Totally Different Structural Programming Programming Languages in ZigZag*

Antti-Juhani Kaijanaho† Benjamin Fallenstein‡

August 14, 2001

Abstract

The ZigZag structure and the ZigZag way of thought provide a medium for writing computer programs that is fairly different from the usual textual approach. It is similar but not identical to visual programming, but instead of using images or text to specify the program, we do it in a pure structure — the ZigZag structure, and thus eliminate a mandatory indirection. This allows the programmer to choose his or her own representation of the structure, whichever suits him or her best. There are several programming languages in ZigZag in the design and prototyping stages, including the zaubertrank, several Greek Clangs and Clasm. These languages, design issues relating to them and future prospects will be discussed.

1 Introduction: Nontextual programming

Traditional programming languages represent programs primarily as text, which is then transformed at compile time into an in-core structure that represents the true nature of the program (the "parse tree"). If a programmer wants to use a different interface to the structure of the program, she must use a custom-built preprocessor.

There have been attempts at freeing the programmer from this prison of text representation. The visual programming languages (Prograph[1] et al., not to be confused with Microsoft's Visual product series) represent programs primarily as diagrams or by using other graphical means. Still, the programmer is confined into the language designer's representation of the

*An invited talk presented at the First International ZigZag Conference, part of ACM Hypertext Conference 2001 in Århus, Denmark on August 14, 2001.

†University of Jyväskylä, Department of Mathematical Information Technology

‡University of Bielefeld, Oberstufen-Kolleg

[1] http://www.pictorius.com/prograph.html

program, and indeed, visual languages do worse than textual languages in being comprehensible when things get complicated.

The ZigZag paradigm provides a uniform metastructure on which one can represent all kinds of structures. We can thus encode the program structure inside a ZigZag structure. This would at first seem to have the same problem as the textual and visual approaches - that the programmer will be constrained to the model chosen by us, the language designers. However, here comes the unique quality of the ZigZag system into play: the system totally decouples views from the structure, so even though there are views that show the structure rather plainly, it is possible to create views that are very different from the conventional visualizations of a ZigZag structure. The second GZigZag implementation (nicknamed "Java" because of the directory name in the code repository) includes a good example of this: the virtualcommunity module, which is unrelated to programming, though.

When the canonical representation of the program is a ZigZag structure instead of a particular textual or visual renderition of it, the programmer can potentially choose from lots of different frontend representations of the same program, depending on her preferences. The choices would range from traditional textual representations to a visual representation to something quite different.

Of course, ZigZag is not the only possible medium for representing programs as structure. We know of at least one language (Flare[2]) that represents the program as an XML document, which also allows decoupling the structure from the presentation. We would not be surprised to find other attempts to do "programming in structure." (We would like to call it "structural programming", but that is too close to the name of a popular programming style.)

In the rest of this presentation, we first discuss the various issues in designing a language for programming in the ZigZag structure and then present some existing designs for ZigZag languages (of which most are implemented). Finally, we describe a specific code view, the zaubertrank.

2 To syntax or not to syntax

A question that comes into mind when discussing languages for programming in structure is whether these languages have syntax. One would think that they don't — we don't need to parse the structure like a textual program to find the parse tree, since we start at the structure. However, this is actually looking at the wrong end of it.

Syntax is not about parsing, although it does help in writing the parser. Syntax is about deciding which things are valid programs and which are not. Very few of the structural languages allow any kind of a structure to

[2] http://flarelang.sourceforge.net/

be a program, so there is a syntax. It is the declaration of valid ways of constructing a program, and it is closely related to the design of the language's semantics.

An example of a syntax possibility is using the parse tree model: a multiway tree consisting of operators at internal nodes and their parameters in the subnodes. This can be represented in a ZigZag structure using the standard multiway tree to binary tree mapping: siblings are connected on one rank, and their parent node is connected to the headcell of that rank along another dimension. In the "Java" version of GZigZag, there is a view for showing this tree structure like it's customarily shown in computer science texts.

Another possibility is to use postfix notation: put everything on one rank, with operands first and operations after the operands. This is similar to the syntax of the textual languages Forth and Postscript (among others). Or one could use prefix notation in the style of LISP, which is also, in a sense, a language for programming in structure. This seems to us like a fairly natural syntax for a ZigZag language, but one will need to figure out a way to represent sublists. One possibility for doing that would be defining a certain cell as representing the sublist within the list that contains it. That cell would then be connected to the sublist itself.

This leads into the matter of the so-called identity cells. These cells are representatives of larger substructures which get used (via a ZigZag connection) in other structures. Conventionally, every unit of a language, such as a procedure, would be represented by a certain identity cell. Now, when the ID cell is included in another substructure, the effect is that of including the whole substructure of the ID cell in the other substructure.

3 Name it — or not?

One problem every textual language needs to solve is the problem of naming. In a large program written by several people and containing stuff written by people who do not communicate with each other, it is too hard to make sure names do not clash. Thus languages have mechanisms for controlling name space: hierarchical naming of modules with a language-level convention for the use of the naming system, namespace boundaries (scope) and name shadowing.

This problem does not exist in languages for programming in structure, since things are identified, not by names, but by identities of the structure members (in ZigZag, cells). This makes every "name" unique throughout the world - assuming the cells itself have global identity, as cells in the third implementation of GZigZag (nicknamed "chartreuse") do.

Thus, names are not needed for referring to things in ZigZag-based languages. However, they can be used as documentation: human program-

mers do need a way to tell things apart. Thus, ZigZag-based=20 languages normally do allow people to name things. But the names do not matter to the language, so there is no need to handle name collisions in the language level.

4 What are we writing for?

Is the language for scripting ZigZag and writing applitudes, or should one be able to write conventional programs in it? This is a decision that has to be made when considering the design goals of a language, because it affects the fundamental assumptions that the language makes about the operating environment. For example, a traditional hosted environment for a medium-level language requires data types for numbers, pointers, characters and certain conventional data abstraction tools, whereas a ZigZag scripting environment can do with cells, cursors and media types. Also the choice affects the design of the language's runtime system: a ZigZag scripting language will want to provide a version of the runtime environment that stores all state (including procedure activation records) in the ZigZag space.

Currently all of the ZigZag-based languages have been designed with scripting in mind. Several have a design that emphasizes the all-state-in-structure approach, and one of them has this implemented.

5 Thales Clang

Thales Clang was one of the first ZigZag-based languages to have been designed. It was supposed to be the first in a series of experimental languages ("clangs" for "cellular languages") named after ancient Greek philosophers. It has never been fully implemented.

The striking feature of Thales Clang is its use of a "sequence dimension". It is the backbone of Thales Clang syntax: it is used to write procedure bodies and also for doing subexpressions. In Thales Clang, everything is an expression and thus can be evaluated to a value. All expressions on one rank along the sequence dimension are evaluated in order. The rank itself forms another expression, whose value is the value of the last expression on the rank.

Subexpressions in for example procedure argument evaluation are formed by having the argument consist of a no-operation cell which is the headcell of a two-cell rank along the sequence dimension. The actual subexpression will be at the end of the rank.

6 Flowing Clang

Flowing Clang was the first ZigZag scripting language in which actual keybindings have been implemented. Due to GZigZag's missing groupwork capabilities at the time, though, these keybindings have never been really used. Flowing Clang was also the first ZigZag language implementation featuring a debugger showing the whole operation of a program inside the ZigZag structure, providing the ability to walk through the program step-by-step. In Flowing Clang, functions with multiple parameters and return values are composed in an assembler-like control structure using sequential execution on a rank plus conditional branching.

The reason for Flowing Clang's name was that viewing the code on the parameter and clone dimensions, one could easily see how the values of variables would be set by one function and read by another, thus showing a structure similar to that of a dataflow language (although the semantics of a dataflow language would not be followed; this was just a different view for the very simple, assembler-like control flow structure).

7 Clasm

Clasm ("clang assembler") is the native scripting language for the third implementation of GZigZag (nicknamed "chartreuse"). It is already implemented, and it will be used to implement some of the essential features of the GZigZag client. Despite its name, it is a medium-level programming language with scoped variables, subexpressions and functions with parameters and return values.

A curious feature of Clasm is that it treats variables as callables: if you call one with no arguments, you "dereference" the variable to get its value, and if you call one with a single argument, you assign the value of the argument to the variable. Clasm is dynamically typed, and almost everything in the language — including variables — is first-class.

8 The zaubertrank

In contrast to the different designs described above, the zaubertrank[3] is not a ZigZag language, but a fancy view and editor usable with different languages. It shows trees of expressions as plain text, composing a text structurally similar to that parsed by text-based languages; it is thus capable of showing a program written in a ZigZag language in a way resembling a text-based language. The main goal of the zaubertrank, though, is to do this showing not computer code, but natural language.

[3] http://www.zaubertrank.org/

In this system, computer code is shown and edited as sentences which, although formal in style, can be understood by people not familiar with computer coding, thus taking the air of magic away from computer programs that currently makes even semantically trivial functionality unintelligible to the average user.

No usable version of the zaubertrank has been implemented so far, but several partial prototypes exist and work on them continues.

9 Conclusion

We have presented the idea of programming not in text, but in structure. We have discussed some of the issues regarding the design of ZigZag-based languages for programming in structure. We have presented some such designs. The field of designing and implementing these languages is virtually uncharted. We hope to have more experience real soon now. Please join us, and share the software.

Acknowledgements

Thank you, we love you all.