EXHIBIT 12: LEWIS IN VIEW OF DYER AND FURTHER IN VIEW OF PROEBSTING

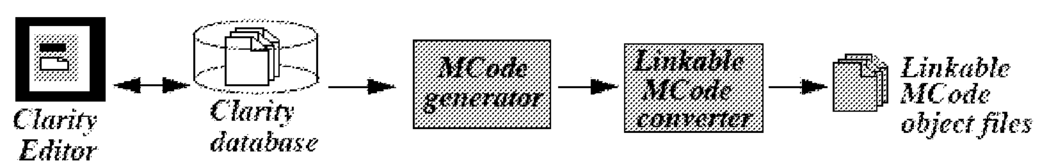
Clarity MCode: A Retargetable Intermediate Representation for Compilation
Brian T. Lewis at L. Peter Deutsch and Theodore C. Goldstein
ACM, IR '95, 1/95, San Francisco, California, USA (1995)
("*Lewis*")

Java Decompilers Compared
Dave Dyer
JavaWorld.com (July 1, 1997)
("*Dyer*")

Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?)
Todd A. Proebsting and Scott A. Watterson
Proceedings of the Third USENIX Conference on
Object-Oriented Technologies and Systems,
Portland, Oregon (June 1997)
("*Proebsting*")

U.S. Patent No. 6,061,520 – Claim 1	***Lewis* in view of *Dyer* and further in view of *Proebsting***
1. A method in a data processing system for statically initializing an array, comprising the steps of:	*Lewis* discloses a method in a data processing system, i.e. the "Clarity MCode compilation system," for statically initializing an array. For example, *Lewis* discloses that the MCode instruction set includes instructions for statically initializing an array, e.g., "AllocArray." "This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode." *Lewis* at 120.

U.S. Patent No. 6,061,520 – Claim 1	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	AllocArray(array type index) ArrayIndex(array type index) ArrayLength(array type index) InvokeOuter(method type index) InvokeDelegated(method type index) Widen(base interface number) Narrow(target type index) BeginSwitch(end swi EndSwitch(end switc BeginExprCase(int) BeginDefaultCase() EndCase(end switch DoBreak(end tag) DoContinue(end tag) Figure 4: The MCode instruction set *Lewis* at 123 Fig. 4 (excerpt). *Dyer* discloses a review of three Java Decompilers. These are programs that convert Java class files into Java source code, effectively reverse engineering compiled code to figure out how the underlying code works. *Dyer* discloses, in most relevant part, an example of the code that would be used to implement a static initialization of an array. *Dyer* not only discloses the concept of the static initialization of an array by way of an instruction provided to the Java virtual machine; *Dyer* also provides examples of actual code to achieve this result. The decompiler of *Dyer* inherently performs a symbolic execution of the compiled bytecode. The functionality of such a decompiler, including the symbolic execution that decompilers use to identify the target of compiled bytecode for the recovery of the underlying Java source code, is disclosed in detail in *Proebsting*. The combination of the *Proebsting* disclosure, which explains the symbolic execution technique of a decompiler, with the actual results of a decompiler as applied to a static initializer for an array, as shown by *Dyer*, render obvious the static initialization of an array as in claim 1.
compiling source code containing the array with static values to generate a class file with a clinit method containing byte codes to statically initialize the array to the static values;	*Lewis* discloses compiling source code containing the array with static values to generate a class file with a clinit method containing byte codes to statically initialize the array to the static values. For example, *Lewis* discloses compiling source code written in the Clarity C++ programming language into MCode object files, and that the MCode instruction set includes instructions for statically initializing arrays.

U.S. Patent No. 6,061,520 – Claim 1	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	"The *Clarity C++* programming language is a dialect of C++ being developed in Sun Microsystems Laboratories. Clarity shares many features with C++ but is less complex and has a more consistent syntax and simpler semantics without loss in expressiveness. . . . Clarity is intended to be a wide-spectrum language suitable for both systems and application programming, particularly of distributed software. To support the compilation of Clarity, we have developed a high-level, machine-independent intermediate representation that we call *MCode* (for "middle code"). We use MCode to compile Clarity programs at execution time (i.e., on-the-fly) into SPARC code for the Solaris operating system. This code generator is designed to be largely machine independent: besides the SPARC code generator, an Intel x86 version is being developed." *Lewis* at 119 (footnote omitted). "This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries."  Figure 1: The development-time portion of the Clarity MCode system *Lewis* at 120-21, Fig. 1.

U.S. Patent No. 6,061,520 – Claim 1	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	"Linkable MCode object files contain a machine-independent *pickle* of an MCode code unit. This is a compact, platform-independent encoding of the MCode information into a sequence of bytes. This pickle can later be internalized or unpickled to reconstruct the original MCode. The MCode for each procedure is pickled separately to support procedure-at-a-time processing. The current encoding is not especially compact although the Linkable MCode object files are still smaller than object files containing machine code. We intend to replace the current pickle format with a more compact one. Linkable MCode object files are platform-standard object files that are processed in the usual way by the platform's standard linker. This means they need to include platform-dependent definitions of global variables and procedures, and descriptions of referenced symbols. We currently encode ("mangle") symbol names in order to ensure that the resulting executables or shared libraries are type-safe with respect to the Clarity language. Eventually, this type-safety will be checked by a Clarity *prelinker*. Besides symbol definitions and references, our Solaris Linkable MCode object files also contain a few machine language instructions for each procedure's entry code. This entry code allows C code to call the MCode procedure. On the SPARC, this entry code consists primarily of a three instruction "trampoline" that redirects the call to the appropriate target procedure chosen by the interpret/compile strategy module in the MCode runtime. The SPARC entry code also has three words used when atomically updating the trampoline. Despite this platform-specific information, the contents of a Linkable MCode file are mostly platform-independent. The Linkable MCode converter itself is also mostly platform-independent. We currently execute the Linkable MCode converter during program development, before a program is distributed. It could also be executed when the program is installed on a particular platform.." *Lewis* at 125-26.

U.S. Patent No. 6,061,520 – Claim 1	***Lewis*** **in view of** ***Dyer*** **and further in view of** ***Proebsting***
	AllocArray(array type index) BeginSwitch(end switch tag) ArrayIndex(array type index) EndSwitch(end switch tag) ArrayLength(array type index) BeginExprCase(int) BeginDefaultCase() InvokeOuter(method type index) EndCase(end switch tag) InvokeDelegated(method type index) Widen(base interface number) DoBreak(end tag) Narrow(target type index) DoContinue(end tag) Figure 4: The MCode instruction set *Lewis* at 123 Fig. 4 (excerpt). "Figures 5 and 6 give an example of generated MCode. The Clarity method *startup* in Figure 5 produces the MCode instructions shown in part in Figure 6."

U.S. Patent No. 6,061,520 – Claim 1	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	(code figure below)

```
ThreadedSimulation: module
{
    work_mutex: Threads::Mutex;
    work_per_worker: int;                 // protected by work_mutex
    extra_work: int;                      // protected by work_mutex

    Worker: type = interface inherits Threads::Thread {};
        // an unusual interface: no methods beyond Thread::startup and the other Thread methods

    WorkImpl: type = implementation of Worker
    {
        implement startup: method (our_workers: in int)
            {   // executed when the thread is started; delegates most of its work to forked sibling workers
                within work_mutex {       // acquire work_mutex for duration of the within statement
                    our_work = work_per_worker;
                    if (extra_work > 0) {our_work += 1; extra_work -= 1;}
                }
                delegated_to_workers: int = (our_workers - 1);
                if (delegated_to_workers > 0) {
                    left_workers = delegated_to_workers/2;
                    right_workers = delegated_to_workers - left_workers;
                    if (left_workers > 0) left_sibling = new WorkerImpl(left_workers);
                    right_sibling = new WorkerImpl(right_workers);
                }
                do_work(our_work);
            };

        // the following declarations are private to the WorkerImpl implementation
        do_work: method (work_to_do: in int) { /* elided */ };

        our_work: int = 0;
        left_workers: int = 0;
        right_workers: int = 0;
        left_sibling: Worker;             // left delegatee; manages "left_workers" workers
        right_sibling: Worker;            // right delegatee; manages "right_workers" workers
    };
    ...
};
```

Figure 5: Part of the Clarity version of the μC++ test program

Lewis at 124, Fig. 5.

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 1</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td>receiving the class file into a preloader;</td><td>Lewis discloses receiving the class file, i.e. the “MCode object files,” into a preloader, i.e., the “Code Generator.”

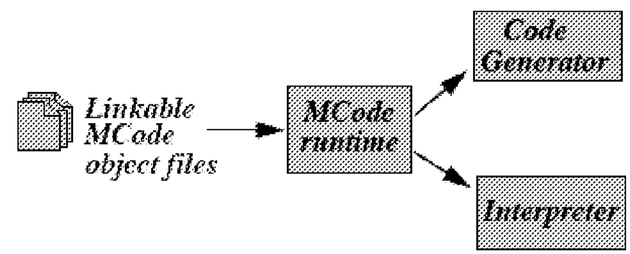
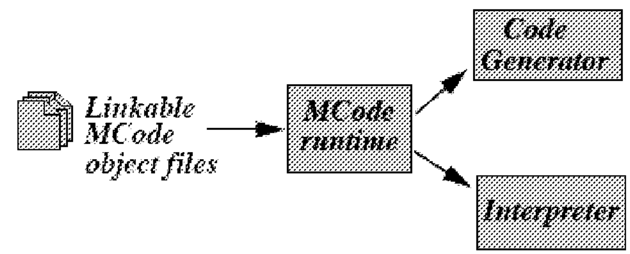


Figure 2: The runtime portion of the Clarity MCode system

“The runtime component of the MCode system is illustrated in Figure 2. The MCode runtime in an MCode-containing executable internalizes the MCode for each procedure as needed, when the procedure is first called. It also implements a interpret/code generate policy separately for each MCode procedure. This policy chooses for each procedure whether to interpret it, generate code, or interpret then later generate code, or generate better code. The code generator currently produces SPARC code of approximately the quality of the SunPRO C compiler at the default -O2 optimization level. A port of the code generator to the x86 is underway. The MCode interpreter interoperates with all SPARC ABI code. Like the compiler, it is reentrant and supports multithreaded programs. It also does extensive checking during program execution, which makes it especially useful for uncovering errors in Clarity programs that are otherwise difficult to detect. The interpreter will also be used by the Clarity debugger that we are developing to evaluate Clarity statements and expressions.” Lewis at 120-21.</td></tr>
<tr><td>simulating execution of the byte codes of the clinit method against a memory without executing the byte codes to identify the static initialization of the</td><td>Lewis discloses simulating execution of the byte codes of the clinit method against a memory without executing the byte codes to identify the static initialization of the array by the preloader. For example, Lewis discloses simulating execution of the MCode instructions in the MCode object files, and that the MCode instruction set includes</td></tr>
</table>

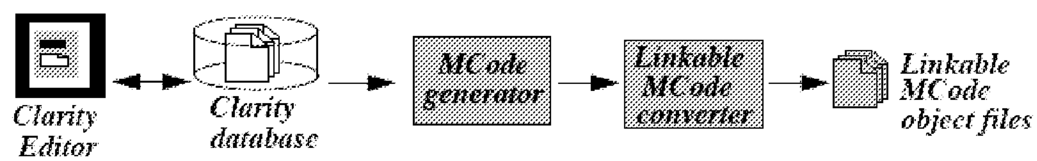
U.S. Patent No. 6,061,520 – Claim 1	*Lewis* in view of *Dyer* and further in view of *Proebsting*
array by the preloader;	instructions for statically initializing arrays.  Figure 2: The runtime portion of the Clarity MCode system *Lewis* at 121. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126.

U.S. Patent No. 6,061,520 – Claim 1	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	AllocArray(array type index) ArrayIndex(array type index) ArrayLength(array type index) InvokeOuter(method type index) InvokeDelegated(method type index) Widen(base interface number) Narrow(target type index) BeginSwitch(end switch tag) EndSwitch(end switch tag) BeginExprCase(int) BeginDefaultCase() EndCase(end switch tag) DoBreak(end tag) DoContinue(end tag) Figure 4: The MCode instruction set *Lewis* at 123 Fig. 4 (excerpt). To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, *Dyer* and *Proebsting* together show that the simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by *Proebsting* discloses a technique for decompilation of Java byte code that simulates the Java virtual machine's stack: "This paper presents our technique for automatically decompiling Java bytecode into Java source." *Proebsting* at Abstract. "*Decompilation* transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. . . . Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.

U.S. Patent No. 6,061,520 – Claim 1	*Lewis* in view of *Dyer* and further in view of *Proebsting*

	We present a technique for transforming low-level Java bytecode into legal Java source code." *Proebsting* § 1.
	
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." *Proebsting* § 5.
	
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." *Proebsting* § 1.
	
	"Symbolic execution of the bytecode creates the corresponding Java source expressions. . . . Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." *Proebsting* § 2.
	
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of *Lewis* with the simulation (symbolic execution) of bytecode against a memory, as disclosed by *Proebsting*. And, even more specifically, the artisan would look to known decompilers, such as those disclosed in *Dyer*. *Dyer* reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. *See Dyer* at 1. In that review, *Dyer* analyzes the decompilation of code, including the decompilation of static initializers:
	
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	
	`public ConsoleWindow(String string, int i1)`
	`{`
	`dead = false;`
	`styles = { "Plain", "Bold", "Italic" };`
	`sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };`
	`...`
	
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these

U.S. Patent No. 6,061,520 – Claim 1	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	initializers are actually implemented by inline code inside constructors, generated by the compiler. When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3. The fact that some of *Dyer*'s decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed -- a technique used to identify the operation of the byte code in order to generate equivalent high-level language expressions. In fact, the '520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (*see* '520 patent at 8:54-9:13)) as an "illegal" Java constant pool entry. The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 . . . is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual machine to recognize an "illegal" instruction such as the new CONSTANT_Array type. Further, during Examination claim terms are to be given their broadest reasonable interpretation (*see* MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact format (whether syntactically correct or not) is not an issue. And in any case, *Dyer* discloses that -- in contrast to the Mocha decompiler's "illegal" code -- the WingDis decompiler produced "syntactically correct code." *Dyer* at 3.
storing into an output file an instruction requesting the static initialization of the array; and	*Lewis* discloses storing into an output file, i.e., the "Linkable MCode object file," an instruction requesting the static initialization of the array.

U.S. Patent No. 6,061,520 – Claim 1	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	"This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries."  Figure 1: The development-time portion of the Clarity MCode system *Lewis* at 120-21, Fig. 1. "Linkable MCode object files contain a machine-independent *pickle* of an MCode code unit. This is a compact, platform-independent encoding of the MCode information into a sequence of bytes. This pickle can later be internalized or unpickled to reconstruct the original MCode. The MCode for each procedure is pickled separately to support procedure-at-a-time processing. The current encoding is not especially compact although the Linkable MCode object files are still smaller than object files containing machine code. We intend to replace the current pickle format with a more compact one. Linkable MCode object files are platform-standard object files that are processed in the usual way by the platform's standard linker. This means they need to include platform-dependent definitions of global variables and procedures, and descriptions of referenced symbols. We currently encode ('mangle') symbol names in order to ensure that the resulting executables or shared libraries are type-safe with respect to the Clarity

U.S. Patent No. 6,061,520 – Claim 1	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	language. Eventually, this type-safety will be checked by a Clarity *prelinker*. Besides symbol definitions and references, our Solaris Linkable MCode object files also contain a few machine language instructions for each procedure's entry code. This entry code allows C code to call the MCode procedure. On the SPARC, this entry code consists primarily of a three instruction 'trampoline' that redirects the call to the appropriate target procedure chosen by the interpret/compile strategy module in the MCode runtime. The SPARC entry code also has three words used when atomically updating the trampoline. Despite this platform-specific information, the contents of a Linkable MCode file are mostly platform-independent. The Linkable MCode converter itself is also mostly platform-independent. We currently execute the Linkable MCode converter during program development, before a program is distributed. It could also be executed when the program is installed on a particular platform." *Lewis* at 125-26. AllocArray(array type index) ArrayIndex(array type index) ArrayLength(array type index) InvokeOuter(method type index) InvokeDelegated(method type index) Widen(base interface number) Narrow(target type index) BeginSwitch(end swi EndSwitch(end switc BeginExprCase(int) BeginDefaultCase() EndCase(end switch DoBreak(end tag) DoContinue(end tag) Figure 4: The MCode instruction set *Lewis* at 123 Fig. 4 (excerpt). To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Proebsting* and *Dyer* to arrive at the claimed feature. Specifically, *Proebsting* discloses that the Krakatoa decompiler writes the source code (which would necessarily include any static array initialization instructions) to an output file: "We have implemented a prototype Java decompiler, Krakatoa, in Java. We have run

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 1</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td>
Krakatoa on a number of class file, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well." *Proebsting* § 5.

"Krakatoa is very efficient at reproducing readable Java source from Java bytecode." *Proebsting* § 6.

Dyer discloses that the output of a decompiler (such as *Proebsting*'s Krakatoa decompiler) may be an instruction that "request[s] the static initialization of the array," as recited in the claim:

"Mocha transformed a static initializer into an elegant, but illegal, construction:

```
public ConsoleWindow(String string, int i1)
{
dead = false;
styles = { "Plain", "Bold", "Italic" };
sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
...
```

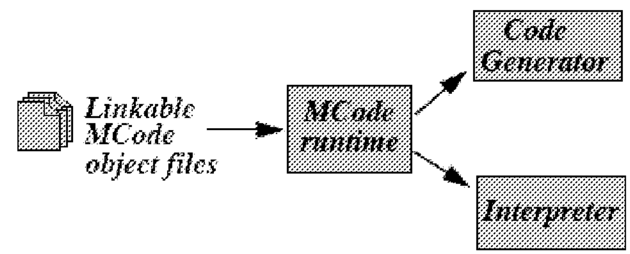
Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.

When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3.

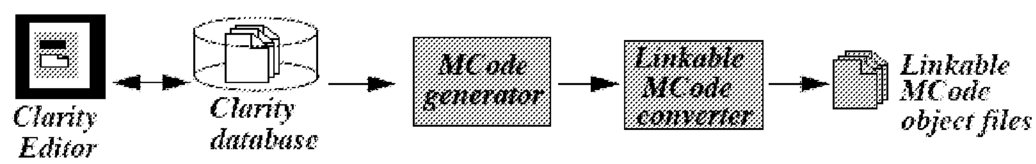
The fact that some of *Dyer*'s decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed -- a technique used to identify the operation of the byte code in order to generate
</td></tr>
</table>

U.S. Patent No. 6,061,520 – Claim 1	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	equivalent high-level language expressions. In fact, the '520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (*see* '520 patent at 8:54-9:13)) as an "illegal" Java constant pool entry. The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 . . . is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual machine to recognize an "illegal" instruction such as the new CONSTANT_Array type. Further, during Examination claim terms are to be given their broadest reasonable interpretation (*see* MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact format (whether syntactically correct or not) is not an issue. And in any case, *Dyer* discloses that -- in contrast to the Mocha decompiler's "illegal" code -- the WingDis decompiler produced "syntactically correct code." *Dyer* at 3. Additionally, the recitation of this method step in Claim 1 of the '520 patent is not limited with respect to the order in which it is performed relative to the other steps of Claim 1. Thus, the combined systems of *Lewis*, *Proebsting*, and *Dyer* may perform this step (i.e., "storing into an output file an instruction") while it generates a class file, i.e., after it performs the step of "compiling source code" and before the class file is received by the preloader. Or it may perform this step after the class file is received by the preloader. Either scenario would read on the claim limitations, which, under a broadest reasonable interpretation (MPEP § 2111), do not require a certain order.
interpreting the instruction by a virtual machine to perform the static initialization of the array.	*Lewis* discloses interpreting the instruction by a virtual machine, i.e., the "MCode interpreter," to perform the static initialization of the array. "While the MCode interpreter is mostly platform-independent, about 20% of its code is platform-specific. For example, in order to fully support procedure interposition and other

U.S. Patent No. 6,061,520 – Claim 1	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	ABI capabilities, the SPARC MCode interpreter does not directly interpret MCode ProcCall or Invoke instructions but instead implements them as SPARC ABI calls. Even MCode calls to other MCode procedures are implemented using SPARC instructions and execute the procedure's machine language entry code. This is necessary because the interpreter can never know whether a called procedure is actually implemented in MCode or in C. (For example, a programmer might have replaced the called procedure using interposition.) This means the interpreter must fully handle all the details required for ABI calls. If a called routine will return an aggregate value, the interpreter must generate a sequence of machine instructions at runtime (a *thunk*) to support the SPARC ABI's calling convention that the returned aggregate's length must be encoded into a SPARC UNIMP instruction just after the call. The interpreter also stores all program values in memory as SPARC values since this is required for ABI interoperation. Recently, a second MCode interpreter has been developed by Mick Jordan. This interpreter executes *system models* written in the Clarity language. These system models precisely describe how a software system is built: the exact versions of its component parts, all options and build parameters, and how the component parts are assembled. This system modeller is intended to replace the Unix *make* tool and to eliminate some of its problems: e.g. the inability to exactly reproduce the construction of a software system. The system modeller's MCode interpreter is specialized to executing these models and to interacting with the Clarity program database. It does not need, for example, to support SPARC ABI interoperation." *Lewis* at 127-28.

U.S. Patent No. 6,061,520 – Claim 1	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	 Figure 2: The runtime portion of the Clarity MCode system *Lewis* at 121.

U.S. Patent No. 6,061,520 – Claim 2	*Lewis* in view of *Dyer* and further in view of *Proebsting*
2. The method of claim 1 wherein the storing step includes step of: storing a constant pool entry into the constant pool.	*Lewis* discloses the method of claim 1 wherein the storing step includes step of: storing a constant pool entry into the constant pool. For example, *Lewis* discloses the entry of a constant as a "CGValue." "The second C++ base class, CGValue, describes values during compilation. The code generator "executes" MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously "executed" subexpressions, and procedure or method calls." *Lewis* at 126. "This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the

U.S. Patent No. 6,061,520 – Claim 2	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries."  Figure 1: The development-time portion of the Clarity MCode system *Lewis* at 120-21, Fig. 1. "Linkable MCode object files contain a machine-independent *pickle* of an MCode code unit. This is a compact, platform-independent encoding of the MCode information into a sequence of bytes. This pickle can later be internalized or unpickled to reconstruct the original MCode. The MCode for each procedure is pickled separately to support procedure-at-a-time processing. The current encoding is not especially compact although the Linkable MCode object files are still smaller than object files containing machine code. We intend to replace the current pickle format with a more compact one. Linkable MCode object files are platform-standard object files that are processed in the usual way by the platform's standard linker. This means they need to include platform-dependent definitions of global variables and procedures, and descriptions of referenced symbols. We currently encode ('mangle') symbol names in order to ensure that the resulting executables or shared libraries are type-safe with respect to the Clarity language. Eventually, this type-safety will be checked by a Clarity *prelinker*. Besides symbol definitions and references, our Solaris Linkable MCode object files also contain a few machine language instructions for each procedure's entry code. This entry code allows C code to call the MCode procedure. On the SPARC, this entry code consists primarily of a three instruction 'trampoline' that redirects the call to the appropriate target procedure

U.S. Patent No. 6,061,520 – Claim 2	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	chosen by the interpret/compile strategy module in the MCode runtime. The SPARC entry code also has three words used when atomically updating the trampoline. Despite this platform-specific information, the contents of a Linkable MCode file are mostly platform-independent. The Linkable MCode converter itself is also mostly platform-independent. We currently execute the Linkable MCode converter during program development, before a program is distributed. It could also be executed when the program is installed on a particular platform." *Lewis* at 125-26.

U.S. Patent No. 6,061,520 – Claim 3	***Lewis* in view of *Dyer* and further in view of *Proebsting***
3. The method of claim 1 wherein	*Lewis* discloses the method of claim 1. *See* claim 1 chart above.
the play executing step includes the steps of: allocating a stack;	*Lewis* discloses the play executing step includes the steps of: allocating a stack. For example, *Lewis* discloses the play execution of MCode instructions to create CGValue entries that represent that state of the entries on the stimulated stack. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Proebsting* to arrive at the claimed feature. Specifically, *Proebsting* discloses simulating the Java virtual machine stack: "Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." *Proebsting* § 1.

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 3</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td>“Symbolic execution of the bytecode creates the corresponding Java source expressions. . . . Symbolic execution simulates the Java Virtual Machine’s evaluation stack with strings that represent the source-level expressions being computed.” Proebsting § 2.

One of ordinary skill in the art would understand that in order to simulate a stack, the Proebsting system necessarily must have “allocat[ed] a stack,” as recited in this claim.</td></tr>
<tr><td>reading a byte code from the clinit method that manipulates the stack; and</td><td>Lewis discloses reading a byte code from the clinit method that manipulates the stack.

“The code generator ‘executes’ MCode instructions in order to maintain a running simulation of the MCode machine’s stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously ‘executed’ subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed.” Lewis at 126.

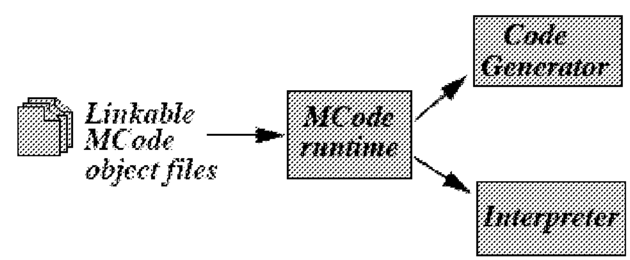


Figure 2: The runtime portion of the Clarity MCode system

Lewis at 121.</td></tr>
<tr><td>performing the stack manipulation on</td><td>Lewis discloses performing the stack manipulation on the allocated stack. For example,</td></tr>
</table>

U.S. Patent No. 6,061,520 – Claim 3	***Lewis* in view of *Dyer* and further in view of *Proebsting***

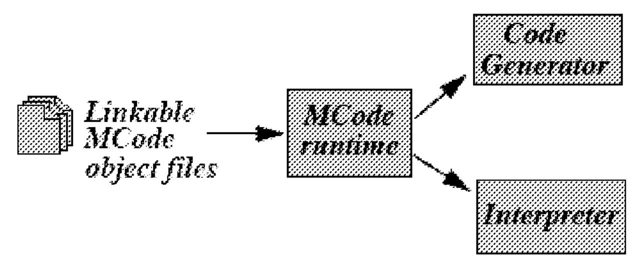
the allocated stack.	*Lewis* discloses the generation of the machine code, which inherently performs stack manipulation on the allocated stack.
	
	"The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126.
	
	To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, this claim element would have been obvious at the time of the invention to one of ordinary skill in the art from the teachings of *Lewis*, either by itself or in combination with other relevant prior art, including, but not limited to the *Dyer* and *Proebsting* references.
	
	Dyer is a review of various Java decompilers, and *Proebsting* is a specific example of a Java decompiler. These decompilers convert Java class files into Java source code.. Running the *Proebsting* decompiler on the Java code disclosed in *Dyer* would lead to the "[s]ymbolic execution [that] simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." *Proebsting* at § 2. *Dyer* expressly identifies static array initializers, which would necessarily (because of the semantic rules of the Java language) be compiled as byte code in a clinit method of the Java class file:
	
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	
	`public ConsoleWindow(String string, int i1)`
	`{`
	`dead = false;`
	`styles = { "Plain", "Bold", "Italic" };`

U.S. Patent No. 6,061,520 – Claim 3	***Lewis* in view of *Dyer* and further in view of *Proebsting***

	`sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };`
	`...`
	
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3.
	
	Thus, the combination of *Proebsting* and *Dyer* would result in "reading a byte code from the clinit method that manipulates the stack; and performing the stack manipulation on the allocated stack," as recited in Claim 3.

U.S. Patent No. 6,061,520 – Claim 4	***Lewis* in view of *Dyer* and further in view of *Proebsting***
4. The method of claim 1 wherein	*Lewis* discloses the method of claim 1. *See* claim 1 chart above.
the play executing step includes the steps of: allocating variables;	*Lewis* discloses the play executing step includes the steps of: allocating variables. For example, *Lewis* discloses the play execution of MCode instructions to create CGValue entries that represent that state of the entries on the stimulated stack. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126.

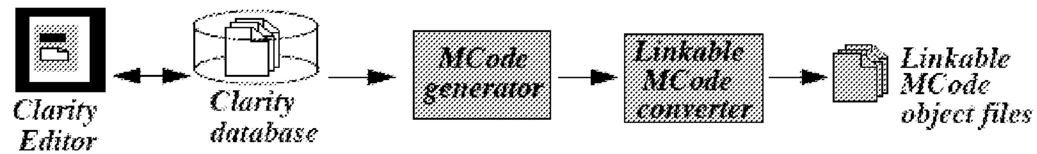
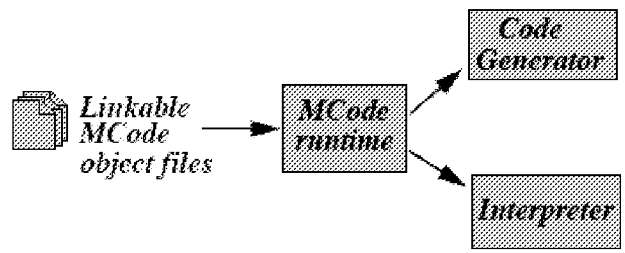
U.S. Patent No. 6,061,520 – Claim 4	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Proebsting* to arrive at the claimed feature. Specifically, *Proebsting* discloses play executing (disclosed as "symbolic execution") that includes allocating variables: "Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed. For instance, iload_1, *which loads the value of the first local variable*—with type int—could be represented on the stack as 'i1'." *Proebsting* § 2 (emphasis added). "For the JVM dup operators, which duplicate stack elements, Krakatoa simply creates a temporary variable to hold the duplicated value. . . . Krakatoa again uses a temporary variable to hold the result of each branch of the conditional expression, and then assigns this temporary value to the conditional expression." *Proebsting* § 5. In light of *Proebsting*'s use of local variables, one of ordinary skill in the art would understand that they must necessarily be allocated, as recited. With respect to the "dup" operator (quoted above), it is particularly noteworthy that the '520 patent discloses that this operation is inherently part of the code that is generated in the static array initialization bytecode. *See* '520 patent at 2:31; *see also* '520 patent at 5:25-43 (Code Table #4 disclosing that "dup" is one of the bytecodes that is recognized by the preloader). Since *Proebsting* was decompiling Java bytecode, it necessarily would have allocated a local variable when encountering the "dup" bytecode that is part of the static array initialization.
reading a byte code from the clinit method that manipulates local variables of the clinit method; and	*Lewis* discloses reading a byte code from the clinit method that manipulates local variables of the clinit method. For instance, *Lewis* discloses reading MCode instructions, which may include initialization instructions. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls.

U.S. Patent No. 6,061,520 – Claim 4	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed.” *Lewis* at 126.  Figure 2: The runtime portion of the Clarity MCode system *Lewis* at 121. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, *Dyer* discloses decompiling static array initialization statements. *See Dyer* at 3. The ‘520 patent is clear that the prior art Java compiler puts all static array initialization bytecode into the clinit method: “As stated above, the class file format cannot instruct the virtual machine to statically initialize arrays. To compensate for this problem, the Java compiler generates a special method, <clinit>, to perform class initialization, including initialization of static arrays.” ‘520 patent at 1:57-61. Accordingly, the decompilation of static array initialization statements, as disclosed in *Dyer*, necessarily reads those bytecodes from the clinit method, as claimed. Further, *Proebsting* discloses manipulating local variables of this code from the clinit method, for example,

U.S. Patent No. 6,061,520 – Claim 4	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	when simulating the "dup" bytecode: "For the JVM dup operators, which duplicate stack elements, Krakatoa simply creates a temporary variable to hold the duplicated value. . . . Krakatoa again uses a temporary variable to hold the result of each branch of the conditional expression, and then assigns this temporary value to the conditional expression." *Proebsting* § 5.
performing manipulation of the local variables on the allocated variables.	*Lewis* discloses performing manipulation of the local variables on the allocated variables. For example, *Lewis* discloses the manipulation of code to create CGValues. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, one of ordinary skill in the art would understand that in order to manipulate the local variables, as disclosed in *Proebsting*, such manipulation would necessarily occur on the allocated variables.

U.S. Patent No. 6,061,520 – Claim 6	***Lewis* in view of *Dyer* and further in view of *Proebsting***
6. A method in a data processing system, comprising the steps of:	*Lewis* discloses a method in a data processing system, e.g., the C++ language and certain dialects thereof, capable of performing data processing. "The *Clarity C++* programming language is a dialect of C++ being developed in Sun Microsystems Laboratories. Clarity shares many features with C++ but is less complex and has a more consistent syntax and simpler semantics without loss in expressiveness. . . .

U.S. Patent No. 6,061,520 – Claim 6	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	Clarity is intended to be a wide-spectrum language suitable for both systems and application programming, particularly of distributed software. To support the compilation of Clarity, we have developed a high-level, machine-independent intermediate representation that we call *MCode* (for "middle code"). We use MCode to compile Clarity programs at execution time (i.e., on-the-fly) into SPARC code for the Solaris operating system. This code generator is designed to be largely machine independent: besides the SPARC code generator, an Intel x86 version is being developed." *Lewis* at 119 (footnote omitted). "MCode has its basis in unpublished work done by L. Peter Deutsch at Sun Microsystems Laboratories in 1992-93. This work consisted of an implementation in Smalltalk of the core of a portable, on-the-fly compiler for a subset of the C language; we will refer to this system as 'CCore.'" *Lewis* at 120. *Dyer* discloses a review of three Java Decompilers. These are programs that convert Java class files into Java source code, effectively reverse engineering compiled code to figure out how the underlying code works. *Dyer* discloses, in most relevant part, an example of the code that would be used to implement a static initialization of an array. *Dyer* not only discloses the concept of the static initialization of an array by way of an instruction provided to the Java virtual machine; *Dyer* also provides examples of actual code to achieve this result. The decompiler of *Dyer* inherently performs a symbolic execution of the compiled bytecode. The functionality of such a decompiler, including the symbolic execution that decompilers use to identify the target of compiled bytecode for the recovery of the underlying Java source code, is disclosed in detail in *Proebsting*. The combination of the *Proebsting* disclosure, which explains the symbolic execution technique of a decompiler, with the actual results of a decompiler, as shown by *Dyer*, render obvious the broad symbolic execution of claim 6.
receiving code to be run on a processing	*Lewis* discloses receiving code, i.e., Linkable MCode object files, to be run on a processing

U.S. Patent No. 6,061,520 – Claim 6	*Lewis* in view of *Dyer* and further in view of *Proebsting*
component to perform an operation;	component to perform an operation.  Figure 1: The development-time portion of the Clarity MCode system  Figure 2: The runtime portion of the Clarity MCode system "This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries. The runtime component of the MCode system is illustrated in Figure 2. The MCode runtime

U.S. Patent No. 6,061,520 – Claim 6	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	in an MCode-containing executable internalizes the MCode for each procedure as needed, when the procedure is first called. It also implements a interpret/code generate policy separately for each MCode procedure. This policy chooses for each procedure whether to interpret it, generate code, or interpret then later generate code, or generate better code. The code generator currently produces SPARC code of approximately the quality of the SunPRO C compiler at the default -O2 optimization level. A port of the code generator to the x86 is underway. The MCode interpreter interoperates with all SPARC ABI code. Like the compiler, it is reentrant and supports multithreaded programs. It also does extensive checking during program execution, which makes it especially useful for uncovering errors in Clarity programs that are otherwise difficult to detect. The interpreter will also be used by the Clarity debugger that we are developing to evaluate Clarity statements and expressions." *Lewis* at 120-21. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Proebsting* to arrive at the claimed feature. Specifically, *Proebsting* discloses receiving Java byte code that was compiled to run on a processing component: "This paper presents our technique for automatically decompiling Java bytecode into Java source." *Proebsting* at Abstract. "*Decompilation* transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. . . . Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file. We present a technique for transforming low-level Java bytecode into legal Java source code." *Proebsting* § 1.

U.S. Patent No. 6,061,520 – Claim 6	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." *Proebsting* § 5.
play executing the code without running the code on the processing component to identify the operation if the code were run by the processing component; and	*Lewis* discloses play executing the code without running the code on the processing component to identify the operation if the code were run by the processing component. For example, *Lewis* discloses simulating execution of the MCode instructions in the MCode object files, and that the MCode instruction set includes instructions for statically initializing arrays. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously "executed" subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. The simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by *Proebsting* discusses "decompiling Java bytecode into Java source." *Proebsting* at Abstract. Specifically: "[s]ymbolic execution of the bytecode creates the corresponding Java source expressions." *Id.* at § 2. This decompilation method works because "[s]ymbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being compounded." *Id.* To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, *Dyer* and *Proebsting* together show that the play execution of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by *Proebsting* discloses a technique for decompilation of Java byte code that play executes the Java virtual machine's stack:

U.S. Patent No. 6,061,520 – Claim 6	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	"This paper presents our technique for automatically decompiling Java bytecode into Java source." *Proebsting* at Abstract. "*Decompilation* transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. . . . Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file. We present a technique for transforming low-level Java bytecode into legal Java source code." *Proebsting* § 1. "We have implemented a prototype Java decompiler, Krakatoa, in Java." *Proebsting* § 5. "Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." *Proebsting* § 1. "Symbolic execution of the bytecode creates the corresponding Java source expressions. . . . Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." *Proebsting* § 2. Thus it would have been obvious to one of ordinary skill in the art to combine the methods of *Lewis* with the play execution (symbolic execution) of bytecode against a memory, as disclosed by *Proebsting*. And, even more specifically, the artisan would look to known decompilers, such as those disclosed in *Dyer*. *Dyer* reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. *See Dyer* at 1. In that review, *Dyer* analyzes the decompilation of code, including the

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 6</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td>

decompilation of static initializers:

“Mocha transformed a static initializer into an elegant, but illegal, construction:

```
public ConsoleWindow(String string, int i1)
{
dead = false;
styles = { "Plain", "Bold", "Italic" };
sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
...
```

Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I’m sure Sun must have had a reason. In any case, it’s apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.

When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code.” *Dyer* at 3.

The fact that some of *Dyer*’s decompiled code may be considered “illegal” should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the ‘520 patent was filed -- a technique used to identify the operation of the byte code in order to generate equivalent high-level language expressions. In fact, the ‘520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (*see* ‘520 patent at 8:54-9:13)) as an “illegal” Java constant pool entry. The reason it wouldn’t have mattered is that the ‘520 patent states that “the virtual machine 222 . . . is modified to recognize the static initialization directives of the preloader.” ‘520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan’s skill set to perform the correct modification to allow the virtual

</td></tr>
</table>

U.S. Patent No. 6,061,520 – Claim 6	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	machine to recognize an "illegal" instruction such as the new CONSTANT_Array type. Further, during Examination claim terms are to be given their broadest reasonable interpretation (*see* MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact format (whether syntactically correct or not) is not an issue. And in any case, *Dyer* discloses that -- in contrast to the Mocha decompiler's "illegal" code -- the WingDis decompiler produced "syntactically correct code." *Dyer* at 3.
creating an instruction for the processing component to perform the operation.	*Lewis* discloses creating an instruction, *e.g.*, "MCode," for the processing component to perform the operation. "The object-oriented architecture of the code generator has significantly simplified our implementation. The MCode machine code generator is designed to be retargetable to a new machine architecture (especially a RISC machine) with relatively little effort. It defines two key C++ base classes that must be subclassed to port the code generator. The first class, CGMachine, represents a target machine for code generation and a code stream for that machine. The basic machine model is a generic, nonwindowed RISC processor. CGMachine subclasses may define variations such as windowed RISC and CISC. These subclasses implement virtual methods that describe the target machine's registers, data types, and instruction properties. CGMachine methods then use those descriptions to generate machine code from MCode. . . . . . . The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126.

U.S. Patent No. 6,061,520 – Claim 6	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	"In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." *Lewis* at 127. "We have described an intermediate representation MCode that is compact, easy to generate, and supports the on-the-fly generation of good quality machine code. Linkable MCode is an encoding of MCode in platform-standard object files that enables full interoperation with C and existing libraries, as well as the full use of all capabilities of standard linkers and other programming tools." *Lewis* at 128. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Proebsting* and *Dyer* to arrive at the claimed feature. Specifically, *Proebsting* discloses that the Krakatoa decompiler writes the source code (which would necessarily include creating any static array initialization instructions) to an output file: "We have implemented a prototype Java decompiler, Krakatoa, in Java. We have run Krakatoa on a number of class file, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well." *Proebsting* § 5. "Krakatoa is very efficient at reproducing readable Java source from Java bytecode." *Proebsting* § 6. *Dyer* discloses that the output of a decompiler (such as *Proebsting*'s Krakatoa decompiler) may be an instruction that "request[s] the static initialization of the array," as recited in the

U.S. Patent No. 6,061,520 – Claim 6	***Lewis* in view of *Dyer* and further in view of *Proebsting***

	claim:
	
	“Mocha transformed a static initializer into an elegant, but illegal, construction:
	
	`public ConsoleWindow(String string, int i1)`
	`{`
	`dead = false;`
	`styles = { "Plain", "Bold", "Italic" };`
	`sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };`
	`...`
	
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I’m sure Sun must have had a reason. In any case, it’s apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code.” *Dyer* at 3.
	
	The fact that some of *Dyer*’s decompiled code may be considered “illegal” should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the ‘520 patent was filed -- a technique used to identify the operation of the byte code in order to generate equivalent high-level language expressions. In fact, the ‘520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (*see* ‘520 patent at 8:54-9:13)) as an “illegal” Java constant pool entry. The reason it wouldn’t have mattered is that the ‘520 patent states that “the virtual machine 222 . . . is modified to recognize the static initialization directives of the preloader.” ‘520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan’s skill set to perform the correct modification to allow the virtual

U.S. Patent No. 6,061,520 – Claim 6	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	machine to recognize an "illegal" instruction such as the new CONSTANT_Array type. Further, during Examination claim terms are to be given their broadest reasonable interpretation (*see* MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact format (whether syntactically correct or not) is not an issue. And in any case, *Dyer* discloses that -- in contrast to the Mocha decompiler's "illegal" code -- the WingDis decompiler produced "syntactically correct code." *Dyer* at 3. Additionally, the recitation of this method step in Claim 1 of the '520 patent is not limited with respect to the order in which it is performed relative to the other steps of Claim 1. Thus, the combined systems of *Lewis*, *Proebsting*, and *Dyer* may perform this step (i.e., "creating an instruction") after it performs the step of "receiving code" and before the step of "reading the byte code." Or it may perform this step after "reading the byte code." Either scenario would read on the claim limitations, which, under a broadest reasonable interpretation (MPEP § 2111), do not require a certain order.

U.S. Patent No. 6,061,520 – Claim 7	***Lewis* in view of *Dyer* and further in view of *Proebsting***
7. The method of claim 6 wherein the operation initializes a data structure, and	*Lewis* discloses the method of claim 6 wherein the operation initializes a data structure. The MCode includes at least instructions and data structures, as shown below. "Although MCode includes instructions and data structures needed to implement some Clarity language-specific constructs such as its exceptions and method calls, the core of MCode is suitable for representing code for C and many other languages." *Lewis* at 119.

U.S. Patent No. 6,061,520 – Claim 7	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	AllocArray(array type index) ArrayIndex(array type index) ArrayLength(array type index) InvokeOuter(method type index) InvokeDelegated(method type index) Widen(base interface number) Narrow(target type index) BeginSwitch(end swi EndSwitch(end switc BeginExprCase(int) BeginDefaultCase() EndCase(end switch DoBreak(end tag) DoContinue(end tag) Figure 4: The MCode instruction set *Lewis* at 123 Fig. 4 (excerpt). "Figures 5 and 6 give an example of generated MCode. The Clarity method *startup* in Figure 5 produces the MCode instructions shown in part in Figure 6."

U.S. Patent No. 6,061,520 – Claim 7	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	(code figure below)

```
ThreadedSimulation: module
{
    work_mutex: Threads::Mutex;
    work_per_worker: int;                // protected by work_mutex
    extra_work: int;                     // protected by work_mutex

    Worker: type = interface inherits Threads::Thread {};
        // an unusual interface: no methods beyond Thread::startup and the other Thread methods

    WorkImpl: type = implementation of Worker
    {
        implement startup: method (our_workers: in int)
            {   // executed when the thread is started; delegates most of its work to forked sibling workers
                within work_mutex {     // acquire work_mutex for duration of the within statement
                    our_work = work_per_worker;
                    if (extra_work > 0) {our_work += 1; extra_work -= 1;}
                }
                delegated_to_workers: int = (our_workers - 1);
                if (delegated_to_workers > 0) {
                    left_workers = delegated_to_workers/2;
                    right_workers = delegated_to_workers - left_workers;
                    if (left_workers > 0) left_sibling = new WorkerImpl(left_workers);
                    right_sibling = new WorkerImpl(right_workers);
                }
                do_work(our_work);
            };

        // the following declarations are private to the WorkerImpl implementation
        do_work: method (work_to_do: in int) { /* elided */ };

        our_work: int = 0;
        left_workers: int = 0;
        right_workers: int = 0;
        left_sibling: Worker;            // left delegatee; manages "left_workers" workers
        right_sibling: Worker;           // right delegatee; manages "right_workers" workers
    };
    ...
};
```

Figure 5: Part of the Clarity version of the μC++ test program

U.S. Patent No. 6,061,520 – Claim 7	*Lewis* in view of *Dyer* and further in view of *Proebsting*

```
startup: method (our_workers: in int)
{     // executed when the thread is started; delegates most of its work to forked sibling worklers
      within work_mutex {      // acquire work_mutex for duration of the within statement
                  LoadGlobal(0)                    work_mutex
                  InvokeOuter(0x00010004)          Thread::enter
            our_work = work_per_worker;
                  LoadGlobal(1)                    work_per_worker
                  StoreGlobal(5)                   our_work
            if (extra_work > 0) {...
                  LoadGlobal(3)                    extra_work
                  LoadInt(0, 0)                    integer constant 0, type 0
                  CompareInt(>)                    extra_work>0
                  SkipThen(cond_false_tag_2)       if(extra_work>0){...
               our_work += 1;
                  GlobalAddr(5)                    our_work
                  Dup()
                  LoadIndirect(0)
                  LoadInt(0, 1)                    integer constant 1, type 0
                  AddInt()                         our_work+=1
                  StoreIndirect(0)                 our_work
               extra_work -= 1;
                  GlobalAddr(3)                    extra_work
                  Dup()
                  LoadIndirect(0)
                  LoadSigned(0, 1)                 integer constant 1, type 0
                  SubInt()                         extra_work-=1
                  StoreIndirect(0)                 extra_work
                  SkipElse(end_if_tag_3)           if(extra_work>0){...
                  BeginElse(cond_false_tag_2)
                  EndIf(end_if_tag_3)
      }
```

U.S. Patent No. 6,061,520 – Claim 7	*Lewis* in view of *Dyer* and further in view of *Proebsting*

```
            LoadGlobal(0)                 work_mutex
            InvokeOuter(0x00010005)       Thread::exit
    delegated_to_workers: int = (our_workers - 1);
            LoadArg(0)                    our_workers
            LoadInt(0, 1)                 integer constant 1, type 0
            SubInt()                      our_workers-1
            StoreLocal(0)                 delegated_to_workers
    ...
    do_work(our_work);
            LoadGlobal(4)                 do_work
            LoadGlobal(5)                 our_work
            ProcCall(7)                   do_work(our_work)
};
            ProcReturn(1)                 method(our_workers:...
```

Figure 6: MCode instructions generated for the µC++ test program's startup method

Lewis at 124-125, Figs. 5-6.

To the extent *Lewis* does not explicitly disclose this limitation of Claim 7, one of ordinary skill in the art would combine the disclosure of *Lewis* with *Dyer* to arrive at the claimed feature. Specifically, *Dyer* shows that an instruction for initializing a data structure (e.g., an array) was widely known in the art, as evidenced by its discussion of static array initialization:

"Mocha transformed a static initializer into an elegant, but illegal, construction:

```
public ConsoleWindow(String string, int i1)
{
dead = false;
styles = { "Plain", "Bold", "Italic" };
sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
...
```

Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these

U.S. Patent No. 6,061,520 – Claim 7	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	initializers are actually implemented by inline code inside constructors, generated by the compiler. When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3.
wherein the play executing step includes the step of: play executing the code to identify the initialization of the data structure.	*Lewis* discloses wherein the play executing step includes the step of: play executing the code to identify the initialization of the data structure. For example, *Lewis* discloses simulating execution of the MCode instructions in the MCode object files, and that the MCode instruction set includes instructions for statically initializing data structures. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. "In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." *Lewis* at 127. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, *Dyer* and *Proebsting* together show that the play execution of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly

U.S. Patent No. 6,061,520 – Claim 7	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	known to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by *Proebsting* discloses a technique for decompilation of Java byte code that play executes the Java virtual machine's stack: "This paper presents our technique for automatically decompiling Java bytecode into Java source." *Proebsting* at Abstract. "*Decompilation* transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. . . . Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file. We present a technique for transforming low-level Java bytecode into legal Java source code." *Proebsting* § 1. "We have implemented a prototype Java decompiler, Krakatoa, in Java." *Proebsting* § 5. "Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." *Proebsting* § 1. "Symbolic execution of the bytecode creates the corresponding Java source expressions. . . . Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." *Proebsting* § 2. Thus it would have been obvious to one of ordinary skill in the art to combine the methods of *Lewis* with the play execution (symbolic execution) of bytecode against a memory, as disclosed by *Proebsting*. And, even more specifically, the artisan would look to known

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 7</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td>decompilers, such as those disclosed in Dyer. Dyer reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. See Dyer at 1. In that review, Dyer analyzes the decompilation of code, including the decompilation of static initializers:

"Mocha transformed a static initializer into an elegant, but illegal, construction:

<pre>public ConsoleWindow(String string, int i1)
{
dead = false;
styles = { "Plain", "Bold", "Italic" };
sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
...</pre>
Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.

When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." Dyer at 3.</td></tr>
</table>

U.S. Patent No. 6,061,520 – Claim 8	*Lewis* in view of *Dyer* and further in view of *Proebsting*
8. The method of claim 6 wherein the operation statically initializes an array and	*Lewis* discloses the method of claim 6 wherein the operation statically initializes an array. For example, *Lewis* discloses that the MCode instruction set includes instructions for statically initializing an array, e.g., "AllocArray." "Although MCode includes instructions and data structures needed to implement some Clarity language-specific constructs such as its exceptions and method calls, the core of MCode is suitable for representing code for C and many other languages." *Lewis* at 119.

U.S. Patent No. 6,061,520 – Claim 8	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	AllocArray(array type index) ArrayIndex(array type index) ArrayLength(array type index) InvokeOuter(method type index) InvokeDelegated(method type index) Widen(base interface number) Narrow(target type index) BeginSwitch(end swi EndSwitch(end switc BeginExprCase(int) BeginDefaultCase() EndCase(end switch DoBreak(end tag) DoContinue(end tag) Figure 4: The MCode instruction set *Lewis* at 123 Fig. 4 (excerpt). "Figures 5 and 6 give an example of generated MCode. The Clarity method *startup* in Figure 5 produces the MCode instructions shown in part in Figure 6."

U.S. Patent No. 6,061,520 – Claim 8	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	See code and Figure 5 below

```
ThreadedSimulation: module
{
    work_mutex: Threads::Mutex;
    work_per_worker: int;              // protected by work_mutex
    extra_work: int;                   // protected by work_mutex

    Worker: type = interface inherits Threads::Thread {};
        // an unusual interface: no methods beyond Thread::startup and the other Thread methods

    WorkImpl: type = implementation of Worker
    {
        implement startup: method (our_workers: in int)
            {   // executed when the thread is started; delegates most of its work to forked sibling workers
                within work_mutex {     // acquire work_mutex for duration of the within statement
                    our_work = work_per_worker;
                    if (extra_work > 0) {our_work += 1; extra_work -= 1;}
                }
                delegated_to_workers: int = (our_workers - 1);
                if (delegated_to_workers > 0) {
                    left_workers = delegated_to_workers/2;
                    right_workers = delegated_to_workers - left_workers;
                    if (left_workers > 0) left_sibling = new WorkerImpl(left_workers);
                    right_sibling = new WorkerImpl(right_workers);
                }
                do_work(our_work);
            };

        // the following declarations are private to the WorkerImpl implementation
        do_work: method (work_to_do: in int) { /* elided */ };

        our_work: int = 0;
        left_workers: int = 0;
        right_workers: int = 0;
        left_sibling: Worker;          // left delegatee; manages "left_workers" workers
        right_sibling: Worker;         // right delegatee; manages "right_workers" workers
    };
    ...
};
```

Figure 5: Part of the Clarity version of the μC++ test program

U.S. Patent No. 6,061,520 – Claim 8	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	(code listing below)

```
startup: method (our_workers: in int)
{     // executed when the thread is started; delegates most of its work to forked sibling worklers
      within work_mutex {     // acquire work_mutex for duration of the within statement
                  LoadGlobal(0)                   work_mutex
                  InvokeOuter(0x00010004)         Thread::enter
            our_work = work_per_worker;
                  LoadGlobal(1)                   work_per_worker
                  StoreGlobal(5)                  our_work
            if (extra_work > 0) {...
                  LoadGlobal(3)                   extra_work
                  LoadInt(0, 0)                   integer constant 0, type 0
                  CompareInt(>)                   extra_work>0
                  SkipThen(cond_false_tag_2)      if(extra_work>0){...
                our_work += 1;
                  GlobalAddr(5)                   our_work
                  Dup()
                  LoadIndirect(0)
                  LoadInt(0, 1)                   integer constant 1, type 0
                  AddInt()                        our_work+=1
                  StoreIndirect(0)                our_work
                extra_work -= 1;
                  GlobalAddr(3)                   extra_work
                  Dup()
                  LoadIndirect(0)
                  LoadSigned(0, 1)                integer constant 1, type 0
                  SubInt()                        extra_work-=1
                  StoreIndirect(0)                extra_work
                  SkipElse(end_if_tag_3)          if(extra_work>0){...
                  BeginElse(cond_false_tag_2)
                  EndIf(end_if_tag_3)
      }
```

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 8</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td>
<pre>
        LoadGlobal(0)   work_mutex
        InvokeOuter(0x00010005)   Thread::exit
  delegated_to_workers: int = (our_workers - 1);
        LoadArg(0)   our_workers
        LoadInt(0, 1)   integer constant 1, type 0
        SubInt()   our_workers-1
        StoreLocal(0)   delegated_to_workers
  ...
  do_work(our_work);
        LoadGlobal(4)   do_work
        LoadGlobal(5)   our_work
        ProcCall(7)   do_work(our_work)
};
        ProcReturn(1)   method(our_workers:...
</pre>
Figure 6: MCode instructions generated for the µC++ test program's startup method

Lewis at 124-125, Figs. 5-6.

To the extent Lewis does not explicitly disclose this limitation of Claim 8, one of ordinary skill in the art would combine the disclosure of Lewis with Dyer to arrive at the claimed feature. Specifically, Dyer shows that an instruction for initializing a data structure (e.g., an array) was widely known in the art, as evidenced by its discussion of static array initialization:

“Mocha transformed a static initializer into an elegant, but illegal, construction:
<pre>
public ConsoleWindow(String string, int i1)
{
dead = false;
styles = { "Plain", "Bold", "Italic" };
sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
...
</pre>
Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I’m sure Sun must have had a reason. In any case, it’s apparent that these
</td></tr>
</table>

U.S. Patent No. 6,061,520 – Claim 8	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	initializers are actually implemented by inline code inside constructors, generated by the compiler. When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3.
wherein the play executing step includes the step of: play executing the code to identify the static initialization of the array.	*Lewis* discloses wherein the play executing step includes the step of: play executing the code to identify the static initialization of the array. For example, *Lewis* discloses that the MCode instruction set includes instructions for statically initializing an array, e.g., "AllocArray." "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. "In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." *Lewis* at 127. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, *Dyer* and *Proebsting* together show that the play execution of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly

U.S. Patent No. 6,061,520 – Claim 8	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	known to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by *Proebsting* discloses a technique for decompilation of Java byte code that play executes the Java virtual machine's stack: "This paper presents our technique for automatically decompiling Java bytecode into Java source." *Proebsting* at Abstract. "*Decompilation* transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. . . . Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file. We present a technique for transforming low-level Java bytecode into legal Java source code." *Proebsting* § 1. "We have implemented a prototype Java decompiler, Krakatoa, in Java." *Proebsting* § 5. "Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." *Proebsting* § 1. "Symbolic execution of the bytecode creates the corresponding Java source expressions. . . . Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." *Proebsting* § 2. Thus it would have been obvious to one of ordinary skill in the art to combine the methods of *Lewis* with the play execution (symbolic execution) of bytecode against a memory, as disclosed by *Proebsting*. And, even more specifically, the artisan would look to known

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 8</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td>decompilers, such as those disclosed in Dyer. Dyer reviews three popular decompilers that were available before the ‘520 patent was filed: DejaVu, Mocha, and WingDis. See Dyer at 1. In that review, Dyer analyzes the decompilation of code, including the decompilation of static initializers:

“Mocha transformed a static initializer into an elegant, but illegal, construction:

<code>public ConsoleWindow(String string, int i1)
{
dead = false;
styles = { "Plain", "Bold", "Italic" };
sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
...</code>

Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I’m sure Sun must have had a reason. In any case, it’s apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.

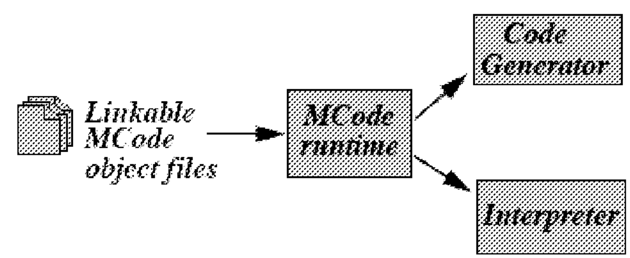
When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code.” Dyer at 3.</td></tr>
</table>

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 9</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td>9. The method of claim 6 further including the step of: running the created instruction on the processing component to perform the operation.</td><td>Lewis discloses the method of claim 6 further including the step of: running the created instruction on the processing component to perform the operation. For example, Lewis discloses that “object files containing MCode . . . are processed by standard linkers or other tools.”

“Runtime generation of machine code offers many advantages. A runtime code generator can take advantage of information about the particular target platform to generate better</td></tr>
</table>

U.S. Patent No. 6,061,520 – Claim 9	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	code. For example, different implementations of the SPARC architecture have different instruction pipeline properties. In the case of one new SPARC implementation, code generated specifically for this new processor can run up to 25% faster than code generated for a 'generic' SPARC processor. A runtime code generator can also take advantage of the specific values used in a program to generate machine code customized for those values. One of our goals is to explore the use of on-the-fly code generation for systems programming within Sun. Our representation of MCode object files is unusual in that we use platform-standard object files instead of some Clarity- or MCode-specific representation. This enables us to fully interoperate with existing C and SPARC ABI code. Object files containing MCode (which we call *Linkable MCode* files) are processed by standard linkers and other tools in the same way as other object files. As an example, our Solaris SPARC implementation supports complete interoperation with all SPARC Application Binary Interface (ABI) compliant code [SPARC ABI]. In particular, interpreted or compiled MCode programs can call C programs and vice versa, addresses of MCode procedures can be passed to C code and later called, and all C data types can be exchanged." *Lewis* at 119.

U.S. Patent No. 6,061,520 – Claim 10	***Lewis* in view of *Dyer* and further in view of *Proebsting***
10. The method of claim 6 further including the step of: interpreting the created instruction by a virtual machine to perform the operation.	*Lewis* discloses the computer-readable medium of claim 18 further including the step of: interpreting the created instruction by a virtual machine to perform the operation. For example, *Lewis* discloses that "MCode calls to other MCode procedures are implemented using SPARC instructions and execute the procedure's machine language entry code." "While the MCode interpreter is mostly platform-independent, about 20% of its code is platform-specific. For example, in order to fully support procedure interposition and other ABI capabilities, the SPARC MCode interpreter does not directly interpret MCode ProcCall or Invoke instructions but instead implements them as SPARC ABI calls. Even MCode calls to other MCode procedures are implemented using SPARC instructions and execute the procedure's machine language entry code. This is necessary because the interpreter can

U.S. Patent No. 6,061,520 – Claim 10	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	never know whether a called procedure is actually implemented in MCode or in C. (For example, a programmer might have replaced the called procedure using interposition.) This means the interpreter must fully handle all the details required for ABI calls. If a called routine will return an aggregate value, the interpreter must generate a sequence of machine instructions at runtime (a *thunk*) to support the SPARC ABI's calling convention that the returned aggregate's length must be encoded into a SPARC UNIMP instruction just after the call. The interpreter also stores all program values in memory as SPARC values since this is required for ABI interoperation. Recently, a second MCode interpreter has been developed by Mick Jordan. This interpreter executes *system models* written in the Clarity language. These system models precisely describe how a software system is built: the exact versions of its component parts, all options and build parameters, and how the component parts are assembled. This system modeller is intended to replace the Unix *make* tool and to eliminate some of its problems: e.g. the inability to exactly reproduce the construction of a software system. The system modeller's MCode interpreter is specialized to executing these models and to interacting with the Clarity program database. It does not need, for example, to support SPARC ABI interoperation." *Lewis* at 127-28.  Figure 2: The runtime portion of the Clarity MCode system *Lewis* at 121.

U.S. Patent No. 6,061,520 – Claim 11	***Lewis* in view of *Dyer* and further in view of *Proebsting***
11. The method of claim 6 wherein the operation has an effect on memory, and wherein the play executing step includes the step of: play executing the code to identify the effect on the memory.	*Lewis* discloses method of claim 6 wherein the operation has an effect on memory, and wherein the play executing step includes the step of: play executing the code to identify the effect on the memory. For example, *Lewis* discloses that the concrete subclasses of CGValue represent the state of the individual entries on the simulated stack, a process which inherently identifies the effect of the play execution on the memory. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, *Dyer* and *Proebsting* together show that the play execution of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention, and that it was used to identify the effect of an operation memory. For example, the decompiler technology disclosed by *Proebsting* discloses a technique for decompilation of Java byte code that play executes the Java virtual machine's stack: "This paper presents our technique for automatically decompiling Java bytecode into Java source." *Proebsting* at Abstract. "*Decompilation* transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type

U.S. Patent No. 6,061,520 – Claim 11	*Lewis* in view of *Dyer* and further in view of *Proebsting*

	information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. . . . Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.
	
	We present a technique for transforming low-level Java bytecode into legal Java source code." *Proebsting* § 1.
	
	"We have implemented a prototype Java decompiler, Krakatoa, in Java." *Proebsting* § 5.
	
	"Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." *Proebsting* § 1.
	
	"Symbolic execution of the bytecode creates the corresponding Java source expressions. . . . Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." *Proebsting* § 2.
	
	Thus it would have been obvious to one of ordinary skill in the art to combine the methods of *Lewis* with the play execution (symbolic execution) of bytecode against a memory, as disclosed by *Proebsting*. This would be for the purpose of "identify[ing] the [code's] effect on memory," as recited in the claim. Further, the artisan would look to known decompilers, such as those disclosed in *Dyer*. *Dyer* reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. *See Dyer* at 1. In that review, *Dyer* analyzes the decompilation of code, including the decompilation of static initializers:
	
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	
	`public ConsoleWindow(String string, int i1)`
	`{`
	`dead = false;`
	`styles = { "Plain", "Bold", "Italic" };`

U.S. Patent No. 6,061,520 – Claim 11	***Lewis* in view of *Dyer* and further in view of *Proebsting***

	`sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };`
	`...`
	
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3.

U.S. Patent No. 6,061,520 – Claim 12	***Lewis* in view of *Dyer* and further in view of *Proebsting***
12. A data processing system comprising:	*Lewis* discloses a data processing system, e.g., the C++ computer programming language, capable of data processing. "The *Clarity C++* programming language is a dialect of C++ being developed in Sun Microsystems Laboratories. Clarity shares many features with C++ but is less complex and has a more consistent syntax and simpler semantics without loss in expressiveness. . . . Clarity is intended to be a wide-spectrum language suitable for both systems and application programming, particularly of distributed software. To support the compilation of Clarity, we have developed a high-level, machine-independent intermediate representation that we call *MCode* (for "middle code"). We use MCode to compile Clarity programs at execution time (i.e., on-the-fly) into SPARC code for the Solaris operating system. This code generator is designed to be largely machine independent: besides the SPARC code generator, an Intel x86 version is being developed." *Lewis* at 119 (footnote omitted). "MCode has its basis in unpublished work done by L. Peter Deutsch at Sun Microsystems Laboratories in 1992-93. This work consisted of an implementation in Smalltalk of the core

U.S. Patent No. 6,061,520 – Claim 12	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	of a portable, on-the-fly compiler for a subset of the C language; we will refer to this system as 'CCore.'" *Lewis* at 120.
a storage device containing:	*Lewis* discloses a storage device. The disclosure of *Lewis* pertains to a computer system with memory stacks, which are inherently storage devices. "Runtime generation of machine code offers many advantages. A runtime code generator can take advantage of information about the particular target platform to generate better code. For example, different implementations of the SPARC architecture have different instruction pipeline properties. In the case of one new SPARC implementation, code generated specifically for this new processor can run up to 25% faster than code generated for a 'generic' SPARC processor. A runtime code generator can also take advantage of the specific values used in a program to generate machine code customized for those values. One of our goals is to explore the use of on-the-fly code generation for systems programming within Sun. Our representation of MCode object files is unusual in that we use platform-standard object files instead of some Clarity- or MCode-specific representation. This enables us to fully interoperate with existing C and SPARC ABI code. Object files containing MCode (which we call *Linkable MCode* files) are processed by standard linkers and other tools in the same way as other object files. As an example, our Solaris SPARC implementation supports complete interoperation with all SPARC Application Binary Interface (ABI) compliant code [SPARC ABI]. In particular, interpreted or compiled MCode programs can call C programs and vice versa, addresses of MCode procedures can be passed to C code and later called, and all C data types can be exchanged." *Lewis* at 119.
a program with source code that statically initializes a data structure; and	*Lewis* discloses a program with source code that statically initializes a data structure. For example, *Lewis* discloses MCode which may contain instruction to initialize a data structure. "Although MCode includes instructions and data structures needed to implement some Clarity language-specific constructs such as its exceptions and method calls, the core of MCode is suitable for representing code for C and many other languages." *Lewis* at 119.

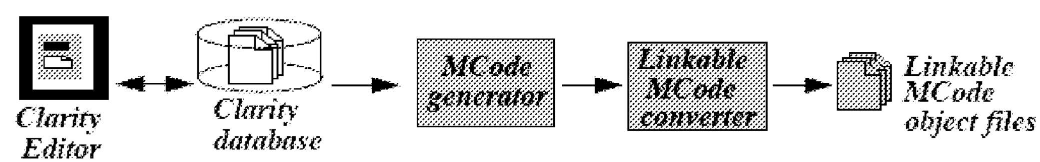
U.S. Patent No. 6,061,520 – Claim 12	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	"Figures 5 and 6 give an example of generated MCode. The Clarity method *startup* in Figure 5 produces the MCode instructions shown in part in Figure 6."

U.S. Patent No. 6,061,520 – Claim 12	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	(code figure below)

```
ThreadedSimulation: module
{
    work_mutex: Threads::Mutex;
    work_per_worker: int;                    // protected by work_mutex
    extra_work: int;                         // protected by work_mutex

    Worker: type = interface inherits Threads::Thread {};
        // an unusual interface: no methods beyond Thread::startup and the other Thread methods

    WorkImpl: type = implementation of Worker
    {
        implement startup: method (our_workers: in int)
            {   // executed when the thread is started; delegates most of its work to forked sibling workers
                within work_mutex {     // acquire work_mutex for duration of the within statement
                    our_work = work_per_worker;
                    if (extra_work > 0) {our_work += 1; extra_work -= 1;}
                }
                delegated_to_workers: int = (our_workers - 1);
                if (delegated_to_workers > 0) {
                    left_workers = delegated_to_workers/2;
                    right_workers = delegated_to_workers - left_workers;
                    if (left_workers > 0) left_sibling = new WorkerImpl(left_workers);
                    right_sibling = new WorkerImpl(right_workers);
                }
                do_work(our_work);
            };

        // the following declarations are private to the WorkerImpl implementation
        do_work: method (work_to_do: in int) { /* elided */ };

        our_work: int = 0;
        left_workers: int = 0;
        right_workers: int = 0;
        left_sibling: Worker;                // left delegatee; manages "left_workers" workers
        right_sibling: Worker;               // right delegatee; manages "right_workers" workers
    };
    ...
};
```

Figure 5: Part of the Clarity version of the μC++ test program

Lewis at 124, Fig. 5.

U.S. Patent No. 6,061,520 – Claim 12	*Lewis* in view of *Dyer* and further in view of *Proebsting*
class files, wherein one of the class files contains a clinit method that statically initializes the data structure;	*Lewis* discloses class files, wherein one of the class files contains a clinit method that statically initializes the data structure. "This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries."  Figure 1: The development-time portion of the Clarity MCode system *Lewis* at 120-21, Fig. 1. "Linkable MCode object files contain a machine-independent *pickle* of an MCode code unit. This is a compact, platform-independent encoding of the MCode information into a sequence of bytes. This pickle can later be internalized or unpickled to reconstruct the original MCode. The MCode for each procedure is pickled separately to support procedure-at-a-time processing. The current encoding is not especially compact although the Linkable MCode object files are still smaller than object files containing machine code. We intend to replace the current pickle format with a more compact one. Linkable MCode object files are platform-standard object files that are processed in the usual way by the platform's standard linker. This means they need to include platform-dependent definitions of global variables and procedures, and descriptions of referenced symbols. We

U.S. Patent No. 6,061,520 – Claim 12	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	currently encode ('mangle') symbol names in order to ensure that the resulting executables or shared libraries are type-safe with respect to the Clarity language. Eventually, this type-safety will be checked by a Clarity *prelinker*. Besides symbol definitions and references, our Solaris Linkable MCode object files also contain a few machine language instructions for each procedure's entry code. This entry code allows C code to call the MCode procedure. On the SPARC, this entry code consists primarily of a three instruction 'trampoline' that redirects the call to the appropriate target procedure chosen by the interpret/compile strategy module in the MCode runtime. The SPARC entry code also has three words used when atomically updating the trampoline. Despite this platform-specific information, the contents of a Linkable MCode file are mostly platform-independent. The Linkable MCode converter itself is also mostly platform-independent. We currently execute the Linkable MCode converter during program development, before a program is distributed. It could also be executed when the program is installed on a particular platform." *Lewis* at 125-26.

U.S. Patent No. 6,061,520 – Claim 12	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	(code listing below)

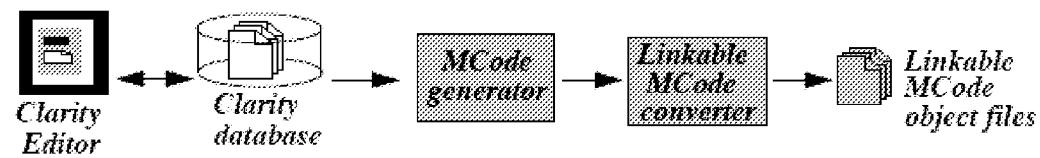
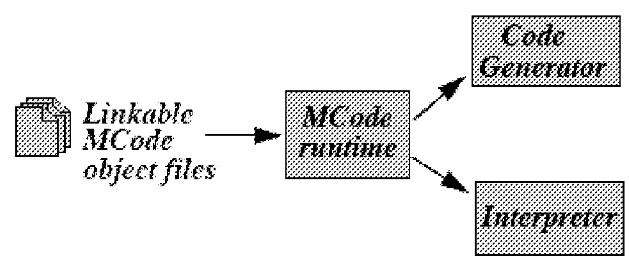
```
startup: method (our_workers: in int)
{     // executed when the thread is started; delegates most of its work to forked sibling worklers
      within work_mutex {      // acquire work_mutex for duration of the within statement
                  LoadGlobal(0)                     work_mutex
                  InvokeOuter(0x00010004)           Thread::enter
            our_work = work_per_worker;
                  LoadGlobal(1)                     work_per_worker
                  StoreGlobal(5)                    our_work
            if (extra_work > 0) {...
                  LoadGlobal(3)                     extra_work
                  LoadInt(0, 0)                     integer constant 0, type 0
                  CompareInt(>)                     extra_work>0
                  SkipThen(cond_false_tag_2)        if(extra_work>0){...
                our_work += 1;
                  GlobalAddr(5)                     our_work
                  Dup()
                  LoadIndirect(0)
                  LoadInt(0, 1)                     integer constant 1, type 0
                  AddInt()                          our_work+=1
                  StoreIndirect(0)                  our_work
                extra_work -= 1;
                  GlobalAddr(3)                     extra_work
                  Dup()
                  LoadIndirect(0)
                  LoadSigned(0, 1)                  integer constant 1, type 0
                  SubInt()                          extra_work-=1
                  StoreIndirect(0)                  extra_work
                  SkipElse(end_if_tag_3)            if(extra_work>0){...
                  BeginElse(cond_false_tag_2)
                  EndIf(end_if_tag_3)
      }
```

U.S. Patent No. 6,061,520 – Claim 12	*Lewis* in view of *Dyer* and further in view of *Proebsting*

	LoadGlobal(0) work_mutex
	InvokeOuter(0x00010005) Thread::exit
	delegated_to_workers: int = (our_workers - 1);
	LoadArg(0) our_workers
	LoadInt(0, 1) integer constant 1, type 0
	SubInt() our_workers-1
	StoreLocal(0) delegated_to_workers
	...
	do_work(our_work);
	LoadGlobal(4) do_work
	LoadGlobal(5) our_work
	ProcCall(7) do_work(our_work)
	};
	ProcReturn(1) method(our_workers:...
	Figure 6: MCode instructions generated for the µC++ test program's startup method
	
	Lewis at 125, Fig. 6.
	
	To the extent *Lewis* does not explicitly disclose this limitation of Claim 12, one of ordinary skill in the art would combine the disclosure of *Lewis* with *Dyer* to arrive at the claimed feature. Specifically, *Dyer* discloses decompiling static array initialization statements. *See Dyer* at 3. The '520 patent is clear that the prior art Java compiler puts all static array initialization bytecode into the clinit method:
	
	"As stated above, the class file format cannot instruct the virtual machine to statically initialize arrays. To compensate for this problem, the Java compiler generates a special method, <clinit>, to perform class initialization, including initialization of static arrays." '520 patent at 1:57-61.
	
	Accordingly, for the decompilation of static array initialization statements, as disclosed in *Dyer*, the bytecode is necessarily in the clinit method, as claimed.

a memory containing:	*Lewis* discloses a memory. "Runtime generation of machine code offers many advantages. A runtime code generator can take advantage of information about the particular target platform to generate better code.

U.S. Patent No. 6,061,520 – Claim 12	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	For example, different implementations of the SPARC architecture have different instruction pipeline properties. In the case of one new SPARC implementation, code generated specifically for this new processor can run up to 25% faster than code generated for a 'generic' SPARC processor. A runtime code generator can also take advantage of the specific values used in a program to generate machine code customized for those values. One of our goals is to explore the use of on-the-fly code generation for systems programming within Sun. Our representation of MCode object files is unusual in that we use platform-standard object files instead of some Clarity- or MCode-specific representation. This enables us to fully interoperate with existing C and SPARC ABI code. Object files containing MCode (which we call *Linkable MCode* files) are processed by standard linkers and other tools in the same way as other object files. As an example, our Solaris SPARC implementation supports complete interoperation with all SPARC Application Binary Interface (ABI) compliant code [SPARC ABI]. In particular, interpreted or compiled MCode programs can call C programs and vice versa, addresses of MCode procedures can be passed to C code and later called, and all C data types can be exchanged." *Lewis* at 119. To the extent *Lewis* does not explicitly disclose this limitation of Claim 12, one of ordinary skill in the art would combine the disclosure of *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, all of these disclosed software systems necessarily disclose a memory.
a compiler for compiling the program and generating the class files; and	*Lewis* discloses a compiler as a part of the "MCode compilation system" for compiling the program and generating the class files. "This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The

U.S. Patent No. 6,061,520 – Claim 12	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries."  Figure 1: The development-time portion of the Clarity MCode system *Lewis* at 120-21, Fig. 1.
a preloader for consolidating the class files, for play executing the clinit method to determine the static initialization the clinit method performs, and for creating an instruction to perform the static initialization; and	*Lewis* discloses a preloader for consolidating the class files, for play executing the clinit method to determine the static initialization the clinit method performs, and for creating an instruction to perform the static initialization. For instance, *Lewis* discloses reading MCode instructions, which may include initialization instructions.  Figure 2: The runtime portion of the Clarity MCode system "The runtime component of the MCode system is illustrated in Figure 2. The MCode runtime in an MCode-containing executable internalizes the MCode for each procedure as needed, when the procedure is first called. It also implements a interpret/code generate policy separately for each MCode procedure. This policy chooses for each procedure whether to

U.S. Patent No. 6,061,520 – Claim 12	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	interpret it, generate code, or interpret then later generate code, or generate better code. The code generator currently produces SPARC code of approximately the quality of the SunPRO C compiler at the default -O2 optimization level. A port of the code generator to the x86 is underway. The MCode interpreter interoperates with all SPARC ABI code. Like the compiler, it is reentrant and supports multithreaded programs. It also does extensive checking during program execution, which makes it especially useful for uncovering errors in Clarity programs that are otherwise difficult to detect. The interpreter will also be used by the Clarity debugger that we are developing to evaluate Clarity statements and expressions." *Lewis* at 120-21. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, *Dyer* and *Proebsting* together show that the play execution of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by *Proebsting* discloses a technique for decompilation of Java byte code that play executes the Java virtual machine's stack: "This paper presents our technique for automatically decompiling Java bytecode into Java source." *Proebsting* at Abstract. "*Decompilation* transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine.

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 12</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td>This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a class file that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. . . . Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.

We present a technique for transforming low-level Java bytecode into legal Java source code.” Proebsting § 1.

“We have implemented a prototype Java decompiler, Krakatoa, in Java.” Proebsting § 5.

“Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations.” Proebsting § 1.

“Symbolic execution of the bytecode creates the corresponding Java source expressions. . . . Symbolic execution simulates the Java Virtual Machine’s evaluation stack with strings that represent the source-level expressions being computed.” Proebsting § 2.

Thus it would have been obvious to one of ordinary skill in the art to combine the methods of Lewis with the play execution (symbolic execution) of bytecode against a memory, as disclosed by Proebsting. And, even more specifically, the artisan would look to known decompilers, such as those disclosed in Dyer. Dyer reviews three popular decompilers that were available before the ‘520 patent was filed: DejaVu, Mocha, and WingDis. See Dyer at 1. In that review, Dyer analyzes the decompilation of code, including the decompilation of static initializers:

“Mocha transformed a static initializer into an elegant, but illegal, construction:

<code>public ConsoleWindow(String string, int i1)</code>
<code>{</code>
<code>dead = false;</code></td></tr>
</table>

U.S. Patent No. 6,061,520 – Claim 12	*Lewis* in view of *Dyer* and further in view of *Proebsting*

	`styles = { "Plain", "Bold", "Italic" };`
	`sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };`
	`...`
	
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3.
	
	Further, *Proebsting* discloses that the Krakatoa decompiler writes the source code (which would necessarily include creating any static array initialization instructions) to an output file:
	
	"We have implemented a prototype Java decompiler, Krakatoa, in Java. We have run Krakatoa on a number of class file, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well." *Proebsting* § 5.
	
	"Krakatoa is very efficient at reproducing readable Java source from Java bytecode." *Proebsting* § 6.
	
	Additionally, *Dyer* discloses that the Java decompiler (such as *Proebsting*'s Krakatoa decompiler) creates an instruction that "perform[s] the static initialization," as recited in the claim. *See Dyer* at 3.
	
	The fact that some of *Dyer*'s decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed --

U.S. Patent No. 6,061,520 – Claim 12	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	a technique used to identify the operation of the byte code in order to generate equivalent high-level language expressions. In fact, the '520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (*see* '520 patent at 8:54-9:13)) as an "illegal" Java constant pool entry. The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 . . . is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual machine to recognize an "illegal" instruction such as the new CONSTANT_Array type. Further, during Examination claim terms are to be given their broadest reasonable interpretation (*see* MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact format (whether syntactically correct or not) is not an issue. And in any case, *Dyer* discloses that -- in contrast to the Mocha decompiler's "illegal" code -- the WingDis decompiler produced "syntactically correct code." *Dyer* at 3.
a processor for running the compiler and the preloader.	*Lewis* discloses a processor for running the compiler and the preloader. *Lewis* discloses the implementation of code that can be generated specifically for certain processors. "Runtime generation of machine code offers many advantages. A runtime code generator can take advantage of information about the particular target platform to generate better code. For example, different implementations of the SPARC architecture have different instruction pipeline properties. In the case of one new SPARC implementation, code generated specifically for this new processor can run up to 25% faster than code generated for a 'generic' SPARC processor. A runtime code generator can also take advantage of the specific values used in a program to generate machine code customized for those values. One of our goals is to explore the use of on-the-fly code generation for systems programming within Sun.

U.S. Patent No. 6,061,520 – Claim 12	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	Our representation of MCode object files is unusual in that we use platform-standard object files instead of some Clarity- or MCode-specific representation. This enables us to fully interoperate with existing C and SPARC ABI code. Object files containing MCode (which we call *Linkable MCode* files) are processed by standard linkers and other tools in the same way as other object files. As an example, our Solaris SPARC implementation supports complete interoperation with all SPARC Application Binary Interface (ABI) compliant code [SPARC ABI]. In particular, interpreted or compiled MCode programs can call C programs and vice versa, addresses of MCode procedures can be passed to C code and later called, and all C data types can be exchanged." *Lewis* at 119. To the extent *Lewis* does not explicitly disclose this limitation of Claim 12, one of ordinary skill in the art would combine the disclosure of *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, all of these disclosed software systems necessarily require a processor to run the disclosed software.

U.S. Patent No. 6,061,520 – Claim 13	***Lewis* in view of *Dyer* and further in view of *Proebsting***
13. The data processing system of claim 12 wherein the preloader includes a mechanism for generating an output file containing the created instruction.	*Lewis* discloses the data processing system of claim 12 wherein the preloader includes a mechanism for generating an output file containing the created instruction, i.e., a code generator that generates good code for SPARC. "The code generator includes a peephole optimizer, does dead code elimination, and generates "leaf procedure" calls on the SPARC. However, little further optimization is done at this time; our immediate concern is generating correct code. Despite this, the code generator generates good code for the SPARC." *Lewis* at 126. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Proebsting* and *Dyer* to arrive at the claimed feature. Specifically, *Proebsting* discloses that the Krakatoa decompiler writes the source code (which would necessarily include any static array initialization instructions) to an output file:

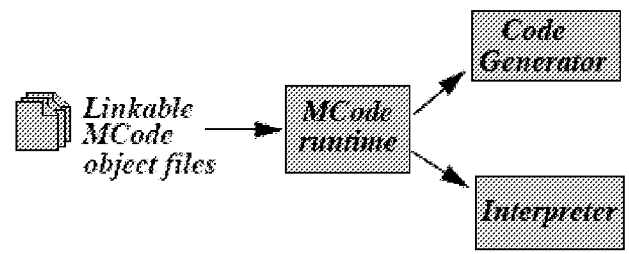
U.S. Patent No. 6,061,520 – Claim 13	***Lewis* in view of *Dyer* and further in view of *Proebsting***

	"We have implemented a prototype Java decompiler, Krakatoa, in Java. We have run Krakatoa on a number of class file, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well." *Proebsting* § 5.
	
	"Krakatoa is very efficient at reproducing readable Java source from Java bytecode." *Proebsting* § 6.
	
	Dyer discloses that the output of a decompiler (such as *Proebsting*'s Krakatoa decompiler) may be an instruction that "request[s] the static initialization of the array," as recited in the claim:
	
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	
	`public ConsoleWindow(String string, int i1)`
	`{`
	`dead = false;`
	`styles = { "Plain", "Bold", "Italic" };`
	`sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };`
	`...`
	
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3.
	
	The fact that some of *Dyer*'s decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was

U.S. Patent No. 6,061,520 – Claim 13	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	filed -- a technique used to identify the operation of the byte code in order to generate equivalent high-level language expressions. In fact, the '520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (*see* '520 patent at 8:54-9:13)) as an "illegal" Java constant pool entry. The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 . . . is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual machine to recognize an "illegal" instruction such as the new CONSTANT_Array type. Further, during Examination claim terms are to be given their broadest reasonable interpretation (*see* MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact format (whether syntactically correct or not) is not an issue. And in any case, *Dyer* discloses that -- in contrast to the Mocha decompiler's "illegal" code -- the WingDis decompiler produced "syntactically correct code." *Dyer* at 3.

U.S. Patent No. 6,061,520 – Claim 14	***Lewis* in view of *Dyer* and further in view of *Proebsting***
14. The data processing system of claim 13 wherein the memory further includes a virtual machine that interprets the created instruction to perform the static initialization.	*Lewis* discloses the data processing system of claim 13 wherein the memory further includes a virtual machine that interprets the created instruction to perform the static initialization. For example, *Lewis* discloses the implementation of an "interpreter that is mostly platform-independent," *i.e.*, a virtual machine. *Lewis* at 127. The interpreter of *Lewis* must be platform independent, *i.e.*, virtual, because "the interpreter can never know whether a called procedure is actually implemented in MCode or in C." *Lewis* at 127-128. This is the same virtual machine functionality described in the '520 patent. For example, the '520 patent describes a "platform-independent code [that] runs on a Java™ virtual machine, which is an abstract computing machine that interprets the platform-independent code." '520 patent at 1:11-14. This virtual machine then interprets the created instruction: "MCode calls to other MCode procedures are implemented using SPARC instructions and execute the

U.S. Patent No. 6,061,520 – Claim 14	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	procedure's machine language entry code." *Lewis* at 127. "While the MCode interpreter is mostly platform-independent, about 20% of its code is platform-specific. For example, in order to fully support procedure interposition and other ABI capabilities, the SPARC MCode interpreter does not directly interpret MCode ProcCall or Invoke instructions but instead implements them as SPARC ABI calls. Even MCode calls to other MCode procedures are implemented using SPARC instructions and execute the procedure's machine language entry code. This is necessary because the interpreter can never know whether a called procedure is actually implemented in MCode or in C. (For example, a programmer might have replaced the called procedure using interposition.) This means the interpreter must fully handle all the details required for ABI calls. If a called routine will return an aggregate value, the interpreter must generate a sequence of machine instructions at runtime (a *thunk*) to support the SPARC ABI's calling convention that the returned aggregate's length must be encoded into a SPARC UNIMP instruction just after the call. The interpreter also stores all program values in memory as SPARC values since this is required for ABI interoperation. Recently, a second MCode interpreter has been developed by Mick Jordan. This interpreter executes *system models* written in the Clarity language. These system models precisely describe how a software system is built: the exact versions of its component parts, all options and build parameters, and how the component parts are assembled. This system modeller is intended to replace the Unix *make* tool and to eliminate some of its problems: e.g. the inability to exactly reproduce the construction of a software system. The system modeller's MCode interpreter is specialized to executing these models and to interacting with the Clarity program database. It does not need, for example, to support SPARC ABI interoperation." *Lewis* at 127-28.

U.S. Patent No. 6,061,520 – Claim 14	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	 Figure 2: The runtime portion of the Clarity MCode system *Lewis* at 121.

U.S. Patent No. 6,061,520 – Claim 15	***Lewis* in view of *Dyer* and further in view of *Proebsting***
15. The data processing system of claim 12, wherein the data structure is an array.	*Lewis* discloses the data processing system of claim 12, wherein the data structure is an array. For example, *Lewis* discloses that the MCode instruction set includes instructions for statically initializing an array, e.g., "AllocArray." AllocArray(array type index) BeginSwitch(end swi ArrayIndex(array type index) EndSwitch(end switc ArrayLength(array type index) BeginExprCase(int) BeginDefaultCase() InvokeOuter(method type index) EndCase(end switch InvokeDelegated(method type index) Widen(base interface number) DoBreak(end tag) Narrow(target type index) DoContinue(end tag) Figure 4: The MCode instruction set *Lewis* at 123 Fig. 4 (excerpt).

U.S. Patent No. 6,061,520 – Claim 15	***Lewis* in view of *Dyer* and further in view of *Proebsting***

	To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* to arrive at the claimed feature. Specifically, *Dyer* discloses that the statically initialized data structure is an array:
	
	“Mocha transformed a static initializer into an elegant, but illegal, construction:
	
	`public ConsoleWindow(String string, int i1)`
	`{`
	`dead = false;`
	`styles = { "Plain", "Bold", "Italic" };`
	`sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };`
	`...`
	
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I’m sure Sun must have had a reason. In any case, it’s apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code.” *Dyer* at 3.

U.S. Patent No. 6,061,520 – Claim 16	***Lewis* in view of *Dyer* and further in view of *Proebsting***
16. The data processing system of claim 12 wherein the clinit method has byte codes that statically initialize the data structure.	*Lewis* discloses the data processing system of claim 12 wherein the clinit method has byte codes that statically initialize the data structure. For example, *Lewis* discloses that the MCode instruction set includes instructions for statically initializing an array, e.g., “AllocArray.”

U.S. Patent No. 6,061,520 – Claim 16	***Lewis* in view of *Dyer* and further in view of *Proebsting***

```
AllocArray(array type index)                  BeginSwitch(end swi
ArrayIndex(array type index)                  EndSwitch(end switc
ArrayLength(array type index)                 BeginExprCase(int)
                                              BeginDefaultCase()
InvokeOuter(method type index)                EndCase(end switch
InvokeDelegated(method type index)
Widen(base interface number)                  DoBreak(end tag)
Narrow(target type index)                     DoContinue(end tag)
```

Figure 4: The MCode instruction set

Lewis at 123 Fig. 4 (excerpt).

To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* to arrive at the claimed feature. Specifically, *Dyer* discloses that the statically initialized data structure is an array:

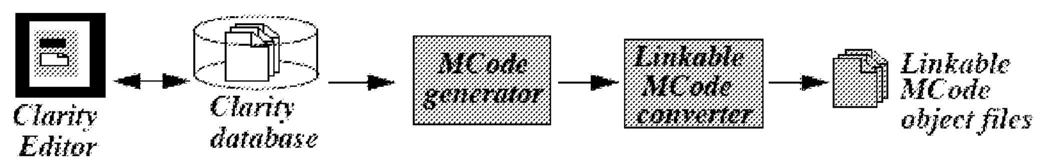
"Mocha transformed a static initializer into an elegant, but illegal, construction:

```
public ConsoleWindow(String string, int i1)
{
dead = false;
styles = { "Plain", "Bold", "Italic" };
sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
...
```

Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.

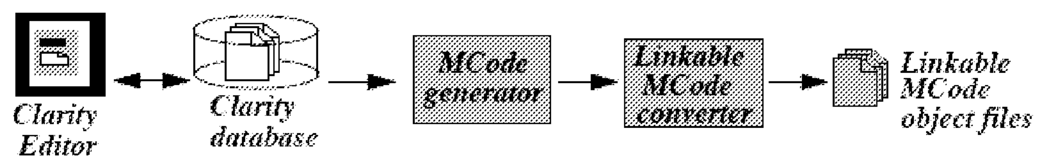
U.S. Patent No. 6,061,520 – Claim 16	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3. Additionally, the '520 patent is clear that the prior art Java compiler puts all static array initialization bytecode into the clinit method: "As stated above, the class file format cannot instruct the virtual machine to statically initialize arrays. To compensate for this problem, the Java compiler generates a special method, <clinit>, to perform class initialization, including initialization of static arrays." '520 patent at 1:57-61. Accordingly, for the decompilation of static array initialization statements, as disclosed in *Dyer*, the bytecode is necessarily in the clinit method, as claimed.

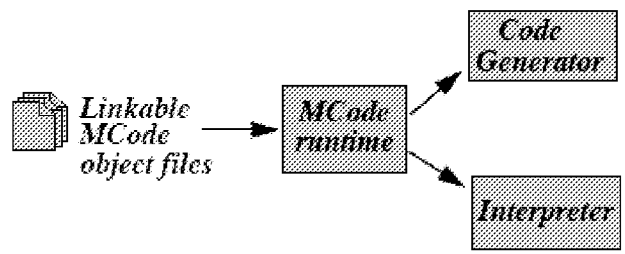
U.S. Patent No. 6,061,520 – Claim 17	*Lewis* in view of *Dyer* and further in view of *Proebsting*
17. The data processing system of claim 12 wherein the created instruction includes an entry into a constant pool.	*Lewis* discloses the data processing system of claim 12, wherein the created instruction includes an entry into a constant pool. For example, *Lewis* discloses the entry of a constant as a "CGValue." "The second C++ base class, CGValue, describes values during compilation. The code generator "executes" MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously "executed" subexpressions, and procedure or method calls." *Lewis* at 126. Compare this to the '520 patent, where "the Java virtual machine 222 is an otherwise standard Java virtual machine," *see* '520 patent at 7:48-49, and wherein the "virtual machines recognize various constant pool entries, such as CONSTANT_Integer, CONSTANT_String, and CONSTANT Long," *see* '520 patent at 7:51-54. The purportedly inventive step involves inserting a "CONSTANT_Array entry in the constant pool." '520

U.S. Patent No. 6,061,520 – Claim 17	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	patent at 7:57-58. Here, *Lewis* implements a CGValue, which represents a simulation of the machine's stack. *Lewis* at 126. This can include an array: "MCode's types currently include integer, real, pointer, array, procedure, bit field, struct, union, interface, implementation, and void." *Lewis* at 122. This is directly analogous to the disclosure of the '520 patent, where a representation of the array is entered into the constant pool as a CONSTANT_Array entry. "This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries."  Figure 1: The development-time portion of the Clarity MCode system *Lewis* at 120-21, Fig. 1. "Linkable MCode object files contain a machine-independent *pickle* of an MCode code unit. This is a compact, platform-independent encoding of the MCode information into a sequence of bytes. This pickle can later be internalized or unpickled to reconstruct the original MCode. The MCode for each procedure is pickled separately to support procedure-at-a-time processing. The current encoding is not especially compact although the

U.S. Patent No. 6,061,520 – Claim 17	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	Linkable MCode object files are still smaller than object files containing machine code. We intend to replace the current pickle format with a more compact one. Linkable MCode object files are platform-standard object files that are processed in the usual way by the platform's standard linker. This means they need to include platform-dependent definitions of global variables and procedures, and descriptions of referenced symbols. We currently encode ('mangle') symbol names in order to ensure that the resulting executables or shared libraries are type-safe with respect to the Clarity language. Eventually, this type-safety will be checked by a Clarity *prelinker*. Besides symbol definitions and references, our Solaris Linkable MCode object files also contain a few machine language instructions for each procedure's entry code. This entry code allows C code to call the MCode procedure. On the SPARC, this entry code consists primarily of a three instruction 'trampoline' that redirects the call to the appropriate target procedure chosen by the interpret/compile strategy module in the MCode runtime. The SPARC entry code also has three words used when atomically updating the trampoline. Despite this platform-specific information, the contents of a Linkable MCode file are mostly platform-independent. The Linkable MCode converter itself is also mostly platform-independent. We currently execute the Linkable MCode converter during program development, before a program is distributed. It could also be executed when the program is installed on a particular platform." *Lewis* at 125-26.

U.S. Patent No. 6,061,520 – Claim 18	***Lewis* in view of *Dyer* and further in view of *Proebsting***
18. A computer-readable medium containing instructions for controlling a data processing system to perform a method, comprising the steps of:	*Lewis* discloses a computer-readable medium containing instructions for controlling a data processing system to perform a method, e.g., the C++ language and certain dialects thereof, capable of performing data processing and performing a method. "The *Clarity C++* programming language is a dialect of C++ being developed in Sun Microsystems Laboratories. Clarity shares many features with C++ but is less complex and has a more consistent syntax and simpler semantics without loss in expressiveness. . . . Clarity is intended to be a wide-spectrum language suitable for both systems and application

U.S. Patent No. 6,061,520 – Claim 18	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	programming, particularly of distributed software. To support the compilation of Clarity, we have developed a high-level, machine-independent intermediate representation that we call *MCode* (for "middle code"). We use MCode to compile Clarity programs at execution time (i.e., on-the-fly) into SPARC code for the Solaris operating system. This code generator is designed to be largely machine independent: besides the SPARC code generator, an Intel x86 version is being developed." *Lewis* at 119 (footnote omitted). "MCode has its basis in unpublished work done by L. Peter Deutsch at Sun Microsystems Laboratories in 1992-93. This work consisted of an implementation in Smalltalk of the core of a portable, on-the-fly compiler for a subset of the C language; we will refer to this system as 'CCore.'" *Lewis* at 120.
receiving code to be run on a processing component to perform an operation;	*Lewis* discloses receiving code to be run on a processing component to perform an operation. See the excerpts below and the discussion above as to claim 1.  Figure 1: The development-time portion of the Clarity MCode system

U.S. Patent No. 6,061,520 – Claim 18	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	 Figure 2: The runtime portion of the Clarity MCode system "This subsection presents an overview of the Clarity MCode compilation system; more details are given in Section 3. There are two major parts to the MCode compilation system: a program development-time part and a runtime part. The development-time portion consists of an MCode generator for the Clarity language and a Linkable MCode converter; see Figure 1. The MCode generator reads semantically decorated Clarity ASTs stored in the Clarity programming environment's database and produces platform-independent MCode. The Linkable MCode converter then wraps a compact encoding of the MCode into a standard object file. Linkable MCode object files are then combined by standard linkers with other object files to produce executables and shared libraries. The runtime component of the MCode system is illustrated in Figure 2. The MCode runtime in an MCode-containing executable internalizes the MCode for each procedure as needed, when the procedure is first called. It also implements a interpret/code generate policy separately for each MCode procedure. This policy chooses for each procedure whether to interpret it, generate code, or interpret then later generate code, or generate better code. The code generator currently produces SPARC code of approximately the quality of the SunPRO C compiler at the default -O2 optimization level. A port of the code generator to the x86 is underway. The MCode interpreter interoperates with all SPARC ABI code. Like the compiler, it is reentrant and supports multithreaded programs. It also does extensive checking during program execution, which makes it especially useful for uncovering errors

U.S. Patent No. 6,061,520 – Claim 18	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	in Clarity programs that are otherwise difficult to detect. The interpreter will also be used by the Clarity debugger that we are developing to evaluate Clarity statements and expressions." *Lewis* at 120-21.
simulating execution of the code without running the code on the processing component to identify the operation if the code were run by the processing component; and	*Lewis* discloses simulating execution of the code without running the code on the processing component to identify the operation if the code were run by the processing component. See the excerpts below and the discussion above as to claim 1. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously "executed" subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, *Dyer* and *Proebsting* together show that the simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by *Proebsting* discloses a technique for decompilation of Java byte code that simulates the Java virtual machine's stack: "This paper presents our technique for automatically decompiling Java bytecode into Java source." *Proebsting* at Abstract. "*Decompilation* transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type information and bytecode. The JVM requires a significant amount of type information

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 18</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td>from the class files for object linking. . . . Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file.

We present a technique for transforming low-level Java bytecode into legal Java source code.” Proebsting § 1.

“We have implemented a prototype Java decompiler, Krakatoa, in Java.” Proebsting § 5.

“Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations.” Proebsting § 1.

“Symbolic execution of the bytecode creates the corresponding Java source expressions. . . . Symbolic execution simulates the Java Virtual Machine’s evaluation stack with strings that represent the source-level expressions being computed.” Proebsting § 2.

Thus it would have been obvious to one of ordinary skill in the art to combine the methods of Lewis with the simulation (symbolic execution) of bytecode against a memory, as disclosed by Proebsting. And, even more specifically, the artisan would look to known decompilers, such as those disclosed in Dyer. Dyer reviews three popular decompilers that were available before the ‘520 patent was filed: DejaVu, Mocha, and WingDis. See Dyer at 1. In that review, Dyer analyzes the decompilation of code, including the decompilation of static initializers:

“Mocha transformed a static initializer into an elegant, but illegal, construction:

<code>public ConsoleWindow(String string, int i1)</code>
<code>{</code>
<code>dead = false;</code>
<code>styles = { "Plain", "Bold", "Italic" };</code>
<code>sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };</code>
<code>...</code></td></tr>
</table>

U.S. Patent No. 6,061,520 – Claim 18	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler. When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3. The fact that some of *Dyer*'s decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed -- a technique used to identify the operation of the byte code in order to generate equivalent high-level language expressions. In fact, the '520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (*see* '520 patent at 8:54-9:13)) as an "illegal" Java constant pool entry. The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 . . . is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual machine to recognize an "illegal" instruction such as the new CONSTANT_Array type. Further, during Examination claim terms are to be given their broadest reasonable interpretation (*see* MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact format (whether syntactically correct or not) is not an issue. And in any case, *Dyer* discloses that -- in contrast to the Mocha decompiler's "illegal" code -- the WingDis decompiler produced "syntactically correct code." *Dyer* at 3.
creating an instruction for the	*Lewis* discloses creating an instruction for the processing component to perform the

U.S. Patent No. 6,061,520 – Claim 18	***Lewis* in view of *Dyer* and further in view of *Proebsting***
processing component to perform the operation.	operation. See the excerpts below and the discussion above as to claim 1. "The object-oriented architecture of the code generator has significantly simplified our implementation. The MCode machine code generator is designed to be retargetable to a new machine architecture (especially a RISC machine) with relatively little effort. It defines two key C++ base classes that must be subclassed to port the code generator. The first class, CGMachine, represents a target machine for code generation and a code stream for that machine. The basic machine model is a generic, nonwindowed RISC processor. CGMachine subclasses may define variations such as windowed RISC and CISC. These subclasses implement virtual methods that describe the target machine's registers, data types, and instruction properties. CGMachine methods then use those descriptions to generate machine code from MCode. . . . . . . The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. "In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." *Lewis* at 127.

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 18</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td>“We have described an intermediate representation MCode that is compact, easy to generate, and supports the on-the-fly generation of good quality machine code. Linkable MCode is an encoding of MCode in platform-standard object files that enables full interoperation with C and existing libraries, as well as the full use of all capabilities of standard linkers and other programming tools.” Lewis at 128.

To the extent Lewis does not explicitly disclose this limitation, one of ordinary skill would combine Lewis with Proebsting and Dyer to arrive at the claimed feature. Specifically, Proebsting discloses that the Krakatoa decompiler writes the source code (which would necessarily include creating any static array initialization instructions) to an output file:

“We have implemented a prototype Java decompiler, Krakatoa, in Java. We have run Krakatoa on a number of class file, including some to which we had no source code access. We examined the output of Krakatoa by hand, and Krakatoa appears to recover high-level constructs very well.” Proebsting § 5.

“Krakatoa is very efficient at reproducing readable Java source from Java bytecode.” Proebsting § 6.

Dyer discloses that the output of a decompiler (such as Proebsting’s Krakatoa decompiler) may be an instruction that “request[s] the static initialization of the array,” as recited in the claim:

“Mocha transformed a static initializer into an elegant, but illegal, construction:

<pre>public ConsoleWindow(String string, int i1)
{
dead = false;
styles = { "Plain", "Bold", "Italic" };
sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
...</pre>
Bracketed initializer lists for arrays are valid only as initializers for variable declarations</td></tr>
</table>

U.S. Patent No. 6,061,520 – Claim 18	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	(either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler. When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3. The fact that some of *Dyer*'s decompiled code may be considered "illegal" should not detract from the fact that the decompilation of static array initialization byte code into a single expression or instruction was a well-known technique at the time the '520 patent was filed -- a technique used to identify the operation of the byte code in order to generate equivalent high-level language expressions. In fact, the '520 patent discloses creating a constant pool entry that is not a standard Java virtual machine construct. Thus, one of ordinary skill in the art would have considered this constant pool construct (i.e., CONSTANT_Array (*see* '520 patent at 8:54-9:13)) as an "illegal" Java constant pool entry. The reason it wouldn't have mattered is that the '520 patent states that "the virtual machine 222 . . . is modified to recognize the static initialization directives of the preloader." '520 patent at 4:46-48. Because the exact form of modification is not disclosed, it must be within the ordinary artisan's skill set to perform the correct modification to allow the virtual machine to recognize an "illegal" instruction such as the new CONSTANT_Array type. Further, during Examination claim terms are to be given their broadest reasonable interpretation (*see* MPEP § 2111), and there are no qualifiers in the claim language as to the generated instruction. Again, because the '520 patent assumes that one of ordinary skill in the art would modify the virtual machine to recognize the generated instruction, the exact format (whether syntactically correct or not) is not an issue. And in any case, *Dyer* discloses that -- in contrast to the Mocha decompiler's "illegal" code -- the WingDis decompiler produced "syntactically correct code." *Dyer* at 3. Additionally, the recitation of this method step in Claim 1 of the '520 patent is not limited with respect to the order in which it is performed relative to the other steps of Claim 1.

U.S. Patent No. 6,061,520 – Claim 18	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	Thus, the combined systems of *Lewis*, *Proebsting*, and *Dyer* may perform this step (i.e., "creating an instruction") after it performs the step of "receiving code" and before the step of "simulating execution." Or it may perform this step after "simulating execution." Either scenario would read on the claim limitations, which, under a broadest reasonable interpretation (MPEP § 2111), do not require a certain order.

U.S. Patent No. 6,061,520 – Claim 19	***Lewis* in view of *Dyer* and further in view of *Proebsting***
19. The computer-readable medium of claim 18 wherein the operation initializes a data structure, and	*Lewis* discloses the computer-readable medium of claim 18 wherein the operation initializes a data structure. See the excerpts below and the discussion above as to claim 1, discussing the initialization of an array, which inherently discloses the initialization of a data structure. "Although MCode includes instructions and data structures needed to implement some Clarity language-specific constructs such as its exceptions and method calls, the core of MCode is suitable for representing code for C and many other languages." *Lewis* at 119. AllocArray(array type index) BeginSwitch(end swi ArrayIndex(array type index) EndSwitch(end switc ArrayLength(array type index) BeginExprCase(int) BeginDefaultCase() InvokeOuter(method type index) EndCase(end switch InvokeDelegated(method type index) Widen(base interface number) DoBreak(end tag) Narrow(target type index) DoContinue(end tag) Figure 4: The MCode instruction set *Lewis* at 123 Fig. 4 (excerpt). "Figures 5 and 6 give an example of generated MCode. The Clarity method *startup* in Figure 5 produces the MCode instructions shown in part in Figure 6."

U.S. Patent No. 6,061,520 – Claim 19	*Lewis* in view of *Dyer* and further in view of *Proebsting*
	(code figure below)

```
ThreadedSimulation: module
{
    work_mutex: Threads::Mutex;
    work_per_worker: int;               // protected by work_mutex
    extra_work: int;                    // protected by work_mutex

    Worker: type = interface inherits Threads::Thread {};
        // an unusual interface: no methods beyond Thread::startup and the other Thread methods

    WorkImpl: type = implementation of Worker
    {
        implement startup: method (our_workers: in int)
            {   // executed when the thread is started; delegates most of its work to forked sibling workers
                within work_mutex {     // acquire work_mutex for duration of the within statement
                    our_work = work_per_worker;
                    if (extra_work > 0) {our_work += 1; extra_work -= 1;}
                }
                delegated_to_workers: int = (our_workers - 1);
                if (delegated_to_workers > 0) {
                    left_workers = delegated_to_workers/2;
                    right_workers = delegated_to_workers - left_workers;
                    if (left_workers > 0) left_sibling = new WorkerImpl(left_workers);
                    right_sibling = new WorkerImpl(right_workers);
                }
                do_work(our_work);
            };

        // the following declarations are private to the WorkerImpl implementation
        do_work: method (work_to_do: in int) { /* elided */ };

        our_work: int = 0;
        left_workers: int = 0;
        right_workers: int = 0;
        left_sibling: Worker;           // left delegatee; manages "left_workers" workers
        right_sibling: Worker;          // right delegatee; manages "right_workers" workers
    };
    ...
};
```

Figure 5: Part of the Clarity version of the μC++ test program

U.S. Patent No. 6,061,520 – Claim 19	*Lewis* in view of *Dyer* and further in view of *Proebsting*

```
startup: method (our_workers: in int)
{     // executed when the thread is started; delegates most of its work to forked sibling worklers
      within work_mutex {      // acquire work_mutex for duration of the within statement
                  LoadGlobal(0)                     work_mutex
                  invokeOuter(0x00010004)           Thread::enter
            our_work = work_per_worker;
                  LoadGlobal(1)                     work_per_worker
                  StoreGlobal(5)                    our_work
            if (extra_work > 0) {...
                  LoadGlobal(3)                     extra_work
                  LoadInt(0, 0)                     integer constant 0, type 0
                  CompareInt(>)                     extra_work>0
                  SkipThen(cond_false_tag_2)        if(extra_work>0){...
               our_work += 1;
                  GlobalAddr(5)                     our_work
                  Dup()
                  LoadIndirect(0)
                  LoadInt(0, 1)                     integer constant 1, type 0
                  AddInt()                          our_work+=1
                  StoreIndirect(0)                  our_work
               extra_work -= 1;
                  GlobalAddr(3)                     extra_work
                  Dup()
                  LoadIndirect(0)
                  LoadSigned(0, 1)                  integer constant 1, type 0
                  SubInt()                          extra_work-=1
                  StoreIndirect(0)                  extra_work
                  SkipElse(end_if_tag_3)            if(extra_work>0){...
                  BeginElse(cond_false_tag_2)
                  EndIf(end_if_tag_3)
      }
```

U.S. Patent No. 6,061,520 – Claim 19	***Lewis* in view of *Dyer* and further in view of *Proebsting***

```
                LoadGlobal(0)                   work_mutex
                InvokeOuter(0x00010005)         Thread::exit
    delegated_to_workers: int = (our_workers - 1);
                LoadArg(0)                      our_workers
                LoadInt(0, 1)                   integer constant 1, type 0
                SubInt()                        our_workers-1
                StoreLocal(0)                   delegated_to_workers
    ...
    do_work(our_work);
                LoadGlobal(4)                   do_work
                LoadGlobal(5)                   our_work
                ProcCall(7)                     do_work(our_work)
};
                ProcReturn(1)                   method(our_workers:...
```

Figure 6: MCode instructions generated for the μC++ test program's startup method

Lewis at 124-125, Figs. 5-6.

To the extent *Lewis* does not explicitly disclose this limitation of Claim 7, one of ordinary skill in the art would combine the disclosure of *Lewis* with *Dyer* to arrive at the claimed feature. Specifically, *Dyer* shows that an instruction for initializing a data structure (e.g., an array) was widely known in the art, as evidenced by its discussion of static array initialization:

"Mocha transformed a static initializer into an elegant, but illegal, construction:

```
public ConsoleWindow(String string, int i1)
{
dead = false;
styles = { "Plain", "Bold", "Italic" };
sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
...
```

Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these

U.S. Patent No. 6,061,520 – Claim 19	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	initializers are actually implemented by inline code inside constructors, generated by the compiler. When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3.
wherein the simulating step includes the step of: simulating execution of the code to identify the initialization of the data structure.	*Lewis* discloses wherein the simulating step includes the step of: simulating execution of the code to identify the initialization of the data structure. See the excerpts below and the discussion above as to claim 1, disclosing the simulated execution of an array, which inherently discloses the simulated execution of a data structure. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. "In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." *Lewis* at 127. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, *Dyer* and *Proebsting* together show that the simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known

U.S. Patent No. 6,061,520 – Claim 19	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by *Proebsting* discloses a technique for decompilation of Java byte code that simulates the Java virtual machine's stack: "This paper presents our technique for automatically decompiling Java bytecode into Java source." *Proebsting* at Abstract. "*Decompilation* transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. . . . Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file. We present a technique for transforming low-level Java bytecode into legal Java source code." *Proebsting* § 1. "We have implemented a prototype Java decompiler, Krakatoa, in Java." *Proebsting* § 5. "Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." *Proebsting* § 1. "Symbolic execution of the bytecode creates the corresponding Java source expressions. . . . Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." *Proebsting* § 2. Thus it would have been obvious to one of ordinary skill in the art to combine the methods of *Lewis* with the simulation (symbolic execution) of bytecode against a memory, as disclosed by *Proebsting*. And, even more specifically, the artisan would look to known decompilers,

U.S. Patent No. 6,061,520 – Claim 19	***Lewis* in view of *Dyer* and further in view of *Proebsting***

	such as those disclosed in *Dyer*. *Dyer* reviews three popular decompilers that were available before the ‘520 patent was filed: DejaVu, Mocha, and WingDis. *See Dyer* at 1. In that review, *Dyer* analyzes the decompilation of code, including the decompilation of static initializers:
	
	“Mocha transformed a static initializer into an elegant, but illegal, construction:
	
	`public ConsoleWindow(String string, int i1)`
	`{`
	`dead = false;`
	`styles = { "Plain", "Bold", "Italic" };`
	`sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };`
	`...`
	
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I’m sure Sun must have had a reason. In any case, it’s apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code.” *Dyer* at 3.

U.S. Patent No. 6,061,520 – Claim 20	***Lewis* in view of *Dyer* and further in view of *Proebsting***
20. The computer-readable medium of claim 18 wherein the operation statically initializes an array and	*Lewis* discloses the computer-readable medium of claim 18 wherein the operation statically initializes an array. “Although MCode includes instructions and data structures needed to implement some Clarity language-specific constructs such as its exceptions and method calls, the core of MCode is suitable for representing code for C and many other languages.” *Lewis* at 119.

U.S. Patent No. 6,061,520 – Claim 20	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	AllocArray(array type index) ArrayIndex(array type index) ArrayLength(array type index) InvokeOuter(method type index) InvokeDelegated(method type index) Widen(base interface number) Narrow(target type index) BeginSwitch(end swi EndSwitch(end switc BeginExprCase(int) BeginDefaultCase() EndCase(end switch DoBreak(end tag) DoContinue(end tag) Figure 4: The MCode instruction set *Lewis* at 123 Fig. 4 (excerpt). "Figures 5 and 6 give an example of generated MCode. The Clarity method *startup* in Figure 5 produces the MCode instructions shown in part in Figure 6."

U.S. Patent No. 6,061,520 – Claim 20	*Lewis* in view of *Dyer* and further in view of *Proebsting*

	<pre>ThreadedSimulation: module
	{
	    work_mutex: Threads::Mutex;
	    work_per_worker: int; // protected by work_mutex
	    extra_work: int; // protected by work_mutex
	
	    Worker: type = interface inherits Threads::Thread {};
	        // an unusual interface: no methods beyond Thread::startup and the other Thread methods
	
	    WorkImpl: type = implementation of Worker
	    {
	        implement startup: method (our_workers: in int)
	            { // executed when the thread is started; delegates most of its work to forked sibling workers
	                within work_mutex { // acquire work_mutex for duration of the within statement
	                    our_work = work_per_worker;
	                    if (extra_work > 0) {our_work += 1; extra_work -= 1;}
	                }
	                delegated_to_workers: int = (our_workers - 1);
	                if (delegated_to_workers > 0) {
	                    left_workers = delegated_to_workers/2;
	                    right_workers = delegated_to_workers - left_workers;
	                    if (left_workers > 0) left_sibling = new WorkerImpl(left_workers);
	                    right_sibling = new WorkerImpl(right_workers);
	                }
	                do_work(our_work);
	            };
	
	        // the following declarations are private to the WorkerImpl implementation
	        do_work: method (work_to_do: in int) { /* elided */ };
	
	        our_work: int = 0;
	        left_workers: int = 0;
	        right_workers: int = 0;
	        left_sibling: Worker; // left delegatee; manages "left_workers" workers
	        right_sibling: Worker; // right delegatee; manages "right_workers" workers
	    };
	    ...
	};</pre>Figure 5: Part of the Clarity version of the μC++ test program

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 20</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td><pre>
startup: method (our_workers: in int)
{   // executed when the thread is started; delegates most of its work to forked sibling worklers
      within work_mutex {   // acquire work_mutex for duration of the within statement
                  LoadGlobal(0)   work_mutex
                  invokeOuter(0x00010004)   Thread::enter
            our_work = work_per_worker;
                  LoadGlobal(1)   work_per_worker
                  StoreGlobal(5)   our_work
            if (extra_work > 0) {...
                  LoadGlobal(3)   extra_work
                  LoadInt(0, 0)   integer constant 0, type 0
                  CompareInt(>)   extra_work>0
                  SkipThen(cond_false_tag_2)   if(extra_work>0){...
               our_work += 1;
                  GlobalAddr(5)   our_work
                  Dup()
                  LoadIndirect(0)
                  LoadInt(0, 1)   integer constant 1, type 0
                  AddInt()   our_work+=1
                  StoreIndirect(0)   our_work
               extra_work -= 1;
                  GlobalAddr(3)   extra_work
                  Dup()
                  LoadIndirect(0)
                  LoadSigned(0, 1)   integer constant 1, type 0
                  SubInt()   extra_work-=1
                  StoreIndirect(0)   extra_work
                  SkipElse(end_if_tag_3)   if(extra_work>0){...
                  BeginElse(cond_false_tag_2)
                  EndIf(end_if_tag_3)
      }
</pre></td></tr>
</table>

U.S. Patent No. 6,061,520 – Claim 20	*Lewis* in view of *Dyer* and further in view of *Proebsting*

```
        LoadGlobal(0)                   work_mutex
        invokeOuter(0x00010005)         Thread::exit
    delegated_to_workers: int = (our_workers - 1);
        LoadArg(0)                      our_workers
        LoadInt(0, 1)                   integer constant 1, type 0
        SubInt()                        our_workers-1
        StoreLocal(0)                   delegated_to_workers
    ...
    do_work(our_work);
        LoadGlobal(4)                   do_work
        LoadGlobal(5)                   our_work
        ProcCall(7)                     do_work(our_work)
};
        ProcReturn(1)                   method(our_workers:...
```

Figure 6: MCode instructions generated for the µC++ test program's startup method

Lewis at 124-125, Figs. 5-6.

To the extent *Lewis* does not explicitly disclose this limitation of Claim 8, one of ordinary skill in the art would combine the disclosure of *Lewis* with *Dyer* to arrive at the claimed feature. Specifically, *Dyer* shows that an instruction for initializing a data structure (e.g., an array) was widely known in the art, as evidenced by its discussion of static array initialization:

"Mocha transformed a static initializer into an elegant, but illegal, construction:

```
public ConsoleWindow(String string, int i1)
{
dead = false;
styles = { "Plain", "Bold", "Italic" };
sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
...
```

Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these

U.S. Patent No. 6,061,520 – Claim 20	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	initializers are actually implemented by inline code inside constructors, generated by the compiler. When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3.
wherein the simulating step includes the step of: simulating execution of the code to identify the static initialization of the array.	*Lewis* discloses wherein the simulating step includes the step of: simulating execution of the code to identify the static initialization of the array. See the excerpts below and the discussion above as to claim 1. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. "In order to do high-quality code generation, we need to rebuild the expression trees from the stack machine. The code generator defers generation until the final target for an expression is known. Much of the process is similar to that of TNBIND algorithm documented in [Wulf 75]. This algorithm gives excellent results and executes extremely efficiently. A final optimization does instruction reordering to minimize RISC processor pipeline execution conflicts. By organizing the code generator into a series of cascading object streams, we are able to consume MCode and generate native machine code in one pass. Our object-oriented architecture provides an efficient way to trade increased memory for speed." *Lewis* at 127. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically, *Dyer* and *Proebsting* together show that the simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known

U.S. Patent No. 6,061,520 – Claim 20	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	to those of ordinary skill in the art at the time of the invention. For example, the decompiler technology disclosed by *Proebsting* discloses a technique for decompilation of Java byte code that simulates the Java virtual machine's stack: "This paper presents our technique for automatically decompiling Java bytecode into Java source." *Proebsting* at Abstract. "*Decompilation* transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. . . . Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file. We present a technique for transforming low-level Java bytecode into legal Java source code." *Proebsting* § 1. "We have implemented a prototype Java decompiler, Krakatoa, in Java." *Proebsting* § 5. "Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." *Proebsting* § 1. "Symbolic execution of the bytecode creates the corresponding Java source expressions. . . . Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." *Proebsting* § 2. Thus it would have been obvious to one of ordinary skill in the art to combine the methods of *Lewis* with the simulation (symbolic execution) of bytecode against a memory, as disclosed by *Proebsting*. And, even more specifically, the artisan would look to known decompilers,

U.S. Patent No. 6,061,520 – Claim 20	***Lewis* in view of *Dyer* and further in view of *Proebsting***

	such as those disclosed in *Dyer*. *Dyer* reviews three popular decompilers that were available before the '520 patent was filed: DejaVu, Mocha, and WingDis. *See Dyer* at 1. In that review, *Dyer* analyzes the decompilation of code, including the decompilation of static initializers:
	
	"Mocha transformed a static initializer into an elegant, but illegal, construction:
	
	`public ConsoleWindow(String string, int i1)`
	`{`
	`dead = false;`
	`styles = { "Plain", "Bold", "Italic" };`
	`sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };`
	`...`
	
	Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I'm sure Sun must have had a reason. In any case, it's apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
	
	When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code." *Dyer* at 3.

U.S. Patent No. 6,061,520 – Claim 21	***Lewis* in view of *Dyer* and further in view of *Proebsting***
21. The computer-readable medium of claim 18 further including the step of: running the created instruction on the processing component to perform the operation.	*Lewis* discloses the computer-readable medium of claim 18 further including the step of: running the created instruction on the processing component to perform the operation. For example, *Lewis* discloses that "object files containing MCode . . . are processed by standard linkers or other tools." "Runtime generation of machine code offers many advantages. A runtime code generator can take advantage of information about the particular target platform to generate better

U.S. Patent No. 6,061,520 – Claim 21	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	code. For example, different implementations of the SPARC architecture have different instruction pipeline properties. In the case of one new SPARC implementation, code generated specifically for this new processor can run up to 25% faster than code generated for a 'generic' SPARC processor. A runtime code generator can also take advantage of the specific values used in a program to generate machine code customized for those values. One of our goals is to explore the use of on-the-fly code generation for systems programming within Sun. Our representation of MCode object files is unusual in that we use platform-standard object files instead of some Clarity- or MCode-specific representation. This enables us to fully interoperate with existing C and SPARC ABI code. Object files containing MCode (which we call *Linkable MCode* files) are processed by standard linkers and other tools in the same way as other object files. As an example, our Solaris SPARC implementation supports complete interoperation with all SPARC Application Binary Interface (ABI) compliant code [SPARC ABI]. In particular, interpreted or compiled MCode programs can call C programs and vice versa, addresses of MCode procedures can be passed to C code and later called, and all C data types can be exchanged." *Lewis* at 119.

U.S. Patent No. 6,061,520 – Claim 22	***Lewis* in view of *Dyer* and further in view of *Proebsting***
22. The computer-readable medium of claim 18 further including the step of: interpreting the created instruction by a virtual machine to perform the operation.	*Lewis* discloses the computer-readable medium of claim 18 further including the step of: interpreting the created instruction by a virtual machine to perform the operation. For example, *Lewis* discloses the implementation of SPARC instructions to execute the procedure's machine language entry code. "While the MCode interpreter is mostly platform-independent, about 20% of its code is platform-specific. For example, in order to fully support procedure interposition and other ABI capabilities, the SPARC MCode interpreter does not directly interpret MCode ProcCall or Invoke instructions but instead implements them as SPARC ABI calls. Even MCode calls to other MCode procedures are implemented using SPARC instructions and execute the procedure's machine language entry code. This is necessary because the interpreter can

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 22</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td>never know whether a called procedure is actually implemented in MCode or in C. (For example, a programmer might have replaced the called procedure using interposition.) This means the interpreter must fully handle all the details required for ABI calls. If a called routine will return an aggregate value, the interpreter must generate a sequence of machine instructions at runtime (a thunk) to support the SPARC ABI's calling convention that the returned aggregate's length must be encoded into a SPARC UNIMP instruction just after the call. The interpreter also stores all program values in memory as SPARC values since this is required for ABI interoperation.

Recently, a second MCode interpreter has been developed by Mick Jordan. This interpreter executes system models written in the Clarity language. These system models precisely describe how a software system is built: the exact versions of its component parts, all options and build parameters, and how the component parts are assembled. This system modeller is intended to replace the Unix make tool and to eliminate some of its problems: e.g. the inability to exactly reproduce the construction of a software system. The system modeller's MCode interpreter is specialized to executing these models and to interacting with the Clarity program database. It does not need, for example, to support SPARC ABI interoperation." Lewis at 127-28.

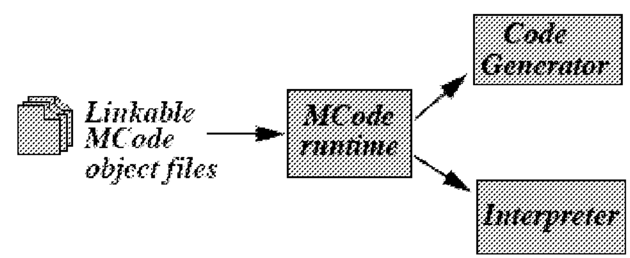


Figure 2: The runtime portion of the Clarity MCode system

Lewis at 121.</td></tr>
</table>

U.S. Patent No. 6,061,520 – Claim 22	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	This claim element would further have been obvious at the time of the invention to one of ordinary skill in the art from the teachings of *Lewis*, either by itself or in combination with other relevant prior art, including, but not limited to the *Dyer* and *Proebsting* references. *Dyer* is a review of various Java Decompilers. These are programs that convert Java class files into Java source code, effectively reverse engineering compiled code to figure out how the underlying code works. *Dyer* discloses, in most relevant part, an example of the code that would be used to implement a static initialization of an array. Applying one of the *Dyer* decompilers to, for example, Java code, would lead to the "[s]ymbolic execution [that] simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." *Proebsting* at § 2.

U.S. Patent No. 6,061,520 – Claim 23	***Lewis* in view of *Dyer* and further in view of *Proebsting***
23. The computer-readable medium of claim 18 wherein the operation has an effect on memory, and wherein the simulating step includes the step of: simulating execution of the code to identify the effect on the memory.	*Lewis* discloses the computer-readable medium of claim 18 wherein the operation has an effect on memory, and wherein the simulating step includes the step of: simulating execution of the code to identify the effect on the memory. For example, *Lewis* discloses that the concrete subclasses of CGValue represent the state of the individual entries on the simulated stack, a process which inherently identifies the effect of the play execution on the memory. "The code generator 'executes' MCode instructions in order to maintain a running simulation of the MCode machine's stack. Concrete subclasses of CGValue represent the state of the individual entries on the simulated stack. These entries include constants, variable references, previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. To the extent *Lewis* does not explicitly disclose this limitation, one of ordinary skill would combine *Lewis* with *Dyer* and *Proebsting* to arrive at the claimed feature. Specifically,

U.S. Patent No. 6,061,520 – Claim 23	***Lewis* in view of *Dyer* and further in view of *Proebsting***
	Dyer and *Proebsting* together show that the simulation of byte codes against a memory to identify underlying code, such as the static initialization of an array, was commonly known to those of ordinary skill in the art at the time of the invention, and that it was used to identify the effect of an operation memory. For example, the decompiler technology disclosed by *Proebsting* discloses a technique for decompilation of Java byte code that simulates the Java virtual machine's stack: "This paper presents our technique for automatically decompiling Java bytecode into Java source." *Proebsting* at Abstract. "*Decompilation* transforms a low-level language into a high-level language. The Java Virtual machine (JVM) specifies a low-level bytecode language for a stack-based machine. This language defines 203 operators, with most of the control flow specified by simple explicit transfers and labels. Compiling a Java class yields a *class file* that contains type information and bytecode. The JVM requires a significant amount of type information from the class files for object linking. . . . Decompilation systems can exploit this type of information and well-behaved property to recover Java source code from the class file. We present a technique for transforming low-level Java bytecode into legal Java source code." *Proebsting* § 1. "We have implemented a prototype Java decompiler, Krakatoa, in Java." *Proebsting* § 5. "Krakatoa uses a stack simulation technique to recover expressions and perform type inference. Expression recovery creates source-level assignments and comparisons from primitive bytecode operations." *Proebsting* § 1. "Symbolic execution of the bytecode creates the corresponding Java source expressions. . . . Symbolic execution simulates the Java Virtual Machine's evaluation stack with strings that represent the source-level expressions being computed." *Proebsting* § 2.

<table>
<tr><th>U.S. Patent No. 6,061,520 – Claim 23</th><th>Lewis in view of Dyer and further in view of Proebsting</th></tr>
<tr><td></td><td>
Thus it would have been obvious to one of ordinary skill in the art to combine the methods of Lewis with the simulation (symbolic execution) of bytecode against a memory, as disclosed by Proebsting. This would be for the purpose of “identify[ing] the [code’s] effect on memory,” as recited in the claim. Further, the artisan would look to known decompilers, such as those disclosed in Dyer. Dyer reviews three popular decompilers that were available before the ‘520 patent was filed: DejaVu, Mocha, and WingDis. See Dyer at 1. In that review, Dyer analyzes the decompilation of code, including the decompilation of static initializers:
“Mocha transformed a static initializer into an elegant, but illegal, construction:
<pre>public ConsoleWindow(String string, int i1)
{
dead = false;
styles = { "Plain", "Bold", "Italic" };
sizes = { "8", "9", "10", "12", "14", "16", "18", "24" };
...</pre>
Bracketed initializer lists for arrays are valid only as initializers for variable declarations (either class or local), not for other assignments. The reason for this differentiation is obscure to me, but I’m sure Sun must have had a reason. In any case, it’s apparent that these initializers are actually implemented by inline code inside constructors, generated by the compiler.
When decompiling this same static initializer, WingDis produced equally beautiful and syntactically correct code.” Dyer at 3.
</td></tr>
</table>