Invited Paper

A Compiler System Supporting Memory Shared by Heterogeneous Machines

Hitoshi Aida[*] and Shiro Kawai[**]

[*]Emeritus Professor, The University of Tokyo, Japan
[**]Scheme Arts, LLC, USA
hitoshi-aida@g.ecc.u-tokyo.ac.jp, shiro@acm.org

Abstract - There are several cases in which non-native data representations should be handled. In this paper, a compiler system to support memory shared by heterogeneous machines is discussed. A keyword standard for representation-type-specifier and another keyword shared for storage-class-specifier are added to language C. A prototype compiler is implemented on SPARCstation (SunOS 4.1.3) and IBM compatible PC (386bsd-0.1). Overhead of endian conversion is small for the machines equipped with byte-swap instruction.

Keywords: Heterogeneous shared memory, C language extension, Representation type, Endian conversion.

1 INTRODUCTION

As shown in Table 1, data representation of compiler systems of computer languages such as C [4] varies in many aspects reflecting CPU architecture, OS environment and so on. Usually, the system only supports the representation by which programs are most efficiently executed. We call it 'native'.

There are several cases, however, representations other than native should be handled. For example, in TCP/IP, the most heavily used communication protocol in the Internet, addresses and data lengths should be represented by most significant byte first order (big endian). In TCP/IP programming, macros such as htonl() (host to network long) or ntohs() (network to host short) are usually used to ensure correct data representation in TCP/IP headers and other places.

The authors used to build a sensor data dispatch system in which the sensor data written by one of the workstations to GM(Global Memory) on its external bus is captured by DBC(Distributed Broadcast Controller), is broadcast through communication bus, and is eventually copied to GM of other workstations. The workstations need not be same machine model. In this environment, in addition to the difference of data representation owing to different CPU architectures, difference of the address of GM seen from those CPUs due to the difference of OS environment becomes problem, especially if pointer variables are placed in GM.

In this paper, assuming that memory is shared by machines with different CPU architecture or OS environment, such as shown above, a compiler system which realizes both efficient access to the shared memory and high source code portability by extending C language is discussed.

Table 1: Example difference of data representation

CPU architecture dependent:
- Byte ordering of multibyte data (endian) [5]
- Alignment of multibyte data
- Representation of negative/floating point numbers

OS dependent:
- Character code set
- Memory mapping scheme

Language dependent:
- Representation of character strings

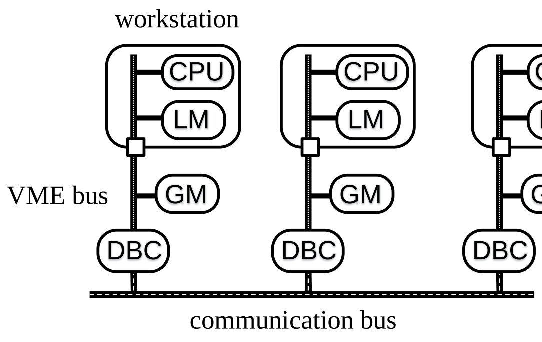


Figure 1: Distributed broadcast memory shared by heterogeneous machines

2 EXTENSION OF C LANGUAGE

2.1 Operational Types and Representation Types

Data type declaration in standard C consists of three parts, 'type-qualifier', 'storage-class-specifier' and 'type-specifier', as roughly shown in Fig 2. Within these, 'type-specifier' is the most essential part which specifies the set of operations applicable to the type and the meaning of those operations. For example, bit shift operations and bitwise logical operations can be applied to char and int, but not to float or double. Another example is that the difference operation of two pointers is only allowed when those pointers point to the same data type except for using explicit cast, and is measured by the size of objects they point.

$$\underset{\substack{\uparrow\\ \text{type-qualifier}}}{\begin{pmatrix}\text{const}\\ \text{volatile}\end{pmatrix}} \underset{\substack{\uparrow\\ \text{storage-}\\ \text{class-}\\ \text{specifier}}}{\begin{pmatrix}\text{auto}\\ \text{register}\\ \text{static}\\ \text{extern}\end{pmatrix}} \underbrace{\begin{pmatrix}\text{signed}\\ \text{unsigned}\end{pmatrix}\begin{pmatrix}\text{short}\\ \text{long}\end{pmatrix}\begin{pmatrix}\text{char}\\ \text{int}\\ \text{float}\\ \text{double}\end{pmatrix}}_{\substack{\text{type-specifier}\\ \downarrow\\ \text{operational-type-specifier}}}$$

Figure 2: Data types in standard C

$$\underset{\substack{\uparrow\\ \text{type-}\\ \text{qualifier}}}{\begin{pmatrix}\text{const}\\ \text{volatile}\end{pmatrix}} \underset{\substack{\uparrow\\ \text{storage-}\\ \text{class-}\\ \text{specifier}}}{\begin{pmatrix}\text{auto}\\ \text{register}\\ \text{static}\\ \text{extern}\end{pmatrix}} \underset{\substack{\uparrow\\ \text{representation-}\\ \text{type-}\\ \text{specifier}}}{\begin{pmatrix}\text{ASCII}\\ \text{EBCDIC}\\ \text{big\_endian}\\ \text{little\_endian}\\ \vdots\end{pmatrix}} \underbrace{\begin{pmatrix}\text{signed}\\ \text{unsigned}\end{pmatrix}\begin{pmatrix}\text{short}\\ \text{long}\end{pmatrix}\begin{pmatrix}\text{char}\\ \text{int}\\ \text{float}\\ \text{double}\end{pmatrix}}_{\substack{\text{operational-}\\ \text{type-}\\ \text{specifier}}}$$

Figure 3: Representation type specifier (not final form)

Here we call the data type specified by 'type-specifier' of standard C as 'operational type'. We also change the name 'type-specifier' to 'operational-type-specifier' for clarity.

As in the case of shared memory described before, different representations of the same operational type could be used. To handle such different representations efficiently and in a portable manner, direct language support is desirable. Here we propose inserting 'representation-type-specifier' in the type declaration in C, such as shown in Fig 3.

When the program accesses variables of which representation type is different from native, automatic conversion is performed. This is much the same as char type variables are converted to int when read and upper bits are truncated when written in standard C.

2.2 Representation Type of Pointers

Pointers are heavily used in C. As for pointers, representation type of both the pointer themselves and the objects pointed by pointers should be considered. Of these, if the representation type of the objects pointed by the pointers are different, the pointers should be supposed to be different operational type. As previously mentioned, in standard C, difference operation of two pointers is only allowed when two pointers point to same data type. Similarly, if pointers are assigned by force using cast to other pointers with different pointed representation type, usage of assigned pointers afterwards may produce unexpected results.

As for the representation type of pointers themselves, in addition to the byte ordering as with integer or floating point data, it should cope with the fact that shared memory might reside different address space reflecting different OS environments. Typical solutions are: (a) represent pointers as the offset from top address of the shared memory, and (b) represent pointers as the relative offset from the pointers themselves. Figure 4 shows a latter example.

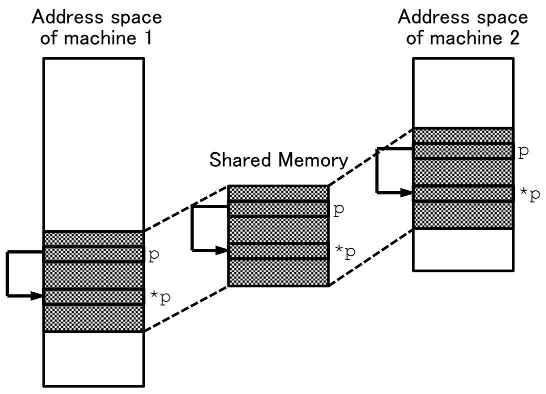


Figure 4: A pointer in the shared memory represented by relative offset

$$\underset{\substack{\uparrow\\ \text{type-}\\ \text{qualifier}}}{*\begin{pmatrix}\text{const}\\ \text{volatile}\end{pmatrix}} \underbrace{\begin{pmatrix}\text{direct}\\ \text{base\_offset}\\ \text{relative}\end{pmatrix}\begin{pmatrix}\text{big\_endian}\\ \text{little\_endian}\end{pmatrix}}_{\text{representation-type-specifier}}$$

Figure 5: Representation type of pointers (not final form)

In standard C, type-qualifiers such as const can be placed both before and after the pointer declaration mark * to distinguish whether the object pointed to the pointer is not allowed to be assigned or the pointer itself is not allowed to be assigned. In the extended C, in addition to type-qualifiers, we allow representation-type-specifiers to be placed after pointer declaration mark * to specify the representation type of the pointer itself.

2.3 Representation Type Standard and Storage Class Shared

Thinking of typical application of heterogeneous share memory system, it is unlikely to use many different representation types in a single application. It is probably enough to use two representation types in each machine: one for shared variables and one for the native representation type of the machine. Thus, in the proposed extended C language, instead of describing explicit representation-type-specifiers such as big_endian or base_offset, we allow only one representation-type-specifier keyword standard in the source code, and the detailed specification of standard representation type is separately fed to the compiler system. This will increase the portability of source code.

All the data in the shared memory is supposed to be standard representation type. On the other hand, in cases such as swapping two data in the shared memory, we want to hold standard representation type data temporally on registers or in the local memory without converting to native representation.

To specify variables be located in the shared memory, additional keyword shared for storage-class-specifier is added to the language. For standard representation of pointers such as the offset from top address of the shared memory to be meaningful, they should point to the shared memory.

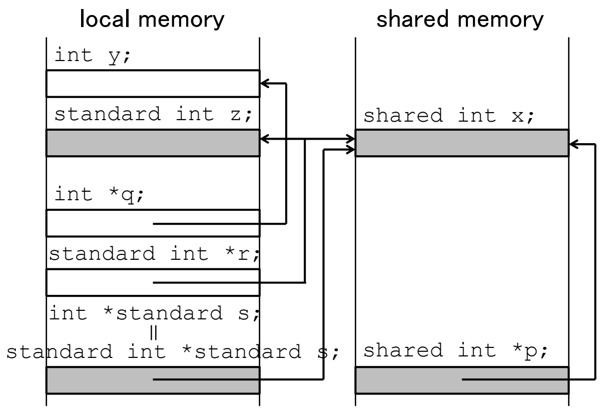


Figure 6: Possible combination of representation type and storage class

Therefore, we assume following constraints between representation type standard and storage class shared:

Constraint 1:
Representation type of variables with storage class shared should be standard.

Constraint 2:
Pointers with representation type standard should point to objects with storage class shared. (For example, they cannot point to functions.)

From these constraints, variables with storage class shared or variables pointed by pointers with representation type standard are implied to have representation type standard:

```
shared int x;
    → shared standard int x;

int *standard s;
    → standard int *standard s;

shared int *p;
    → shared standard int *standard p;
```

There are three combinations for whether the variables are located in the local memory or in the shared memory, and whether the representation type is native or standard for non-pointer variables, and four combinations for direct pointers.

As for structures, all the members of the structures with representation type standard automatically become standard representation type. On the other hand, non-standard representation type structures may have mixture of standard representation type members and non-standard representation type members.

```
standard struct {
    char c;
    int i, j;
    double d;
} t;
            ↓
standard struct {
    standard char c;
    standard int i, j;
    standard double d;
} t;
```

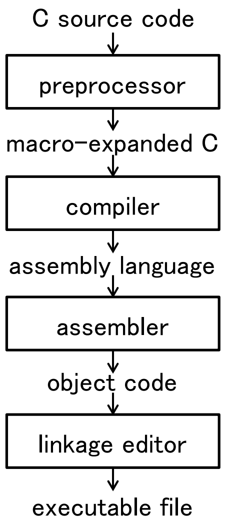


Figure 7: Typical compilation steps for standard C

3 IMPLEMENTATION

Compiler systems of standard C usually consist of several steps as shown in Fig 7. To implement extended C language, best performance is achieved by modifying the compiler in the narrow sense, the second step. However, in cases such as the source code of the compiler for the target CPU architecture is not available, fairly large portion of the features of the extended language can be handled by the preprocessor, the first step. In the following sections, we use preprocessor approach as an example to explain the essence of implementation.

3.1 Representation Type Conversion

Conversion of representation types is the most essential part in the implementation of extended C language.

First of all, compiler system should allocate just enough space for standard representation typed variables. If the conversion is processed by the preprocessor and standard typed pointers are represented as either offset from the top address of shared memory area or relative offset from the pointers themselves, it is natural to convert them to declarations of simple integer variables rather than pointers:

```
standard int x, *p;
        ⇩
long x, p;
```

To access standard representation typed variables, representation type conversion from/to native representation type is needed. In the following, we borrow TCP/IP notations ntohl() for conversion from standard representation to native representation and htonl() for conversion from native representation to standard representation.

```
standard int x;
int y;
```

```
    y = x;
    x = y;
            ⇩
long x;
int y;

    y = ntohl(x);
    x = htonl(y);
```

To access the objects pointed by standard-type pointers, conversions are needed for both pointers themselves and the objects pointed by pointers. The following illustrates the possible preprocessor conversion when standard-type pointers are represented by the offsets from top address of the shared memory area. Here _GM denotes the top address of the shared memory area.

```
int *standard p;
int y;

    y = *p;
            ⇩
extern void *_GM;
long p;
int y;

    y = ntohl(*(long *)(_GM + ntohl(p)));
```

If standard pointers are represented by relative addresses, preprocessor conversion may look like the following:

```
int *standard p;
int y;

    y = *p;
            ⇩
long p;
int y;

    y = ntohl(*(long *)
            ((void *)&p + ntohl(p)));
```

In this case, even simple assignments of pointers needs arithmetic operations:

```
int *standard p, *standard s;

    p = s;
            ⇩
long p, s;

    p = htonl(ntohl(s) +
            ((void*)&s - (void*)&p));
```

Byte order (endian) conversion between standard and native representation types can be implemented as C functions. Figure 8(a) shows an example using bit shift and bitwise logical operations, while Fig 8(b) shows an example using a union.

```
long convert_endian(long data)
{
    return (((data << 24) & 0xff000000) |
            ((data << 8)  & 0x00ff0000) |
            ((data >> 8)  & 0x0000ff00) |
            ((data >> 24) & 0x000000ff));
}
```

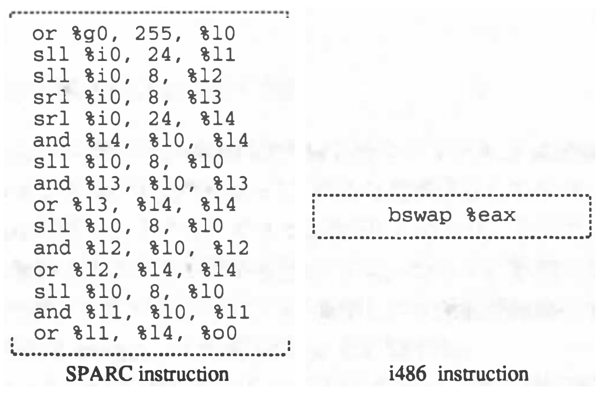
(a)

```
long convert_endian(long data)
{
    union { long l; char c[4] } in, out;

    in.l = data;
    out.c[0] = in.c[3];
    out.c[1] = in.c[2];
    out.c[2] = in.c[1];
    out.c[3] = in.c[0];
    return out.l;
}
```

(b)

Figure 8: C source code examples for endian conversion



```
or %g0, 255, %l0
sll %i0, 24, %l1
sll %i0, 8, %l2
srl %i0, 8, %l3
srl %i0, 24, %l4
and %l4, %l0, %l4
sll %l0, 8, %l0
and %l3, %l0, %l3
or %l3, %l4, %l4
sll %l0, 8, %l0
and %l2, %l0, %l2
or %l2, %l4, %l4
sll %l0, 8, %l0
and %l1, %l0, %l1
or %l1, %l4, %o0
```

```
bswap %eax
```

Figure 9: Inline assembly expansion for endian conversion

To reduce the overhead of the conversion, expansion of macros similar to Figure 8(a) instead of function call is probably effective.

If the compiler itself can be modified or the existing compiler accepts inline assembler expansions, conversion overhead can be further reduced by embedding machine instructions such as shown in Fig 9. In contrast to SPARC [6] CPU architecture which needs fifteen instructions roughly corresponding to Fig 8(a) for conversion, i486 [7] needs only one instruction which can be executed in about single machine cycle.

3.2 Alignment of Structure Members

For the structures of which representation type is standard, all the offsets of the members from top of the structures should meet the most severe alignment requirement of the machines. To meet this requirement by preprocessor, dummy members are inserted to the declaration of standard structures.

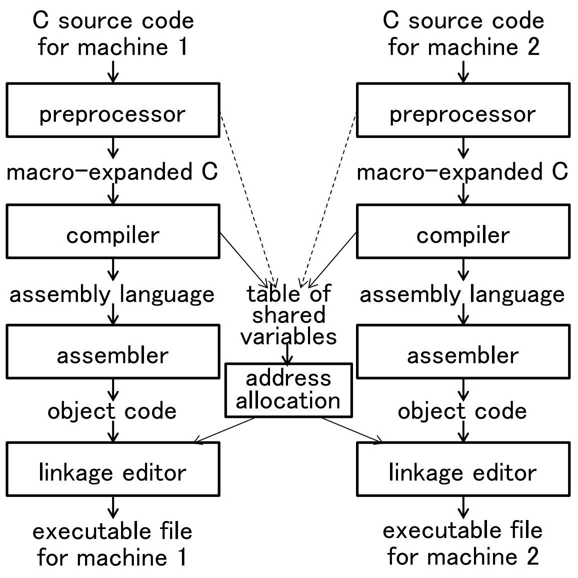

C source code for machine 1
C source code for machine 2
preprocessor
macro−expanded C
compiler
assembly language
table of shared variables
assembler
address allocation
object code
linkage editor
executable file for machine 1
executable file for machine 2

Figure 10: Compilation steps for extended C language

```
standard struct {
   char c;
   int i, j;
   double d;
} t;

struct {
   char c, _dummy1[3];
   long i, j, _dummy2;
   double d;
} t;
```

3.3 Allocation of Shared Variables

If compiled separately, layout of variables in the shared memory may become different among machines. As shown in Fig 10, the compiler (or the preprocessor) is modified to collect all declarations of shared storage class variables in all source files and another compiler step is added to fix the layout of shared memory satisfying the most severe alignment requirement of the machines, which is fed to the linkage editors. Furthermore, a routine to map the shared memory to the address determined at the compilation time should be added to the startup time of the execution in each machine.

4 PERFORMANCE EVALUATION

As one of the experiments to show the practicality and effectiveness of proposed compiler system, we measured the overhead of representation type conversion. In the experiment, two SPARCstations (big endian, SunOS 4.1.3) and one IBM PC compatible machine called i486pc (little endian, 386bsd-0.1) are connected by replicated type shared memory called SCRAMNet [8]. Parallel sorting algorithm known as Batcher's Merge-Exchange Sort [9] was implemented on the hardware. Figure 11 shows performance of single processor execution by relative value to the case of:

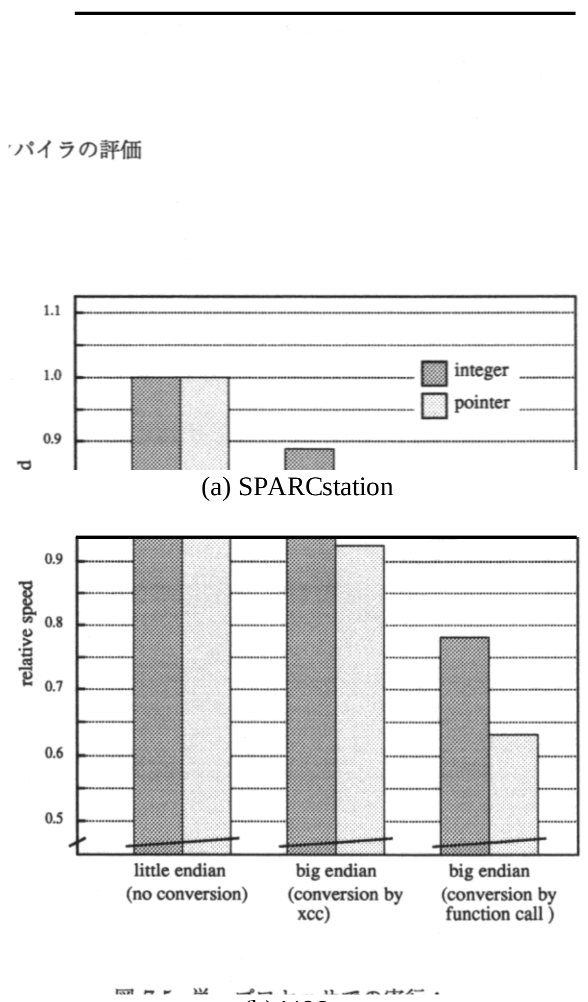


Figure 11: Overhead of endian conversion

- standard representation type is set to the same endian as the native representation type and compiled by the proposed compiler system (denoted as ‘no conversion’).

The other two cases are:

- standard representation type is set to the opposite endian to the native representation type and compiled by the proposed compiler system (denoted as ‘conversion by xcc’)

- standard representation type is set to the opposite endian to the native representation type, conversion functions are written in C, and compiled by GNU CC (denoted as ‘conversion by function call’)

In all the three cases, measurement is done for both sorting integer data themselves in the shred memory (denoted as ‘integer’), and sorting shared pointers to the data rather than data themselves (denoted as ‘pointers’).

For both SPARCstation and i486pc, overhead of representation type conversion is reduced by embedding instructions for representation type conversion by the proposed compiler system.

Table 2: Number of instructions in the innermost loop

	SPARCstation		i486pc	
	integer	pointer	integer	pointer
Total number of instructions	52	86	33	42
Number of instructions for endian conversion	30	62	2	8

Especially for i486pc, which has special instruction for endian conversion, overhead is reduced to less than 10%. On the other hand, for SPARCstation, even when the proposed compiler system embed instructions for representation type conversions, overhead reaches 15～25%. Table 2 shows the number of instructions in the innermost loop in the assembly code file. For SPARCstation, the ratio of number of instructions for representation type conversion to the total number of instructions is very large, which results in the large overhead.

Note that, on the hardware used here, absolute performance (MIPS) of SPARCstation exceeds that of i486pc even considering overhead of representation type conversion. To select which endian to use for standard representation type, many factors such as performance of the processors, load distribution, access frequency of shared variables should be considered.

5 CONCLUSION

In this paper, a compiler system to support memory shared by heterogeneous machines is discussed. A keyword standard for representation-type-specifier and another keyword shared for storage-class-specifier are added to language C. A prototype compiler is implemented on SPARCstation (SunOS 4.1.3) and IBM compatible PC (386bsd-0.1). Overhead of endian conversion is small for the machines equipped with byte-swap instruction.

REFERENCES

[1] T. Saito, H. Aida and S. Kawai, "A Compiler System Supporting Heterogeneous Shared Memories", Journal of Faculty of Engineering, The University of Tokyo, A-31, pp.46-47 (1993) (in Japanese).
[2] S. Kawai, "A Language Processing System for Heterogeneous Distributed Shared Memory", Master's Thesis, Graduate School of Engineering, The University of Tokyo (1993) (in Japanese).
[3] S. Kawai, H. Aida and T. Saito, "A Compiler System for Heterogeneous Distributed Shared Memory", The Special Interest Group Technical Reports of IPSJ, 1992-DPS-058, pp.189-196 (1992) (in Japanese).
[4] B. W. Kernighan and D. M. Ritchie, "The C Programming Language (Second Edition)", Prentice-Hall (1988).
[5] D. Cohen, "On holy wars and a plea for peace", IEEE Computer, Vol.14, No.10, pp.48-54 (1981).
[6] SPARC International Inc., The SPARC Architecture Manual Version 8 (1992).
[7] Intel Corporation, i486^{TM} Processor Programmer's Reference Manual (1990).
[8] SYSTRAN Corporation, SCRAMNet^{TM} Network Reference Manual (1991).
[9] D. E. Knuth, "The Art of Computer Programming", Volume 3, Addison-Wesley (1973).

(Received: January 27, 2024)
(Revised: February 29, 2024)

Hitoshi AIDA received his Doctor's degrees in electrical engineering from the University of Tokyo in 1985. He joined the University of Tokyo as a research associate in 1985, and was promoted to a lecturer, an associate professor and a professor of the same university in 1986, 1990 and 1999 respectively. He stayed SRI international as an international fellow from 1988 to 1990. His research area includes high quality telecommunication and parallel and distributed computing. He is a fellow of IPSJ and IEICE, a senior member of IEEE, and a member of ACM, EAJ, JSSST, JSAI and IEIEJ.

Shiro Kawai is a software consultant and an actor. He received a Ph.D. in Engineering from the University of Tokyo and worked for interactive graphics, digital content production pipelines, and domain-specific programming languages. His credits include "Final Fantasy" videogame and CG movie franchise at Square USA. He then founded Scheme Arts, LLC, and developed a Scheme scripting engine "Gauche". He also appears as an actor in films, TV shows, and theatres, including "Go For Broke", "Running for Grace", and "Hawaii Five-0". He is a member of ACM and SAG-AFTRA.