Structure Sharing and Structure Copying Revisited

Xining Li
Department of Computer Science
Lakehead University
Thunder Bay, Canada
E-mail: xli@flash.lakeheadu.ca

Keywords

Unification, Term Representation, Abstract Machine

Abstract

Various Prolog systems can be classified into two categories: *Structure Sharing* (SS) and *Structure Copying* (SC). The fundamental distinction of SS and SC is the way of representing structures. SS represents a structure instance by a two-pointer molecule with one to the structure skeleton and the other to a binding environment. On the other hand, SC makes a concrete copy of a structure whenever the structure is matched against a variable. SS was used in earlier Prolog implementations while SC has been accepted as the *de facto* standard in modern Prolog implementations. However, analysis and practical comparison of SS and SC shown that programs can be written which make any one method almost arbitrarily worse than the other. In this paper, I propose a new Prolog implementation approach - *Program Sharing* (PS). The major contribution of this work is that PS has the advantages of both SC (representing terms of different types to fit in the size of a machine word) and SS (low overhead in constructing a dynamic structure instance), and the concept of program sharing could be used to realize all special-case instruction-driven unification. PS has been adopted in the design of a new Prolog abstract machine - the LAM$^{\frac{1}{2}}$. I have implemented an experimental LAM$^{\frac{1}{2}}$-emulator in C. Benchmarks show that this new approach is very promising in both performance and memory utilization.

1. Introduction

For more than twenty years, two very different methods - Structure Sharing (SS) and Structure Copying (SC), have been used to implement term unification in various Prolog systems. SS was first introduced by Boyer and Moore [1] and used in earlier Prolog implementations, such as DEC-10 Prolog [2] and MProlog [3]. The first report of SC implementation came from Bruynooghe [4] and has been accepted as the *de facto* standard in modern Prolog implementations. The abstract machines dedicated to Prolog, such as the WAM [5][6] and the VAM [7], adopt SC as the fundamental component to implement efficient unification. Most of the high performance Prolog systems, such as Aquarius, Parma, BIM, Quintus, SICStus and wamcc, are based on the WAM or the WAM data structure with refined instructions for further efficiency [8][9][10][11].

The principal distinction of SS and SC is the way of representing structures. SS represents a structure instance by a two-pointer molecule with one to the structure skeleton and the other to a binding environment. On the other hand, SC makes a concrete copy of a structure whenever the structure is matched against a variable. Analysis and practical comparison of SS and SC shown that programs can be written which make any one method almost arbitrarily worse than the other. In this paper, I will propose a new term representation method - *Program Sharing*. The idea of PS is originated from SS. The significant differences, however, are that PS only needs one pointer to represent a dynamic instance of a structure, and the shared resources are no longer structure skeletons but executable code. The major contribution of this work is that PS has the advantages of both SC (representing terms of different types to fit in the size of a machine word) and SS (low overhead in constructing a dynamic structure instance), and the concept of program sharing could be used to realize all special-case instruction-driven unification.

This paper is organized as follows. In section 2, I briefly compare SS and SC, revisit the principle of SS. In section 3, I present PS - its term representation, memory allocation, pair-wise instruction-driven unification - in detail. Section 4 gives a comparison of memory utilization of different methods and a brief performance analysis. The performance evaluation came from a PS-based Prolog abstract machine - LAM$^{\frac{1}{2}}$. An experimental C-emulator of the LAM$^{\frac{1}{2}}$ has been implemented. I will present the empirical results of the emulator on a small set of hand-translated benchmarks (the translator of Prolog to LAM$^{\frac{1}{2}}$ instructions is in progress). Two existing Prolog systems, SICStus 2.1 (emulated) and BIN-Prolog 3.30, have been compared under the same environment. Performance evaluation shows that the LAM$^{\frac{1}{2}}$ emulator is at least as efficient as the WAM-based systems in general while exhibits much better performance to those tasks involving large, complex data structures.

2. SC and SS Revisited

The performance and memory utilization of a Prolog system are greatly influenced by the way of how logic terms are represented. In a SC system, terms of different types fit in the size of a machine word/register. Non-structure terms can be handled quite efficiently in most cases. Their unification operations could be simplified into matches and assignments. When a variable comes to stand for a structure, however, a concrete *instance* of the structure must be created on the heap, which includes copying the ground description of the structure and allocating a heap cell for every argument in the structure. Copying a complex structure consumes not only time, but also space because superfluous memory cells must be allocated to constants and pointers to shared variables. Furthermore, SC conceals the knowledge about structures. As soon as a structure has been copied onto the heap, the information about its arity and argument types becomes indirectly accessible. Only the instance pointer can be carried around in the future unification. The efficient special term treatments are no longer applicable to a structure instance and its arguments. Therefore, when two terms to be unified are both structure instances, a general unification procedure will be invoked to carry out the stack-based full unification. This procedure forces each pair of terms to go through a sequence of unavoidable operations: push, pop, dereferencing, tag-checking and unification.

On the other hand, a SS system takes advantage of the fact that different instances of the same term could share a single prototype and differ only in their variable bindings. The cost of constructing a new structure instance is quite low: it only needs an environment allocation plus a *molecule* assignment. Why has this scheme been abandoned in the latter Prolog implementations? A major reason is that a molecule, which is used to represent a dynamic instance of a structure,

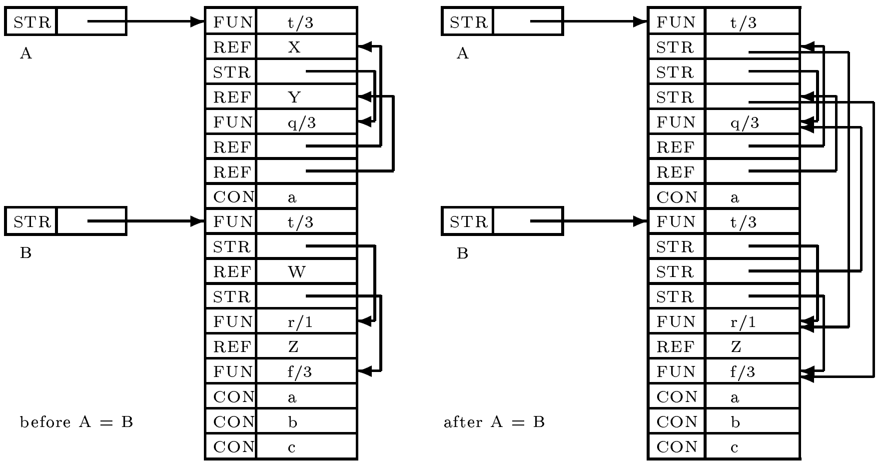

Figure 1: Term Representations of Structure-Copying

consists of two components: a pointer to the skeleton of the structure together with a pointer to a global stack frame which contains the variable instances of the structure. There are three commonly used methods for handling molecules: to embed these two components in a single machine word, to allocate two machine words to each variable catering to the contingency that it might be bound to a structure instance, or to allocate a single machine word to each variable which either refers to a non-structure binding or points to a two-cell molecule created on the heap. The problems involved with these methods are that the first scheme makes it impossible to cope with large address space on 32-bit machines, the second method wastes half of the local/global space on non-structure bindings, while the third increases the usage of the global space.

Let us consider the following Prolog example:

A structure creation and unification example

```
?- p(A), q(B), A = B.
p(t(X, q(X, Y, a), Y)).
q(t(r(Z), W, f(a, b, c))).
```

The execution of the query will create two structure instances carried by variables A and B and unify A and B afterwards. Fig. 1 and Fig. 2 illustrate term representations of SC and SS, before and after A = B. In Fig. 1, two structure instances are copied onto the heap as flattened records. Each record starts with a main functor followed by an array of cells identifying its arguments. Multiple occurrences of a shared variable are equated by pointers to a self-referential cell which represents a single occurrence. Pointers to a shared variable are superfluous and increase dereferencing level. Fig. 2 gives the SS term representation by the third molecule handling method, implemented in MProlog. A molecule is represented in two successive machine words delimited by a dotted line. When a structure becomes the binding of a variable, a molecule is created on the heap and its address with a tag MRF is assigned to the variable. Each molecule is constructed by a skeleton pointer and an environment pointer. A special molecule with a nil environment pointer indicates that the structure is ground (environment independent). In addition, Fig. 2 also shows the structure skeletons in directed graphs, where *off:i* indicates the offset of the ith variable in its environment. For this example, SC needs 18 global cells while SS requires 14 global cells to carry out the query.

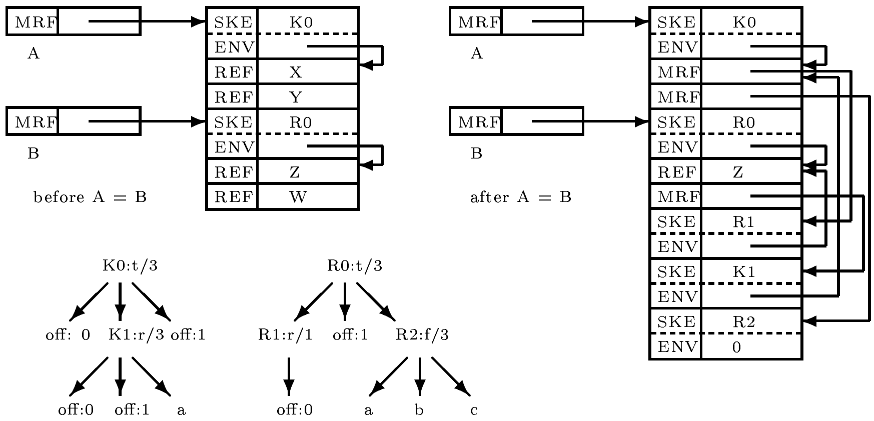

Figure 2: Term Representations of Structure-Sharing

SC and SS have been thoroughly investigated in [13] [14]. It is faster to create terms in a SS system while it is faster to access (unify) terms in a SC system. Comparison between them shown that programs can be written which make any one method almost arbitrarily worse than the other. As Mellish remarked:"the comparison between structure sharing and its alternative is not a simple one, and no quick answer can be given as to which approach is best. It is interesting, however, that a significant factor in the decision is the relationship between the word size and address size of the machine on which the system is implemented. Unfortunately, neither of the systems discussed is optimal in its use of the local and global stacks. It remains to be seen whether mixed approach can be devised that have the advantages of both"[13]. David Warren also indicates that "Although the WAM is a distillation of a long line of experience in Prolog implementation, it is by no means the only possible point to consider in the design space. For example, whereas the WAM adopts *structure copying* to represent Prolog terms, the *structure sharing* representation used in the Marseille and DEC-10 implementation still has much to recommend it"[12].

3. Program Sharing

Prolog is a dynamic typing language in the sense that variables may hold dynamically created data objects of any type. The creation and manipulation of dynamic data objects cost both time and space during execution. For example, when a variable is unified with a complex structure, SC duplicates the whole structure no matter it is a ground or a partially ground structure. In contrast with SC, SS tries to extract static information from a structure during compilation. Consequently, the static information, *i.e.*, the structure skeleton, could be shared by all instances of the structure, if only care is taken to let them have different dynamic variables. I adopt the same idea in my proposal - Program Sharing. The significant differences, however, are that PS only needs one pointer to represent a dynamic instance of a structure, and most importantly, the shared static information is no longer structure skeletons but executable code.

With PS, a Prolog program is compiled into separate code segments for control and unification. The core of PS is the way of implementing unification. It mirrors the pair-wise term unification algorithm cost-effectively. Informally speaking, Prolog terms consists of three basic types: variable, constant and structure. Correspondingly, for a pair of given terms, its unification could be simplified into one of three basic operations: assignment, matching and structure expansion. ¿From PS point of view, there is virtually no *data* in a pure Prolog program. All terms are compiled into and handled as instructions. Like SC, PS represents terms of different types to fit the size of a machine word. Unlike SC, PS sees terms as executable instructions instead of tagged data. Unification (term) instructions have a single format: <opcode, operand>, and they are executed in pair-wise, that is, a unification operation is determined by the combination of a pair of opcodes. Let V denote a variable offset (it could be a register variable, a local variable or a global variable, however, the usage of global variable - indicated by G, is emphasized in this paper), E a memory address and C a constant, Table 1 gives the unification instructions.

opcode	operand	meaning
VAR	V	an uninitialized variable
VAL	V	a variable's value
SCD	E	a selector†
CCD	G	a constructor
FUN	C	a functor
CON	C	a constant†
REF	E	a unbound variable‡
DCD	E	a constructor‡

Table 1: Unification Instructions

Unification instructions can be divided into two classes: static and dynamic. Static instructions are used to encode terms during compilation while dynamic instructions are constructed during execution. In Table 1, instructions marked by ‡ are dynamic instructions, by † are dynamic/static instructions, and the rest are static instructions. A dynamic instruction is one which can become the binding of a variable. Dereferencing operation terminates when a dynamic instruction is reached.

Since PS is originated from SS, they face a same question: how many structure instances will be created during execution. Here I take remarks from [3] to answer this question:“ In general it is impossible to know when, and how many molecules are created - this considerably complicates the management of the global stack. We have noticed, however, that in many cases (which occur frequently in Prolog programs) it is in fact possible to know about the molecules that will be created during execution and therefore the space for molecules can be allocated during compile time. This is the case for structures appearing in head arguments in an *output* position and for all structures in call arguments.”

For reasons of efficiency and minimizing memory usage, PS supports and encourages *mode declaration*. The mode information can be obtained by user-declaration or through global analysis during compilation, where the latter has become a practical tool in some high performance Prolog implementations[8][9]. Based upon the mode declaration (*in*/+, *out*/$\wedge$), structure arguments can be classified into *selectors* and *constructors*. A structure is a selector if it cannot, under any circumstances, be used to form bindings of variables outside the clause; or a constructor otherwise. A special type of constructors is the kind of statically ground structures. They are handled as selectors for a reason to be explained later. Let Ci be a constructor and Si be a selector. In compiling a clause, structure arguments are classified by the following algorithm (assume that all non-ground structures in goal arguments are constructors):

1. Scan arguments of the clause from left to right. For each structure argument, flatten the structure and identify each flattened (sub)structure by a unique name Ci;

2. Scan the flattened structures in reverse order:

i) change Ci to Si if Ci's mode is +, or

ii) change Ci to Si if Ci's arguments consist of ground terms and Sj's only.

Mapping Prolog terms to PS code is straightforward. The following is a simple example of how to generate unification code, where the offset calculation will be discussed later.

Mode: $\{\wedge\}$

Term: {f(a, g(X, h(b, j(c, d))), Y)}

Flattening: C1 = f(a, C2, Y), C2 = g(X, C3), C3 = h(b, C4), C4 = j(c, d)

Scanning: change C4 to S4, then C3 to S3; C2 and C1 remain as constructors.

Results: C1 = f(a, C2, Y), C2 = g(X, S3), S3 = h(b, S4), S4 = j(c, d).

Unification code		
P1:	FUN	f/3
	CON	a
	CCD	offset(C2)
	VAL	offset(Y)
P2:	FUN	g/2
	VAL	offset(X)
	SCD	P3
P3:	FUN	h/2
	CON	b
	SCD	P4
P4:	FUN	j/2
	CON	c
	CON	d

Selectors can be handled very efficiently in PS. A selector code is identified by an instruction <SCD, code-entry>. In general, a selector code is executed under the current environment without delay - this is guaranteed by its input mode. A special case that a selector could be carried by a variable is that the selector is a statically ground structure. To *construct* such a structure, we simply assign the selector instruction to that variable. In this case, we say that the execution of the selector is delayed. Such a selector can be carried to and invoked at any place, because its code is environment independent.

The problem of using two-cell molecules in SS is that we do need a structure skeleton pointer (structure code entry in PS), and an environment pointer for a constructor, since a constructor instance might be carried by a variable to an arbitrary place and the delayed execution of the constructor code needs to access global variables which were allocated in some environment different from the current one. How do I handle these two pieces of information without using molecules? I use a so called *constructor code stub* mechanism to solve the two-pointer problem. When a procedure is called, an integral global frame is allocated to hold global variables as well as the constructor code entries occurred in the matched clause. I call the global cell holding a constructor code entry as the constructor code stub, or just the *stub* in our latter discussion. A static constructor is identified by an instruction <CCD, stub-offset>. When a free variable is matched against a CCD instruction, it is assigned by a dynamic constructor instruction of the form <DCD, stub-address>. A stub will serve on two purposes: its address is the environment base for executing the constructor code, and its content gives the code entry. For example, if a variable's binding is a DCD instruction, then by accessing the stub we have the entry to the code segment which defines the necessary instructions for the structure unification, and by the stub-address we get the environment which will be consulted during unification to access global variables occurred in the structure.

With this setup, Fig. 3 shows the term representation of PS for the example presented in the previous section. When procedure p/1 is called by goal p(A), four global cells are allocated as an

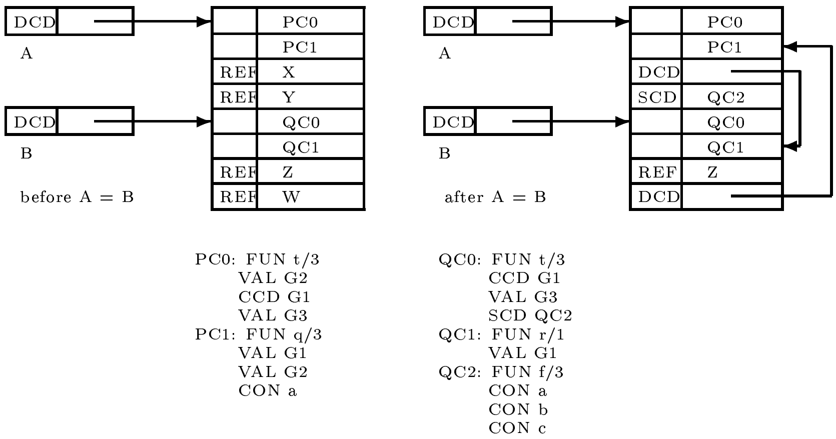

Figure 3: Term Representations of Program-Sharing

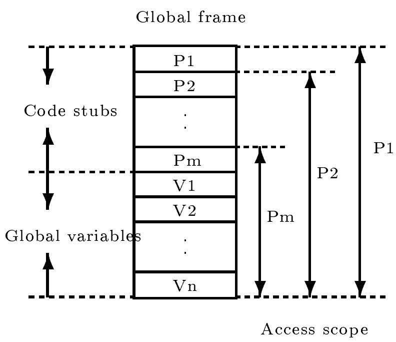

Figure 4: Frame Allocation and Access Scope of PS

integral heap frame to hold global variables and stubs occurred in p/1, where the first two cells are initialized by stubs (PC0 and PC1) and the next two cells are unbound variables (X and Y). The execution of p/1 will assign the address of stub PC0 plus an opcode DCD to variable A. The similar behavior happens for the call of q(B). The goal of A = B thus involves four basic pair-wise operations: a functor matching and three assignments. For this example, PS requires only 8 heap cells to carry out the query, while SC and SS need 18 and 14 heap cells respectively.

In SS, variable indices are calculated against a common frame base. If a variable is allocated at the ith cell of the frame, then *off:i* will be used in all structure/substructure skeletons where this variable occurs. For example, *off:0* refers to the same variable X in both skeletons of K0 and K1. On the other hand, PS introduces a scope rule to compute the offsets of variables/stubs occurred in each code segment. Let P1, P2, ..., Pm be constructor entries and V1, V2, ..., Vn be global variables occurred in a clause, Fig. 4 shows the global frame allocated upon the invocation of the clause and illustrates the access scopes of different code segments, where the first m cells represent stubs of P1 to Pm, and the next n cells are allocated to global variables. Let G_i represent the ith frame cell

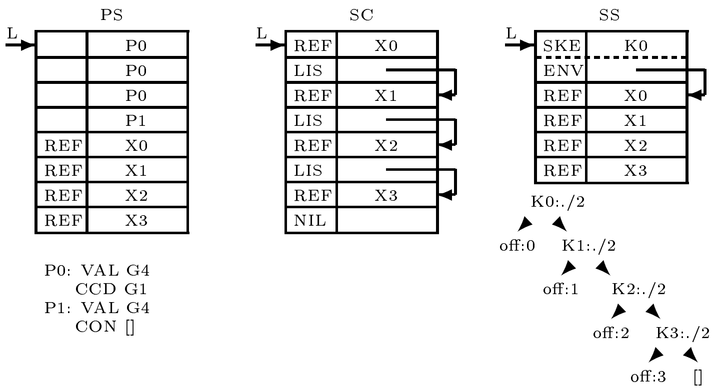

Figure 5: Variable List Representations of PS, SC and SS

against a stub address, then G_{m+1}, G_{m+2}, ..., G_{m+n} refer to V1, V2, ..., Vn in code P1, whereas G_1, G_2, ..., G_n refer to V1, V2, ..., Vn in code Pm. In other words, each constructor code segment uses the scoped index calculation to decide the offsets of its arguments. The code segments in Fig. 3 have revealed such a fact, for example, G_2 denotes variable X in PC0 whereas G_1 represents the same variable in PC1. It is worth to note that the number of stubs, global variables as well as their indices can be calculated during compilation.

To illustrate the index calculation and global frame allocation, let us examine another simple example:

Constructing a variable list

```
:- vlist(∧).
vlist([X0, X1, X2, X3]).
```

Fig. 5 shows the term representations of different schemes upon the execution of a goal *vlist(L)*. For this example, PS and SC require the same amount of heap cells to construct list L. Although it seems that SS needs fewer heap cells, three more molecules (six heap cells) will probably be created when L is accessed in the execution of successive goals. An interesting phenomenon in PS representation is that the code labeled by P0 is shared by three stubs. This is because the first three flattened lists, *i.e.*, C0 = [X0, C1], C1 =[X1, C2], and C2 =[X2, C3], happen to use the same instructions and the same (scoped) indices to encode their arguments. Sharing a code pattern is not just on rare occasions in PS, and is of benefits to code locality.

4. Memory Utilization and Performance Analysis

It is certain that both PS and SS consume less local space than SC, this has been discussed by [13]. As regards global space utilization, PS is clearly better than SS, because it uses one-cell stubs to replace two-cell molecules. Now, I will compare the global space utilization of PS and SC by two typical examples.

A worst case example of SC

```
?- construct([k1, k2, ...], L).
:- construct(+, ∧).
construct([], []).
construct([K|T], [G|L]):- record(K, G), construct(T, L).
:- record(+, ∧).
record(k1, f(a1, a2, a3, a4, a5)).
record(k2, f(b1, b2, b3, b4, b5)).
....
```

The first example collects a list of records from a database through a list of given keys. On this example, PS will gain by arbitrary amounts. Suppose that the total number of records to be collected is N, then PS will use $3 * N$ global space to construct the record list, where N cells for stubs and $2 * N$ cells for global variable instances extracted from [G|L]. On the other hand, SC will use $2 * N + (5 + 1) * N + 2 * N$ global cells where the first $2 * N$ for the record list, $(5 + 1) * N$ for N records and the last $2 * N$ for constructing the input key list. If the arity of the database records is an arbitrary M instead of 5, SC will use $(M + 2) * N$ more heap cells than PS for this particular example.

A worst case example of PS

```
?- nreverse(L, [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
               17,18,19,20,21,22,23,24,25,26,27,28,29,30]).
:- nreverse(∧, +).
nreverse(L, [X|L0]) :- nreverse(L1, L0), concatenate([X], L, L1).
nreverse([], []).
:- concatenate(_, ∧, +).
concatenate(L1, [X|L2], [X|L3]) :- concatenate(L1,L2,L3).
concatenate(L, L, []).
```

By running the second example, we found that nreverse/2 is called 30 times and concatenate/3 is called 435 times. PS uses 2 cells to construct an instance of [X] (one for a stub and another for X), and 3 cells to construct an instance of [X|L2]. The total amount of heap cells consumed by PS is $435 * 3 + 30 * 2 = 1365$. SC also needs 2 cells for each [X] (X and []), but only 2 cells for each [X|L2]. Furthermore, SC requires additional $30 * 2$ cells for constructing the input integer list. Thus the total is $435 * 2 + 30 * 2 + 30 * 2 = 990$.

For a given Prolog program, the number of structure instances to be created in execution is somehow independent of term representation methods (assuming mode analysis as a practical tool). However, different term representation methods require a variable amount of global space for constructing an individual instance. In general, to construct a structure instance $\mathcal{T}$ of the form $f(T_1, T_2, ..., T_m)$, the global cells required by PS and SC are shown in Table 2.

	selector($\mathcal{T}$)	constructor($\mathcal{T}$)	constructor($\mathcal{T}$ == [X|L])
SC	$m + 1$	$m + 1$	2
PS	0	$\leq m + 1$	3

Table 2: Global cells required in constructing a structure instance

If $\mathcal{T}$ is a constructor, it must contain variable arguments. However, it is possible that some of T_i's are ground terms and some are repeated variables, thus the global cells required by PS is less than or equal to $m + 1$. We use $m + 1$ to indicate the number of global cells, the 1 in SC represents the cell allocated to the functor of $\mathcal{T}$ whereas in PS it represents the cell assigned to the stub of $\mathcal{T}$. The worst case of PS is to construct a list structure of the form [X|L]. Three cells are needed where one cell for its stub and two cells for X and L. An interesting point is that if SC did not offer a special treatment to a list structure (different tags are used to distinguish structures and lists such that a list is constructed as a special structure without a main functor), then it should also need three cells for creating an instance of [X|L]. ¿From this analysis, we conclude that PS will take no more than 1.5 times of global space as much as SC in the worst case whereas SC will lose by arbitrary amounts in its worst case.

PS not only bridges the gap between SS and SC, but also provides a novel basis for designing new Prolog abstract machine. The LAM$^{\frac{1}{2}}$ is a PS-based Prolog abstract machine (space limit does allow a detailed discussion). I have implemented a LAM$^{\frac{1}{2}}$ C-emulator which copes with a small set of builtin predicates and without stack overflow checking. The LAM$^{\frac{1}{2}}$ emulator is compiled using gcc 2.7.2 with the -O option. Timings of a small set of hand-translated benchmarks are in millisecond measured on a SUN SPARC IPC with 8 megabytes of memory. Table 3 gives the LAM$^{\frac{1}{2}}$ benchmark statistics which include the number of logic inferences (procedure calls), control instructions and unification instruction-pairs executed, average instructions per logic inference, maximum global and local space (words) consumed, execution time and KLIPS. Table 4 compares the performances of the LAM$^{\frac{1}{2}}$ emulator, SICStus 2.1 (emulated) and BIN-Prolog 3.30.

	nrev30	qsort50	tak(18,10,6)	queen8	serialise	mu	puzzle	zebra
Logic Inferences	496	373	8889	1680	227	605	4693	15708
C-instructions	1550	2331	75556	12219	1530	4036	25709	80828
U-Instr-pairs	2475	2069	35556	9252	2706	6055	48542	247403
Instr-per-LI	8.11	11.8	12.5	12.8	18.7	16.7	15.8	20.1
Max global	1365	816	0	89	364	84	258	35
Max local	215	59	201	147	549	119	502	168
Execution time(ms)	3.67	3.85	121	19.3	4.43	11.36	80.0	332
KLIPS	135	97	73	87	51	53	58	47

Table 3: LAM$^{\frac{1}{2}}$ Benchmark Statistics

	nrev30	qsort50	tak(18,10,6)	queen8	serialise	mu	puzzle	zebra
LAM	3.67	3.85	121	19.3	4.43	11.36	80	332
SIC	4.00	7.00	241	35.2	5.68	9.80	140	551
BIN	5.50	8.00	250	37.0	7.50	12.16	140	700

Table 4: Performance Comparison

Here I should point out that my experimental results only give a rough performance evaluation. First, the LAM$^{\frac{1}{2}}$ emulator is not a complete system. It does not perform stack-overflow checking. Adding this function could influence the empirical results. However, an idea proposed by [11] suggests to use UNIX memory manager to raise an exception signal in case of stack overflow, and therefore eliminates costly software tests. Secondly, benchmarks are small and hand-translated with optimizations based on annotated mode. This is unfair towards the compared systems because mode information helps to generate better code. Despite these factors, the performance evaluation suggests that the LAM$^{\frac{1}{2}}$ emulator is as efficient as the WAM-based systems in general while exhibits much better performance to those tasks involving large, complex data structures, as evidenced by the *zebra* (Zebra Puzzle by Claude Sammut) and the *puzzle* (Music Man Puzzle by Paul Tarau) benchmarks. Interesting readers might want to compare these two benchmarks with the performances of some well-known Prolog compilers, they will find that the LAM$^{\frac{1}{2}}$ C-emulated results are even competitive with the execution times of the native code.

Although there is a suggestion that it is unreliable to use LIPS as the performance unit, I believe that the LIPS does somehow reflect the performance of a Prolog system for real Prolog programs. Some Prolog systems have remarkable peak performances. For programs which are mainly deterministic and involve simple data-structures, these systems could achieve over 1 MLIPS. However, for programs which are backtracking/unification intensive and use complex data-structures, their performances degenerate to several tens KLIPS. The reason of this, I believe, is not only because different procedures carry out different amounts of work, but also because these systems adopt SC as their term representation method which is not an ideal model for handling large, complicated data structures. On the other hand, the difference of the LIPS measurements under the LAM$^{\frac{1}{2}}$ emulator is much smaller.

As evidenced by our discussion, to create a structure instance in execution, SC turns static information to dynamic; SS maintains static information but involves great overhead for handling molecules; whereas PS takes the advantages of both SC and SS, *i.e.*, it makes logic terms to fit the size of a machine word, represents static/dynamic terms as directly executable instructions and spends much less overhead for handling dynamics. PS is very promising in both time and space utilization, especially for those tasks involving large, complex data structures (with mixed ground arguments and repeated-variables).

5. Conclusion

In this paper, I have briefly compared two commonly used term representation methods in implementing Prolog, and proposed a new approach - Program Sharing. The major contribution of this research is that PS has the advantages of both SC (representing data object in a single machine word) and SS (low overhead in constructing a dynamic structure instance). PS could be viewed as a superior alternative to SC and SS in implementing high performance logic programming systems.

Based on PS, a Prolog abstract machine - the LAM$^{\frac{1}{2}}$ has been designed. Generally speaking, a Prolog program is translated into a set of LAM$^{\frac{1}{2}}$ code which specify the control and unification. The control instructions similar to the WAM's counterpart, however, unification instructions are defined and implemented in a totally different style. All terms are compiled into and handled as executable instructions. Dynamic objects stored in execution environment (local stack and global stack) are no longer tagged data but either directly executable unification instructions or code segment entries. Unification is purely pair-wise instruction driven. Performance evaluation shows that this new approach is very promising in both time and space utilization, especially for applications which intensively access structured database and manipulate large, complex data structures.

Study on this subject is now being concentrated on the refinement of the instruction set, completing the emulator to cope with full Prolog, and the design of a LAM$^{\frac{1}{2}}$-based compiler. In addition, PS offers a great potential in the design of a simple but efficient logic machine in hardware. This will be another subject of my future work.

Acknowledgments

I would like to express my appreciation to the Natural Science and Engineering Council of Canada for supporting this research. Thanks to anonymous referees for valuable comments and suggestions. Special thanks to Dr. Bart Demoen for providing benchmark statistics with SICStus.

References

[1] R. S. Boyer and J. S. Moore. The Sharing of Structure in Theorem Proving Programs. *Machine Intelligence 7 (B. Meltzer and D. Miche, eds.)*, Edinburgh University Press, 1972, pp. 101-116.

[2] D. H. D. Warren. Logic Programming and Compiler Writing. *Technique Report: DAI 44, University of Edinburgh*, 1977.

[3] Z. Farkas, P. Koves and P. Szeredi. MProlog: An Implementation Overview. *Implementations of Logic Programming Systems, Kluwer Academic Publishers*, 1994, pp. 103-117.

[4] M. Bruynooghe. An Interpreter for Predicate Programs: Part 1. *Technique Report CW 16*, Katholieke Universiteit Leuven, 1976.

[5] D. H. D. Warren. An Abstract Prolog Instruction Set. *Technical Note 209*, SIR International, 1983.

[6] H. Ait-Kaci. Warren's Abstract Machine: a Tutorial Reconstruction. *MIT Press*, 1991.

[7] A. Krall and U. Neumerkel. The Vienna Abstract Machine. In *PLILP'90, LNCS. Springer*, 1990.

[8] P. Van Roy and A. M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer*, Vol. 25, No. 1, 1992, pp. 54-68.

[9] A. Taylor. LIPS on a MIPS: Results from a Prolog compiler to RISC. In *6th ICLP, MIT Press, Cambridge* , 1990, pp. 174-185.

[10] E. Tick and G. Succi. Implementations of Logic Programming Systems. *Kluwer Academic Publishers*, 1994.

[11] P. Codognet and D. Diaz. wamcc: Compiling Prolog to C. *PLILP'95, The MIT Press*, 1995.

[12] D. H. D. Warren. Foreword in *Warren's Abstract Machine: a Tutorial Reconstruction. MIT Press*, 1991, pp. xiii-xiv.

[13] C. S. Mellish. An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter. *Logic Programming, Academic Press*, 1982.

[14] M. Bruynooghe. The Memory Management of Prolog Implementations. *Logic Programming, Academic Press*, 1982.