A New Term Representation Method for Prolog

Xining Li

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 free 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 claim that programs can be written which make any one method almost arbitrarily worse than the other. In this paper, I propose a new Prolog term representation 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 LAMπ. I have implemented an experimental LAMπ-emulator in C. Benchmarks show that this new approach is very promising in memory utilization and reasonably close to a very good SC-based system in performance.

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 wamce, 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 free (unbound/uninitialized) variable. Analysis and practical comparison of SS and SC show 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†. An experimental C-emulator of the LAM† 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† instructions is in progress). A comparison is made with SICStus 2.1 (emulated) under the same environment. The empirical results show that the LAM† emulator achieves better memory utilization, performs close to an efficient WAM-based system.

2. A Review of SC and SS

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 each 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 and nested structures. Furthermore, SC conceals the knowledge about structures. As soon as a structure has been copied onto the
heap, the information about its arguments 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, 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.

**Example 1: Structure Creation and Unification**

?- p(A), q(B), A = B.
p(t(X, q(X, Y, a), Y)).
q(t(r(Z), W, f(a, b, c))).
Figure 2: Term Representations of Structure-Sharing

Now, let us consider a simple Prolog program in Example 1. 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. Nested structures are represented by tagged data with pointers to their corresponding records. 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 $i$th variable in its environment. For this example, SC needs 18 global cells while SS requires 14 global cells to carry out the query.

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 show 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 free
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 differ-
ences, 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 mir-
rors the pair-wise term unification algorithm cost-effectively. Informally speaking,
Prolog terms consist of three basic types: variable, constant and structure. Cor-
respondingly, for a pair of given terms, its unification could be simplified into one
of three basic operations: binding, matching and structure expansion (or applying
full unification to a pair of structure instances). From PS point of view, there is
virtually no data in a pure Prolog program. All terms are compiled into and han-
dled as instructions. Like SC, PS represents terms of different types to fit the size
of a machine word. Unlike SC, PS sees terms, no matter statically compiled or
dynamically created, as executable instructions instead of tagged data. It is im-
portant to remark that the concept of executable unification instructions comes from
the abstract machine point of view. There is no fundamental difference between
instructions and tagged data for an intermediate code interpreter because decoding
of unification instructions is almost the same as dispatching tagged terms in general
unification.

<table>
<thead>
<tr>
<th>opcode</th>
<th>operand</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR</td>
<td>V1</td>
<td>an uninitialized VARIABLE</td>
</tr>
<tr>
<td>XAR</td>
<td>v</td>
<td>a variable's Value</td>
</tr>
<tr>
<td>SCD</td>
<td>B</td>
<td>a Selector Code Delegate (\dagger)</td>
</tr>
<tr>
<td>CCD</td>
<td>G1</td>
<td>a static Constructor Code Delegate</td>
</tr>
<tr>
<td>CLD</td>
<td>G1</td>
<td>a static Constructor ([R</td>
</tr>
<tr>
<td>PUN</td>
<td>C</td>
<td>a FUnctor</td>
</tr>
<tr>
<td>CON</td>
<td>C</td>
<td>a COntant (\ddagger)</td>
</tr>
<tr>
<td>REF</td>
<td>B</td>
<td>a REFerence [or an unbound when self-referential](\dagger)</td>
</tr>
<tr>
<td>DECD</td>
<td>D</td>
<td>a Dynamic Constructor Delegate</td>
</tr>
<tr>
<td>DLLD</td>
<td>D</td>
<td>a Dynamic ([R</td>
</tr>
</tbody>
</table>

Table 1: Unification Instructions

Unification instructions have a single format: \(<\text{opcode}, \text{operand}>\), and they are
executed in pair-wise, that is, a unification operation is determined by the combi-
nation of a pair of opcodes. Let \(V_i\) denote an offset that may be indifferently a
register, a local or a global offset, however, the usage of global offset - indicated by Gi, is emphasized in this paper, E a memory address and C a constant, Table 1 gives the unification instructions. According to their usage, unification instructions are 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 self-referential REF instruction or a non-REF 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/\, and in\_out/?), structure arguments can be classified into selectors and constructors. A structure is a selector if it can not, under any circumstances, be bound to a variable outside the clause; or a constructor otherwise. Therefore, \_moded and ?-moded structure arguments must be translated as constructors. 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 record by a unique name Ci;

2. Scan the flattened records 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 constants and Si’s only.

Mapping Prolog terms to PS code is straightforward. Example 2 exhibits a complete procedure of unification code generation, where instructions with a trailing star denote the last instructions of their corresponding code segments. Note that a complete understanding of the final code (under the title of After offset calculation) may rely upon my further discussion. Selectors can be handled very efficiently in PS. A selector is identified by an instruction <SCD, code_entry>. In general, code of a selector is executed under the current environment without delay - this is guaranteed by its input mode.
Example 2: Unification Code Generation

\[ \text{p}(+, \wedge) \]
\[ \text{p}(X, f(a, g(X, h(b, j(c, d))), Y)). \]

Analysis

Model \{+, \wedge\}
Arguments \{X, 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)
Seamings: change C4 to S4, then C3 to S3; C2 and C1 remain as constructors.
Result: head = \{X, C1\}, C1 = f(a, C2, Y), C2 = g(X, C3), S3 = h(b, S4), S4 = j(c, d).

<table>
<thead>
<tr>
<th>Before offset calculation</th>
<th>After offset calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{p}/2.u.head: VAR \text{offset(X)}</td>
<td>\text{p}/2.u.head: VAR G2</td>
</tr>
<tr>
<td>CCD* \text{offset(C1)}</td>
<td>CCD* Go</td>
</tr>
<tr>
<td>\text{p}/2.u.1: FUN f/3</td>
<td>\text{p}/2.u.1: FUN f/3</td>
</tr>
<tr>
<td>CON a</td>
<td>CON a</td>
</tr>
<tr>
<td>CCD offset(C2)</td>
<td>CCD G1</td>
</tr>
<tr>
<td>VAL* \text{offset(Y)}</td>
<td>VAL* G3</td>
</tr>
<tr>
<td>\text{p}/2.u.2: FUN g/2</td>
<td>\text{p}/2.u.2: FUN g/2</td>
</tr>
<tr>
<td>VAL offset(X)</td>
<td>VAL G1</td>
</tr>
<tr>
<td>CCD* p/2.u.3</td>
<td>CCD* p/2.u.3</td>
</tr>
<tr>
<td>\text{p}/2.u.3: FUN h/2</td>
<td>\text{p}/2.u.3: FUN h/2</td>
</tr>
<tr>
<td>CON b</td>
<td>CON b</td>
</tr>
<tr>
<td>CCD* p/2.u.4</td>
<td>CCD* p/2.u.4</td>
</tr>
<tr>
<td>\text{p}/2.u.4: FUN j/2</td>
<td>\text{p}/2.u.4: FUN j/2</td>
</tr>
<tr>
<td>CON c</td>
<td>CON c</td>
</tr>
<tr>
<td>CON* d</td>
<td>CON* d</td>
</tr>
</tbody>
</table>

The reason of using two-cell molecules in SS is that we do need a skeleton pointer (a 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 or nested structures 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 hol fold 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 \langle CCD, stub, offset >. When a free variable is matched against an CCD instruction, it is assigned by a dynamic constructor instruction of the form \langle 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 and nested stubs.

A special case that we can change a constructor into a selector is that the constructor is a statically ground structure. When such a structure is matched against a free variable, 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 compiler must follow the above algorithm to generate unification code. Here the particular ordering of flattening and traversal is important, because the purpose
of the algorithm is to find all nested selectors within constructors. The change from constructors to selectors not only improves performance but also saves global space. For example, suppose \( f(X, g(a, h(b))) \) is an \( \land \)-moded argument, then it will be used as a constructor. However, by going through the code generation algorithm, we found that subterms \( g/2 \) and \( h/1 \) can actually be processed as selectors. Thus only two global cells will be allocated: one for the stub of \( f/2 \) and another for \( X \). On the other hand, for a constructor of the form \( f(a, g(b, h(X))) \), no change can be made, and four global cells (three stubs and one variable) are required.

In SS, variable indices are calculated against a common frame base. If a variable is allocated at the \( i \)th cell of the frame, then \( \text{off}:i \) will be used in all structure/substructure skeletons where this variable occurs. For instance, \( \text{off}:0 \) refers to the same variable \( X \) in both skeletons of \( K_0 \) and \( K_1 \) in Fig. 2. On the other hand, PS introduces a scope rule to compute the offsets of variables/stubs occurred in each code segment. Let \( P_1, P_2, \ldots, P_m \) be constructor entries and \( V_1, V_2, \ldots, V_n \) be global variables occurred in a clause, Fig. 3 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 \( P_1 \) to \( P_m \), and the next \( n \) cells are allocated to global variables. Let \( G_i \) represent a global offset against a stub address, the right part of Fig. 3 shows the offsets used in each constructor code segment. For example, suppose that a clause \( p/1 \) has an \( \land \)-moded argument \( f(X, g(a, h(Y))) \). During compilation, we found that five global cells are needed for an instance of \( f/2 \). The compiler will arrange these locations to stubs and variables in the order of \([f/2\_stub, g/2\_stub, h/1\_stub, X, Y]\). Subsequently, three constructor code segments will be generated and each segment is encoded by using the scoped index calculation to decide the offsets of its arguments. For instance, \( f/2 \) has two arguments: a variable \( X \) and a nested structure \( g/2 \). Its code segment thus involves three instructions \((\text{FUN} f/2, \text{VAL} G_3, \text{CCD}\* G_1)\) where \( G_3 \) and \( G_1 \) are global offsets against the location of \( f/2\_stub \). The same principle applies to the code generation of \( g/2 \) and \( h/1 \). Their code will be \((\text{FUN} g/2, \text{CON} a, \text{CCD}\* G_1)\) and \((\text{FUN} h/1, \text{VAL}\* G_2)\) respectively. Finally, the compiler will generate \((\text{CCD}\* G_0)\) - the unification code of \( p/1 \). When \( p/1 \) is invoked during execution, a chunk of five global cells will be allocated and initialized with the code entries and self-

![Global Frame](image-url)
referential variables. If the caller’s argument is a free variable, it will be bound to the instruction \(<\text{DCD, f/2, stub address}>\) which represents a concrete instance of \(f/2\). Interested readers might want to reexamine Example 2 where the final code has revealed the results of offset calculation. It is worth to note that the number of stubs, global variables as well as their offsets are completely determined during compilation.

Based upon the above discussion, Fig. 4 shows the PS term representation of Example 1. When procedure p/1 is called by goal p(A), four global cells are allocated as an integral 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 last 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. By measuring the execution performance statically, it is easy to find that SC requires more efforts in copying structure instances whereas SS needs more overheads for constructing molecules.

Structure instructions, CCD and SCD, are sufficient to encode various compound terms (including lists). However, if we use an CCD to identify a constructor of the form \([H|T]\), we have to allocate three global cells where one cell for its stub and two cells for \(H\) and \(T\). On the other hand, a SC-based system only requires two heap cells because a list instance has a special tag and is handled as a structure without a main functor. For this reason, the earlier version of PS [15] claims that PS will consume no more than 1.5 times of global space as much as SC in the worse case. Even though this has narrowed the gap between SS and SC, it still causes more global space consumption (compare with SC) in cases where lists are frequently constructed. In order to reduce the extra overhead in constructing a \([H|T]\)-type constructor, a special instruction CLD and its dynamic form DLD are introduced.

![Figure 4: Term Representations of Program-Sharing](image-url)
in this improved PS method. Like an CCD instruction, an CLD instruction also has a global offset as its operand. However, this operand is no longer a stub offset, instead, it is the offset to the first of two adjacent global cells allocated to an \([H|T]\) instance. Furthermore, each CLD instruction is implicitly associated with a builtin code segment which consists of two instructions: VAL G0 and VAL\(^*\) G1. In other words, there is no need to allocate and initialize a stub for an \([H|T]\) instance, an CLD/DLD instruction automatically carries an implicit code entry and an explicit environment offset/pointer.

Of course, having introduced these special instructions for an \([H|T]\) constructor will require the compiler to arrange \(H\) and \(T\) always adjacent and in head-tail order. A new problem arises from this constraint: when more than one \([H|T]\)-type constructor in a clause share some variables, what strategy the compiler should take? An easy solution is to use an CLD instruction to identify one instance and to use CCD instructions to represent the rest. However, a better solution is to borrow the idea from SC, i.e., to allocate and initialize extra global cells to equate shared variables. Although the second solution somehow violates the principle of PS - multiple occurrences of a variable share a single instance, it turns out to be more economical.

In summary, based on the mode declaration, compound terms occurred in a clause are flattened and translated to unification code segments. They can be identified by three PS instructions: SCD, CCD, and CLD. An SCD instruction represents a selector and carries its code entry. It can be assigned to a free variable only if it represents a statically ground structure, otherwise the selector code must be invoked under the current environment. An CCD instruction represents a constructor. Its operand is a stub offset through which we can find an environment base and a code entry. It can be bound (in DCD form) to a free variable and its carried code can be invoked at any execution point. An CLD instruction is a variant of CCD specially designed for the \([H|T]\)-type constructors. It is associated with an implicit (builtin) code entry and its operand gives a direct reference to it environment. Moreover, the design of a PS-based compiler should take the constraint on the use of CLD into account.

4. Memory Utilization and Performance Analysis

It is clear that both SS and PS require less local variables than SC does, because a SC-based system must create a local image to bridge each heap variable which occurs many times across the body of a clause, whereas SS and PS hold direct (base plus index) access to global variables.

In regard to global space utilization, PS is generally better than SS, because it uses one-cell stubs to replace two-cell molecules. An extreme case that PS could be arbitrarily worse than SS is that a huge amount of complicated structures are constructed but their nested constructors will never be created. On this, PS might consume more global space because all stubs, no matter they will be used or not, must be created and initialized. On the other hand, SS may only create molecules to represent the top-level structures without concerning their nested components. In other words, the amount of global space PS uses for a constructor is proportional to the sum of the number of stubs plus the number of variables whereas SS uses global space proportional to the number of variables only. Fortunately, this case
seems less likely in practice. Moreover, if a SS-based system (as suggested by [3]) allocates all possible molecules (including molecules for nested structures) during compilation, then PS behaves even better in this extreme case.

Now, I will compare the global space consumption of PS and SC by analysis and empirical results. 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 make up a structure instance $\mathcal{T}$ of the form $f(A_1, A_2, \ldots, A_m)$, the global cells required by PS and SC are shown in Table 2.

|          | selector($\mathcal{T}$) | constructor($\mathcal{T}$) | selector($A_1 | A_2$) | constructor($A_1 | A_2$) |
|----------|-------------------------|---------------------------|------------------|-------------------------|
| SC       | $m + 1$                 | $m + 1$                   | 2                | 2                       |
| PS       | 0                       | $\leq m + 1$              | 0                | $\leq 2$                |

Table 2: Global Space Requirements

In Table 2, $m + 1$ is the upper bound of global cells needed in constructing an instance of $\mathcal{T} \equiv f(A_1, A_2, \ldots, A_m)$, where the 1 in SC represents the cell for the functor of $\mathcal{T}$ whereas in PS the cell for the stub of $\mathcal{T}$. Clearly, if $\mathcal{T}$ is a constructor, it must contain variables or nested constructors. The only case that PS needs $m + 1$ global cells is that all $A_i$'s are distinct variables. However, it is possible that $A_i$'s also include constants, nested structures, and repeated variables. In this case, the number of global cells required by PS is less than $m + 1$ because constants and nested structures are static features of $\mathcal{T}$ and a repeated variable only needs one global cell to be its dynamic instance. When $\mathcal{T}$ stands for a list constructor $[A_1 | A_2]$, things become a little complicated. Let ANY be an instruction other than VAR and VAL. In general, we have four possible combinations: [ANY][ANY], [ANY][VAL], [VAL][ANY], and [VAL][VAL] (note: VAR instructions represent uninitialized variables and could not be used in a constructor). PS uses CCD to identify the first three combinations, and CLD to the last combination. Thus the global cells allocated to different combinations are: 1 cell for the first case (a stub), 2 cells for the second and the third case (a variable plus a stub), and 2 cells for the last case (2 variables as required by an CLD instruction).

From this analysis, I conclude that for well-moded Prolog programs, PS will consume at most as much global space as SC does in any case whereas SC will lose by arbitrary amounts in its worst case. Example 3 shows a worst case of SC.

Example 3: A worst case example of SC

?- construct([k1, k2, ...], L).
\[\text{?- construct}(+, \wedge).\]
\[\text{construct}(\{\}, [])].\]
\[\text{construct}([\mathcal{T}], [G|L]).\]
\[\text{record}(K, G), \text{construct}(\mathcal{T}, L).\]
\[\text{?- \\text{record}(\{\}, [\mathcal{T}], [G|L]).\}
\[\text{record}(k1, f(a1, a2, a3, a4, a5)).\]
\[\text{record}(k2, f(k1, k2, k3, k4, k5)).\]

This 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 uses only $2 \times N$ global space to construct the output list [G|L]. On the other hand, SC will use $2 \times N + (5 + 1) \times N + 2 \times N$ global cells where the first $2 \times N$ for the input key list, $(5 + 1) \times N$ for $N$ records
and the last 2 + N for the output list. If the arity of the database records is an
arbitrary M instead of 5, SC will use (M + 3) * N more heap cells than PS for this
particular example. Note that this analysis is based on the principle of SC without
considering possible optimal implementations.

PS not only bridges the gap between SS and SC, but also provides a novel ba-
sis for designing new Prolog abstract machine. The LAM[::-] is a PS-based Prolog
abstract machine[16]. It separates control and unification code and cooperates two
engines: an one-program-counter (1P) control engine and a two-program-counter
(2P) unification engine. The 1P engine executes control instructions which are
similar to the WAM’s counterpart, which include stack allocation, initialization,
exection control, nondeterministic control, and environment manipulation. Unifi-
cation instructions, however, are executed like the VAM2P which eliminates the
register interface by unifying goal and head arguments in one step. Although the 2P
engine looks like the VAM2P in the sense that they both use the merged caller-callee
unification, yet an essential difference is that the VAM2P adopts a lazy structure
copying strategy, whereas the LAM[::-] works on the program sharing model. To
enable fast decoding of a pair of instructions, the VAM2P defines separate sets of
instructions for head and goal arguments and the sum of a pair of op-code must be
unique. However, when two terms to be unified are dynamic structure instances, a
full unification procedure must be invoked. On the other hand, the LAM[::-] specifies
a set of neutral unification instructions with the same format (a single word). This
instruction set is used not only for encoding (static) program terms but also for
representing dynamic term instances. The advantage is that the 2P engine can ex-
cute any static/dynamic instruction combination. However, the costs of instruction
decoding and operand calculation in the LAM[::-] emulator are more expensive. The
VAM2P and LAM[::-] are well-suited for an intermediate code interpreter or a Pro-
log virtual machine. Unfortunately, using two program counters almost precludes
native code compilation. Whether the PS model can be used to generate highly
optimized, native code, such as the work done by Aquarius Prolog, remains to be
investigated.

I have implemented an experimental LAM[::-] C-emulator. The LAM[::-] 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 and 4 show the comparison of memory utilization
and performance with the LAM[::-] emulator and SICStus 2.1 (emulated).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>nrev30</th>
<th>qsort50</th>
<th>lisp18,10,6</th>
<th>queen8</th>
<th>serialise</th>
<th>mkt</th>
<th>puzzle</th>
<th>zebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAM[::-]</td>
<td>930</td>
<td>544</td>
<td>0</td>
<td>60</td>
<td>288</td>
<td>63</td>
<td>211</td>
<td>35</td>
</tr>
<tr>
<td>SICStus</td>
<td>991</td>
<td>651</td>
<td>0</td>
<td>77</td>
<td>335</td>
<td>83</td>
<td>516</td>
<td>348</td>
</tr>
</tbody>
</table>

Table 3: Comparison of Global Space Consumption (words)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>nrev30</th>
<th>qsort50</th>
<th>lisp18,10,6</th>
<th>queen8</th>
<th>serialise</th>
<th>mkt</th>
<th>puzzle</th>
<th>zebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAM[::-]</td>
<td>4.30</td>
<td>7.03</td>
<td>196</td>
<td>25.9</td>
<td>5.53</td>
<td>12.7</td>
<td>99</td>
<td>410</td>
</tr>
<tr>
<td>SICStus</td>
<td>4.00</td>
<td>7.00</td>
<td>241</td>
<td>35.2</td>
<td>5.68</td>
<td>9.8</td>
<td>140</td>
<td>551</td>
</tr>
</tbody>
</table>

Table 4: Comparison of Performance (ms)

The empirical results of global space consumption give a further proof that PS
is superior to SC, as evidenced by the less memory requirements of all benchmarks.
In regard to performance, I should point out that my experimental results only
suggest a rough evaluation. First, the LAM\textsuperscript{\ddagger} emulator is not a complete system. Its major purpose is to verify the feasibility of PS. It only copes with a small set of built-in predicates. Secondly, benchmarks are small and hand-translated with optimizations based on annotated mode. This is unfair towards the compared system because mode information helps to improve performance. In order to put the benchmark results into proper perspective - as an anonymous referee suggested, I have tried to implement this experimental emulator as close as possible to the compared system. For example, the emulator is written in ANSI C and it involves code for stack overflow checking.

Although some benchmarks reveal better performance under the LAM\textsuperscript{\ddagger}, taking into account of mode declaration and hand-translated optimization, it is still hard to say that the LAM\textsuperscript{\ddagger} outperforms SICStus on these benchmarks. In addition, an anonymous referee indicates that for benchmark zebra, if we rewrote the program such that all term creation is done before any backtracking choice is created, this benchmark could speed up by a factor of two for SICStus. Benchmark mu is particularly interesting because SICStus performs much better than the LAM\textsuperscript{\ddagger}. In this benchmark, most of the execution time is spend on rule/3 with an input mode (\&, ?, +). The current implementation of the LAM\textsuperscript{\ddagger} (statically) translates each \?-mode structure as a constructor no matter its dynamic input is an unbound variable or not. Unfortunately, experiments show that most dynamic input cases are structures (lists) instead of unbound variables. Thus, the LAM\textsuperscript{\ddagger}'s performance is diluted because the overhead of manipulating a constructor (stub allocation, initialization and indirect invocation) is much higher than handling a selector (direct invocation). On the other hand, a SC-based system, such as SICStus, checks the input dynamically, and only creates a concrete structure instance when the input is an unbound variable. This example exposes a shortcoming of PS: the strong dependence on static mode declaration and the weak flexibility on dynamic varieties. One possible way to moderate this weakness is to generate separate code segments to deal with different inputs. This problem will be studied in my future work. From the above analysis, my conclusion is that the performance of a PS-based system could be reasonably close to a very good SC-based system, and therefore PS might be considered as an alternative model in the design space of high-performance logic programming.

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. As it turns out, PS is a mixed approach of SS and SC which has the advantages of both and greatly narrows the gap between them. PS could be viewed as an alternative to SC and SS in implementing high performance logic programming systems. Based on PS, a Prolog abstract machine - the LAM\textsuperscript{\ddagger} has been designed. The LAM\textsuperscript{\ddagger} is well suited for a bytecode emulator. Benchmarks show that this new approach is very promising in memory utilization and reasonably close to a very good SC-based system in performance.

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\textsuperscript{\dagger} based compiler. Several practical issues remain to be solved. For example, an abstract Prolog machine must be able to support built-in predicates like \texttt{assert/1, setof/3, copy_term/2}, etc. to make the system fully functional. At this stage, I am not sure whether a PS-based machine can support these predicates as efficiently as implemented in SC-based systems. An idea is that if we could design an abstract machine which blends PS and SC models, existing techniques could be fully inherited. Next, due to efficiency consideration, most Prolog implementations perform unification without the occur-check. I have found that the LAM\textsuperscript{\dagger} behaves with respect to occur-check similar to the extended WAM of [17] with no or reduced efforts on occur-check. Strategies for this problem are under investigation. Finally and most importantly, as the LAM\textsuperscript{\dagger} uses a different heap-allocation scheme, the traditional garbage collection algorithms [18][19] do not always allow an optimal compaction of the heap. A new algorithm which I called the Chronological Garbage Collection has been developed and is under testing.

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, especially for their suggestions of how to put the performance analysis into proper perspective. Special thanks to Dr. Bart Demeen and an anonymous referee for providing performance and global space consumption data with SICStus.

REFERENCES


