provides persistent identifiers and unshifting positions within sequences. But using field trees with conventional programming languages and development environments requires more effort than the ad hoc editing they seek to supplant. Field trees are therefore extended into *differential trees*, which integrate artifacts and their transformations into a unified representation.

Categories and Subject Descriptors D.2.11 [*SOFTWARE ENGINEERING*]: Software Architectures; D.2.7 [*SOFTWARE ENGINEERING*]: Distribution, Maintenance, and Enhancement; D.2.3 [*SOFTWARE ENGINEERING*]: Coding Tools and Techniques; D.2.6 [*SOFTWARE ENGINEERING*]: Programming Environments

General Terms Languages, Design

Keywords Generative programming, transformations, refinement, customization, templates

1. Introduction

Complex systems are often built from multiple specialized languages and representations [8, 24]. For example a web system might use Java, SQL, HTML, and PHP. There are good reasons for language diversity: specialized syntax and semantics can make it easier to express certain aspects of a design, leverage specialized tools, and standardize interfaces between systems. Unfortunately diversity can also lead to disharmony. This paper focuses specifically on the problem of modularity between artifacts written in multiple languages. Changes in one artifact can lead to the need for manual changes in others because there is duplicated information or interdependencies between internal structures.

For example, web pages are often generated with a template language like PHP that embeds fragments of executable language syntax within HTML. These fragments can dynamically extract fields from a database record and expand them into HTML form fields. But when the database schema changes, as when adding or deleting fields, many templates may need to be manually adjusted. We could make the system more modular by generating HTML pages directly from the database schema. However to vary the design of the generated pages we would have to customize or parameterize the generator, which can be complicated and difficult. In contrast it is easy to implement arbitrary customizations with ad hoc edits to templates. It seems that we face a conflict between modularity and flexibility.

Such conflicts have been intensively studied within single languages, resulting in effective techniques, but the multi-language case is less well developed. This paper makes four contributions:

1. The above web programming scenario is refined into a simple challenge problem in order to evaluate different approaches to multi-language modularity.
2. To address common problems in existing approaches, a new abstraction of sequential structure is proposed: *positional sequences*, which provide unshifting positions.
3. Positional sequences are generalized into *field trees*, a model of software artifacts that solves the challenge problem by abstracting generation and customization into modularly composable transformations.

```
{id: "1234",  name: "John Smith",  phone: "555-1212"}
```

Figure 1. The Data object.

```
<table>
  <tr>
    <td>id</td>
    <td><input type="text", value="1234"/></td></tr>
  <tr>
    <td>name</td>
    <td><input type="text", value="John Smith"/></td></tr>
  <tr>
    <td>phone</td>
    <td><input type="text", value="555-1212"/></td></tr>
</table>
```

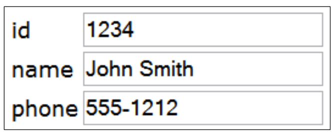


Figure 2. Generic HTML: Form.

4. Field trees are extended into *differential trees*, which integrate artifacts and their transformations into a unified representation.

The Subtext project [12, 14] is developing a programming environment based on differential trees that seeks to make modular transformation of software artifacts nearly as easy to use as ad hoc editing. The guiding hypothesis of this research is that competing with the ease and flexibility of ad hoc editing will require an approach with the utmost simplicity and conceptual coherence. This paper reports on progress made toward that goal.

2. The Challenge

The purpose of this challenge is to present an example of a commonplace multi-language modularity problem that is as simple as possible, so that the problem can be more clearly seen, and potential solutions more easily compared. Figure 1 defines an object Data as a JavaScript object literal with three text fields: id, name, and phone. Figure 2 shows an HTML fragment, called Form, that displays these fields in a table (the surrounding boilerplate HTML has been elided). The browser rendering of Form is shown below it.

We want to customize this generic layout to move the id field below the name field, and to change the label on the name field to be customer. Note that form customizations in practice often involve more complex structural changes, such as forming groups, dividing into multiple frames, aligning to a grid, etc. Figure 3 shows the customized HTML, called CustomForm, with deletions struck-through and insertions in bold. These two HTML forms could be produced in a number of ways, the most common in practice being the use

<table>
~~<tr>~~
~~<td>id</td>~~
~~<td><input type="text", value="1234"/></td></tr>~~
<tr>
<td>~~name~~**customer**</td>
<td><input type="text", value="John Smith"/></td></tr>
<tr>
<td>id</td>
<td><input type="text", value="1234"/></td></tr>
<tr>
<td>phone</td>
<td><input type="text", value="555-1212"/></td></tr>
</table>

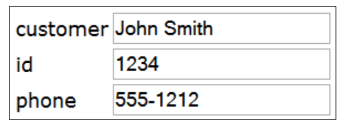


Figure 3. Customized HTML: CustomForm.

{~~id~~ **account**: "1234", **company: "Acme, Inc.",**
name: "John Smith", ~~phone: "555-1212"~~}

Figure 4. Evolved object Data′.

```
<table>
  <tr>
    <td>account</td>
    <td><input type="text", value="1234"/></td></tr>
  <tr>
    <td>company</td>
    <td><input type="text", value="Acme, Inc."/></td></tr>
  <tr>
    <td>name</td>
    <td><input type="text", value="John Smith"/></td></tr>
</table>
```

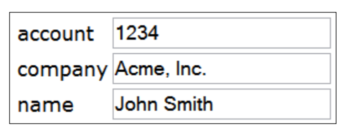


Figure 5. Evolved generic HTML: Form′.

of a template language like PHP that extracts the data fields dynamically, using distinct templates for each form. The CustomForm template would typically be constructed from Form by copy and paste edits. High levels of duplication in web applications have been confirmed by a study [30].

The difficulty arises when the database schema evolves. Figure 4 shows the evolved version of Data, called Data′. The id field has been renamed to account, the company field has been inserted, and the phone field has been deleted. We expect the forms to evolve correspondingly, into Form′ in

```
<table>
  <tr>
    <td>account</td>
    <td><input type="text", value="1234"/></td></tr>
  <tr>
    <td>company</td>
    <td><input type="text", value="Acme, Inc."/></td></tr>
  <tr>
    <td>namecustomer</td>
    <td><input type="text", value="John Smith"/></td></tr>
  <tr>
    <td>account</td>
    <td><input type="text", value="1234"/></td></tr>
</table>
```

company	Acme, Inc.
customer	John Smith
account	1234

Figure 6. Evolved customized HTML: CustomForm′.

$$
\begin{tikzcd}
\text{Data} \arrow[r, "\text{Evolve}"] \arrow[d, "\text{Generate}"] & \text{Data}' \arrow[d, "\text{Generate}"] \\
\text{Form} \arrow[d, "\text{Customize}"] & \text{Form}' \arrow[d, "\text{Customize}"] \\
\text{CustomForm} & \text{CustomForm}'
\end{tikzcd}
$$

Figure 7. Transformations.

Figure 5 and CustomForm′ in Figure 6. If we were using templates, we would need to manually edit both of them to get the correct result. That is easy in this case, but in practice such changes can be far more widespread, laborious, and error-prone.

> The challenge is to provide a solution that adapts automatically to any schema evolution involving insertion, deletion, and renaming of fields.

2.1 Modular generation and customization

The approach that will be proposed in this paper is to provide two transformations Generate and Customize that compose as shown in Figure 7. We will refer to a *generating transformation* as one that maps artifacts across languages in a multi-language system, and a *customizing transformation* as one that maps between artifacts in the same language.

It is easy to write the Generate transformation: simply iterate over the data fields and map them into the corresponding HTML in the same order. The heart of the problem is getting the Customize transformation to do the same thing on Form′ as on Form: move the id field (which has been renamed to account) to be after the name field, whose label is changed to customer. This can be seen as a problem of modularity: being able to abstract the intentions of Customize so that they are preserved when composed with Generate on different sources.

2.2 The difficulty

As will be discussed in the related work section, the challenge is difficult because it undermines the basis of many approaches:

1. *Textual identifiers.* The renaming of id to account will baffle code looking for the id field by name.
2. *Ordinal position in sequences.* Inserting the company field shifts the position of the name field, so if the id field is moved to the same ordinal position as before it will end up above the name field, not below it.

Both of these issues warrant further clarification. First, it is acceptable to handle the renaming of id by an automatic refactoring on the transformation code that replaces all occurrences of id with company. But refactoring is not possible if the code uses the string literal "id", which in general can not be distinguished from uses of the same string as data, error messages, etc. Some mechanism for identifying or parameterizing field names is needed to support refactoring.

The second clarification is about the requirements on the move operation. It is required that the ordering between fields in CustomForm must be preserved in CustomForm′, regardless of arbitrary insertions, deletions, and renamings in Data′ (including deleting name). This requirement might be criticized as too strict, because the order of fields on a screen is not really important. Try telling that to a UI designer! More to the point, order is of the essence in many software artifacts, as in the order of statements in a program. It is reasonable to expect that software transformations not alter relative orderings when exposed to changes in their sources.

2.3 An ad hoc solution

An ad hoc solution can be assembled from some common engineering practices, but it does not generalize well. Basically, "markers" are added into the source and generated artifacts so that transformations can get handles on them. Each <tr> tag in Form can be given a *name marker* as an ID attribute naming the Data field it corresponds to. This allows Customize to find the HTML related to id in order to move it. To handle field renaming, Customize can not rely on literal strings to match names, but must use a level of symbolic indirection, perhaps via a set of static constants defining each name's string. Customize needs to know where to move id to. This position could be determined by inserting a *positional marker* into Data after the name field, wrapped inside a specially formatted JavaScript comment. Generate would notice such comment markers, and map them through into the generated HTML, wrapped as an HTML comment, or perhaps with some tag that browsers will ignore.

While technically a solution to the challenge, this approach does not generalize well, for a number of reasons. The syntax for embedding markers is different for each language, and comment-wrapped markers are fragile and invisible to many tools. The programmer must invent unique marker names for source locations involved in a transformation, an annoying imposition. Generated markers must be made unique, often using concatenative "name mangling", which depends upon language and application conventions. Generating markers is problematic in cases such as the concatenation of two sequences. Customizing transforms need markers within generated artifacts, but these can only be created by inserting markers within source artifacts at the proper place. The proper place depends upon the internal implementation of the generator, and may not even exist, as in the case of concatenating a sequence with itself.

Marker techniques are common in practice, but the author knows of no tool or framework that can solve the challenge problem without being first extended. The general solution proposed in this paper can be seen as an attempt to "do markers right".

3. A General Solution

A general solution emerges from the observation that the difficulties of the challenge problem surround notions of *identity*. Textual identifiers can be renamed, and so do not provide a stable identity. Ordinal positions in sequences can shift, and so do not provide a stable way to locate elements.

We can provide a stable replacement for textual identifiers by storing our artifacts in some kind of database which assigns persistent internal IDs. A specialized editor coupled to the database must be provided to allow editing of the artifacts. A number of the approaches described in the related work section also take this step, which offers additional software engineering benefits beyond the scope of this paper.

3.1 Positional sequences

The problem of shifting positions in sequential structures is rooted in the fact that we treat sequences as integer-indexed arrays. To solve this problem we introduce an alternative abstraction of sequential structure. A *positional sequence* is a sorted map from a domain of identifiers called *positions*. Positions are totally ordered and dense, like the rational numbers. To insert an element into a sequence, a unique new position is allocated between the positions of the adjacent elements. Assigning an element to a position in a sequence replaces the current element at that position, or inserts it if there is none. The crucial property provided by positions is that insertions, deletions, and lookups at different preallocated positions are independent of each other, regardless of execution order. This property helps transformations on sequences to be composed modularly.

We are interested in *global positioning*: all positions used within a database are totally ordered, and new positions are globally unique. Global positioning makes sequences commensurable, which as we will see is crucial to abstracting the Customize transform. A brute force implementation could use infinite precision rational numbers, but would have to track them in an index, and would suffer from the fact that the bit length of rationals grows unboundedly under repeated insertion at a fixed location. The current implementation [14] assigns an incrementing 64 bit serial number to positions for global uniqueness. Positions are organized into a tree, with the children of a position being ordered before it, by ascending serial number. Positions are encoded as a path of serial numbers from the root of this notional tree. More compressed encodings of positions could be achieved with periodic global repositioning.

Positional sequences provide the essential missing ingredient to solve the challenge problem. It can be solved by storing the artifacts in a persistent data model that incorporates both stable identifiers for objects and stable positions within sequences. One approach would be to add positional sequences as a new kind of collection class in an OODB. Alternatively, we could take a database tailored for software artifacts like Molhado [29] and substitute positional sequences for its array-based ones. Such approaches could solve the challenge problem with the least incremental investment in infrastructure. However the goal of this paper is not to minimize cost but to optimize conceptual coherence. Accordingly, we will generalize positional sequences into a uniform model for representing software artifacts: *field trees*.

3.2 Field trees

Field trees use nested positional sequences to unify the notions of objects, fields, and sequences. A field tree is a tree whose non-root nodes are labeled with positions. The nodes are called *fields*. Fields with the same parent must have different positions. Non-root fields are optionally assigned *values* drawn from some fixed set (here numbers and strings). We can make several observations:

1. Fields are uniquely identified by the path of positions down from the root of the tree. Position paths serve as objects ID's.
2. A field serves as a positional sequence of its children, which are said to be contained in it.
3. Fields in different containers can have the same position. A container thus serves as a record/structure, with positions identifying the members unambiguously.

Positions can optionally be named with strings, which need not be unique. To simplify the presentation we will temporarily assume that all positions have unique names.

```
1 id: "1234"
2 name: "John Smith"
3 phone: "555-1212"
```

a) Data

```
01 tag: "table"
02 contents
03   id
04   | tag: "tr"
05   | contents
06   |   c1
07   |   | tag: "td"
08   |   | contents
09   |   |   c2: "id"
10   |   c3
11   |     tag: "td"
12   |     contents
13   |       c4
14   |         tag: "input"
15   |         type: "text"
16   |         value: "1234"
17   name [...]
18   phone [...]
```

b) Form

Figure 8. Data and Form as field trees.

```
01 tag: "table"
02 contents
03   id [...]
04   name
05   | tag: "tr"
06   | contents
07   |   c1
08   |   | tag: "td"
09   |   | contents
10   |   |   c2: "name" "customer"
11   |   c3
12   |     tag: "td"
13   |     contents
14   |       c4
15   |         tag: "input"
16   |         type: "text"
17   |         value: "John Smith"
18   id2 [...]
19   phone [...]
```

Figure 9. CustomForm as a field tree.

3.3 The field tree solution

Figure 8 shows how Data and Form are represented as field trees, displayed as outlines that a structure-aware editor might use. Each field is on a separate line, with the name of its position indented to show the containment structure of the tree. The root of the tree is implicit — its children are the un-indented lines. Vertical lines connect sibling fields. The Data object is the trivial tree in (a) containing three leaf fields with positions named id, name, and phone in that order. The values assigned to the fields are shown following a colon to the right of their position names.

To represent HTML as a field tree, a DOM-like convention for encoding XML into a regular tree structure is adopted, shown for Form in Figure 8(b). Each XML tag is encoded as a subtree containing the field tag, which is assigned the name of the tag as a string value, as in line 1. The XML contents of the tag are placed within the contents field on line 2. XML attributes are encoded as sibling fields of tag and contents, as in the <input> tag on lines 15 and 16. A conventional XML DOM would treat the contents field as an array or collection, which in field trees are replaced by positional sequences. The arbitrarily named positions c1, c2, c3, and c4 have been allocated to properly sequence the elements of the contents fields. These arbitrary names will be discarded later when we discuss anonymous positions.

Note that the contents of the <table> tag use the same positions as the Data fields that they correspond to: id on line 3, name on line 17, and phone on line 18. For brevity, the name and phone subtrees have been collapsed, indicated by the ellipsis buttons on their right. As will be seen, the positional correlation between Data and Form is the key to solving the challenge. Field trees enable this correlation by unifying records and sequences. The Generate transform simply iterates over the Data fields and maps each one to a corresponding position in Form, plugging in the proper <td> sub-tree, as shown in lines 3–16 for the id field.

Figure 9 shows how Customize transforms Form into CustomForm. The label of the name field is changed on line 10. The id subtree on line 3 has been deleted. A new position id2 is allocated between name and phone, which is used to insert the id subtree of the form at line 18. The schema evolution will allocate a new position company in between id and name, which will map through to the form, but it is guaranteed not to disrupt the relative order of the other form fields, including id2. This is because the same positions are used in the data and the form, and allocating a new position does not change the relative ordering of existing positions. Note that these new positions are allocated once when the transformations are written, not every time they execute.

A Java implementation of the solution is presented in Appendix A. It is assumed that the field tree database provides a Position enum that symbolically defines each position with the proper internal identifier. The id component of the form is found unambiguously despite its being renamed during the evolution because the name of a position can be changed without changing its identity. Code compiled with the old version of the Position enum will still run correctly. Preferably, renamings in the field tree would invoke a rename refactoring on the enum to co-evolve the code.

3.4 Anonymous positions

We must revisit the earlier assumption that all positions have unique names. The crucial use of these names was in the transformation code as symbolic constants for referenced positions. The positions c1, c2, . . . are particularly troubling, as we don't want to have to invent symbolic names every time we insert something in a sequence. But we also don't want to hard-code hex strings for internal position IDs.

This problem could be solved by storing the Java transformation program itself inside the same field tree database, perhaps by encoding its AST as a field tree. References to positions would become special AST nodes containing the internal identifier of the position. The position would be fed to the compiler as an expression encoding the position's internal identifier, but would be presented in the program editor using the position's symbolic name. If the position lacked a name, non-textual techniques could be used, such as a hyperlink to a field with the proper position. The effect is to allow "non-textual literals" in the program — constants defined by the environment without a textual encoding. This approach would require a big infrastructure investment, but it establishes that in principle we do not need symbolic names for positions. It also eliminates the need to refactor the code when a position is renamed — position names become a feature of the user interface, not the semantics.

Field tree summary: Field trees solve the challenge problem by providing stable identities and positions so that transformations expressed in those terms can be composed modularly. The programmer is not required to provide markers because they are already built in. Field trees serve as a common *medium* for artifacts in different languages that supports modular transformations between them.

4. An Integrated Solution

Field trees allow the challenge problem to be solved with a conventional language like Java, but at a cost in complexity that is still too high to compete with ad hoc editing of templates. Coding transformations in Java or even specialized transformation languages is complex to start with, and field trees add the need to properly map positions through the transformations. In pursuit of simplicity and conceptual coherence, field trees can be generalized into *differential trees*, which integrate field trees and declarative specifications of their transformations into a single unified representation.

Differential trees declaratively specify field tree transformations that automatically establish stable positional correspondences between artifacts. They also offer deep copying and overriding of trees, as well as arbitrary functional computation. As with the prior Java solution, in order to have non-textual literal references into field trees, the transformation language must itself be embedded in the field tree. This necessity is made a virtue: differential trees are "the LISP" of field trees, living within them and using them as its "S-expressions". The result is a unified model of software artifacts and their transformations.

Differential trees extend field trees in two ways:

1. Fields can point to other fields. Their value can be not only a string or number but also a path of positions denoting another field.
2. The value assigned to a field defines its contents and usage in four different ways, called modes, to be detailed below. These modes are distinguished in the outline with variants of the colon character: a double colon, a colon-equals, and a double-colon-equals.

4.1 Differential trees by example

Figure 10 shows a series of examples explaining the interpretation of differential trees. The assignments of values to fields are seen as a set of *definitions* whose implications are worked out, a process called *integration*.

Figure 10(a) shows an example of two definition modes: colon and double-colon. The fields a and b contained in x are defined with a single colon to be leaf nodes with the value 1. The field y is defined using a double-colon to be a deep copy of the field x. The result of integrating this tree is diagrammed below as it might appear in a user interface for differential trees.[1] The fields a and b have been copied from x into y in lines 5 and 6. The arrows on the right indicate references between fields, and serve as non-textual literals for anonymous fields.

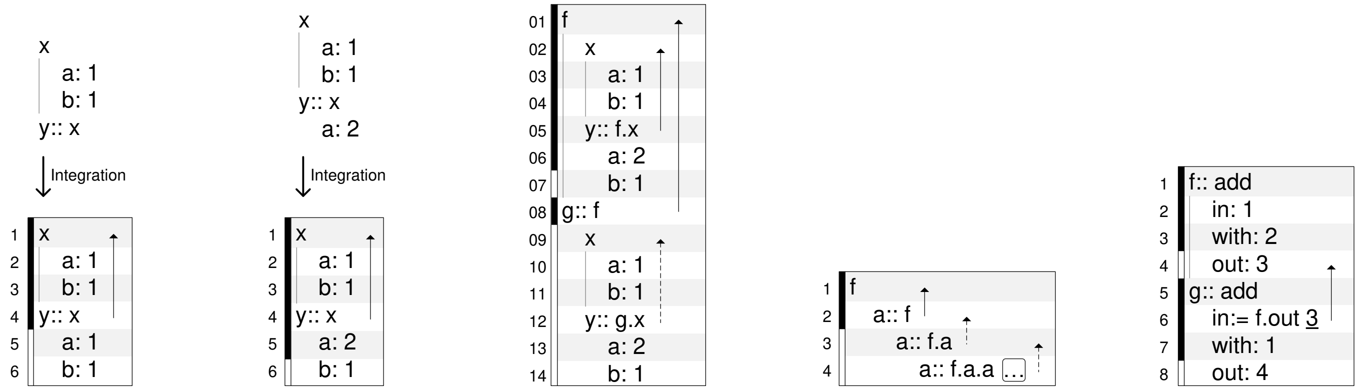
Note the black bars in the left margin. Integration is declarative, and only "fills out" the initial differential tree with additional derived definitions, never altering the initial definitions. The initial definitions are distinguished from the derived ones by the black bars. The fact that integration leaves the original program intact means that programming can be done by directly editing the "live" execution displayed in the outline, as with a spreadsheet. Accordingly, the initial state of the differential tree will be omitted in subsequent examples, as it is equal to the barred lines.

Figure 10(b) shows how copied structures are incrementally overridden. The copy of x into y from the previous example has its a field overridden to 2. Only the b field on line 6 gets inherited. Differential trees construct transformations by layering overriding copies in this way, somewhat like inheritance and overriding in OO languages except that it can delve deeply into sub-trees. Every definition expresses an overriding difference between its containers and their definitions — hence the term "differential".

Figure 10(c) shows what happens when the entire tree from 10b is copied. Field f on line 1 contains the prior tree in lines 2–7. Field g on line 8 copies f, yielding lines 9–14. Note that when the definition of y on line 5 gets copied to line 12, its referenced value changes from f.x to g.x to maintain the same relative location within the copy. The end result is the same in this case, but would not have been if there were some change inside g.x. Copying preserves the internal structure of copying and overriding isomorphically, and can be said to be "higher-order" [16]. References outside of the tree being copied are not changed but are "captured", as in a closure. The arrow on the right of line 12 is dashed to indicate it has been inherited.

Figure 10(d) shows that copying can be recursive. Copying a structure into itself produces an infinitely deep tree.

[1] These diagrams are "paper prototypes" of the UI under development



a) copying b) overriding c) higher-order copying d) recursive copying e) computation

Figure 10. Differential trees by example.

Integration is implemented lazily, and limits tree depth analogously to a stack depth limit. The outline display shows the use of an expander button hiding a portion of the tree.

Higher-order copying and recursion make differential trees Turing-complete, demonstrated by an embedding of Lambda calculus [13]. Although theoretically unnecessary, for convenience we add primitive functions. Figure 10(e) shows an example of the add primitive. Field f on line 1 is a call to add, which is a structure containing three fields: in, with, and out (used by convention in all functions). The in and with fields are expected to have rational number values. When the tree is integrated, the out field on line 3 is defined as the value of the sum.

The field g on line 5 is another call to add which adds 1 to the result of the prior addition. The two additions are linked by the definition of g.in on line 6 using the := mode. That mode tells the addition function to use as its argument the value of f.out, which will be 3. A chain of := definitions will be followed until a : or :: definition is found. Values pass between := definitions like the way values pass through conventional variable assignments, hence the use of the standard assignment operator :=. To visualize function execution, the resolved value 3 is displayed underlined on the right of line 6. The fourth definition mode, ::=, combines tree copying with value chaining, but will not be used in this paper.

Appendix B contains a formal semantics of differential trees as used in this paper. Two alternative implementations of differential trees were used in prior research on programming environments [12, 14]. These implementations added a number of convenience and safety features such as private fields that cannot be overridden in instances. A technical report [13] describes advanced features of the implementations not considered here, including mutable state and the incorporation of change history. Of note is that mutable state allows user inputs on forms to be mapped backwards through generation and customization transformations to the source data.

4.2 The differential tree solution

The differential tree solution of the challenge problem is shown in Figure 11. It does the same thing as the prior Java solution in field trees, but more succinctly and declaratively. The Generate transform at line 5 uses the map primitive to apply a function over each of the Data fields. The map primitive's in parameter is bound to Data on line 6, and the func parameter is defined on lines 7–23. The contents of the <table> tag in Form are copied from the collected outputs of the mapped functions at line 26. CustomForm is derived from Form at line 27. The id component of the form is deleted at line 29 by assigning the special value delete to it. It is reinserted at location id2 at line 35. The label of the name field is overridden to be "customer" at line 34.

Figure 12 drills into the execution of the mapping of the id field. The mapping is constructed inside the body structure on lines 8–29, where each element takes the position of an input element and contains an instance of the mapped function (on lines 9, 28, and 29). The mapping of the id element on line 9 is fully expanded. Its in field is linked to the corresponding input element on line 10. The mapped function is expected to construct an appropriate HTML tree in its out field at line 14. Lines 15–27 are essentially a template of an HTML <tr> tag, where the label for the field is defined at line 20 and its value is defined at line 27. The label definition uses a call to the reflective primitive valueName at line 11, which determines the string name of the input field's position. This internal call is given the position name f1 to make the presentation clearer, but in practice would be anonymous. The value definition on line 27 just copies the value from the input field on line 10.

The outputs of each mapped function are collected into the overall output of the mapping at lines 30–33, each automatically given the same position as the corresponding input elements. As we saw in the previous section, it is this positional correspondence between Data and Form that allows

```
Data
  id: "1234"
  name: "John Smith"
  phone: "555-1212"
Generate:: map
  in:: Data
  func
    f1:: valueName
      in: Generate.func.in
    out
      tag: "tr"
      contents
        c1
          tag: "td"
          contents
            c2:= Generate.func.f1.out
        c3
          tag: "td"
          contents
            c4
              tag: "input"
              type: "text"
              value:= Generate.func.in
Form
  tag: "table"
  contents:: Generate.out
CustomForm:: Form
  contents
    id:= delete
    name
      contents
        c1
          contents
            c2: "customer"
    id2:: Form.contents.id
```

Figure 11. Differential tree solution of challenge problem.

```
Data [...]
Generate:: map
  in:: Data
    id: "1234"
    name: "John Smith"
    phone: "555-1212"
  func [...]
  body
    id:: Generate.func
      in:= Generate.in.id "1234"
      f1:: valueName
        in: Generate.body.id.in
        out: "id"
      out
        tag: "tr"
        contents
          c1
            tag: "td"
            contents
              c2:= Generate.body.id.f1.out "id"
          c3
            tag: "td"
            contents
              c4
                tag: "input"
                type: "text"
                value:= Generate.body.id.in "1234"
    name:: Generate.func [...]
    phone:: Generate.func [...]
  out
    id:: Generate.body.id.out [...]
    name:: Generate.body.name.out [...]
    phone:: Generate.body.phone.out [...]
Form
  tag: "table"
  contents:: Generate.out [...]
CustomForm:: Form [...]
```

Figure 12. Detailed mapping of id.

Customize to survive the evolution of Data. All transformation primitives create such correspondences. For example, concatenation of sequences p and q generates tupled positions $\langle 1, p_i \rangle \ldots \langle 2, q_i \rangle$. Note that new positions can be stably inserted between such synthetic positions. Ensuring stable unique correspondences through transformations are a key benefit of differential trees.

While Figure 12 may appear complex, consider that what we are doing is essentially browsing an execution trace of a typical higher-order mapping, showing the complete detail of all computations and data flows. One benefit is "debugging by browsing" [12]. Another is the automatic provision of traceability [11].

4.3 From templates to transformations

The Subtext project is developing a programming environment based on differential trees with the goal of making *transformative programming* as easy as ad hoc copy and paste editing of templates. Ironically, differential trees are similar to templates in the way they intermix literal artifact structure with computation. But differential trees express their computations declaratively, with layered overriding copies that, in a sense, are abstractions of those same copy and paste operations. The core concept of differential trees is that all structure arises from higher-order abstract copy and paste operations. The hypothesis is that this conceptual coherence, and its congruence with actual program editing practices, will enable a compelling realization of transformative programming.

5. Related Work

Visual UI builders have long used marker techniques, as described in section 2.3, to allow their generated code to be customized within limits. Code generation techniques are the subject of Generative Programming [10], while Model Driven Engineering [21] focuses on generation from models. These approaches handle customization through Roundtrip Engineering [4], which attempts to map such customizations back into the source representation so that regeneration will preserve them. The limitation of Roundtrip Engineering is that it only handles customizations which can be expressed in the source domain, implying that the target domain adds no expressive power. But a major reason for the use of multiple languages is precisely that they add expressive power. Another view of Roundtrip Engineering is that it extends the source language with annotations or metadata, which may include fragments from target languages. But that still depends on there being some place in the source artifact to attach such metadata.

The Magritte [31] web framework is representative of this latter approach, providing a Smalltalk dialect of "descriptions" which can be attached as metadata to a schema. The challenge problem might be addressed by attaching to each data field a description containing its HTML label string and its ordinal position in the form. But when the company field is added, the ordinals must be manually re-numbered, violating the requirement of automatic adaptation. The metadata approach does not generalize to more complicated customizations that alter the structure of the HTML, such as dividing the fields across multiple tables, where a simple ordinal would no longer suffice.

Language features to modularize customization have been intensively studied, but only language-neutral approaches relevant to multi-language systems will be cited here. These include Architecture Description Languages [26], hyperslices [33], feature models [9, 23], aspectual features [2, 18, 27], XVCL [19], and invasive software composition [3]. A representative approach is AHEAD [5], which uses nested equations to specify hierarchical overriding as in differential trees. XAK [34] refines XML by requiring name markers to be inserted within a base XML document to establish refinement points. Feature Software Trees [1] represent the hierarchical structure of artifacts like field trees, and provide hierarchical overriding, called superimposition [6], as in differential trees. All of the approaches in this paragraph focus on the customization of source artifacts through linking, overriding, or weaving, applied to named program points. It is ideal when such named points exist, but presuming upon programmers to provide such points, especially if refactoring is required, risks losing out to copy and paste. These approaches do not construct arbitrary generating transformations, nor address how to furnish generated artifacts with named customization points.

The need for persistent identifiers in software artifacts has long been recognized, and is supported in software repositories like Molhado [29] and modeling facilities like the OMG MOF. There are a number of approaches to providing stable positions in sequences, although they have not been used in databases or repositories. Emacs maintains stable markers in text buffers. Text editing frameworks offer similar constructs, for example javax.swing.text.Position. Operational transformation [15] and Darcs [32] make sequence operations commutative through compensating adjustments to their ordinal positions. Interruptible Iterators [25], and the Apache CursorableLinkedList provide stable collection iterators. C5's views [22] can serve as cursors. None of these constructs are intended to be persistent identifiers, though perhaps they could be serialized as such. They offer only sequence-local positions, which do not help correlate generated artifacts with their sources as global positions do.

There are many specialized transformation languages, like XSLT, ATL [20], and Stratego/XT [7]. They depend upon parsing or pattern matching of the source, making it hard to abstract customizations in the presence of renaming and shifting. Recent bidirectional transformation languages [17, 28, 35] have appealing functional semantics and address the mapping of changes through transforms. They lack a model of stable sequence positions, but perhaps could be applied to field trees as an alternative to differential trees. Differential trees can make deep changes within structures, whereas these languages take the functional approach of deconstructing top-down and then reconstructing bottom-up.

6. Conclusions

This paper spotlights a problem that deserves greater attention: the expression of ad hoc customizations as abstract transformations. Differential trees are a novel approach, but it is hoped that others will be stimulated by the challenge. The conclusions of this paper are:

1. Multi-language systems could be made more modular while retaining flexibility by replacing copy and paste programming with composable transformations that generate and customize.
2. Many customizations refer to locations not stably defined by the standard techniques of textual identifiers and ordinal positions. These difficulties are distilled by the challenge problem, and overcome with the help of a new abstraction of sequential structure: *positional sequences*.
3. *Field trees* generalize positional sequences into a common medium for artifacts in different languages that supports modular transformations between them.
4. *Differential trees* unify field trees and their transformations into a single coherent model as the basis of *transformative programming*.

Acknowledgments

Discussions with Daniel Jackson, Derek Rayside, Emina Torlak, and Eunsuk Kang were helpful. Helpful comments were offered by Damien Pollet, Kevin Reid, Nat Pryce, and John Zabroski.

A. Java field tree solution

```
import java.util.TreeMap;
import java.util.Map.Entry;

public class Main {
  // simulate database generation of symbolic
  // position values
  enum Position {
    id, name, id2, phone, tag, contents, type,
    value, c1, c2, c3, c4
  }

  /* Field trees are represented with a
   * TreeMap<Position, Object>, where the
   * Objects are either sub-trees or boxed leaf
   * values.
   */
  static class Tree extends
      TreeMap<Position, Object> {
    Tree deepCopy() {
      Tree copy = new Tree();
      for (Entry<Position, Object> e :
          this.entrySet()) {
        if (e.getValue() instanceof Tree) {
          Tree value =
            ((Tree) (e.getValue())).deepCopy();
          copy.put(e.getKey(), value);
        } else {
          copy.put(e.getKey(), e.getValue());
        }
      }
      return copy;
    }
  }

  public static void main(String[] args) {
    // Simulate load of data from database
    Tree data = new Tree();
    data.put(Position.id, "1234");
    data.put(Position.name, "John Smith");
    data.put(Position.phone, "555-1212");
    Tree form = generate(data); // generate
    Tree form2 = customize(form); // customize
  }

  static Tree generate(Tree in) {
    Tree tableContents = new Tree();
    for (Entry<Position, Object> e :
        in.entrySet()) {
      // name of data field
      Tree c1Contents = new Tree();
      c1Contents.put(Position.c2,
        e.getKey().toString());
      // input field loaded from data
      Tree c1 = new Tree();
      c1.put(Position.tag, "td");
      c1.put(Position.contents, c1Contents);
      Tree input = new Tree();
      input.put(Position.tag, "input");
      input.put(Position.type, "text");
      input.put(Position.value,
        in.get(e.getKey()));
      Tree c3Contents = new Tree();
      c3Contents.put(Position.c4, input);
      Tree c3 = new Tree();
      c3.put(Position.tag, "td");
      c3.put(Position.contents, c3Contents);
      Tree trContents = new Tree();
      trContents.put(Position.c1, c1);
      trContents.put(Position.c3, c3);
      // map to data's position
      Tree tr = new Tree();
      tr.put(Position.tag, "tr");
      tr.put(Position.contents, trContents);
      tableContents.put(e.getKey(), tr);
    }
    Tree out = new Tree();
    out.put(Position.tag, "table");
    out.put(Position.contents, tableContents);
    return out;
  }

  static Tree customize(Tree in) {
    Tree out = in.deepCopy();
    Tree contents =
      (Tree) out.get(Position.contents);
    // move id field
    contents.put(Position.id2,
      contents.remove(Position.id));
    // change name->customer
    Tree t = contents;
    t = (Tree) t.get(Position.name);
    t = (Tree) t.get(Position.contents);
    t = (Tree) t.get(Position.c1);
    t = (Tree) t.get(Position.contents);
    t.put(Position.c2, "customer");
    return out;
  }
}
```

B. Differential Tree Semantics

This appendix defines the semantics of differential trees as used in the paper. The goal is to precisely explain the essential nature of differential trees, not to prove formal properties, nor to model an actual implementation. A "big-step" style is used that is mute about errors, lapsing into undefinedness. The more complex small-step semantics in a prior technical report [13] detects error conditions explicitly and extends the semantics in several directions.

B.1 Definitions

We take a set of positions $\mathcal{P}$ containing the rationals $\mathbb{Q}$, Booleans $\mathbb{B}$, characters $\mathcal{C}$, and all position tuples $\langle p_1, \ldots\ldots p_n \rangle$. Strings are character tuples. $\mathcal{P}$ also contains the predefined positions add, map, valueName, in, with, out, body, and delete. $\mathcal{P}$ has a total dense ordering $\leq$, which is consistent with the natural orders of $\mathbb{Q}$, $\mathbb{B}$, $\mathcal{C}$, and the dictionary order on tuples. $\mathcal{P}$ contains extra positions in between all of the aforementioned ones to allow arbitrary insertions, but we will treat all positions as preallocated here.

A $Path$ is a finite non-empty sequence of positions, written using the dot operator as $p_1 . p_2 \ldots p_n$. Notation will be abused to treat positions interchangeably with the singleton path containing them, and the dot operator is overloaded to append positions as well as concatenate paths. The length of a path x is $\mathrm{len}(x)$. The last position of a path x is $\mathrm{leaf}(x)$. The name of a position p is $\mathrm{name}(p)$, which is the empty string for anonymous positions, and is the expected print string for integers, Booleans, strings, and tuples.

B.2 Differential trees as relations

A differential tree over P can be seen as a subset of $Path \times Mode \times Path$ where each tuple represents a definition. The left hand paths must be unique, and $Mode$ is the set $\{:, ::, :=, ::=\}$. To express the semantics of differential trees, we will add a natural number qualifying each definition, called its *provenance*. Because definitions can be stacked at multiple heights in the tree, inheritance can occur in multiple overriding layers. A definition of a field x with provenance n has been inherited from the definition of the n^{th} container of x, whose path is the prefix of x with length $(\mathrm{len}(x) - n)$. If $n = \mathrm{len}(x)$, the definition is inherited from the root of the tree, which means it is an initial definition specified by the programmer. If $n = 1$, then the definition was inherited from its immediate container. If $n = 0$, then the definition was not inherited at all, but was internally computed by a primitive function. The rule is that the definition with the highest provenance overrides all others.

We express the semantics as inference rules on the relation $|\,| \subseteq Path \times \mathbb{N} \times Mode \times Path$ where the natural numbers are the provenances. This relation contains all the initial definitions, with their provenance set to the length of the left hand path, which guarantees they will override all derived definitions.

Overriding is determined by the quaternary predicate $\lceil\,\rceil$ defined as:

$$\lceil x\, n\, d\, y \rceil \equiv |x\, n\, d\, y| \wedge \forall m. \left(|x\, m\, \_\, \_\,| \;\Rightarrow\; m \leq n \right)$$

B.3 Integration

Integration is defined by the single inference rule:

$$\frac{\lceil x\, \_\, \_\, y \rceil \quad \lceil y.z\, n\, d\, w \rceil \quad n \geq \mathrm{len}(z)}{|x.z\, \mathrm{len}(z)\, d\, \phi\,|}$$

$$\text{where } \phi = \begin{cases} x & \text{if } w = y \\ x.u & \text{if } \exists u \,|\, w = y.u \\ w & \text{otherwise} \end{cases}$$

This rule states that if x is defined as the path y (after overriding), and somewhere within y there is another definition, then the corresponding location within x will inherit that definition, subject to two provisos. The first proviso is that only definitions with a provenance at least as high as y will be inherited from it. In other words, inheritance from internal definitions within y will be ignored, since they will be recapitulated within x, perhaps differently. The other proviso is that the value of the definition to be inherited depends on whether it is located within y or not, which is the conditional definition of ϕ. If the value is located within y it is mapped to the corresponding location within x. Otherwise it is "captured" as is.

B.4 Primitive functions

Primitive functions are specified in additional rules that create 0-provenance definitions, which prevents them from being inherited rather than being recalculated. Recall that function parameter fields can be linked to other fields with the $:=$ and $::=$ definition modes. The helper function ref determines the value to be used, called the field's *reference*, by chasing down those links.

Determining the reference of a field involves another complication: the value of a field is allowed to be a path that traverses into a leaf node. The path beneath the leaf will be followed starting at the value of the leaf. This means that the value of the leaf is being "dereferenced" — allowing a path to represent an arbitrary traversal of pointers within the tree. Dereferencing is done by the loc helper function.

Note that dereferencing is deferred untill a function needs it, rather than being taken care of during integration (as in the implementation). What this means is that a leaf field, defined by a $:$ or $:=$ definition, may not physically be a leaf: any substructure of its value will be copied into it, but then later ignored by the loc function. This approach makes the integration rule simpler, and in fact corresponds to the conceptual model of the user interface, where a leaf can be expanded to see the contents of its value, just as if it had been copied into it.

$$\mathrm{loc}(p) = p$$

$$\mathrm{loc}(x.p) = \begin{cases} y.p & \text{if } \lceil \mathrm{loc}(x)\ \_\ :\ y \rceil \\ y.p & \text{if } \lceil \mathrm{loc}(x)\ \_\ :=\ y \rceil \\ \mathrm{loc}(x).p & \text{otherwise} \end{cases}$$

where $p \in \mathcal{P}$.

$$\mathrm{ref}(x) = \begin{cases} y & \text{if } \lceil \mathrm{loc}(x)\ \_\ :\ y \rceil \\ y & \text{if } \lceil \mathrm{loc}(x)\ \_\ ::\ y \rceil \\ \mathrm{ref}(y) & \text{if } \lceil \mathrm{loc}(x)\ \_\ :=\ y \rceil \\ \mathrm{ref}(y) & \text{if } \lceil \mathrm{loc}(x)\ \_\ ::=\ y \rceil \\ \bot & \text{otherwise} \end{cases}$$

The add function adds the values of its in and with fields and sets the sum into its out field:

$$\frac{\mathrm{ref}(x) = \mathsf{add} \quad \mathrm{ref}(x.\mathsf{in}) = n \in \mathbb{Q} \quad \mathrm{ref}(x.\mathsf{with}) = m \in \mathbb{Q}}{|x.\mathsf{out}\ 0\ :\ (n+m)|}$$

The valueName function returns the name of the leaf of the value of a location, which is often used to represent the value symbolically in print strings and the UI.

$$\frac{\mathrm{ref}(x) = \mathsf{valueName} \quad \mathrm{ref}(x.\mathsf{in}) = y \quad \lceil y\ \_\ \_\ z \rceil}{|x.\mathsf{out}\ 0\ :\ \mathrm{name}(\mathrm{leaf}(z))|}$$

The map function rule fires for each non-deleted sub-field p of its in parameter. It instantiates a copy of the func parameter as body.p, binding its body.p.in parameter to the sub-field. The output of the function is collected into out.p. Unlike the implementation, deletions are not filtered out immediately during integration, but later by the map and other functions that enumerate sequences.

$$\frac{\begin{array}{cc} \mathrm{ref}(x) = \mathsf{map} & p \in \mathcal{P} \\ \lceil x.\mathsf{in}.p\ \_\ \_\ \_ \rceil & \neg \lceil x.\mathsf{in}.p\ \_\ :=\ \mathsf{delete} \rceil \end{array}}{\begin{array}{l} |x.\mathsf{body}.p\ 0\ ::\ x.\mathsf{func}| \\ |x.\mathsf{body}.p.\mathsf{in}\ 0\ :=\ x.\mathsf{in}.p| \\ |x.\mathsf{out}.p\ 0\ ::\ x.\mathsf{body}.p.\mathsf{out}| \end{array}}$$

References

[1] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *ETAPS Intl. Symp. on Software Composition*, 2008.

[2] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE*, 2006.

[3] U. Aßmann. *Invasive Software Composition*. Springer-Verlag, 2003.

[4] U. Aßmann. Automatic Roundtrip Engineering. *Electronic Notes in Theoretical Computer Science*, 82(5), 2003.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. on Software Engineering*, 30(6), 2004.

[6] J. Bosch. Superimposition: a component adaptation technique. *Information and Software Technology*, 41 (5), 1999.

[7] M. Bravenboer, K. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming (to appear)*, 2008.

[8] P. Clements and L. Northrup. *Software product lines*. Addison-Wesley, 2002.

[9] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE*, 2005.

[10] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications*. Addison-Wesley, 2000.

[11] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.

[12] J. Edwards. Subtext: Uncovering the simplicity of programming. In *OOPSLA*, 2005.

[13] J. Edwards. First Class Copy & Paste. Technical report, MIT CSAIL, 2006. URL http://hdl.handle.net/1721.1/32980.

[14] J. Edwards. No ifs, ands, or buts: uncovering the simplicity of conditionals. In *OOPSLA*, 2007.

[15] C. Ellis and S. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2), 1989.

[16] E. Ernst. Higher-order hierarchies. In *ECOOP*, 2003.

[17] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL*, 2005.

[18] W. Harrison, H. Ossher, and P. Tarr. General Composition of Software Artifacts. In *Intl. Symp. on Software Composition*, 2006.

[19] S. Jarzabek and L. Shubiao. Eliminating redundancies with a "composition with adaptation" metaprogramming technique. In *ESEC/FSE*, 2003.

[20] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Model Transformations in Practice Workshop at MoDELS'05*, 2005.

[21] S. Kent. Model Driven Engineering. In *Intl. Conf. on Integrated Formal Methods*, 2002.

[22] N. Kokholm and P. Sestoft. *The C5 generic collection library for C# and CLI*. The IT University of Copenhagen, 2006.

[23] M. Laguna, B. González-Baixauli, and J. Marqués. Seamless development of software product lines. In *GPCE*, 2007.

[24] R. Lammel and E. Meijer. Mappings Make Data Processing Go 'Round. In *Generative and Transformational Techniques in Software Engineering*, 2005.

[25] J. Liu, A. Kimball, and A. Myers. Interruptible iterators. *POPL*, 2006.

[26] J. Magee and J. Kramer. Dynamic structure in software architectures. In *FSE-4*, 1996.

[27] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *FSE-12*, 2004.

[28] S. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. *ASIAN Symposium on Programming Languages and Systems*, 2004.

[29] T. Nguyen, E. Munson, and J. Boyland. The Molhado hypertext versioning system. In *ACM conf. on Hypertext & hypermedia*, 2004.

[30] D. C. Rajapakse and S. Jarzabek. An investigation of cloning in web applications. In *Intl. Conf. on Web Engineering*, 2005.

[31] L. Renggli, S. Ducasse, and A. Kuhn. Magritte – a meta-driven approach to empower developers and end users. *Intl. Conf. On Model Driven Engineering Languages And Systems*, 2007.

[32] D. Roundy. Darcs: distributed version management in Haskell. In *ACM SIGPLAN workshop on Haskell*, 2005.

[33] P. Tarr, H. Ossher, W. Harrison, and J. Sutton, S.M. N degrees of separation: multi-dimensional separation of concerns. *ICSE*, 1999.

[34] S. Trujillo, D. Batory, and O. Diaz. Feature refactoring a multi-representation program into a product line. In *GPCE*, 2006.

[35] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *ASE*, 2007.