User-Friendly Functional Programming for Web Mashups

Rob Ennals David Gay

Intel Research Berkeley
2150 Shattuck Avenue, Penthouse Suite
Berkeley, CA 94704
{robert.ennals,david.e.gay}@intel.com

Abstract

MashMaker is a web-based tool that makes it easy for a normal user to create web mashups by browsing around, without needing to type, or plan in advance what they want to do.

Like a web browser, Mashmaker allows users to create mashups by browsing, rather than writing code, and allows users to bookmark interesting things they find, forming new widgets — reusable mashup fragments. Like a spreadsheet, MashMaker mixes program and data and allows ad-hoc unstructured editing of programs

MashMaker is also a modern functional programming language with non-side effecting expressions, higher order functions, and lazy evaluation. MashMaker programs can be manipulated either textually, or through an interactive tree representation, in which a program is presented together with the values it produces.

In order to cope with this unusual domain, MashMaker contains a number of deviations from normal function languages. The most notable of these is that, in order to allow the programmer to write programs directly on their data, all data is stored in a single tree, and evaluation of an expression always takes place at a specific point in this tree, which also functions as its scope.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Very high-level languages; H.4.3 [*Information Systems Applications*]: Information Browsers

General Terms Languages, Human Factors, Design, Management

Keywords Mashup, web, end-user programming, browser

1. Introduction

There has recently been lots of interest in so called "mashup sites" – web sites that combine information, processing, or visualizations from several web sites to provide information the user could not easily obtain by manually browsing the base web sites separately. One of the first such sites was HousingMaps.com, which uses a map from Google Maps to visualize houses available for rent on Craigslist.org. Other examples include WeatherBonk.com which combines various sources of information about weather, BidNearby.com which finds items being sold nearby on sites such as EBay.com and Craigslist.org, DiggDot.com which combines Digg.com with Slashdot.org, and WikiMapia.com which combines WikiPedia.org with Google Maps to provide information about physical locations. At the time of writing, ProgrammableWeb.com lists around 2,100 known mashup sites and the number is growing rapidly.

While there are a large number of mashup sites in existence, there are many more that could usefully be created. For example, as far as the authors are aware, no existing mashup site can answer any of the following questions[1]:

- Which of these houses on Craigslist has lots of good restaurants nearby according to Yelp, and would be less than a 30 minute commute to work according to Google Maps?
- Which of these events on Upcoming clash with things on my Google Calendar?
- How much would each of these recipes from Epicurious cost to make if I bought the ingredients at Safeway?
- How much of my weekly expenditure according to Bank of America goes to companies who donate money to political parties I don't like according to OpenSecrets?
- What is the best route through town according to GoogleMaps that allows me to visit highly rated shops according to Yelp that sell suggested Christmas presents appropriate for each of my friends according to FindGift?

It seems reasonable to assume the number of such useful questions users might ask is huge. New web sites, containing new information, appear on the web every day, and the number of possible ways of combining them is huge, particularly when one considers the affect of combining three, or four, or more different web sites to answer a question. At the time of writing, ProgrammableWeb lists over 450 web sites with published APIs intended to be used by mashups, and millions more web sites can be used via scraping (Schrenk 2007).

If each mashup site were only able to answer a single question, based on a fixed set of source web sites, then it seems the number of mashup sites needed would be impractical — both for the programmers who create them, and the users who have to find them. The obvious solution seems to be generic mashup sites that allow end users to easily combine information from multiple web sites to answer a wide range of questions. Indeed several groups have built such generic mashup creation sites (see Section 5).

In this paper, we describe MashMaker, a tool that makes it easy to create mashups, for users ranging from naïve to expert. The

ICFP'07 October 1–3, 2007, Freiburg, Germany.

[1] If such a mashup site does exist then that merely goes to underscore one of our other points — having such a large number of mashup sites makes it very hard to find a site that can answer a specific question.

design of Mashmaker is guided by a number of principles that support this goal:

- Program as you browse: creation of mashups should be viewed as an extension of the normal web browsing habits.
- Direct manipulation: users should be able to work directly on the data that they are interested in, without having to think about abstract concepts such as programs.
- Least surprise: local changes should produce local effects.
- Pay as you go: unskilled users should be able to gain some benefit with very little effort, but more skilled users should be able to do powerful things.
- Code sharing: mashups, and elements used to create mashups should be shared across users.

MashMaker is a combination of a custom functional programming language and a web-based user interface. We believe mashups are an excellent application domain for a programming language. Moreover, many web sites can be modeled as functions from form parameters to structured results. As a result, the majority of mashups combine side-effect-free queries and list manipulation, making them an excellent application domain for a functional programming language.

In addition to functional languages, Mashmaker draws inspiration from spreadsheets, web browsers, and file systems. Time has shown that these are all metaphors that normal users are able to use productively, and as a result, all three have been very successful. We explain in Section 2 how each of these tools has influenced the design of MashMaker, in ways that support the principles outlined above.

1.1 The Language and the Interface

The heart of MashMaker is a functional programming language. A simplified grammar for this language is given in Figure 1. MashMaker has several significant differences from previous functional languages. Most of these differences stem from the fact that the MashMaker language is designed to be accessed through a spreadsheet-style user interface, rather than as a textual program.

Like a spreadsheet, MashMaker stores every value that is computed in a single, central data structure (in this case a tree). There are no intermediate values that get thrown away. This tree contains every intermediate value that was computed on the way to producing the final result. Even the values computed internally by a function can be made visible by expanding the function call.

MashMaker has no concept of a scope, or of a local variable. An expression is evaluated at a specific location in the data tree, and all variable references refer to fields at the current location. This is again similar to a spreadsheet, in which cell references are relative to the current location, and temporary values are stored directly in the data.

The language also has some more conventional features: it is dynamically typed, function arguments are bound by name rather than order, evaluation is lazy, and functions are first class values. We describe the MashMaker language in more detail in Section 3.

The novel features of MashMaker are not just in the core language, but also in the way that we expose this language through a user interface. In particular:

- MashMaker's tree view shows the value that the current expression evaluates to. Each node in this tree contains an internal reference back to the subexpression that defined that node. This is made practical by the fact that the MashMaker language considers each evaluation to have a logical position in the data tree. (Section 2.3).

Value			
v	$::=$	c	constant (file)
	$\mid$	$\emptyset$	empty directory
	$\mid$	$v \ltimes (k \mapsto v)$	extension
	$\mid$	$s, \lambda(x_0 \ldots x_n).e.x$	closure with body e, env s and result selector x
Subnode Key			
k	$::=$	x	Property with name x
	$\mid$	$\bullet$	Unnamed child
Expression			
e	$::=$	c	constant
	$\mid$	$\emptyset$	empty directory
	$\mid$	$e \ltimes (q \mapsto e)$	extension
	$\mid$	$\lambda(x_0 \ldots x_n).e.x$	lambda
	$\mid$	$e\ (e)$	function application
	$\mid$	this	current context
	$\mid$	p	link to another node
	$\mid$	X	global
Extension Path			
q	$::=$	k	new subnode
	$\mid$	$*.q$	for all children
	$\mid$	$x.q$	inside property x
Reference Path			
p	$::=$	x	variable
	$\mid$	$x.p$	inside property x
	$\mid$	$!.p$	inside parent
Scope			
s	$::=$	$v_0 : \ldots : v_n$	stack of parent dir values v

Figure 1. Simplified grammar for the MashMaker core language

- MashMaker allows operations such as map, fold, and filter to be applied through direct manipulation of data (Section 2.3.1).
- MashMaker automatically suggests functions that a user might apply to their data, based on the functions that other users have applied to similar data. This allows less skilled users to create mashups simply by clicking on suggestions, without having to do any programming or having to think about what websites might connect together (Section 2.4).
- MashMaker's user interface allows users to write expressions in a simplified form. This form allows users to avoid quoting constants, and automatically infers arguments for lambda expressions (Section 3.5).
- All data is "live", meaning that functions automatically recompute their values in response to changing data (Section 3.6).
- MashMaker bundles functions up together with associated metadata to form "widgets" — reusable mashup fragments (Section 2.4).
- Users can interact with MashMaker at a number of different levels making it useful for anyone from a complete beginner to a skilled programmer (Section 2.5).

1.2 Why this Paper is Interesting

This paper makes several interesting and novel contributions:

- We propose the use of a functional language for the creation of web mashups.

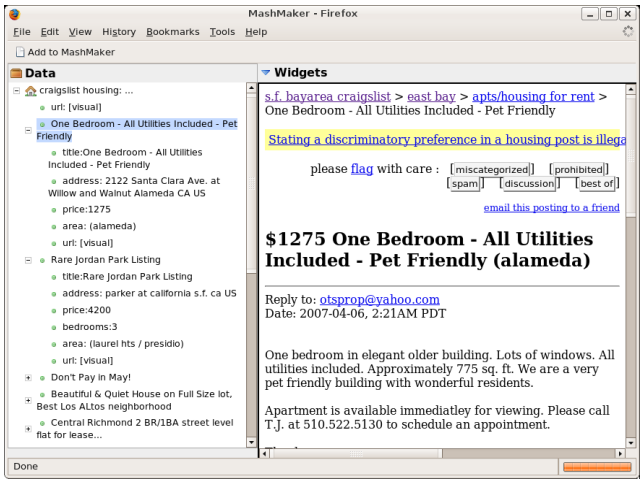

Figure 2. Craigslist apartment listing imported into MashMaker

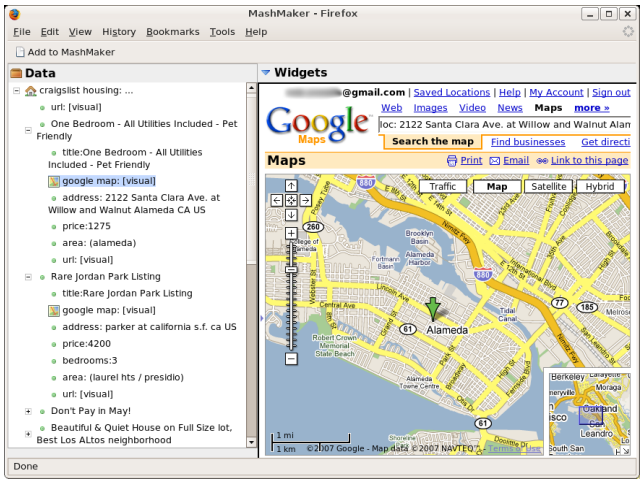

Figure 3. Using a Google Maps visualization for an apartment

- We propose several changes to the standard functional paradigm, in order to make such languages more suitable for this domain. In particular, we present a model made on *tree extension*, in which an expression is evaluated with respect to a location in a data tree.
- We demonstrate that this approach is practical, through the creation of MashMaker, which is, in our opinion[2], the first truly general purpose mashup creation tool that is usable by normal users. MashMaker is currently undergoing a closed beta test program, and we plan to open it up to the general public soon.

2. The MashMaker Design

In this section we sketch the overall design of the MashMaker system, and explain how it draws inspiration from file systems, spreadsheets, and web browsers.

2.1 An Illustrative Example

Since MashMaker is quite different from most functional languages, it is perhaps helpful to set the scene with a walk through of an actual session using the current version of the MashMaker tool.

1. Alice is planning to rent an apartment, so she navigates her normal web browser to Craigslist.com and has a look at the apartments listed there. Alice looks at the apartments listed by Craigslist and would like to know more about them. In particular, she would like to know which apartments have good restaurants nearby. Since Craigslist can't do this itself, Alice realizes she needs to use MashMaker.
2. Alice clicks on the "Add to MashMaker" button on her web browser bookmark bar to launch MashMaker and view the Craigslist housing web page within the MashMaker proxy-browser. MashMaker starts up as an AJAX web application, within Alice's web browser, hosted by the central MashMaker webserver (Figure 2). The right hand window shows the web page Alice is looking at, and the left hand side shows a tree representation of the web site. In this case, there is a node for the Craigslist query, with a child node for each apartment. Each of these apartment nodes has a set of property nodes, expressing properties of that apartment such as its price and the number of bedrooms.

[2] Terms such as "truly general purpose" and "usable by normal" are hard to define formally, so some might disagree with this statement.

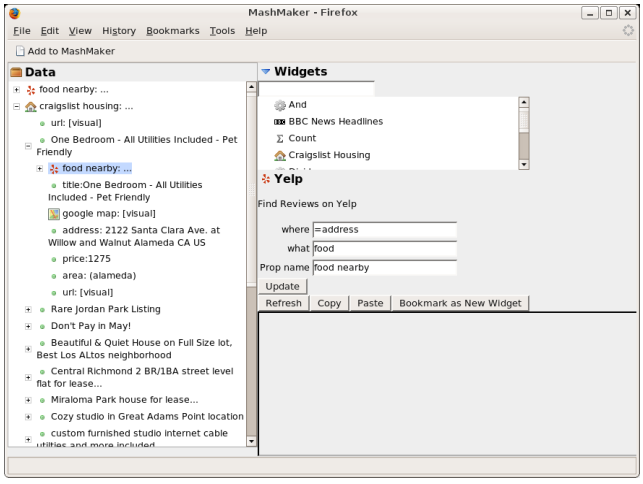

Figure 4. Joining Yelp to Craigslist

3. MashMaker notices that Craigslist apartments are things that users have previously displayed on maps, so it provides a button at the top of the window allowing Alice to add a map to each apartment. Alice clicks on this button to get a map for each apartment (Figure 3).
4. Alice would like to see Yelp restaurant reviews near each apartment, but is disappointed to see that MashMaker has not provided a button to do this automatically. She will thus need to teach MashMaker about this connection herself.
5. Alice navigates to Yelp and searches for food at an arbitrary address. When the result of the query appears, she clicks on "Add to MashMaker" to suck this query page into MashMaker. The resulting Yelp node is called "food nearby" and contains nodes for all restaurants near to the specified address.
6. Alice now has both Craigslist and Yelp in her MashMaker scratch space but they are not yet connected. Alice copies the Yelp node into one of the apartments[3]. She then expands the "form panel" to reveal the form defining the Yelp query, and changes the "address" field of the Yelp form to be "=address" – a reference to the address property of the enclosing apartment

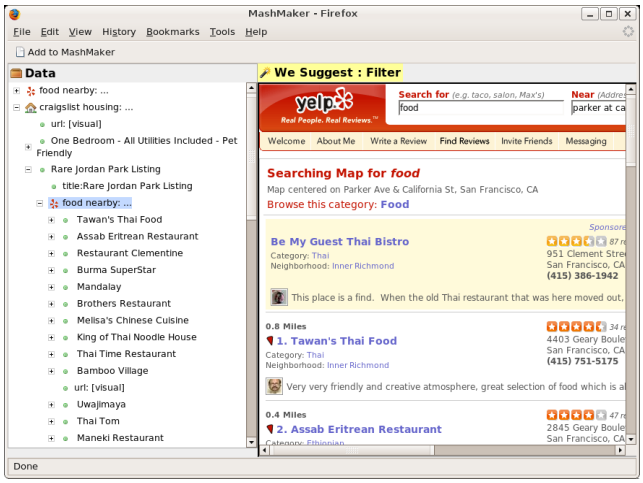

Figure 5. Yelp displays restaurants near the apartment

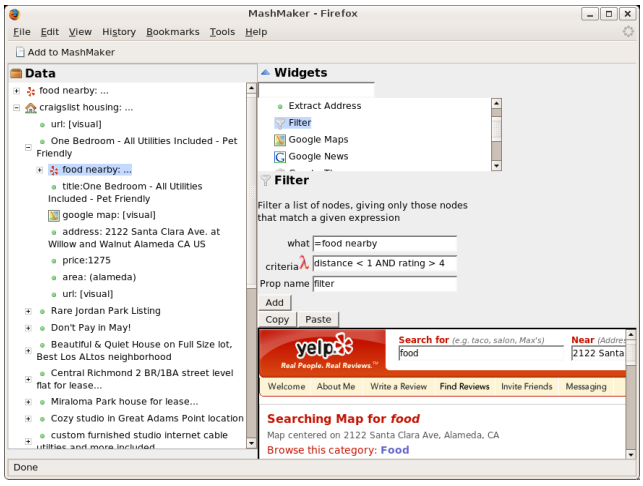

Figure 6. Filtering apartments using a lambda expression

node (Figure 4). Each apartment now has a "food nearby" property, showing the restaurants near to that apartment (Figure 5).

7. Now that Alice has created this "food nearby" node, MashMaker learns that this is a property that users might want to define for Craigslist apartments. In the future, if another user views a Craigslist apartment, MashMaker will provide a "food nearby" button to allow other users to add this same property.

8. Currently "food nearby" shows all restaurants near to each apartment, but Alice is picky, and so is only interested in highly rated restaurants very near to the apartment. MashMaker knows that previous users have applied a filter operation to Yelp listings, so it suggests that Alice apply a filter operation here (Figure 5). Alice clicks on this button and types "`distance < 1 AND rating > 4`" as the condition[4] (Figure 6). Alice could alternatively have used an interactive user interface to compose this query. Alice renames the result of the filter to "good restaurants within walking distance".

9. This new property is dependent on the existence of a "food nearby" property, and so will not be suggested by MashMaker for Craigslist apartments that lack this property.[5] Thinking that this property might be useful for other users searching for apartments on Craigslist, Alice decides to bundle up the new property, together with the other properties it depends on, as a new widget. To do this, she clicks "bookmark as new widget", and tells MashMaker which of the properties that this property depends on should be considered to be arguments, rather than being internal to the widget. In this case, the address is an argument, and "food nearby" is internal. In the future, when another user browses a Craigslist apartment, "good restaurants within walking distance" may be suggested to them.

10. To allow themselves to get an overall view of the quality of each apartment on offer, Alice adds a number of additional properties (commute time to work, crime level, average income) and uses the "calculator" widget to define a scoring metric by combining these features. She then sorts all the data by this metric.

Section 3.2 shows the functional program that this editing session created.

2.2 Learning from File Systems

Like a file system, MashMaker presents all data as a tree (Figure 2). All nodes in this tree are immutable. The underlying implementation uses sharing to avoid duplicating data, but users are encouraged to think of their data as being in a tree.

Like a file system, each node either has its own content (a file), or subnodes (a directory). The content can be of arbitrary type, for example, some text, a number, an expression, an image, a URL. Each subnode is either a *property* with an explicit name, or a *child* with no name. It is assumed that all children will represent things of roughly equivalent type, whereas the types of properties will vary, and correspond to their names.

MashMaker's tree view shows a text summary for each node. If the node is a named property, then this summary is preceded by the property name. The summary for a simple file node is a text summary of the content (e.g. the text for a text node). The summary for a directory node is a user-configurable combination of the summaries for its properties — by default the summary of the first property.

MashMaker's right hand pane shows a visualization of the selected node. If the node is a file, then this will be a visual representation of the file's data. In some cases, this will be a computed representation of other data in the tree, for example a map, a graph, or a table. If the node's content is a URL then that URL is shown. Similarly, text, images, and other visualizations can be shown in this panel..

Figure 1 shows the notation for such tree values in MashMaker's underlying language. We write $\emptyset$ to denote an empty directory, and $v \ltimes (k \mapsto v')$ to denote the directory that results from extending the directory v with a new subnode v' – where k is $\bullet$ if the subnode is a child and a property name if the subnode is a property. As the underlying language is purely functional, all dependencies between nodes are explicit, preserving the principle of least surprise.

For convenience, one can write $[k_0 \mapsto v_0, \ldots, k_n \mapsto v_n]$ as an abbreviation for $\emptyset \ltimes (k_0 \mapsto v_0) \ltimes \ldots \ltimes (k_n \mapsto v_n)$. To illustrate the way this notation works, here is the notation for the value shown in Figure 5:

[3] This corresponds to a "map" operation (Section 2.3.1).

[4] This is actually a lambda expression, despite the absence of lambda symbol or explicit arguments (Section 3.5).

[5] See Section 2.4 to see why.

```
[
  food nearby ↦ . . . ,
  craigslist housing ↦ [
    url ↦ . . . ,
    • ↦ [
      title ↦ "Rare Jordan Park Listing",
      food nearby ↦ [
        • ↦ {title ↦ "Tawan's Thai Food", . . .},
        • ↦ {title ↦ "Assab Eritrean ...", . . .}
      ],
      google map ↦ . . . , . . .
    ], . . .
  ], . . .
]
```

Somewhat unusually for a programming language, but entirely in keeping with a file system, MashMaker allows identifier names to contain spaces.

MashMaker's data representation is also heavily influenced by XML[6]. In particular, the idea of distinguishing between properties and children is taken from XML. The key differences are that, unlike XML, MashMaker allows properties to be arbitrary subtrees, rather than just simple text, and MashMaker nodes do not have tag names. We believe that this representation is simpler for users to understand than XML, but it is close enough to XML that it is easy to encode each in the other.

2.3 Learning from Spreadsheets

One of the key principles behind mashmaker is that users should create mashups through "direct manipulation" of their data, rather than writing an abstract program. To support this goal, we have taken much inspiration from spreadsheets.

Spreadsheets have achieved remarkable success in allowing normal users to write relatively complicated programs. One of the key features that has allowed spreadsheets to do this is their avoidance of unnecessary separation between a program and the data it is working with. In a conventional programming language, the programmer writes a program in isolation of any argument data. As they write this function, they must imagine what it will do with arguments they might give it. By contrast, a spreadsheet avoids this separation by allowing the programmer to apply operations directly to a specific piece of data and then copying the operation to other data if it seems to work.

MashMaker borrows six key features from spreadsheets:

- Program and data mixed in one central workspace.
- All data is stored explicitly in the workspace. There are no "local variables".
- An expression is evaluated at a specific location, and refers to other data relative to its current location.
- Map and fold are described through direct manipulation of data, rather than using abstract functions.
- Text entered by a user is assumed to be a constant, unless proceeded by "=".
- Expressions re-evaluate automatically in response to changing arguments[7].

[6] Originally our plan was to use XML as the data model, but we eventually decided that a simplified model would be easier for users.

[7] There are some systems issue here. In particular, how does one determine that a web site has changed, and how does one avoid querying a web site too often? However these are off-topic for this paper, so we will not discuss them further.

The tree view on the left side of the MashMaker window shows the value that the user's current program evaluates to. Each node is either a *defined node*, meaning that there is a user-specified expression defining its value, or it is a *result node*, meaning that it is part of the result of evaluating a parent defined node. In the user interface, result nodes have a green ball as their icon while defined nodes have an icon corresponding to the function that defines them. A defined node contains an internal reference to the expression that defines it, allowing the user to easily edit this expression. Each expression is evaluated by referring to the values of other nodes near it in the tree.

Just as a spreadsheet allows a user to edit a formula using the formula bar, but avoids displaying the formula in the normal display, MashMaker allows a user to edit the form arguments for a *defined node* using the *form panel* which pops down from the top of the right hand side when a user clicks on its icon. For advanced users, MashMaker also allows the user to turn on a spreadsheet-style formula bar, which displays the current expression textually, in MashMaker's underlying functional language.

2.3.1 Map and Fold

An appealing feature of spreadsheets is the way they allow a user to map a single expression over a collection of objects by simply writing a formula in one cell, and then copying the expression to all data rows below. This approach allows the user to easily see how their expression is evaluated for each data element. MashMaker takes a similar approach. If a user adds a new property to a node, then similarly defined properties are automatically created for all sibling nodes. For example, in Section 2.1, when the user added a google map to one apartment, a map was automatically added to all other apartments. All these expressions use the same defining expression, and when the user uses the form panel to change the arguments they change the arguments for all the other replicas too. As in a spreadsheet, a MashMaker user can easily look at a specific application of the expression to some data and see directly how it evaluates.

Similarly, spreadsheets provide an easy way for users to `fold` an operation over a collection of objects. `fold` is infamous within the functional programming community as being a difficult function for beginners; they either can't remember the argument order, or they have difficulty thinking about exactly what a function will do when folded over a collection of data. However those same users have little difficulty performing `fold` operations in a spreadsheet. In a spreadsheet, all a user has to do to fold their function over data rather than simply map it is write an expression that refers to the cell above, and then read out the final value from the bottom. Following this example, MashMaker allows users to write fold-like operations using the built in "prev" property to refer to the previous sibling. If there is no previous sibling then "prev" returns the value of the user-defined "init" property, or an empty value if this is not defined.

MashMaker also appropriates spreadsheet syntax for distinguishing constants from expressions within the user interface. By default, any text entered by a user is assumed to be a string constant, unless it is proceeded by "=". We chose this default partly because it is what users are familiar with and partly to avoid the need for beginning users to learn about expressions before they can edit form arguments. This feature is not part of the textual representation of the MashMaker language — it is simply part of the user interface.

2.4 Learning from Web Browsers

Another guiding principle of MashMaker is "program as you browse", meaning that creating a mashup should feel like the normal web browsing process.

The web browser is perhaps the most successful user interface of modern times. This simple interface allows users to find information and perform sophisticated queries by merely following a sequence of links and occasionally typing data into forms.

MashMaker follows this model as much as possible. When a user is looking at a particular node, MashMaker will automatically suggest additional functions that they might want to apply by providing buttons across the top of the view pane (Figure 5). Clicking on one of these buttons will insert a new node whose defining expression extracts data from the data already available (e.g. "food nearby" finds its address argument from the existing address property).

The expectation is that most users of MashMaker will never use the keyboard or expand the widget pane. Instead they will explore their data entirely by clicking on MashMaker suggestions. In effect, MashMaker extends the web browsing experience by adding new links that users can follow.

The suggestions that MashMaker makes are derived from observations of functions that other users have previously applied to similar-looking data, following the code sharing principle. For example, in Section 2.1, once one user had added a node to a Craigslist apartment whose defining expression was `food nearby = Yelp([address = address, what = "food"])`, MashMaker will automatically suggest adding a node with the same definition for all other Craigslist apartments.

Another behavior that MashMaker borrows from web browsers is bookmarks. If a user is using a web browser, they can use bookmarks to remember interesting pages that they have found, or to share an interesting page with friends. The equivalent behavior in MashMaker is user-defined widgets. To create a user defined widget, the user navigates to an interesting result that they would like to remember for later or share with friends and clicks "bookmark as new widget" (Figure 4). MashMaker will then prompt the user to select which of the nodes that the result depends on should be considered arguments and which should be considered internal to the function. Non-expert users will typically opt for the default — all nodes the result depends on are internal, causing the entire mashup to be bookmarked, just as if the user had bookmarked a normal web page.

This approach allows users to experiment with their function on real data before abstracting it as a function. Perhaps more importantly, it allows users to browse around aimlessly, looking for something interesting, without necessarily having to think in advance that they might be going to create a function at the end. Even when they do create a function, our intention is that they think of what they are doing as bookmarking an interesting discovery, rather than writing a function.

2.5 Multi-Level MashMaking

We have designed MashMaker with the intention that it should be usable by anyone from a complete novice to an expert programmer (following the "pay as you go" principle). In particular, we anticipate that users will use MashMaker at the following levels, where each level requires a little more skill from the user and allows the user to do more powerful things:

1. Basic Users: Never unfold the widget panel. They explore their data purely by clicking on "Add to MashMaker" in their browser, and clicking on suggestion buttons to add enhancements to their data. Basic users will sometimes bookmark things they find, using the default bookmark settings (Section 2.4). Basic users do not even have to see the tree on the left, since all the information they need is visible in the view pane on the right.
2. Normal Users: Occasionally expand the widget panel to edit form parameters. The changes they make flow through into the suggestions made to all users.
3. Skilled Users: Connect up new sites that have not previously been connected, using copy and paste and simple expressions that refer to other properties.
4. Semi-Expert Users: Use semi-automated scraper-creation tools (not yet written) to create scraper widgets for new web sites.
5. Expert Users: Write complex expressions directly in MashMaker's core language.
6. Gurus: Teach MashMaker how to understand the content of new websites, either by uploading a hand-written XML description (the current state), or by using an interactive tool (the future).

We expect that each category will contain an order of magnitude fewer people than the previous category. However, even though the number of highly skilled users may be small, their presence is essential since it is they who import the web sites and write the functions that less skilled users later use.

3. The MashMaker Language

In the previous sections we have explained the general model of how MashMaker works and the mental model that it presents to a user. In this section we will describe in more detail the functional programming language that is at the heart of MashMaker, and in particular the concept of directories and tree extension (Section 3.2).

3.1 Core Syntax

Figure 1 gives the grammar for expressions in MashMaker's core language. This grammar deals with a simplified version of the full MashMaker language. In particular we omit the expression forms for special built-in functions such as `prev`.

Values were described in Section 2.2. A value is either a constant file value (of arbitrary type), a directory with property and child subnodes, or a lambda expression. A directory is either empty, or is a smaller directory that has been extended with an additional subnode[8]. It is legal for a directory to contain multiple properties with the same name, however one will only be able to look up the value of the last one.

MashMaker is dynamically typed, in common with the Lisp family of languages, and, of course, spreadsheets. If a dynamic type error occurs then the erroneous node's text summary and view pane will explain what went wrong.

An expression is evaluated with respect to its location in the data tree. The location is represented as a stack of parent values, each of which is a directory. If an expression looks up a variable, the lookup is relative to the current location. The innermost directory value in the location is known as the *context*.

Expressions take the following forms:

- A literal constant c
- An empty directory $\emptyset$
- An extension $e \ltimes (q \mapsto e')$ adds a new subnode to e. e is required to evaluate to a directory, and q is an *extension path* that says where the new subnode should appear. The expression e' is evaluated at e, so e' can refer to any properties in the directory that e evaluates to. We discuss extensions in more detail in Section 3.2.

[8] Note that, unlike normal list concatenation, we add new elements to the end, rather than the beginning of a directory.

For convenience, one can write $[q_0 \mapsto e_0, \ldots, q_n \mapsto e_n]$ as an abbreviation for $\emptyset \ltimes (q_0 \mapsto e_0) \ltimes \ldots \ltimes (q_n \mapsto e_n)$.

- A lambda expression $\lambda(x_0 \ldots x_n).e.x$ defines a closure value whose arguments have names $x_0 \ldots x_n$ and whose body is the expression e. The return value is computed by evaluating e to a directory and then selecting the property called x. MashMaker identifies function arguments by their label, rather than their order.

  For syntactic convenience, one can omit the final "$.x$" part of a lambda expression, meaning that the result is the entire value, rather than some property of it.
- A function application $e\ (e')$ applies e to e'. e must evaluate to a function and e' must evaluate to a directory with properties matching the names of all the arguments of e.
- Writing "`this`", allows an expression to refer directly to the current location as a directory value.
- A path reference p looks up a property relative to the current location. By default, only properties in the current context are matched, but the $!x$ form allows one to search the parent locations.

3.2 Tree Extensions

The key new construct in the MashMaker language is the tree extension:

$$e \ltimes q \mapsto e'$$

This construct is key to the connection between the core MashMaker language and the MashMaker user interface, since it denotes the action of adding new computed nodes to the data tree. In this construct, e is the base of the current tree, q is a path to the position at which the user has inserted a new node, and e' is the expression used to define the new node.

The path q can include the wildcard symbol, "$*$", meaning that the rest of the path should be applied to all children. In fact, there is no way to add a new subnode for just one child node. If one wishes to add a subnode to one child node then one must add it to all of them.

Here is the simple expression that the user created interactively in the example in Section 2.1.

$$
\begin{array}{l}
[\\
\quad \text{houses} \mapsto \text{Craigslist Housing}([\text{area} \mapsto \text{sfbay}]), \\
\quad \text{houses.*.map} \mapsto \text{Google Maps}([\text{address} \mapsto \text{address}]), \\
\quad \text{food nearby} \mapsto \text{Yelp}([\text{what} \mapsto \text{“food”}, \text{where} \mapsto \ldots]), \\
\quad \text{houses.*.food nearby} \mapsto \\
\qquad \text{Yelp}([\text{what} \mapsto \text{“food”}, \text{where} \mapsto \text{address}]), \\
\quad \text{houses.*.good food nearby} \mapsto \\
\qquad \text{Filter}([\text{what} \mapsto \text{food nearby}, \text{how} \mapsto \\
\qquad\quad \lambda(\text{distance}, \text{rating}).(\text{distance} < 1) \wedge (\text{rating} > 5)]), \\
]
\end{array}
$$

Here, we see that the user took Craigslist Housing and extended it to add a new defined node called "map" inside each apartment. The value of each map was computed with reference to the "address" property of that apartment. The user also added an additional top-level node called "food nearby" and defined "food nearby" and "good food nearby" nodes for each apartment. Within the user interface, each of these defined nodes contains an internal reference to the program subexpression that defines it, allowing the user to easily edit function arguments.

The user in that example also defined the following global function:

$$
\begin{array}{l}
\text{Good Food Nearby} = \lambda(\text{address}).(\\
\quad [\\
\quad\quad \text{food nearby} \mapsto \text{Yelp}([\text{what} \mapsto \text{“food”}, \text{where} \mapsto \text{address}]), \\
\quad\quad \text{good food nearby} \mapsto \text{Filter}([\text{what} \mapsto \text{food nearby}, \text{how} \mapsto \\
\quad\quad\quad \lambda(\text{distance}, \text{rating}).(\text{distance} < 1) \wedge (\text{rating} > 5)]), \\
\quad].\text{good food nearby})
\end{array}
$$

We can see that this function was created by copying the "food nearby" and "good food nearby" properties out of the scratch-space data tree, recording the fact that "address" was an external input, and noting that "good food nearby" is the result value.

3.3 Semantics

Figure 7 gives a big-step operational semantics for strict evaluation of MashMaker expressions. The real semantics for MashMaker is lazy, however this strict semantics allows us to more easily present language constructs such as tree extension, without the distraction of lazy evaluation. We discuss lazy evaluation more in Section 3.4.

The evaluation relation is of the following form:

$$s, e \Downarrow v$$

where

- s is the *location* at which the expression is evaluated. This is a stack of parent values, each of which is a directory value. The innermost parent value is known as the *context*. We write $v : s$ to denote a location with context v on top of the rest of the location.
- e is the expression being evaluated
- v is the value that e evaluates to in the location s

Most of these rules should be easy to follow:

- *(CONST)* and *(EMPTY)* are already values, and so do nothing
- *(EXTEND-NEW)* adds a new subnode to a directory. The new node is evaluated in a context that includes all previous properties, but not any properties added to the directory subsequently.
- *(EXTEND-EMPTY)*, *(EXTEND-ALL)* and *(EXTEND-SKIP)* add an extension to all child nodes. *(EXTEND-SKIP)* skips over property nodes, *(EXTEND-ALL)* extends the last child node, and then recursively extends the others.
- *(EXTEND-PROP1)* and *(EXTEND-PROP2)* extend a named property. *(EXTEND-PROP1)* matches the last property and extends it. *(EXTEND-PROP2)* skips over a non-matching property.
- *(LAM)* builds a closure, stashing the current location as the environment.
- *(APP)* applies e to e'. e is evaluated to a closure with body e''. e' is evaluated to a directory v''. A new context is built by extending the closure environment with properties from v'' that match the closure arguments. The closure body e'' is evaluated in this context and the property x is selected from the result.
- *(THIS)* simply grabs the current context value.
- *(VAR)*, *(FIELD)*, and *(PARENT)* follow a path from the current location.

3.4 Lazy Evaluation

Like Haskell (Peyton Jones 2003b), all MashMaker expressions are evaluated lazily. The current consensus in the programming language community seems to now be that lazy evaluation is the wrong evaluation model for conventional programming languages (Peyton Jones 2003a). This is because the bookkeeping overhead of lazy

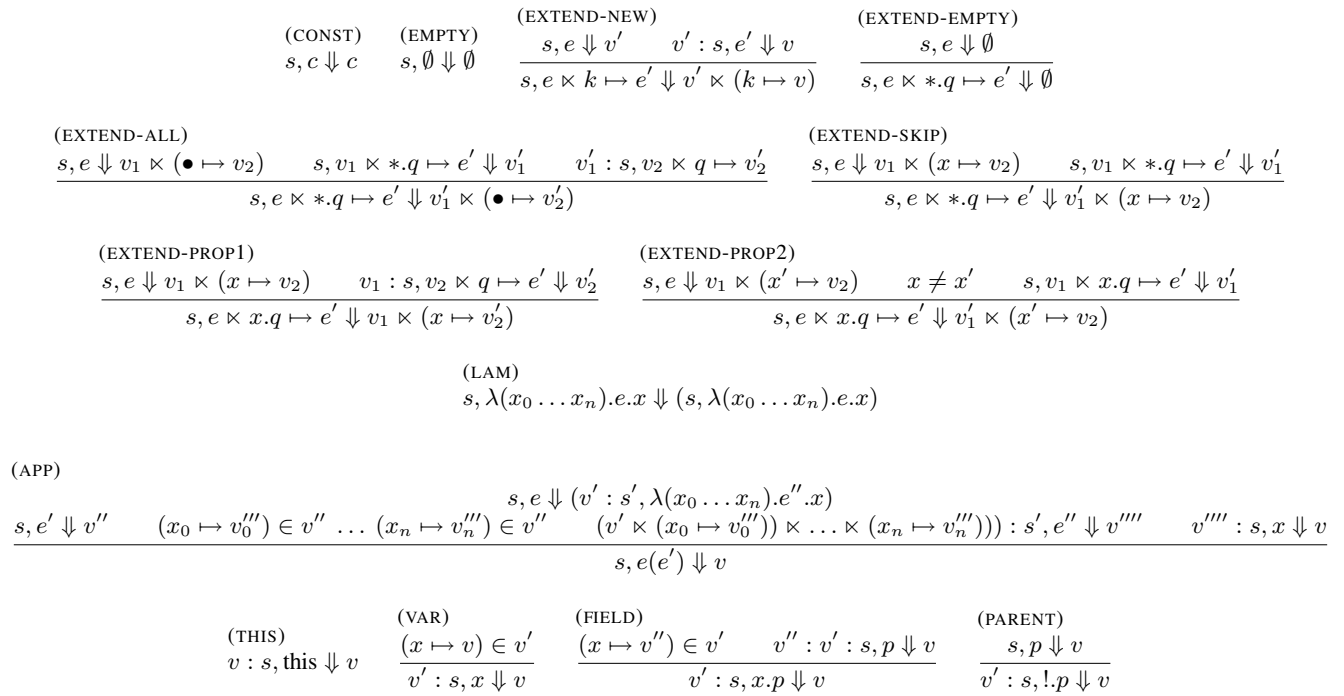

Figure 7. Operational semantics for strict MashMaker evaluation

evaluation makes programs run slowly, the complex evaluation behavior makes performance hard to predict, and programmers often have to battle with space leaks due to long chains of lazy thunks.

MashMaker, however, is not a conventional programming language. We believe that the unusual application domain that MashMaker works in makes lazy evaluation highly appropriate. In particular:

- In the case of web mashups, the bookkeeping cost of remembering how to evaluate something is tiny compared to the massive cost of fetching and scraping a web site, thus it is only necessary for a very small number of expressions to be unneeded for the bookkeeping cost to be more than paid back.
- Even if fetching a web site was cheap, it is important for us to minimize the number of queries we make to a remote server, to avoid overwhelming a server (Section 3.7).
- Typical mashup programs work with relatively small amounts of data that are not directly presented to the user, and so space leaks are far less of a problem.
- Many web sites are already essentially lazy. For example when one makes a search using Google, it does not return all results in one page, but instead produces results lazily as one presses the "next" buttons.

MashMaker's lazy evaluation works largely as one would expect. The value of a node is only evaluated when it is either demanded by the evaluation of another node, or the user attempts to view it through the graphical interface.

3.5 Lambda Expressions in the User Interface

The MashMaker user interface has somewhat unusual treatment of lambda expressions. While the syntax in the underlying core language is fairly conventional, with lambda expressions explicitly marked as such and arguments explicitly listed, the user interface attempts to hide this from users as much as possible.

Although MashMaker is dynamically typed, the Widget metadata for a function includes a flag for each argument saying whether it is a closure. If an argument is a closure then the forms UI interprets text entered for that argument a little differently than for non-closure arguments. Any text entered is assumed to be the body of a lambda expression, and any variables in the expression that are not bound at the current location are assumed to be lambda arguments. Advanced users can tell that this alternative text handling is in use by noticing a λ icon next to the argument text box.

3.6 Live Data

All data in MashMaker is *live*, meaning that it may change over time and will react to changes in other parts of the data tree. If a tree is the result of a web query, then this tree will update over time, as the source web site changes [9].

The MashMaker language is designed to handle changing data well. In particular, since MashMaker overlays extensions over generated data (Section 3.2), rather than modifying it in-place, these extensions will be automatically applied to new versions of the underlying data. Also, since the MashMaker extension construct automatically adds new properties to all children of a node, these properties will also apply to any new children that are added to the tree.

3.7 Throttling

One important practical issue that has to be dealt with whenever one creates a mashup is the need to avoid placing too much load on the web sites supplying data. If one has an agreement with the web site provider then it is likely to specify a maximum load, and if one

[9] Either by polling the web site at a fixed frequency, or waiting until the user asks for a refresh.

does not have an explicit agreement then placing too much load on a server could cause the owners to block the mashup system's IP address.

As a result of this, it is necessary for MashMaker to throttle the rate at which requests can be made to external web sites. Indeed this rate is one of the primary issues that determines the performance of a Mashup, since if a mashup needs to make too many requests, then it will have to slow itself down in order to avoid sending requests too rapidly. This performance restriction has motivated MashMaker not only to use lazy evaluation (Section 3.4), but also to use a number of other tricks (not discussed in this paper) to minimize the number of requests that need to be made to external web servers.

3.8 When Websites turn Bad

One limitation of MashMaker, as with most other tools that scrape information from web sites, is that mashups can break if the underlying websites change. If a website changes the structure of the data it produces, or changes its HTML such that the current scraper no longer understands it, then mashups that depend on this data will no longer function correctly. In the long term, we hope this this problem will become less severe as websites increasingly publish semantic information in well defined data formats.

More generally, MashMaker is not intended to be used for "mission critical" applications where data integrity is essential. Instead, its focus is on applications where it is more important to be able to produce interesting data than to be certain that the data is correct.

4. Evaluating Usability

Following Peyton Jones et al. (2003), we evaluate the usability of MashMaker using the *Cognitive Dimensions of Notations* (CDs) framework (Blackwell et al. 2001). CDs provide a vocabulary that enumerates concepts important to users who are engaged in programming tasks. While evaluation against cognitive dimensions is subjective, and is not a substitute for thorough user testing, these concepts have been shown over time to be important to human problem solving and it is important to consider each when designing a usable interface. We list the cognitive dimensions in Figure 8 and evaluate MashMaker against these dimensions below:

- **Abstraction Gradient:** MashMaker can be used at a number of different levels of abstraction, allowing use by users ranging from complete beginner to experienced programmer (Section 2.5).
- **Consistency:** New widgets are created using the same mechanism as creating simple expressions. All work in MashMaker is done using the same simple mechanism of applying widgets and setting form parameters.
- **Error Proneness:** Unlike normal spreadsheets, MashMaker automatically ensures that when a user adds a property to a set of children the defining expression is identical for all children.
- **Hidden Dependencies and Role Expressiveness:** When a node is selected, all dependent or source nodes are automatically highlighted to make it clear that there is a dependency.
- **Premature Commitment:** Users do not have to decide in advance what they are looking for, but instead can wander aimlessly, looking for something useful. If they find something they like, they can bookmark it as a new function widget, but they need not decide in advance that this is what they are going to do.
- **Progressive Evaluation:** There is no requirement that a program be in any sense "complete" in order for the user to look at its result. Similarly, like a spreadsheet, if some evaluations fail then this does not affect the behavior of non-dependent parts of the program.
- **Viscosity:** MashMaker's support for user-defined functions, automatic synchronization of property definitions across multiple children, and its general preference for linking of data rather than copying, make it easy to make widespread changes.
- **Visibility and Juxtaposability:** Unlike conventional programming languages, MashMaker juxtaposes program and data together, so the programmer can easily see the effects of evaluating their expressions. While MashMaker does not allow one to view multiple forms or multiple view panels in the same window, MashMaker does allow one to view the same data store with multiple browser windows, allowing one to put arbitrary information side-by-side.

Abstraction gradient	What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
Consistency	When some of the language has been learnt, how much of the rest can be inferred?
Error-proneness	Does the design of the notation induce 'careless mistakes'?
Hidden dependencies	Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
Premature commitment	Do programmers have to make decisions before they have the information they need?
Progressive evaluation	Can a partially-complete program be executed to obtain feedback on "how am I doing"?
Role-expressiveness	Can the reader see how each component of a program relates to the whole?
Viscosity	How much effort is required to perform a single change?
Visibility and juxtaposability	Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to compare any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

Figure 8. Cognitive Dimensions (taken from Peyton Jones et al. 2003)

Based on this analysis, and also our personal experiences using MashMaker, we believe that our design is fundamentally sound. However, in order to demonstrate this objectively, we need to perform a proper user study, and indeed we intend to do this in the near future.

5. Related Work

In this section, we explain how MashMaker relates to previous work on Mashup creation and end-user programming in general.

In prior work, we described the basic MashMaker tool from a database angle (Ennals and Garofalakis 2007).

5.1 Mashup Creation Tools

Mashups are an increasingly hot topic, and thus there have been many efforts to simplify their creation. Relative to MashMaker, these previous tools generally fall into two groups: those which are easy to use, but can only create a limited family of mashups; and those which are relatively difficult to use, but can create a wide range of mashups.

Google MyMaps [10] and MapCruncher [11] make it easy for end users to create mashups involving maps. Swivel.com makes it very easy for end users to create graph mashups from multiple data tables. However, while each of these tools is easy to use, and excellent at producing mashups of a specific type, none of them is as general purpose as MashMaker.

Yahoo Pipes [12] is a powerful tool that allows users to process data from RSS feeds. While, at the time of writing, the small set of operations available in Pipes makes it less flexible than MashMaker, it seems likely that the tool will be extended to give it equivalent expressive power. The key difference between Pipes and MashMaker is that, unlike MashMaker, pipes presents the program as an explicit graphical dataflow graph, rather than mixing it with the data being browsed.

Marmite (Wong and Hong 2006) takes a pipeline-based approach, similar to Apple's Automator [13]. The data from a web site is routed through a sequence of pipeline stages, each of which is configurable, and can produce data of a different type. Like Yahoo Pipes, the program is separated from the data and presented as a graph, rather than being embedded in the data like a spreadsheet. Like MashMaker, Marmite will automatically suggest operations to apply to data. Unlike MashMaker, these suggestions are based on the type of the data (similar to Jungloids (Mandelin et al. 2005)), rather than based on the behavior of previous users. Anthracite [14] is similar to Marmite, but requires that the user be familiar with complex concepts such as HTML and regular expressions.

Creo (Faaborg and Lieberman 2006) augments web pages with additional links that can obtain additional information about items on a web page. Like MashMaker, Creo will automatically make suggestions and can learn by example from things that users do with their data. Unlike MashMaker, Creo is limited to adding additional hyperlinks to web pages and cannot perform bulk data processing tasks.

Google Mashup Editor, Plagger.org, Ning.com, Javascript Dataflow Architecture (Lim and Lucas 2006), and Web Mashup Scripting Language (Sabbouh et al. 2007) are powerful tools for creating mashups, but they require that the user write code.

ClearSpring.com, Widsets.com, WidgetBox.com, and Apple's Dashboard [15] allow users to write small graphical web widgets and then lay them out together on a screen. DataMashups.com additionally allows users to connect these widgets together (e.g. the output of this widget is the input to that widget), but complex tasks require considerable programmer skill.

HunterGatherer (Schraefel et al. 2002) and Internet Scrapbook (Sugiura and Koseki 1998) allow users to extract parts of multiple web sites and composite them together, but are not able to perform complex processing on these sites and extract collections of data.

Like MashMaker, C3W (Fujima et al. 2004) uses a spreadsheet metaphor. C3W uses a standard flat two-dimensional spreadsheet to connect web sites together. If a user defines values for a web site's input cells, then it will produce results in its output cells, clipped from the web page. Unlike MashMaker, C3W uses a flat two dimensional grid, rather than a tree. This prevents one writing mashups that produce nested data, such as producing a list of restaurants for each of several apartments.

TreeSheet (Leonard 2004) represents data as an XML tree, but unlike MashMaker, programming is done using imperative scripts, rather than functional overlays.

Within the database community, SEMEX (Cai et al. 2005) and DataSpaces (Franklin et al. 2005) have looked at data-integration – how to get transform various data sources into a suitable structure so that they can be combined with queries.

5.2 End-User Programming Tools

MashMaker also bears some similarity to a number of end-user programming tools that have not been used for creating web mashups:

Programmable Structured Documents (PSDs) (Takeichi et al. 2003; Hu et al. 2004; Liu et al. 2005) allow one to extend a standard XML document by embedding elements in the tree that are computed from other elements. An expression defining an XML node can refer to other nodes using XPath expressions and then process the nodes using arbitrary Haskell functions. Like MashMaker, PSDs are based on a functional language (in this case Haskell) and are evaluated lazily. Unlike MashMaker PSDs deal with static XML documents, rather than live data. Indeed, since PSDs include expressions directly in a document, rather that overlaying changes on top of generated data in a way that can be automatically re-applied, they could not be used to add additional properties to live data without changes to the model (Section 3.6).

Subtext (Edwards 2005) is a programming tool that allows one to look at a program together with the results of its evaluation. In Subtext, every node in the data tree corresponds to the execution of a single program line with specific data and is annotated with the value produced. Function calls are expanded as subtrees and function definitions contain example arguments that the programmer can adjust to interactively see how their program will behave. Like MashMaker, subtext allows programmers to easily see how their program will behave when applied to particular arguments. Unlike MashMaker, the Subtext interface is program-centric, rather than data-centric — meaning that data is layered on top of a program, rather than overlaying a program on top of data.

MashMaker's function creation system is influenced by the work of Peyton Jones et al. (2003) in extending Microsoft Excel to support user-defined functions. Like MashMaker, they allow one to define a new function by selecting a result cell and then using a graphical interface to specify which other cells are arguments.

MashMaker's suggestion system is influenced by Jungloids (Mandelin et al. 2005) and Google Suggest [16]. Like Jungloids, MashMaker suggests operations that are appropriate to the data one has at hand. Like Google Suggest, MashMaker learns from the behavior of other users.

More generally, MashMaker draws on past work on Programming by Example (Cypher et al. 1993; Lieberman 2001), and previous work on programming approaches for beginners (Kelleher and Pausch 2005).

6. Conclusions

We have presented MashMaker, a tool that allows end-users to easily create web mashups. While MashMaker is, at its core, a functional language, it contains a number of deviations from the standard functional paradigm. By taking ideas from such popular tools as file systems, spreadsheets, and web browsers, we have produced a tool that we believe is well suited to the task of mashup creation.

This project is interesting as a programming language research project both because it approaches an application domain for which programming languages have not historically been seen as the solution, and also because, in the process of fitting our language to this

[10] http://maps.google.com

[11] http://research.microsoft.com/mapcruncher

[12] http://pipes.yahoo.com

[13] http://www.apple.com/

[14] http://www.metafly.com/products/anthracite

[15] http://www.apple.com/

[16] http://labs.google.com/suggest/

domain, we have produced a language that has many differences from previous functional languages.

We have implemented MashMaker as an AJAX web application, currently made available within our organization as part of a closed beta program. We plan to make it publically available in the near future. In the long term, the success of MashMaker will be judged based on the extent to which real users adopt it, and the scale of the benefit they are able to obtain from it.

For more information on MashMaker, and access to the public beta when it opens, please go to the following url:

`http://berkeley.intel-research.net/rennals/mashmaker/mashmaker.html`

Acknowledgments

This work has benefited from the input of many people. Particular thanks should go to Minos Garofalakis, Eric Paulos, and Ian Smith.

References

Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong, and Richard M. Young. Cognitive dimensions of notations: Design tools for cognitive technology. In *CT '01: Proceedings of the 4th International Conference on Cognitive Technology*, pages 325–341, London, UK, 2001. Springer-Verlag. ISBN 3-540-42406-7.

Yuhan Cai, Xin Luna Dong, Alon Halevy, Jing Michelle Liu, and Jayant Madhavan. Personal information management with SEMEX. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 921–923, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-060-4. doi: http://doi.acm.org/10.1145/1066157.1066289.

Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-03213-9.

Jonathan Edwards. Subtext: uncovering the simplicity of programming. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 505–518, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094851.

Robert Ennals and Minos Garofalakis. Mashmaker : Mashups for the masses (demo paper). In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD'2007)*, 2007.

Alexander Faaborg and Henry Lieberman. A goal-oriented web browser. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 751–760, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-372-7. doi: http://doi.acm.org/10.1145/1124772.1124883.

Michael Franklin, Alan Halevy, and David Maier. From databases to dataspaces: A new abstraction for information management. In *SIGMOD Record*, 2005.

Jun Fujima, Aran Lunzer, Kasper Hornbæk, and Yuzuru Tanaka. Clip, connect, clone: combining application elements to build custom interfaces for information access. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 175–184, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-957-8. doi: http://doi.acm.org/10.1145/1029632.1029664.

Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 178–189, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-835-0. doi: http://doi.acm.org/10.1145/1014007.1014025.

Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, 2005. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1089733.1089734.

Thomas Leonard. *Tree-Sheets and Structured Documents*. PhD thesis, University of Southampton, 2004.

Henry Lieberman. *Your wish is my command: programming by example*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1-55860-688-2.

Seung Chan Slim Lim and Peter Lucas. Jda: a step towards large-scale reuse on the web. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 586–601, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-491-X. doi: http://doi.acm.org/10.1145/1176617.1176631.

Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. An environment for maintaining computation dependency in XML documents. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 42–51, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-240-2. doi: http://doi.acm.org/10.1145/1096601.1096616.

David Mandelin, Lin Xu, Rastislav Bodik, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 48–61, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-056-6. doi: http://doi.acm.org/10.1145/1065010.1065018.

Simon Peyton Jones. Wearing the hair shirt: a retrospective on Haskell (invited talk). In *ACM SIGPLAN Conferenge on Principles of Programming Languages (POPL'03)*, 2003a.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, may 2003b.

Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in Excel. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 165–176, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-756-7. doi: http://doi.acm.org/10.1145/944705.944721.

Marwan Sabbouh, Jeff Higginson, Danny Gagne, and Salim Semy. Web mashup scripting language (poster). In *16th International World Wide Web Conference*, 2007.

M. C. Schraefel, Daniel Wigdor, Yuxiang Zhu, and David Modjeska. Hunter gatherer: within-web-page collection making. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 826–827, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-454-1. doi: http://doi.acm.org/10.1145/506443.506617.

Michael Schrenk. *Webbots, Spiders, and Screen Scrapers*. No Starch Press, 2007.

Atsushi Sugiura and Yoshiyuki Koseki. Internet scrapbook: automating web browsing tasks by demonstration. In *UIST '98: Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 9–18, New York, NY, USA, 1998. ACM Press. ISBN 1-58113-034-1. doi: http://doi.acm.org/10.1145/288392.288395.

Masato Takeichi, Zhenjiang Hu, Kazuhiko Kakehi, Yashushi Hayashi, Shin-Cheng Mu, and Keisuke Nakano. TreeCalc: towards programmable structured documents. In *Japan Society for Software Science and Technology*, 2003.

Jeffrey Wong and Jason Hong. Marmite: end-user programming for the web. In *CHI '06: CHI '06 extended abstracts on Human factors in computing systems*, pages 1541–1546, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-298-4. doi: http://doi.acm.org/10.1145/1125451.1125733.