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:

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

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.

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