Abstraction Considered Harmful*

Wouter van Oortmerssen

May 1, 2005

Abstract

Abstraction facilities have always been considered the number one prerequiste for a programming language, without much debate. This paper presents the KNOFLOOK language where abstraction is much more implicit, and we look at the benefits this would have for the programmer.

1 Goals

The KNOFLOOK programming language fullfills three purposes:

- To attempt to radically change the way we create code from a software engineering point of view.
- To create a language more suitable for beginning programmers or even "non-programmers".
- To create a syntax that works better as a graphical programming language than any previous attempts.

This document presents the concept of the language, an exact language definition is to follow later

1.1 Software Engineering & Abstraction

Abstraction[1] is such a fundamental part of programming that nobody ever questions it. *Abstraction is good*. Or is it? Certainly, getting your abstractions right in a large codebase may also be one of the hardest parts of programming.

*Provocative on purpose, inspired by a certain classic paper by Edsger Dijkstra

[1]When referring to abstraction facilities, our main focus is on functions (or methods), but also classes or any additional such constructs a language may have. To a lesser extent, for the purpose of the discussion here, variables can also be considered an abstraction facility.

Beginning programmers always fall into the trap of *underabstraction*, a.k.a. *cut & paste code*. Abstraction is hard, just replicating your code and modifying it is easy. Every programmer has been through this stage and remembers how it ended: with your first large program that became totally unmaintainable because of lack of abstraction.

Experienced programmers have left the days of underabstraction behind, and have traded it in for another mistake: *overabstraction*. Scared of underabstraction, and experienced enough that creating an abstraction is no longer hard, they have decided the holy grail is now to abstract as much as possible, preferably designed up front, before a single line of implementation is written. They are in denial about how hard design is, especially in the face of change, and their abstractions create enormous amounts of redundancy that has the power of choking maintainability almost as badly as underabstraction does. Sadly, the majority of programmers never get beyond this phase. But hey, abstraction is *fun*!

Expert programmers, through deep experience with large, under & overabstracted codebases, are now in the final phase: that of *perfect abstraction*. The principle is simple: abstractions should be created such that there is no redundancy: no underabstraction and no overabstraction. Reaching this elusive goal however is very hard, as it requires constant intensive refactoring as an application grows, something which very few programmers are able & prepared to do to the level required for success[2].

So abstraction is on the one hand extremely hard, yet on the other hand its purpose so incredibly simple: it serves to remove redundancy in program code, as redundancy is the dominating source of errors. Finding redundancy is normally something that computers do well (see e.g. compression algorithms), so maybe having humans do abstraction is the wrong thing altogether? Maybe it can be done automatically?

Maybe. There is hope that one day we will have a system that completely automatically maintains & refactors code to be non-redundant, and thus abstracts as needed. But the disconnect between the mental model of the programmer and code factoring the "compression" algorithm comes up with may be counter productive if it is too mechanic, or require complex artificial intelligence. As such, KNOFLOOK is a very simple language that is halfway between classical languages, and fully automated factoring: it presents a model where abstraction is implicit and code automatically stays close to perfectly factored as it is edited.

1.2 Humans & Abstraction

Humans have a hard time thinking in terms of abstractions. They like to think in terms of concrete cases. To learn something new, they first require to see concrete

[2] Methodologies such as XP try to support this using methods such as OnceAndOnlyOnce, YouArentGonnaNeedIt and DoTheSimplestThingThatCouldPossiblyWork. Probably one of the reasons why XP is not a bigger success story is because of the thoroughness with which these rules have to be applied for them to have the full effect.

things over and over again, before the general pattern emerges. If humans could work with concrete code rather than abstractions, their understanding of new code would be smoother and faster, and thus their general capacity of working with larger, more complex programs increased. Though this benefits any level of programmer, it certainly benefits beginning programmers most, and may even lower the barrier to entry so much that what previously were "non-programmers" can now ease into writing code. Designing a language for non-programmers is still somewhat of a holy grail as there are vast amounts of people that could benefit from being able to do minor amounts of programming.

1.3 Graphical Programming & Abstraction

Research on graphical programming has a rich history of research, mostly centered around representing dataflow languages visually, but also grid & graph rewriting based languages. A central problem here again is how to deal with abstraction. Dataflow is popular because it deals with variables very well (they become lines), but doesn't really have a good way of doing functions. Rewriting languages do functions well (they become rules, allowing visual pattern matching), but still struggle with variables/placeholders. KNOFLOOK hopes to do well on both accounts.

2 Implicit Abstraction

Some of the features that are about to be introduced would be problematic if edited or represented in a classical textual programming language. As such, the environment for KNOFLOOK is an editor that allows structural control over editing and special purpose rendering to express structure. The base idea is somewhat like a tree structure editor, but its actual implementation could be anything from something that tries to stay as close to textual code as possible, to a full visual programming language.

In most programming languages, a "program" consists of a list of abstractions (usually functions and/or classes). Since KNOFLOOK doesn't have explicit abstraction, the entire program is a single tree structure (like one giant expression). Nodes of that tree structure are all operations, where operations can be what in other languages are operators, builtin functions, value constructors etc., all concrete at the bottom level. Evaluating the program just evaluates the tree, much like one giant expression. Without functions or variables, how can one possibly express an interesting program in such a language? this is where KNOFLOOK's two main features come in: *sharing*, and *overriding*.

2.1 Sharing

Sharing is the simplest of the two, and allows the programmer to reuse computed values, essentially making the tree internally into a DAG. This is closely coupled with

editing & representation: the user can simply drag any tree node with the mouse, and release it somewhere else (or alternatively, use copy/paste). That tree is now shared between the source where it was taken from and the destination where it was released. The DAG is still rendered as a tree, with the shared node rendered twice if both occurrences happen to be on screen at once. The editor shows visually (using a box and/or different colour background) that a particular tree is shared, signalling the user that it is used in more than one place, so that if he changes anything inside it, he is aware that he is changing all occurrences. If one of the two occurrences gets overwritten, then sharing is automatically undone. Sharing is a bit like creating & using local variables rolled into one, or using global functions/constants.

Example: starting expression `2 * 3 + 1`, user clicks on the `*` node, drags it over the `1`, and releases it. The new code now looks like (assuming we denote a shared area in ascii using square brackets): `[ 2 * 3 ] + [ 2 * 3 ]`. The equivalent traditional language code would be something like: `localvar = 2 * 3; return localvar + localvar`. Now the user clicks on any of the two 3's and types a `4`. The code now looks like `[ 2 * 4 ] + [ 2 * 4 ]`, the other shared copies are automatically updated.

2.2 Overrides

Changing something in a shared node by default changes all occurrences. So what happens if you only wanted to change your "local copy" (i.e. the shared node from the perspective of a particular parent only) ? You can do this by creating an override, which is as simple as a normal drag or paste action, but now holding an additional modifier key. It is called overriding, because essentially what you are doing is saying that you want to reuse a certain node, except you want to replace a certain part of it (without affecting all other parents of that node). Overrides are rendered in-place, i.e. as part of the node that it overrides a part of. Overrides are a bit like function arguments, formal parameters, and function definition all merged into one, as they allow you to reuse a block of code and factor out a part of it on the fly, without requiring any extra work or thought.

Example: lets continue our previous example of `[ 2 * 4 ] + [ 2 * 4 ]`, and assume that user would like to change the second occurrence of 4 only, but he would like to keep the sharing of the `2 *` part (it helps to think of this example as involving more code than it does). He clicks on the second `4` using a modifier key[3], and now types a `5`. The new code, assuming we would display overrides in ascii using curly braces, would look like: `[ 2 * { 4 } ] + [ 2 * { 5 } ]`. The `4` and the `5` are now independent even though they sit in the same shared area. The equivalent traditional language code would be something like: `f(x) { return 2 * x }; return f(4) + f(5)`. Note how the functional abstraction

[3] How exactly this works is an UI design question: the modifier key could be held at selection time, or during dragging of a tree. Or an entirely different mousebutton can be used. This will have to be intuitive. A modifier is intuitive because the user is forcing an exception to the default behaviour of normal click & drag.

was introduced transparently by just dealing with sharing and unsharing. If the user deletes one shared instance, or changes the override `{ }` node to something else, then the override (and sharing) will automatically be undone.

By default when you create an override, you are overriding from the perspective of the parent of closest containing shared node, i.e. in the above example, the `{ }` are local to the closest surrounding `[ ]`. However, when sharing is nested (which can happen easily, the equivalent of nested function calls/definitions or variables), sometimes it makes sense to have the override be dominated by a parent one sharing level up. When an override is selected, the sharing node which it is operating upon is graphically visible, and an operation is available for moving it up or down to the next parent. If more than one shared occurrence are dominated by this parent, then the override will automatically hold for all of them (this is a bit like dynamic scoping).

In the above example, notice how making the 5 an override also made the 4 an override. What if there had been 3 shared nodes, as in `[ [ 2 * { 4 } ] + [ 2 * { 4 } ] + [ 2 * { 5 } ] ]`, are the two 4's now both independent? The answer is, that by default, the remaining 4's would be overrides with respect to the closest enclosing shared area that encloses them all. In that sense, the two 4's are still shared with eachother (because they are the same override), but not with the 5. This gets harder to express in ascii, but the equivalent traditional code for the situation above would be something like: `g(y) { f(x) { return 2 * x }; return f(y) + f(y) + f(5) }; return g(4)`. Similarly, if the structure would have been `[ 2 * { 4 } ] + [ [ 2 * { 4 } ] + [ 2 * { 5 } ] ]`, the user could elect to move the override of 5 up one level, which would cause the expression to become `[ 2 * { 4 } ] + [ [ 2 * { 5 } ] + [ 2 * { 5 } ] ]` where the two 5's are the same override, in the same way as the previous example[4].

3 Benefits

Sharing and overriding combined allow the user to create code by simply dragging around the blocks of code they need, only occasionally stopping to think wether they want to modify all copies, or a subset. By doing so, (s)he unknowingly is creating abstractions, but without requiring any thought or planning. The central achievement here is that *the simplest, default actions maintain perfect factoring, to create bad abstractions you have to work hard*. This as opposed to classical languages where bad abstraction is easy and perfect abstraction is hard work.

Overabstraction is hard or impossible, because there is no way to create a "function" unless you already have the implementation as concrete code, and are needing it

[4]Selecting, or hovering the mouse over an override visually shows which enclosing block the override is relative to. It is hoped that this will be intuitive enough to allow the user to quickly edit and still understand how it affects the sharing of values. As with all UI issues, better representations of overriding may be found.

a second time. It discourages underabstraction, as abstraction is now so easy and natural:

- code can go from being a normal block of code to being a "function" without having to move/reformat it. You almost never need to move code, you can simply keep on writing!
- additional "paramaters" can be added without disturbing the callers that don't need it. "functions" can be used at so many different levels and combinations that goes well beyond default parameters, in fact, overrides provide a more powerful "abstraction facility" than classical functions.
- code for repeating occurances can now sit right next to its additional calls without having to be moved out of its function scope

Even if it were possible for bad abstraction to creep in (which is hard, but possible, nothing stops you from creating the same code twice manually), in a system such as this it would be terribly easy to check for redundant code and either fix it automatically or make suggestions to the user.

4 Semantics

The above describes the core of the language, but it is clear that there is quite a lot of choice left in the details of the semantics of the rest of the language. This section provides some of them.

4.1 Evaluation

As it stands, the most natural model for the language is that of a purely functional language, as the whole program is one giant expression, and there is no natural way to express globals, assignment, or any kind of state or object identity. Evaluation could be eager or lazy. For eager evaluation, sharing by default causes the block of code the be evaluated only once, and reused thereafter, except when some of it is overridden, in which case it has to be evaluated for each individual override (which may be less than the amount of occurrences of the shared node (!)). If the language were to have side effects, then a side effect inside a shared area would force it to be evaluated for every occurrence.

4.2 State

If a language with state was desired, there are a number of ways in which this can be achieved

- Using (concurrent) tree space evaluation, much like in the Aardappel language (a Linda-like mechanism). This fits the functional basis of the language quite well, but has the huge disadvantage that it would require some sort of identifiers for identifying items in the tree space.
- Using a special "replace" operator. What this would do is to "replace" the value of an arbitrary tree node specified. As long as that tree node is shared, changes would propagate through the program. This has the advantage that it fits in better with the no-abstraction style of the language, but may make the program harder to read as the global value is identified by its initial value.

Neither of these is very attractive.

4.3 Data structures

the natural model for data structures in KNOFLOOK would be the usual functional constructor and selector functions. It may well be that a better model can be found.

5 Editor specifics

The main issue here is how navigating & editing such a huge tree can be made manageable. The following features could contribute:

- Easy (un)folding of any node, specifically shared nodes. Additionally, shared nodes with overrides can be folded such that the shared code is not visible but the overrides contained in it are (this is more akin to a function call).
- More compact representation for small non-DAG leaf areas of the tree, shown linearly more like normal programming text.
- Having a stack of "panes" where a node could be shown in a new pane as an alternative to be shown inline. This would reduce clutter and speed up navidation
- Having a more "fisheye" like tree display where contextual parts of the tree automatically get compact representation around the edges as you navigate.