

Project Ideas for Amit Agarwal and Joseph D. Kulisics

Joseph D. Kulisics

2008 April 13

Abelson and Sussman make the unsubstantiated claim in their classic textbook, *The Structure and Interpretation of Computer Programs*, that garbage collection is incompatible with imperative programming, and the reasons for the claim become clear in implementations of the observer design pattern in garbage-collected, imperative and object-oriented programming languages like Java. An object-oriented application typically has one of two execution profiles: either the application creates a fixed number of objects upon initialization exiting as soon as any of the objects reaches a terminal state, or the application dynamically creates many objects during its run exiting on a condition independent of the states of most of the objects in the application. In the first instance, a programmer can write a static implementation of the message-passing style and make call-backs by directly connecting through references all of the objects of the application. In the second case, the programmer cannot statically connect all of the objects in the application, and the observer design pattern exists to provide a single, static object, the observer, through which all dynamically created objects can connect and pass messages. In dynamically managing references to objects in an observer, a program becomes difficult to statically understand and analyze with respect to garbage collection efficacy, and in a language like Java, large systems often waste memory in a phenomenon known in the Java programming community as unintentional object retention, which resembles a memory leak in simpler interpreters and compilers requiring the programmer to explicitly manage memory.

The problem of garbage collection manifests itself in the observer design pattern when the application programmer takes a naïve view of garbage collection assuming that the garbage collector can reclaim all memory to which the programmer no longer statically has access as opposed to the actual guarantee of garbage collection that the garbage collector will eventually return to the system all memory that it cannot reach from the registers of the virtual machine on which the program executes. In effect, programmers incorrectly understand garbage collection as freeing them from having to understand memory. Unfortunately, imperative and object-oriented programming expose to the programmer a model of the underlying machine in which there is a fundamental and accessi-

ble distinction between values resulting from computations and the store of the machine, which contains mutable data, a model of the machine in which one must necessarily understand memory. (By contrast, functional programming deals only in values as the results of computations drawing no distinction between values and cells in a store of memory, and functional programming with the pure, functional core of a language like Lisp or Scheme conceals the von Neumann machine and equivalent Turing machine abstraction of the computer safely behind the interpreter, the only element of the runtime system able to manage memory.) As evidence of the generality of the problem, the connection of the problem to imperative features of language, and the difficulty in statically detecting and correcting the problem, we offer two examples, the first a simple implementation in Java of the observer design pattern containing an error resulting in unintentional object retention and the second the same example written in Scheme in a message-passing style using imperative features outside the functional core of Scheme. The examples are available at the following web addresses:

http://copper.chem.ucla.edu/~kulisics/txt/echo_memory_leak.java

<http://copper.chem.ucla.edu/~kulisics/txt/ObserverMemoryLeak.scheme>

Drawing on observations by Wirth, instead of viewing unintentional object retention as a fundamental incompatibility of imperative and object-oriented models of programming with garbage collection as Abelson and Sussman might view the problem, one could consider the problem to result from references and aliasing. A programmer equipped with the full power of references, which are not strictly necessary to the imperative and object-oriented models, can dynamically bind memory through aliases making the memory uncollectable, and in the spirit of Wirth, asking of programmers how the desire for references arises rather than asking of programmers what they require to complete their work, references appear necessary in the observer design pattern only as a means to implement call-backs. The observation suggests constraining or eliminating the use of references entirely in a new imperative or object-oriented programming language.

In our first and most developed area of research, we offer two proposals to contain the damage caused by aliasing in the presence of garbage collection. Our first proposal is to merge into the control structure of a simple interpreter of a Scheme-like language with imperative features a messaging facility intended to act as a broadcast medium and obviate the need for an observer in an application dynamically creating a large number of objects during execution. The second proposal entails constraining the use of references. The environment context of computation in statically scoped languages resembles a tree, and in the unconstrained use of references, the programmer can pass references to the root of the environment context and ultimately laterally through the tree. We propose to constrain a simple interpreter of a Scheme-like language containing imperative features to forward passing of references. By one of the two approaches, we hope either to eliminate the need for an observer design pattern and references as a device for inter-object communication or to show that a memory leak in

an observer-based design becomes difficult to casually write and easy to correct should the problem occur.

As a second area of research, we consider the view of languages offered by Abelson and Sussman as consisting of primitive elements, means of combination, means of abstraction, and an interpreter of the language with a control structure containing effective, procedural interpretations of the means of combination and means of abstraction of the language. The control structures of the most elementary programming models, the functional and imperative models, offer little expressive power to the programmer over the basic operations of the underlying Turing machine. (In fact, the functional model presents to the programmer a more impoverished view of computation than the view contained in the Turing machine.) Most of the language-level constructs of the functional and imperative programming languages are syntactic sugar for the memory operations, jumps, and stack operations of a Turing machine. The first nontrivial, conceptual extension of the control structure of a language occurs in the nondeterministic model of programming in which the control structure of the interpreter of a language merges with a mechanism for automatically branching and backtracking, and the extension to the control structure comes with a language-level extension allowing the programmer to explicitly invoke the extension to the basic control structure. Constraint-based systems and Prolog merge with the control structures of their interpreters effective, procedural interpretations of arithmetic operations and effective, procedural interpretations of the operations of a restricted propositional calculus, respectively. Other extensions of the control structure of a language based on operations of another domain might be possible, and we propose to extend the control structure of a simple, interpreted language derived from Scheme to add a pattern matcher and to incorporate a language-level extension to the language to allow the programmer to explicitly invoke the pattern matcher. We hope that by a careful language-level extension of the language, we might be able to allow programmer to express ideas like parameterized types and dynamic dispatch without actually encoding support for the concepts in the interpreter of the language.

As a third and final area of research, we consider that computations come with an environment context, and in modern, statically scoped languages, trees are used to model the environment context of computation offering more structure than the most obvious models of the environment context of computation supported directly by a Turing machine, the array, and subsuming the full power of the raw model of the environment context of computation as trees subsume arrays as data structures. The data structures supporting static scoping can support language-level access to the environment context of computation and user-selectable scoping characteristics simply because the data structures behind static scoping are more flexible than the arrays providing the support sufficient for dynamic scoping. Computations also come with a control context, but the data structures behind the model of the control context of computation have not tended to the richness of the structures supporting the environment contexts of computation in modern languages. A call stack typically models

the control context of computation, and in a sense, the stack presents fewer possibilities for understanding and manipulating the control context of computation than even arrays. Recent efforts at extending the control context of computation attempt to clone the control context with its underlying model for the purpose of parallelization of computation as in the case of threading. We propose to integrate into the control structure of a language a model of the control context of computation having all of the richness of the trees modeling the environment context of computation in statically scoped languages. Some languages like Scheme provide language-level access to the control context of a computation—in Scheme, the *call-with-current-continuation* special form allows a programmer to modify the control context of computation—and further, we intend to offer language-level access to our enriched control contexts. Though we cannot imagine the immediate applications of enriching the model of the control context of the language, we hope that experimentation in the area has the potential to enhance the expressive power of a programming language and simplify the work of programmers.

Thank you,
Joseph D. Kulisics