EXHIBIT 10: LEWIS

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*")

U.S. Patent No. 6,061,520 – Claim 14	***Lewis***
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.	With respect to the limitations of claim 12, *Lewis* discloses a data processing system, e.g., the C++ computer programming language, capable of data processing. *Lewis* discloses a data processing system: "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 "MCode has its basis in unpublished work done by L. Peter Deutsch at Sun Microsystems Laboratories in 1992-93.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 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. *Lewis* discloses a storage device:

U.S. Patent No. 6,061,520 – Claim 14	*Lewis*
	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. <u>*Lewis* discloses a program with source code that statically initializes a data structure:</u> As one 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

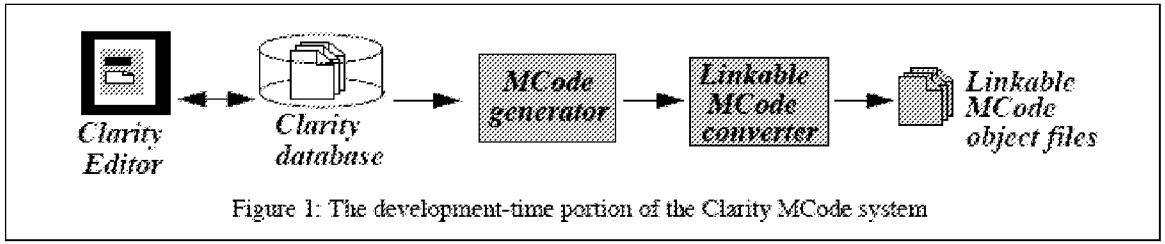
U.S. Patent No. 6,061,520 – Claim 14	*Lewis*
	MCode is suitable for representing code for C and many other languages." *Lewis* at 119. "Figures 5 and 6 give an example of generated MCode. The Clarity method *startup* in Figure 5 produces the MCode instructions shown in part in Figure 6."

U.S. Patent No. 6,061,520 – Claim 14	*Lewis*

```
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 14	*Lewis*
	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 14	*Lewis*
	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 14	*Lewis*

```
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
                  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 14	*Lewis*

```
                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:...
```

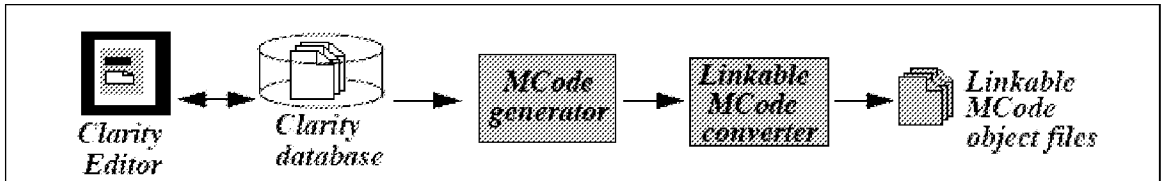
Figure 6: MCode instructions generated for the μC++ test program's startup method

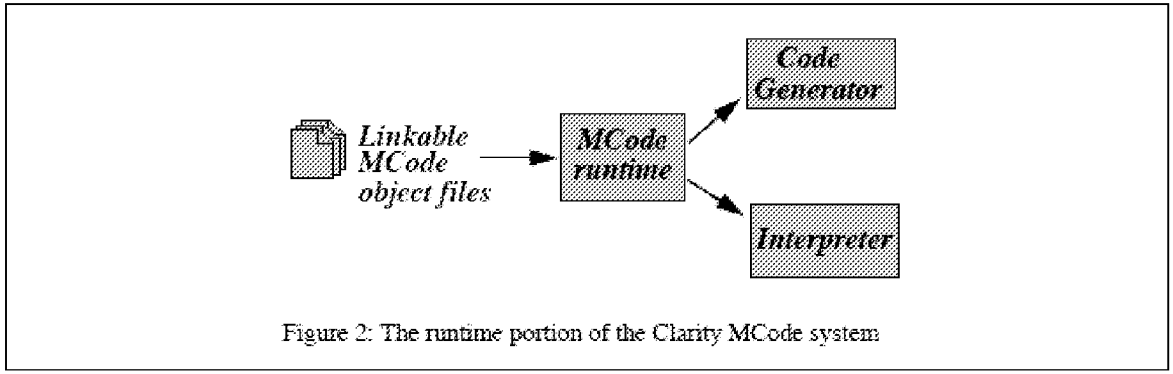
Lewis at 125, Fig. 6.

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. 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

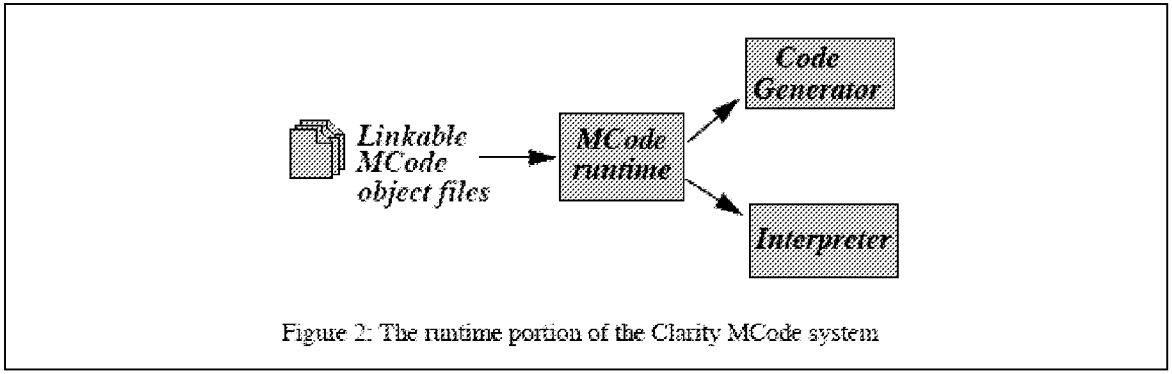
U.S. Patent No. 6,061,520 – Claim 14	*Lewis*
	complete interoperation with all SPARC Application Binary Interface (ABI) compliant code [SPARC ABI]. In particular, interpreted or compiled MCode programs can call C programs and vice versa, addresses of MCode procedures can be passed to C code and later called, and all C data types can be exchanged." *Lewis* at 119. *Lewis* discloses a compiler for compiling the program and generating the class files: *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 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. *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:

U.S. Patent No. 6,061,520 – Claim 14	*Lewis*
	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 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,

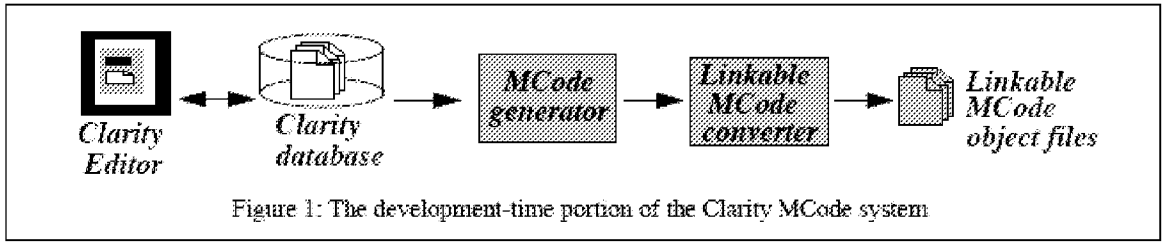
U.S. Patent No. 6,061,520 – Claim 14	*Lewis*
	previously 'executed' subexpressions, and procedure or method calls. The simulated stack records information about operands until the MCode instructions that use them are encountered. Machine code for (sub)expressions is only generated when the value of those expressions is needed." *Lewis* at 126. *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. 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. With respect to the limitations of claim 13, *Lewis* discloses the data processing system of

U.S. Patent No. 6,061,520 – Claim 14	*Lewis*
	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. And with respect to the limitations of claim 14, *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 procedure's machine language entry code." *Lewis* at 127. *See also* the disclosures quoted below: "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 14	*Lewis*
	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*
	 Figure 2: The runtime portion of the Clarity MCode system *Lewis* at 121.

U.S. Patent No. 6,061,520 – Claim 17	*Lewis*
17. The data processing system of claim 12 wherein the created instruction includes an entry into a constant pool.	(Requester notes that a SNQ was found in the related reexamination (Control No. 90/011,489) with respect to claim 2 which is similar to claim 17.) As outlined directly above with respect to claim 14, *Lewis* discloses each and every limitation of the data processing system of claim 12. And with respect to the limitations of claim 17, *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.

U.S. Patent No. 6,061,520 – Claim 17	*Lewis*
	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 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. <u>*See also* the disclosures quoted below:</u> "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

U.S. Patent No. 6,061,520 – Claim 17	*Lewis*
	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 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.