Integrated Computer-Aided Engineering 16 (2009) 181–191
DOI 10.3233/ICA-2009-0312
IOS Press

Computer aided software design via inference and constraint propagation

Gordon S. Novak Jr.*
Department of Computer Sciences, University of Texas at Austin, Austin, USA

Abstract. Engineered systems, including computer programs, are mainly composed of versions of known components. We describe systems that produce computer software by composition and specialization of generic software components. The work of the human designer is reduced because components can infer or default many specifications, propagating them to connected components. Specialized programs can be constructed rapidly and easily using a graphical programming interface. We describe systems that can capture and understand data from the web, allowing a user to write programs easily to analyze the data. Web pages that perform calculations or data lookup can be treated as remote procedure calls, allowing calculations, proprietary data and real-time data to be used. Examples are presented to illustrate construction of programs for analysis of web data. These capabilities provide connectivity between web data, codified knowledge, and generic programs, providing a pathway to fulfillment of the promise of integration of the vast amount of information on the web.

1. Introduction

Engineered systems, including computer programs, are mainly composed of versions of known components. Components that are connected mutually constrain each other. Engineering design involves selection and parameterization of components such that the goals of the system are satisfied, constraints of components are met, and cost is minimized. Detailed design is difficult due to a large number of components, parameters, and constraints.

We describe systems that produce computer software by composition and specialization of generic software components. Computer software is unique because the manufacturing cost of software is near zero. We have compiler technology that can produce specialized versions of generic software components. Generics are parameterized using *views* that describe how the concrete data of an application relate to the abstract types in terms of which the generics are written. A single generic can be specialized to produce program code for a variety of data types in a variety of programming languages.

Since production of software code from parameterized generics is mechanized, the process of programming can be raised a level to selection of components and specification of parameters, rather than writing of code. A potential difficulty is that if a generic is to be flexible enough to meet all the needs of designers, it must have many parameters, and specification of so many parameters will remain a burden. Our approach reduces the input required from the human designer. Components are able to infer or default many of their parameters; parameters that become defined in one component are propagated to connected components, often leading to further inference and propagation. We have developed a graphical programming system that allows rapid and easy construction of specialized programs from generic components.

Our work is based on reuse and specialization of generic procedures through views. A *view* is a set of mappings that translate between a concrete type and the features expected by an abstract type. A generic procedure defined in terms of abstract types can then be specialized through views, so that the specialized procedure operates on data of the concrete types. This

*Address for correspondence: Prof. Gordon S. Novak Jr., Department of Computer Sciences, TAY 2.124, C0500, University of Texas at Austin, Austin, TX 78712, USA. Tel.: +1 512 471 9569; Fax: +1 512 471 8885; E-mail: novak@cs.utexas.edu.

ISSN 1069-2509/09/$17.00

ability allows knowledge, in the form of generic procedures, to operate on data in arbitrary formats.

The web provides large amounts of data and specialized procedures, but it is not easy to integrate data and procedures from separate sources to perform a desired analysis. We describe several systems that make it easy to acquire data from the web, understand it, connect it with a knowledge base, and write programs to perform a desired analysis using the data.

XML and HTML are widely used for structuring data in tree or table form, and a great deal of formatted data is available on the web. However, the tags and table headers of these formats do not adequately describe the data. We have written a *data grokker* that both parses data into a program-usable form and also develops data type descriptions that link the data to knowledge associated with common data types; this makes the knowledge usable in generating programs that operate on the data.

There are numerous web pages that provide real-time data (e.g. stock prices), proprietary database access, or calculations (e.g. converting latitude/longitude coordinates to UTM). We have developed an interface that makes it easy to treat such a web page as a remote procedure call, passing data to the web page as input and then parsing the result out of the web page that is returned. When such web pages are linked to known data types in the knowledge base, the web pages can be used as part of analysis programs.

We will describe each of these systems and illustrate the process with working examples. We begin with an example of graphical program development to illustrate how programs can be constructed quickly and easily. Next, we describe the process by which the minimal specifications provided by the user via the graphical interface are expanded into a complete and consistent program specification and then converted into executable code. All of the programs described in this paper have been implemented as shown.

The approach of parameter inference and propagation that we describe seems applicable to other kinds of engineering design as well. Our system uses a set of procedures to perform several kinds of inferences. It would also be possible to include small expert systems to make choices of components and parameters. For example, in the software domain, an association list is preferred for small lookup tables (simple code), while an AVL-tree might be preferred for large data sets (better performance). Given information about expected data set sizes, a small expert system could make the appropriate choice automatically. Engineering design tradeoffs, including negative constraints, could be incorporated as needed for different engineering domains.

2. Graphical programming

We begin with an example to illustrate graphical programming; later sections describe the technical details that are required to implement the simplified interface presented here.

Today there are billions of web pages, and powerful search engines make it relatively easy to find data relevant to a user's needs; however, the data may not be in the units or form needed. In order to make use of data obtained from the web, users need to write custom programs to analyze and combine such data. Since traditional programming requires a precise fit between data and program, there is almost no chance that a prewritten program will work with web data. We have implemented a graphical user interface that allows easy and rapid interactive construction of a custom program by specialization of reusable components.

The interface is mixed-initiative; sometimes it prompts the user for inputs, and sometimes the user clicks on a diagram button to select a filler for an interface. Some parameters automatically receive default values; the user can override defaults by clicking on a box and selecting the parameter and its value from menus. An example of the program diagram for an analysis of web data (a list of data from a music CD catalog) is shown in Fig. 1.

In this example, the user wishes to analyze a catalog of music CD's, producing data about CD's from each different country. The user clicks a command button to create a program, then selects from a menu a program in the class `iterate-accumulate`; such a program will iterate over a collection of items and accumulate something about each item. The user selects the starting type as `catalog`, which was produced by the system when it read the data from the web. A `catalog` contains a set of `cds`, so this is selected as the collection over which to iterate; the system can infer that the type of an item is `cd`. Next, the user clicks on the `Acc` button of the `iter-acc` box, to specify the accumulation to be performed; the system responds with a menu of possible accumulators. To analyze CD's separately by country, the user selects `find-update` as the accumulator. `find-update` looks up an item in a database, then performs some update to the information associated with the item. `find-update` asks

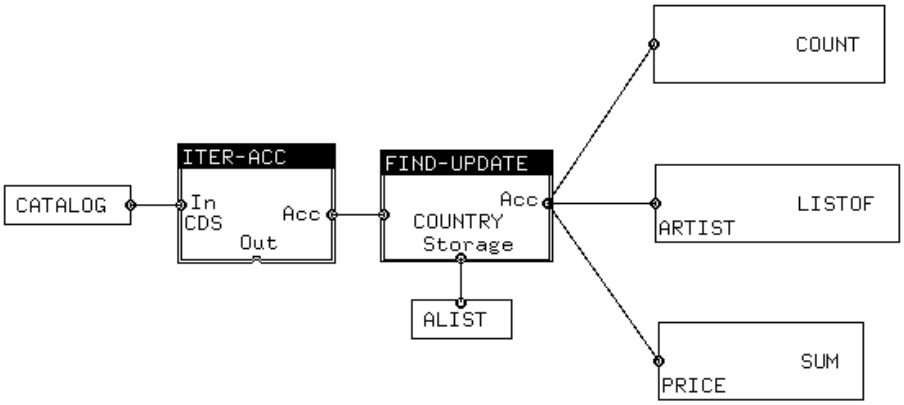

Fig. 1. Program specification diagram.

the user for the `key` to be used in matching the item, presenting a menu of `title`, `artist`, `country`, `company`, and `year`; these choices are derived from the data type of `cd`, which was inferred from the web data. The user selects `country` as the `key` value. `find-update` also has an `Acc` button that accepts accumulators. The user selects counting the number of CD's for each country, making a list of the names of artists, and summing the price of the CD's; each of these is shown as a separate box in the diagram. Finally, the user clicks on the `Storage` button of the `find-update` box and selects `alist` (association list, a simple kind of database) as the storage.

Although this process is lengthy when described in words, it takes the user less than a minute to understand the data from the web, enter the diagram and produce the program, and run the program to get the answer; it is fast because it does not require any textual programming. The system produces a program, named `gpfn12`, that can be applied to the data from the web to produce the desired analysis:

```
>(gpfn12 *xmldata*)

(("Norway" 1 ("Jorn Hoel") 7.90)
 ("EU" 5
   ("Simply Red" "Kim Larsen"
    "Savage Rose" "Andrea Bocelli"
    "Eros Ramazzotti")
   46.60)
 ... )
```

The result is a list of lists; each sublist contains the name of the country, the number of CD's from that country, a list of the artist names, and the total price of the CD's.

An important feature of the user interface is that it uses context to help the user make most choices by menu selection. The system knows the types of data at each point and uses these types to derive menus of sensible options from which the user can choose. These menus include knowledge associated with the types by inheritance, e.g. if the data contains a zip code, properties associated with the zip code can be included in the program.

The style of program design used here involves selection and connection of boxes, followed by further specification of their properties. This fits with the notion of *progressive deepening* that Herbert A. Simon and colleagues observed in human designers [10].

Graphical programming allows a user to create a custom analysis of web data based on the user's understanding of the data, but does not require the user to know the details of textual programming.

3. Program specialization through views

Programming languages generally associate fixed or tightly parameterized types with the arguments of a procedure; unfortunately, the rigidity of these types prevents easy reuse of the procedure, causing a lack of connectivity between data, programs, and knowledge. Although some languages allow the use of a *functor*, such as a `Comparator` in Java [26], a human programmer must understand the data and the interface and construct the functor, and there is a runtime penalty of interpretive overhead. This makes reuse difficult, slow, and costly [13]. A better mechanism is needed to connect the types of the new data with the types expected by reusable procedures.

A *view* [16,17] is a data type that is used to connect a concrete (application) type to generic procedures. The view has the concrete type as its stored data; it defines methods that make the concrete type look like the abstract type assumed by the generic procedures. Compiling a generic with respect to the view type produces a specialized version of the generic that operates directly on the application data. Views are somewhat similar to functors in Java, but partial evaluation during compilation causes the translations specified by the views to become hard-coded and efficient, as compared to creation of interpreted functor objects in Java. Views allow reuse of generic procedures with arbitrary data that may be obtained from the web. Since a view is purely *gedanken* and does not add anything to the data, the same data can have multiple views. The specialized versions of generic procedures are mechanically translated into the desired programming language; currently, Lisp, C, C++, Java, and Pascal are available.

Although views are somewhat complex, we have developed automated systems to help construct views for data structures [17] and mathematical representations [16]. Views are essential for making programs reusable with web data in arbitrary formats. Automation of the view creation process makes the use of views transparent to the user.

4. System architecture

The architecture of our system is shown in Fig. 2. The user interacts with the graphical programming system to produce a declarative representation of the program design as a network of *component instances*, each of which represents a kind of component described by a *component specification*. There can be multiple instances of the same kind of component in a network. The component instances can be mechanically translated into a set of views. Generic components are converted, by compilation through views and partial evaluation, into a program in the desired programming language. From the perspective of this paper, the network of component instances is the finished specification of the program, since everything else is produced mechanically by compilation processes.

Components are connected via interfaces, each of which contains multiple parameters. In some cases, an interface may be filled by multiple components, as shown in Fig. 1 where several accumulators are connected to the `find-update` box. The network of component instances is reflected in the boxes and connections shown by the graphical user interface.

A component specification describes the types, functions, and constants needed by an instance, how they are related, and how some of them may be inferred.

A component instance is usually created when the user selects it from a menu. It will receive parameter values by user input, or by propagation from other components to which it is connected. When a parameter value becomes defined, the component specification is examined to see whether other parameters can be derived by inference; if so, these values may be propagated to other components via connections.

5. Generic programs

Although languages such as C++, Java, and Haskell allow some degree of type parameterization of procedures, and therefore some degree of genericity, we find that they do not allow enough for easily reusable generics. While it is possible, with sufficient effort, to write any program in any language, effective reuse requires that it be easier to obtain a program by reuse than to write it from scratch [11]. This section describes features needed for effective reusability.

Any *assumption* made by a generic becomes a *requirement* for its reuse; if a generic makes multiple assumptions, there will have to be a combinatoric number of generics with different sets of assumption choices [4]. Thus, it is better for a generic to make few assumptions, but this leads to a larger number of parameters. For example, a sorting program that uses the $<$ operator can only sort objects that can be compared by $<$ and can only sort them in one way; but a sorting program that allows the comparison function to be specified has an additional parameter.

Our approach is to try to minimize the assumptions made by generics, so that a single generic will be widely reusable. The interface between a generic and its data will always be mediated by views. This leads to two potential problems: inefficiency and a large specification. The inefficiency that would be caused by runtime interpretation is eliminated by partial evaluation that expands interface functions in-line during compilation and optimizes the resulting code. The burden of a large specification is removed from the user by inference and propagation of parameters.

A further need is an ability for generics to create data structures that incorporate data from other generics. Other languages provide an ability to specify a

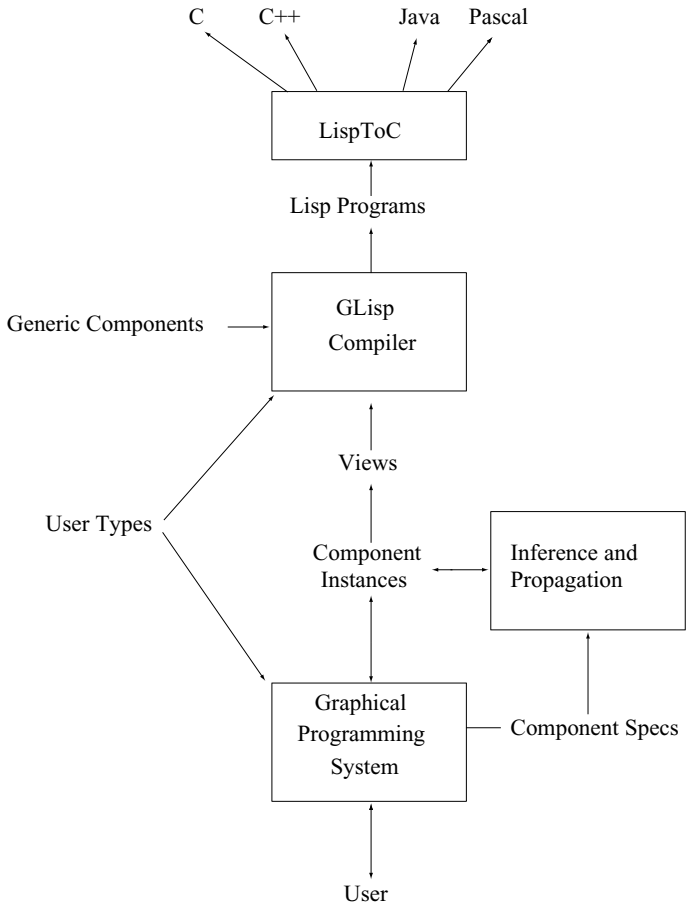

Fig. 2. System Architecture.

type parameter, e.g. to allow a container type that contains application data. We find that applications require an ability to construct data structures that combine data from multiple generics and to propagate data structures between generics. While other languages generally require that type parameters be specified manually by humans, our system derives the necessary types by inference and propagates them between components.

The overall structure of a program is based on a program framework, such as `iterate-accumulate` or `heuristic-search`. The framework usually has interfaces for attaching related components such as accumulators or database components. A set of commonly useful accumulators has been implemented, including `sum`, `product`, `count`, `average`, `max`, `min`, `argmax`, `argmin`, `statistics`, and `histogram`. Data accumulators such as `listof` and `queue` allow individual values to be accumulated. The database components include `alist` (association list), `avl-tree`, `array`, and `histo`. Accumulators can

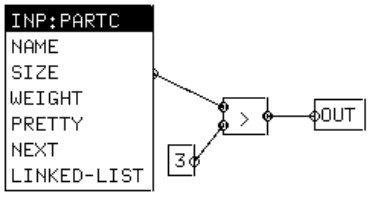

Fig. 3. Graphical Predicate Specification: `size > 3`.

be chained, so that it is possible to create an array of queues of values.

Generic programs often have places where mappings can be inserted, e.g. from an item to the value to be accumulated. In addition to selecting a field or computed value of the item, it is also possible to insert bits of arbitrary code in several ways: a separately written function can be named, a fragment of source code can be included, or a function can be created graphically as shown in Fig. 3.

In the following section, we describe the inference

and propagation processes that perform much of the work of specification.

6. Inference and propagation

The ease of use of the graphical programming interface is based on the system's ability to infer and propagate data types and other parameters as the program is constructed. A program such as the example described in Fig. 1 is composed of reusable generic components such as `iterate-accumulate`, `find-update` and `alist`. The way in which these components are combined, and the data types on which they operate, are unknown until the program structure is created by the user. It is not easy to combine generic components in a variety of ways and make them work together on arbitrary data. We believe that understanding and managing the interactions between components is one of the main tasks performed by human programmers. In analyzing a hand-written program [18], we found that although the program was composed of identifiable and well-understood abstract components, the code of the program scrambled together fragments of code corresponding to the different components. The software components interact strongly and parameterize each other. The fact that information from a simple design decision becomes spread throughout program code was noted by Balzer [2].

In order for reusable components to work together with arbitrary data, the components must have many parameters. For example, the `average` component, which averages data values over a set of items, has an interface function that maps from the item type to the value to be averaged and a predicate to test whether an item should be included in computing the average. In addition, the averaging component has some intermediate data used to sum data for the average and to count the number of instances; this data, in turn, must be stored by the data structure of the database component selected as a parameter to `find-update`. The type of the sum data will be the same as the type of data being averaged, which may include units of measurement as well as numeric type.

Our approach is to write very general components with many parameters. Most of these parameters can be defaulted, and optimization by means of partial evaluation will remove unused code. It is easier to customize general code by defaulting and eliminating unused code than to synthesize and integrate new code.

There is a serious problem in constructing programs from components: the components have many parameters, and these parameters must be mutually consistent so that the components can inter-operate. If the user had to specify all the parameters, it would be as difficult as ordinary programming. The key insights are that many of the parameters can be inferred, and that the structure of the program can be exploited to propagate parameters between components and simplify the user interface.

Components infer parameters based both on other parameter values and on information in their component specifications. For example, `iterate-accumulate` iterates over a collection of items; once the type of the input is known, *type filtering* is used to restrict the possibilities for the collection to parts of the input whose type is some kind of a collection, such as an array or linked list. If there is only one possibility, it can be chosen automatically; otherwise, the user is asked to choose from a menu. Because of type filtering, only sensible choices are presented in the menu.

Once the system knows the collection type, it can infer the item type. Parameter values that become known in one component are propagated to other components that are connected to it, leading to further inference and propagation. Thus, the item type inferred by `iterate-accumulate` is sent to `find-update` as the type of its input. Some parameters are defaulted, e.g. most accumulator components have a `test` (to decide whether to include an item in the accumulation) that defaults to `true`. A default can be over-ridden by specifying the parameter explicitly. Known parameter values are used to help the user specify other parameters, e.g. a known source type, together with requirements on the result (e.g., only types that support both + and / can be averaged), can be used to construct a filtered menu of possible values for the parameter. Selection of a value to be averaged implies the type of the accumulator used in the average.

An important kind of inference is construction of data structures within the program. In our example, `find-update` is connected to an arbitrary set of accumulators, each of which has its own kind of data, and to some kind of database, which imposes its own structure on its data. Propagation allows these separate sources of data type information to be combined. Each accumulator sends the types of the data it needs to `find-update`. `find-update` combines these into a record; it passes the type of the ensemble of accumulator data to the database program, `alist`, as well as passing back to the accumulator an access func-

tion to tell it how to access its data within the record. `alist` adds to the data provided by `find-update` the data that `alist` needs, creating a composite data type that is passed back to `find-update` as the type of the database as a whole. `find-update` passes the database type to `iterate-accumulate`, which must hold the data for the program. Although there is much activity of this sort behind the scenes, the user does not need to be concerned with it. Automatic construction of data structures is only triggered when all the inputs are available, so it will always work; as new components are added, it may be triggered again and re-derive the data structures.

Most of the inferences performed by components are rather simple. However, it would be possible to have a small expert system to perform some kinds of inference and component selection based on facts about the interfaces, e.g. to select an appropriate implementation for the database based on the expected size of the data set.

Propagation occurs when a new value is determined for a parameter that is connected to another component; the new value will be sent to the other component, which will cause its inference procedures to be invoked. Inference is performed when a new parameter value becomes defined; each item in the component specification that depends on the new value is examined, possibly inferring another new value. This process propagates new information and changes through the network; it is analogous to the constraint propagation in Waltz filtering [25] and MOLGEN [23] and performs the progressive deepening noted by Simon [10].

A key requirement for integration of web data with reusable programs is usability: the user should not have to do detailed programming in order to create a custom analysis of web data. Because our system understands web data and creates and propagates data types, the user sees only menu selections that make sense and can easily and rapidly create the desired programs.

7. Data understanding

Large amounts of data are available on the web in tree-structured XML or in HTML tables. However, before this data can be used, it must be parsed, understood, and reduced to a form in which the meaning of the data is usable. We have developed software that can parse and create data types for well-structured XML data obtained from the web, making the data usable with the program generation tools described previously. We believe that similar techniques could be extended to HTML tables.

The meaning of data is most often implicit. Some clues to the meaning can be found in XML tags and in the data values themselves, but important facts about data (such as type or units of measurement) are typically left unspecified. In order for web data to be used in an analysis, these facts must be determined, and our system attempts to infer some of them. Certain data values are recognizable as naming members of finite sets, e.g. month names or state abbreviations; finding such data values implies the type of the data. Our system recognizes special data formats such as phone numbers, URL's, and dates (in several commonly used formats). Even purely numeric data formats may be recognizable, such as currency values or zip codes, especially when the data form is confirmed by an appropriate tag name (e.g., "price" suggests a currency value). Data that cannot be parsed as a recognized data type are left as is and described as `string`; there is no attempt to parse natural language.

As data values are parsed from a web page, the system converts them into forms that are usable for analysis. Numbers are converted into machine binary so that they can be used in calculations. State names or abbreviations are converted so that they link to knowledge stored for each state (e.g. population, land area).

Dates occur in a number of different formats; each is converted into a common structured form so that operations on dates (e.g. comparisons, calculations of time between dates) can be done, even on dates with different syntaxes on the web.

As data are being parsed from an XML file, a structured description of the data is built. The data itself is translated into Lisp data structures that can be used in computations, and the data type description is constructed in the form used by the GLisp compiler [14]. Since XML data is structured, parsing a single XML file often results in multiple data types. Repetitions of similar data are often found; these are converted into a `listof` structure for which iterators exist. Once the data has been parsed into Lisp form, it would be easy to re-format and output the data for use by a program in another programming language. Since the data has been understood and related to factual knowledge and procedures, this re-formating could include conversions such as converting units of measurement [15] or coordinate systems.

A simple function call containing the URL of web data is all that is needed to both parse the data into internal form and create the data structure descriptions, e.g.:

```
(groknroll
 "www.w3schools.com/xml/cd_catalog.
 xml")
```

This call reads and understands XML data from the web representing a catalog of music CD's, as used in our examples. There also is a command button on our programming interface that accepts a URL and then retrieves and understands the data.

The data understanding software described here is sufficient to parse well-structured XML data and allow programs to be generated over the data using the program generation tools described earlier. Even though the computer may not have a full understanding of the data, the XML tags that become field names in the data types are likely to be descriptive enough that the menus shown to a human user during program construction will be meaningful.

While HTML files can be complicated, it is often the case that a web page contains a large table surrounded by many other things. It should fairly easy to extract the data from the table in a manner similar to that used for XML data, while ignoring the rest of the HTML data.

Not all XML data is well-structured; the techniques described here would be less useful on poorly structured data.

8. Web pages as remote procedure calls

Many web pages provide real-time data (e.g., Yahoo Finance will provide price and other data for a stock upon request), access to a proprietary data base, or specialized calculations such as converting latitude/longitude coordinates to UTM. Such data often are only available dynamically via the web: stock prices change with time, and companies may provide a few responses from a proprietary database but not give access to the whole database.

We have developed an interface that makes it easy to treat a web page as a remote procedure call. The `curl` utility program [6] retrieves a web page given a `http` or `ftp` URL; arguments may be passed to the web page as part of the request. Our program constructs an appropriate `curl` command, retrieving as a file the web page that is returned. We have a set of parsing programs that can use a specified grammar to parse the web page and to extract the desired information. Once a web page has been encapsulated as an RPC, it can be used with the other tools described earlier, e.g. to apply it iteratively over a set of data derived from the web.

In order to use a web page as a remote procedure call, one should first use the page manually, noting the arguments passed in the URL and saving the returned web page as a file. Next, a grammar is written to extract the desired information from the file; this grammar is designed to skip over most of the web page down to the desired information, often by looking for keyword patterns. A function is then written to pass the arguments and retrieve the data; such a function is remarkably small and easy to write. For example, the following function retrieves a stock price using Yahoo Finance:

```
(defun stockprice (symbol)
  (webrpc "http://finance.yahoo.com/
           q"
     (list "s" symbol)
     '(car (seq (skipto "Last Trade:")
                (skipto "<b>")
                (num))) ) )
```

This function passes the ticker symbol of a stock to the web page as the "s" argument. The result is returned as a `.html` file; the grammar specifies that the parser should skip to the text ``Last Trade:'', then skip to ``<b>'', then parse a number; that number is returned as the answer. Given this small function, stock prices become as easily accessible as square roots:

```
>(stockprice 'ibm)

113.94
```

Some computer expertise is required to analyze a web page, write a grammar to extract the desired information, and write the function as shown above. Someone with the appropriate expertise can write a function for a web page in a few minutes. Once written, the function can made available as knowledge associated with a data type and can be used by any user.

9. Associating knowledge with data

Web data often involve standard data types such as state names, dates, telephone numbers, zip codes, etc. These known data types, in turn, potentially have a great deal of procedural and factual knowledge associated with them, especially when linked to web pages as described above. The type system of our language allows knowledge to be associated with these types; this can include constant values, values looked up in tables, and procedures. Procedures can include such things as how to compare two dates, as well as retrieval of web data:

```
>(zipcode 94025)

("Menlo Park" CA 94025 37.452
-122.184)
```

In this example, knowledge that an integer represents a zip code allows retrieval of the associated city, state, and location coordinates from a web site. By linking the zip code to this related data, additional knowledge becomes available, e.g. there is a function to compute distance between latitude/longitude positions, and this is used to transparently define distance between zip codes. This linking of knowledge about data types allows very flexible linking of information, e.g. a telephone number can be looked up to find a zip code, making all of the knowledge associated with zip codes also available for phone numbers.

We have written programs [16] to facilitate the creation of *views* that connect structured data to known types that have knowledge associated with them. For example, a `business` might have definitions of properties such as debt/equity ratio, defined in terms of other parameters. By putting the definition of a business as described in web data into correspondence with the known types, this knowledge can be reused. It is possible that web data uses different but equivalent data to that used in the known types (e.g. radius of a circle used in the known type, diameter used on the web); our system allows such differences in terminology to be translated.

10. Future work

Our current system can infer types for some basic data; it would be useful to extend this inference to match web data against an ontology [5] to allow more accurate classification and understanding of data. Web data often describe well-known kinds of objects such as people, addresses, companies, products, etc. If web data could be matched against an ontology that describes these classes, a richer understanding of the data could be obtained. For example, synonyms in XML tags could be recognized, e.g. either “price” or “cost” might be used to describe the price of a product. The units of data are often implicit and might be inferred, e.g. if the height of a person is 180, the units could be inferred to be centimeters rather than feet.

Ontologies to match against could be chosen as those that share the most terms with the data. Of course, it will not always be possible to match terms correctly; however, a partial match could be presented to a human user for editing and selection of additional matching terms.

We plan to expand the range of programs that can be created graphically using our system, including an expanded library as well as geometric and physics analysis, and creation of intermediate data structures that are then used by additional programs. Expert systems to choose data structures and algorithms, based on features such as expected data set size, could simplify the task of specifying programs.

Engineering design in many disciplines is based on idea of connected components, each of which is an instance of a standard class of components. The techniques presented here seem to be applicable for other kinds of engineering in addition to software.

11. Related work

KIDS [20] transforms problem statements in first-order logic into efficient programs for combinatorial problems. The user selects transformations and supplies a formal domain theory for the application; [21] describes design theories that represent abstract design concepts such as divide-and-conquer. These systems are interesting and powerful, in some cases requiring mathematical sophistication of the user. Smith [22] describes combination of components in terms of category theory, with specialization of components described as functors and combination of components as colimit of categories consisting of types and functions. There is some similarity in the exchange of information between components in our system and in their colimits. However, while Kestrel’s computation of colimits is basically a unification process done all at once, our system performs the information exchange incrementally and uses the partial information to help the user specify the remaining parameters.

Krueger [11] surveys software reuse, with criteria for effective reuse. Mili [13] extensively surveys software reuse, emphasizing technical challenges.

The LabVIEW^{TM} system [9] allows graphical programming by connecting components; the connections represent data flow of streams of numeric values. Our connections are bi-directional and represent design-time information such as types.

ML [19] and Haskell [24] include polymorphic functions and functors (functions that map structures of types and functions to structures). While generics can be expressed in these languages, the complexity of the textual type specifications makes the functions harder

to reuse. Eiffel [12], the Standard Template Library of C++ [8], and Java provide container classes.

Batory [3] describes the AHEAD tool suite, which performs *step-wise refinement* on hierarchical structures of program and other files, somewhat analogous to aspect-oriented programming. Refinements are implemented by inclusion of code sections and expressed by equations in terms of functional composition. "Columns", representing different kinds of features, are composed separately. The use of layers whose interfaces are carefully specified allows the developer to ensure that the layers will interface correctly. We have focused on adapting interfaces so that generics can be reused for independently designed data.

Czarnecki and Eisenecker [7] overview methods of generative programming, including implementation of generics as templates in C++. Their technique creates a C++ template for each statement of a programming language, e.g. `if` statement. This technique relies on a standard compiler, but the highly nested structure of templates is not easy to understand and debug.

Akers, Kant et al. [1] describe SciNapse, which generates programs to simulate partial differential equations, e.g. for analysis of derivative securities. SciNapse transforms a small specification into a much larger program; it propagates parameters between components.

12. Discussion

Traditional technologies, including subroutines, templates, and OOP, inhibit software reuse because they require a precise match between types used in the application and in the reused software. The tendency in programming languages has been toward increasingly complex specifications for software modules, with increasingly strict type checking to ensure that modules and their uses match correctly; the complex specifications are written by humans. The compiler becomes a harsh master that criticizes but does not help. In addition, the user has to know too many details about library software in order to reuse it, adding a high learning cost to the cost of reuse. The system described here helps overcome these problems by adapting generics to the application (rather than requiring that the application conform to the library) and by using knowledge about both the generics and the application types to help the user make choices by menu (rather than requiring the user to remember exact names, procedure arguments, etc.). The programs that result are strongly typed and get compiled with strict type checking; but instead of humans writing the complex types, the computer generates correct types automatically.

The techniques that we describe are somewhat independent of the GLisp language. Similar techniques could be used to generate functors for components implemented in a language such as Java. However, our compiler generates optimized application code similar to code written by human programmers; Java libraries using many interface functors would incur significant overhead due to runtime interpretation.

The techniques described in this paper allow users to rapidly and easily capture data and procedural knowledge from the web and to reuse a library of generic procedures to generate custom analyses of web data. This provides a pathway to fulfillment of the promise of integration of the vast amount of information on the web.

Acknowledgment

We thank the anonymous reviewers for many helpful comments that have improved this paper.

References

[1] R. Akers, E. Kant, C. Randall, S. Steinberg and R. Young, Scinapse: A problem-solving environment for partial differential equations, *IEEE Computational Science and Engineering* **4**(3) (July–Sept 1997), 32–42. http://www.scicomp.com.

[2] R. Balzer, A 15 year perspective on automatic programming. IEEE Trans, *Software Engineering* **11**(11) (November 1985), 1257–1267.

[3] D. Batory, J. Sarvela and A. Rauschmayer, Scaling step-wise refinement, *IEEE Trans Software Engineering* **30**(6) (June 2004), 355–371. http://www.cs.utexas.edu/users/dsb/.

[4] D. Batory, V. Singhal, J. Thomas and M. Sirkin, Scalable software libraries, in: *Proc ACM SIGSOFT '93: Foundations of Software Engineering*, Association for Computing Machinery, December 1993.

[5] P. Clark and B. Porter, Building concept representations from reusable components, in: *Proc National Conf on Artificial Intelligence* (*AAAI-97*), American Association for Artificial Intelligence, 1997, pp. 369–376.

[6] Curl web site, http://curl.haxx.se/.

[7] K. Czarnecki and U. Eisenecker, Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000. http://www.generative-programming.org.

[8] S. Graphics, Standard template library programmer's guide, http://www.sgi.com/tech/stl/.

[9] N. Instruments. http://www.ni.com/labview/.

[10] D. Klahr and K. Kotovsky, Complex Information Processing: The Impact of Herbert A. Simon, Lawrence Erlbaum, 1989.

[11] C.W. Krueger, Software reuse, *ACM Computing Surveys* **24**(2) (June 1992), 131–184.

[12] B. Meyer, *Object-Oriented Software Construction*, (2nd ed.), Prentice-Hall, 1997.
[13] H. Mili, F. Mili and A. Mili, Reusing software: Issues and research directions, *IEEE Trans Software Engineering* **21**(6) (June 1995), 528–562.
[14] G.S. Novak, Glisp: A lisp-based programming system with data abstraction, *AI Magazine* **4**(3) (Fall 1983), 37–47.
[15] G.S. Novak, Conversion of units of measurement, *IEEE Trans Software Engineering* **21**(8) (August 1995), 651–661.
[16] G.S. Novak, Creation of views for reuse of software with different data representations, *IEEE Trans Software Engineering* **21**(12) (December 1995), 993–1005.
[17] G.S. Novak, Software reuse by specialization of generic procedures through views, *IEEE Trans Software Engineering* **23**(7) (July 1997), 401–417. http://www.cs.utexas.edu/users/novak/.
[18] G.S. Novak, Interactions of abstractions in programming, in: *Proc SARA 2000, LNAI 1864*, Springer-Verlag, July 2000, pp. 185–201.
[19] L.C. Paulson, *ML for the Working Programmer*, Cambridge University Press, 1991.
[20] D.R. Smith, Kids: A semiautomatic program development system, *IEEE Trans Software Engineering* **16**(9) (September 1990), 1024–1043.
[21] D.R. Smith, Toward a classification approach to design, in: *Proc Fifth Int'l Conf on Algebraic Methodology and Software Technology*, AMAST'96, Springer-Verlag, LNCS 1101, 1996, pp. 62–84.
[22] D.R. Smith, A generative approach to aspect-oriented programming, in: *Proc Third Int'l Conf on Generative Programming and Component Engineering* (*GPCE 04*), *LNCS 3286*, Springer-Verlag, 2004, pp. 39–54.
[23] M. Stefik, Planning with constraints, *Artificial Intelligence* **16**(2) (May 1981), 111–170.
[24] S. Thompson, *Haskell: The Craft of Functional Programming*, Addison-Wesley, 1996.
[25] D. Waltz, Understanding line drawings of scenes with shadows, in: *The Psychology of Computer Vision*, P.H. Winston, ed., McGraw-Hill, 1975.
[26] M.A. Weiss, *Data Structures and Problem Solving Using Java*, Addison Wesley, 2005.