TLP **12** (1–2): 35–66, 2012. © Cambridge University Press 2011
doi:10.1017/S1471068411000482 First published online 12 September 2011

SICStus Prolog—The first 25 years

MATS CARLSSON and PER MILDNER
SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
(*e-mail:* `matsc@sics.se`)

submitted 4 October 2009; revised 1 March 2010; accepted 22 November 2010

Abstract

SICStus Prolog has evolved for nearly 25 years. This is an appropriate point in time for revisiting the main language and design decisions, and try to distill some lessons. SICStus Prolog was conceived in a context of multiple, conflicting Prolog dialect camps and a fledgling standardization effort. We reflect on the impact of this effort and role model implementations on our development. After summarizing the development history, we give a guided tour of the system anatomy, exposing some designs that were not published before. We give an overview of our new interactive development environment, and describe a sample of key applications. Finally, we try to identify key good and not so good design decisions.

KEYWORDS: Prolog, logic programming system, virtual machine, compilers, memory management

1 Introduction

SICStus Prolog[1] is a Prolog system that has evolved for nearly 25 years. In this article, we revisit the factors affecting the choice of language dialects and APIs, and summarize the more important developments that have taken place over this time period. We also give an in-depth description of the anatomy of the system and its development environment. Some key applications are briefly described. Several design choices that were never published before are described herein. We reflect on these choices, and try to learn some lessons.

The rest of the article is structured as follows. In Section 2, we review and motivate the main phases of development. In Section 3, we give our perspective on two important role models for the SICStus Prolog language, APIs and implementation: the Prolog standardization effort and Quintus Prolog. In Section 4, we describe the parts of the system that are the most interesting from a design and implementation point of view, going into details where warranted. In Section 5, we describe our Integrated Development Environment (SPIDER). In Section 6, we briefly describe some key applications. Finally, we conclude with some lessons learned from the whole endeavor.

---

[1] `http://www.sics.se/sicstus`

2 Development history

SICStus Prolog is a Prolog system that "just happened" as opposed to being planned in advance. We now review the main phases of development.

1983. The Warren Abstract Machine (WAM) is published and later becomes a cult tech report (Warren 1983), fascinating many including the first author.

1985–1990. SICS is founded and recruits the first author, who joins the Logic Programming Systems laboratory, headed by Seif Haridi. The laboratory's first and main field of research was or-parallel execution of Prolog. The first author's first task at SICS is to develop the Prolog engine that will be the subject of parallelization (Gupta *et al.* 2001). This happens in the informal Aurora project (Lusk *et al.* 1990) involving David H.D. Warren and researchers from Manchester and ANL, who provide schedulers and visualizers. Subsequently, another SICStus-based or-parallel effort, MUSE (Ali and Karlsson 1990a; Ali and Karlsson 1990b), doing more copying and less sharing than Aurora, is being pursued by other SICS researchers. At the same time, SICS begins distribution of SICStus Prolog, which quickly becomes popular mainly in the academy. Visitors Carl Kesselman and Ralph Haygood develop execution profilers and native code compilers, respectively.

1988–1991. A national funding agency and several companies (see the Acknowledgements) fund the industrialization of SICStus Prolog. This provides the resources to add several pieces of necessary or desirable functionality, including indexed interpreted code, persistent term store, and multiple library modules.

1991–2010. The first author becomes fascinated by Boolean and finite domain constraint solvers, and such solvers appear in SICStus Prolog (Carlsson 1991; Carlsson *et al.* 1997). The SICStus Prolog finite domain solver eventually grows into a sizable subsystem. More on this in Section 4.10.

1995. The ISO Core Prolog standard is published, the first author having been an active member of the standardization committee. Although the standard is not perfect, contains things that would better have been left out, and lacks other dearly needed items, we decide to comply. This leads to the release of SICStus Prolog 3, a dual mode system: its syntax and semantics can be switched dynamically between ISO and pre-ISO.

1998. Jesper Eskilson devotes his master's thesis to a message-passing-based design of multi-threaded execution for SICStus Prolog (Eskilson and Carlsson 1998). A prototype implementation is finished, but does not quite make it into a release. When Jesper leaves SICS, the effort runs out of steam.

1998. SICS acquires Quintus Prolog from a UK company, which had acquired it from Quintus Corp. The reason for this move is partly economical, partly to get access to documentation and design choices that can be integrated into SICStus Prolog, and partly service to the community: the nitty-gritty of WAM technology was not in the UK company's area of expertise. SICS makes bold plans to fuse SICStus Prolog and Quintus Prolog into the Grand Unified Prolog by the year 2000. This is not to happen, but the work on a successor of SICStus Prolog 3 is started, influenced in part by the Quintus Prolog architecture. At the same time, Quintus Prolog assets begin to make their way into the SICStus Prolog 3 system.

2007. The shortcomings of SICStus Prolog 3 and the need for a successor were evident since early on: in particular, its dual dialect and other dynamic aspects are difficult to defend and maintain; by design it can only use 256M of virtual memory, way too little for many applications. After a major redesign, the successor version SICStus Prolog 4 is deemed ready for release.

2009. The first author finally sees the advantage of logical loops (Schimpf 2002), and they appear in SICStus Prolog 4.1. Also, it has been clear for a long time that users have come to expect more from an integrated development environment than what Emacs can provide. After a considerable implementation effort by the second author, we release SPIDER, our Eclipse-based IDE.

3 Standards and role models

SICStus Prolog was conceived in a context of multiple, conflicting Prolog dialect camps and a fledgling standardization effort. The first author's first encounter with a Prolog system was with DECsystem-10 Prolog, i.e. with the Edinburgh tradition, so there was never any question, which camp to align to. Later, Quintus Prolog arrived on the scene in the same tradition, by the same lead designer, and emerged as the de-facto standard, due to its industrial quality and speed. Quintus Prolog was also among the first systems to provide designs for features such as foreign language interface, embeddability, customization through hook predicates and functions, and module system. Since Quintus Prolog seemed to be doing everything right, it seemed pointless to try to come up with alternative designs for these features. Instead, in the design of SICStus Prolog, we opted for the "imitation is the sincerest (form) of flattery" principle (Colton 1825).

The ISO Prolog standardization effort started late, too late. The Prolog dialects had already diverged: basically, there were as many dialects as there were implementations, although the Edinburgh tradition, which had grown out of David H.D. Warren's work was always the dominant one. Every vendor had already invested too much effort and acquired too large a customer base to be prepared to make radical changes to syntax and semantics. Instead, every vendor would defend his own dialect against such radical changes. Finally, after the most vehement opposition had been worn down in countless acrimonious committee meetings, a compromise document that most voting countries could live with was submitted for balloting and was approved.

Although far from perfect, we wanted to promote the standard. At the same time, our users had already developed vast amounts of non-compliant code, which we had no right to break. Our solution to this dilemma was to provide a dual dialect system, SICStus Prolog 3.

4 System anatomy

This section is more or less a white paper of the current system architecture, covering the parts of the system that are the most interesting from a design and implementation point of view. This description is necessarily incomplete, and the omission of some system component does not at all mean that its design and implementation is trivial or uninteresting.

Before and especially after our take-over of Quintus Prolog, a lot of designs and assets have migrated into SICStus Prolog, including: instruction set details, tagging scheme, `structs` and `objects` modules, foreign language interface, message and query systems, and memory manager. So in the rest of this article, we will not credit Quintus Prolog each time.

4.1 Modes of execution

Prolog code can be executed in three different modes, and each variant comes with its pros and cons.

Interpreted. Prolog clauses are stored in a form that is close to the source code and are executed by an interpreter written either in the host language or in Prolog itself. Such an interpreted is an excellent base for debuggers and is virtually necessary for bootstrapping purposes even in the presence of a compiler. The main disadvantage is slow execution.

Native code. Early, successful implementations such as (Warren 1979; Farkas *et al.* 1994) showed that Prolog is amenable to compilation to native machine code with modest to good execution speed. Later work (Taylor 1991; Van Roy and Despain 1992) demonstrated that excellent execution speed can be achieved with global analysis. The main drawbacks of native code compilation are: the large amount of work that has to be invested, slow compilation, difficulty of using stand-alone assembler and linker tools in the compilation chain, and its inherent lack of portability. Also, a variant of Amdahl's law (Amdahl 1967) applies: the speedup available from compiling code to native code is limited by the time spent elsewhere in the runtime system and application code.

Virtual code. This approach can be seen as a compromise between the above two extremes. Its feasibility has been demonstrated by a vast number of programming languages including Pascal, Forth, Lisp, ML, and Java. Most if not all contemporary implementations of Prolog use this approach, exclusively or in combination with the above two.

4.2 Virtual machine

SICStus Prolog was not bootstrapped the classical way, with an interpreter written in a host language. First came a virtual code (WAM) compiler, developed on another Prolog system, a WAM emulator written in C, and a meta-interpreter.

The original WAM report only treated the Horn clause subset of Prolog, so of course the instruction set had to be enriched with instructions to support cut, arithmetic functions, arithmetic tests, term comparison, etc. Also, some deviations from the original WAM design were made and are described and motivated below. Specific features of the SICStus Prolog VM include the following:

`get_constant_x0` *t*	`get_nil_x0`
`get_structure_x0` *f/a*	`get_list_x0`
`get_large_x0` *n*	

Fig. 1. Specialized `get` instructions for indexable clauses. Each instruction encodes a principal functor. The compiled clause for such clauses begins with one such instruction, instead of, e.g. `get_constant` *t*`,0`. If the given clause is called with a non-variable first argument, indexing will kick in and only try clauses that match the given principal functor. Hence, these instructions become no-ops, and the indexing mechanism arranges to skip them. If called with a variable first argument, however, these instructions are not skipped and act as normal `get` instructions. *t* denotes an atomic term; *n* denotes a float or bignum; and *f/a* denotes the functor of a compound term.

Indexing. In SICStus Prolog, clause indexing is performed as part of the predicate call operations (`call` and `execute`), which index on the first argument if the callee is of the appropriate kind. This is done by means of a per-predicate data structure (essentially, a hash table) that maintains an index over the clauses. This is in contrast to the original WAM, which provides instructions to perform such indexing. This design decision was made mainly for convenience of incremental compilation, which deals with one clause at a time, but also to reduce emulator overhead. However, incremental compilation is by no means incompatible with having indexing instructions; witness, e.g. Quintus Prolog. Furthermore, indexable clauses use `get` instructions specialized for matching the first argument, as shown in Figure 1.

Backtracking. Taking the next alternative of a choicepoint, and removing the choicepoint if the last alternative was taken, is done as part of a general backtracking routine. This is again in contrast to the original WAM, which provides instructions for these purposes. This design decision was made for the same reasons as for the indexing issue. However, SICStus Prolog has retained a `try` instruction, which creates a choicepoint if multiple clauses match a procedure call.

Inlined operations. The instruction set directly supports primitives for cut, if-then-else, arithmetic functions and comparisons, type tests, term comparisons, passing values to and from foreign functions, and basic built-in predicates.

It is worth going into some detail about arithmetic, as the design has changed quite a bit. In SICStus Prolog 3, every binary arithmetic function had a corresponding instruction with two input and one output operand (temporary registers) and a corresponding implementation in a C function to dereference the inputs, compute the value depending on the types of the inputs, and store the value (see Figure 2). SICStus Prolog 4 uses the Quintus Prolog design, which is based on two accumulators holding untagged values throughout the evaluation of an expression, and instructions falling into four categories, each item illustrated by the corresponding part of Figure 3:

(1) Loading constants and variables into one of the accumulators; unspilling intermediate results.

(2) Applying a function to the accumulators. The case where the operands are integers (except bignums) is handled inline in the core emulator.

`function_1` f, s_1, d	
`function_2` f, s_1, s_2, d	`function_2_imm` f, s_1, i_2, d

Fig. 2. SICStus Prolog 3 arithmetic instructions (sample). Every arithmetic function is implemented by a C function that dereferences and untags the inputs, computes the value depending on the types of the inputs, tags it, and handles any stack overflows. The virtual machine merely retrieves the function to call and its inputs from the operands, and stores the computed value in the destination. The right-hand side shows the special case where a binary function takes an immediate second argument. f denotes the C function implementing the instruction; s_1 and s_2 are source registers; i_2 is a source immediate value; and d is the destination register.

`first_constant` i	`later_constant` i
`first_large` n	`later_large` n
`first_x_value` x	`later_x_value` x
`first_y_value` y	`later_y_value` y
`binop_add`	`binop_add_imm` i
`binop_subtract`	`binop_subtract_imm` i
`binop_multiply`	`binop_multiply_imm` i
`binop_divide`	`binop_divide_imm` i
`binop_idivide`	`binop_idivide_imm` i
	`store_constant` i
	`store_large` n
`store_x_variable` x	`store_x_value` x
`store_y_variable` y	`store_y_value` y
`equal_to` ℓ	`equal_to_imm` i, ℓ
`less_than` ℓ	`less_than_imm` i, ℓ
`greater_than` ℓ	`greater_than_imm` i, ℓ
`not_equal_to` ℓ	`not_equal_to_imm` i, ℓ
`not_less_than` ℓ	`not_less_than_imm` i, ℓ
`not_greater_than` ℓ	`not_greater_than_imm` i, ℓ

Fig. 3. SICStus Prolog 4 arithmetic instructions (sample). Let A and B denote the two arithmetic accumulators. Top: instructions that untag and load a number into A (left) or B (right). Second left: binary operations on A and B, leaving a value in A. Second right: binary operations on A and an immediate operand, leaving a value in A. Third left: instructions that tag and store the contents of A into a Prolog variable. Third right: instructions that compare the contents of A with a given value, and fail if they differ. Bottom left: instructions that compare the contents of A and B, and branch if the comparison fails. Bottom right: instructions that compare the contents of A and an immediate operand, and branch if the comparison fails. i denotes a size-limited integer constant; n denotes a float or bignum; x and y denote a temporary and a permanent variable, respectively; and ℓ denotes an "else" label.

(3) Storing or unifying the value of an expression; spilling intermediate results.
(4) Comparing the values of two expressions.

In addition, for both designs, instruction variants with immediate operands exist, as an example of instruction merging. Thus, the SICStus Prolog 4 design may seem to optimize non-trivial expressions involving intermediate values, but with a higher

```
incmax(X,Y,Z) :- Z is max(X+1,Y).
```

```
function_2_imm add,x(0),1,x(0)
function_2 max,x(0),x(1),x(0)
unify_value x(0),x(2)
proceed
```

```
first_x_value x(0)
binop_add_imm 1
later_x_value x(1)
binop_maximum
store_x_value x(2)
proceed
```

Fig. 4. Top: a Prolog clause containing arithmetics. Middle: the corresponding SICStus Prolog 3 VM instruction sequence. Bottom: the corresponding SICStus Prolog 4 VM instruction sequence.

```
lifetime_map(_, Map) :- var(Map), !.
lifetime_map(DUs, Map) :-
    lifetime_map(DUs, 0, Map).
```

```
lifetime_map/3:
    var x(1) else L1
    cut
    proceed
L1: get_x_variable x(2),x(1)
    put_constant 0,x(1)
    execute lifetime_map/3
```

Fig. 5. Top: a Prolog clause containing a test allowing to branch directly into the next clause if the test fails, bypassing general backtracking. Bottom: the corresponding SICStus Prolog 4 VM instruction sequence. Execution starts at the first instruction, without creating any choicepoint.

setup cost due to the initial load and final store. Experiments have shown that the SICStus Prolog 4 design is significantly faster also on code doing only simple integer arithmetic. Figure 4 shows an example of the compilation of arithmetics.

Conditionals. Type and arithmetic test instructions are equipped with an “else” branch, which is taken if the test fails. Often, the else branch can go to the next clause, bypassing general backtracking. This is a “leaner and meaner” variant of shallow backtracking (Carlsson 1989), which was implemented in an early version. These else branches somewhat complicate incremental compilation. For example, suppose that the first clause of predicate P/N contains such an else branch. The compiler back-end will make it point to the general backtracking routine. But to enable this optimization, after the second clause of P/N has been compiled, the back-end must revisit the else branch of the first clause and make it point to the second clause. Finally, the second clause must not be threaded into the general backtracking chain of the first clause. An example is shown in Figure 5.

General disjunctions and logical loops (Schimpf 2002) are “flattened” by the compiler into anonymous predicates. Backtracking from one disjunct to another can use the general backtracking mechanism as well as else branches.

Garbage collection support. The question as to what is the best garbage collection algorithm for Prolog is a controversial one. For SICStus Prolog, we chose to implement a mark-and-sweep algorithm (Appleby *et al.* 1988; Carlsson and Sahlin 1990); see also Bevemyr and Lindgren (1994) for a detailed algorithm summary. As shown in Bevemyr and Lindgren (1994) and elsewhere, mark-and-copy can run faster than mark-and-sweep, especially if there is little live data, even if the optimization in Chung *et al.* (2000) is applied. However, there is a property that, although not enforced by the ISO standard, a lot of existing Prolog code relies on: preservation of variable order. This property is maintained by construction by mark-and-sweep, but not by mark-and-copy. In Bevemyr and Lindgren (1994), several methods to cope with this problem are listed, and they all boil down to either disabling mark-and-copy in the presence of term comparisons or adding extra data structures to the VM for supporting variable order. Although we are convinced that mark-and-copy is a viable alternative to mark-and-sweep, we found that the benefits do not outweigh the extra complexity of having to maintain a *fromspace* and a *tospace*, the extra support necessary for maintaining variable order, the less effective memory reclamation by backtracking, and the risk of running into unforeseen problems, what with mutables, trailed goals, attributed variables, and everything. Last but not least, we were guided by the “if it ain’t broke, don’t fix it” principle.

The VM handles stack overflows as follows. At procedure calls, if the global stack has less than a prescribed amount of free space, it is expanded and/or garbage collected. The inlined operation instructions also check this. Finally, the compiler emits an instruction to perform this test elsewhere if needed, which is rarely the case. We have taken the approach to ensure that all memory reachable by the garbage collector contain valid terms. This is in contrast to, e.g. Quintus Prolog, which does not make such a guarantee, and uses runtime tests to determine whether or not terms are valid. The main issue with ensuring validity of terms concerns *permanent variables*, which are often uninitialized at the time garbage collection occurs. However, uninitialized locations can be discriminated from initialized ones by scanning the VM code for past and future operations, and this is the approach taken by SICStus Prolog 4; see Section 4.3. In SICStus Prolog 3, we handled this issue by ensuring that all permanent variables be initialized before any garbage collection could be invoked.

As we will see later, there are several conditions that cause the execution to be suspended at the next procedure call or inlined operation. The VM has a conceptual “generic overflow flag”, which is the disjunction of all such conditions, and a “generic overflow handler”, which “pushes” the current execution state, and then checks and handles each condition in detail.

Coroutining support. SICStus Prolog supports goals being suspended on attributed variables (Holzbaur 1992). Binding an attributed variable will set the generic overflow

`get_large` n, x `put_large` n, x `unify_large` n	`get_large_x0` n
`first_large` n `store_large` n	`later_large` n

Fig. 6. SICStus Prolog 4 instructions encoding occurrences of floats and bignums. The top four instructions encode unification with such numbers. The bottom three encode arithmetic with such numbers. n denotes a float or bignum; x denotes a temporary register.

flag, after which the generic overflow handler will arrange for the suspended goals to be run. This mechanism is described in more detail in Section 4.10.

Interrupt handling. A Prolog predicate can be linked to a UNIX signal or similar. To ensure that the VM is in a secure state when the interrupt is serviced, a two-stage solution is used: when the interrupt arrives, a primary interrupt handler sets the generic overflow flag; and at the first opportunity, the general overflow handler services the interrupt.

Floats and bignums. Such numbers are represented as “boxes” on the global stack, in a way so that they can be distinguished from regular terms. Certain instructions encode their occurrences in Prolog code (see Figure 6). As Prolog terms, they use the same basic tag as structures, but are distinguished by non-standard functors.

Profiling support. Profiling in SICStus Prolog is done by instrumenting the virtual code with *counter* instructions. When executed, such instructions simply increment a private counter. After execution of a benchmark, the relevant counter values are easily gathered by scanning the virtual code. This scheme was described in Gorlick and Kesselman (1987) and was first prototyped on an early SICStus Prolog version. The instrumentation is done at compile time, but could have been done directly on existing virtual code.

Although this scheme provides exact information about the number of predicate calls and backtracks, it cannot know exactly how much time is spent where in the code. To overcome this obvious limitation, one would have to monitor the VM program counter using clock interrupts, like `gprof`.

Another current limitation is that no call graph is maintained. It is often of interest to know not only how many times a predicate was called, but also where it was called from. Such information could be readily provided by a small piece of extra profiling, since at every predicate call operations (`call` and `execute`), the VM stores the caller location in a register, for use by the source-linked debugger.

Low-level considerations. The layout of the VM code was partly designed, partly evolved, to minimize emulator overhead. Pointers and constants are word aligned, but instructions are half-word aligned, which implies that instructions that contain a pointer or constant need to exist in a (word) *aligned* and an *unaligned* variant, where one of the two variants includes a padding half-word. Operands denoting registers are encoded with offsets off the base address of a register bank as opposed to just integers. The instruction dispatch loop makes use of gcc’s computed goto extension: the instruction opcode is encoded as an offset into a table of labels. The table has one *read mode* and one *write mode* entry per instruction. Thus, to select mode, one just adds an offset to the opcode. On 64-bit platforms, instructions and their fields are twice their size on 32-bit platforms, except operands encoding bignums and floats.

Instruction merging and specialization. These are two well-known transformations of VM instruction sets, aiming at saving time as well as space. In Nässén *et al.* (2001), we performed an extensive study of these two transformations and their impact on the SICStus Prolog VM. The current instruction set was finalized based on that study. Briefly, we use specialization to a very limited extent, only for the special first argument `get` instructions mentioned above, and for frequent instructions that move a value from one virtual register to another. Merging, on the other hand, was found to pay off more and is used extensively. Instruction pairs as well as patterns involving longer sequences are subject to merging.

Tagging schemes. All Prolog implementations need to use some means of runtime typing of its terms. Most implementations, including SICStus Prolog, use tagged pointers, i.e. machine addresses with a few bits or even an extra word replaced by a bit-field that denotes the type of term pointed to, but tagged object implementations also exist, e.g. Tarau and Neumerkel (1994) and Brady (2005). SICStus Prolog 3 reserved the four most significant bits, with the rationale that fewer bits would not suffice for encoding the basic types, including bignums, floats, and attributed variables. The implementation settled on using nine different tags. Moreover, the two least significant bits were reserved for use by the garbage collector. The main disadvantage of this choice was that it limited the address range of non-atomic terms to 256M on 32-bit platforms, which is much too little for many applications. SICStus Prolog 4, and the original WAM report, instead reserve the two least significant bits, plus a third bit when the pointer is not a machine address, i.e. an integer or an atom. With this design, no address space problems arise. Bignums and floats use the same tag as structures, but are distinguished by non-standard functors. All types of variables use the same tag. The garbage collector still needs to store two bits for every word, so the question is, where? The SICStus Prolog 4 solution is to reserve a small part of each Prolog stack as a bit array for use by the garbage collector.

4.3 A note on code scanning

One of the advantages of VMs is the ease with which various information can be extracted from the virtual code, usually in time linear in the length of the code. This is for example the case for the use-definition analysis (Aho *et al.* 1986) that the garbage collector performs. SICStus Prolog 4 uses this technique in the following contexts:

- As mentioned before, test instructions are equipped with an “else” branch, which is taken if the test fails. The compiler back-end must scan code containing such “else” branches, making them point to the next clause.

- The garbage collector needs to identify uninitialized local stack locations. It also needs to know which temporary registers are live, if a global stack overflow occurred in the middle of VM code. Code scanning solves both of these tasks.
- SICStus Prolog supports a binary file format for precompiled code. When creating such files, VM code and other pieces of the memory image are dumped, together with relocation information. Code scanning is used to find what relocation information to write to the file. When loading such files, the VM code is not only scanned but also relocated. Relocatable information includes pointers to predicates, atom numbers, and endianness.
- All Prologs that the authors are aware store atoms in a table for purposes of representation sharing and $O(1)$ time identity test. Since the table can fill up, many Prologs provide an atom garbage collector, which disposes of atoms that are no longer in use anywhere. The atom garbage collector needs to scan all relevant memory areas, including the VM code, to discover which atoms are still in use.
- As mentioned before, SICStus Prolog provides a counter-based execution profiler. If told to instrument code for profiling, the compiler inserts special counter instructions at certain places in the VM code. The profiler later uses code scanning to reset those counters prior to profiling and to gather their values afterwards.
- If an arithmetic instruction encounters an invalid argument at runtime, for example an atom, an error exception is raised. By scanning the code around the program location, one can reconstruct a goal that is semantically if not syntactically identical to the source code where the error occurred. The decompiled goal is part of the error exception.

4.4 Native code

Native code compilation for SICStus Prolog has a long history. Starting in the 1980s, we developed compilers from WAM code to Motorola 68K and SPARC. We used a fixed mapping of WAM registers to machine registers, and took care to seamlessly integrate all three execution modes:

- Native code calling non-native code and vice versa.
- Native code returning to non-native code and vice versa.
- Native code backtracking into non-native code and vice versa.

The compilation was not a mere macro expansion of the WAM instructions. In particular, read and write mode instruction streams for compound term unification were kept separate and reasonably optimized. The target code was rich in calls to runtime routines, but operations like dereferencing, `allocate`, `deallocate`, stack trimming, and write mode unification were inline. Speedups by a factor of 3 over virtual code were not uncommon.

Later, Clark Haygood overhauled the native code compilers, the main inventions being the intermediate languages *SICStus Abstract Machine* (SAM) and *RISCified*

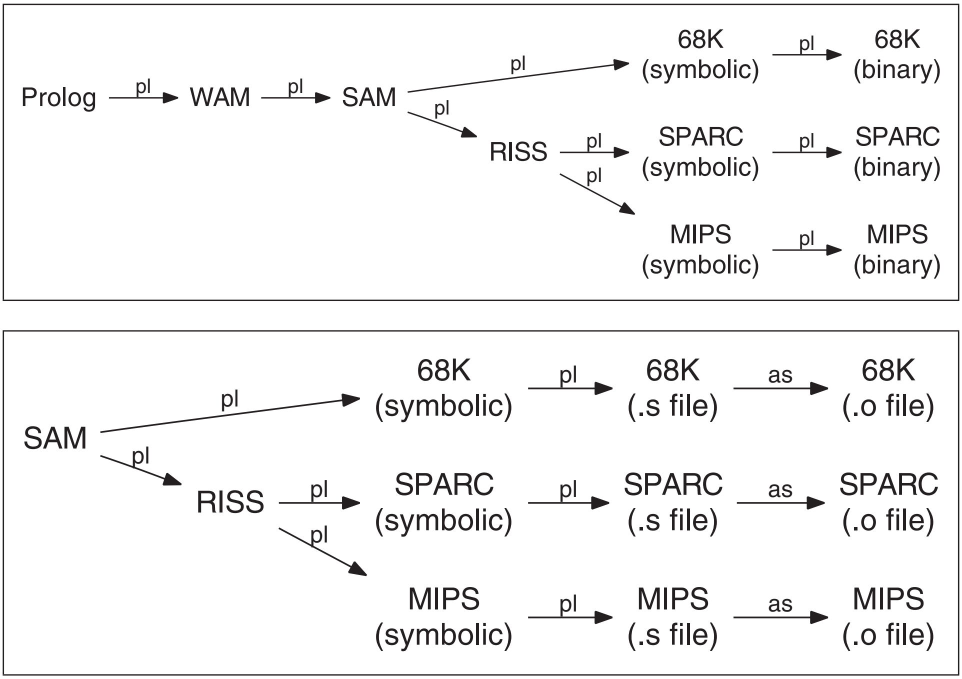


Fig. 7. Top: native code compilation path for Prolog code. Bottom: compilation path for the native code kernel. The standard assembler `as` is used in the native code kernel compilation path. Everywhere else, Prolog with the appropriate back-ends in C is used.

SAM (RISS) (Haygood 1994). SAM was not only an intermediate language; it was also a macro assembly language for the native code runtime kernel, containing all the runtime routines. He also added a MIPS back-end. The compilation paths from Prolog code, respectively, the runtime kernel to binary code are shown in Figure 7.

Eventually, the M68K and MIPS back-ends were dropped. The current SICStus Prolog 3 release only supports the SPARC back-end. Native code was completely dropped in SICStus Prolog 4 for lots of reasons, including:

- Amdahl's law, which tends to dominate as applications scale up.
- The inevitably large number of wheels that tend to get reinvented: assembler functionality, instruction scheduling, register allocation, etc.
- The difficulty of saving relocatable code in binary files and doing the relocation upon loading such files.
- Scanning native code for information listed in Section 4.3 is extremely cumbersome.
- The instruction cache easily gets confused if native code is modified on the fly.
- When an architecture goes extinct, a huge investment in code development is lost.

Of course, the potential of getting significant speedup of time-critical code is a baby that should not be thrown out with the bathwater. JIT compilation is a well-known scheme that avoids most of the above problems, and has been used for Prolog (da Silva and Costa 2007). We may well explore this approach in the future.

4.5 Managing dynamic code

Prolog makes a difference between *dynamic* predicates, whose clauses may be asserted, retracted, or inspected by the running program, and *static* predicates, where such operations are not allowed. In practice, dynamic predicates will be represented as interpreted in the sense of Section 4.1, since accessing and inspecting clauses are a central operation of the interpreter. There are several issues with interpreted and/or dynamic clauses. We now describe how we deal with them.

Indexing. SICStus Prolog uses the scheme for indexing of dynamic clauses on the first argument in linear space that was described in Demoen *et al.* (1989).

Semantics. The paper (O'Keefe and Lindholm 1987) proposed, and the ISO Prolog standard later confirmed, a semantics for dynamic clauses that are asserted or retracted during execution. The authors also invented a clever mechanism that allows to implement the semantics in almost constant time. The mechanism is based on a global clock register, two time-stamps per dynamic clause, and a time-stamp per dynamic choicepoint. Note that a retracted clause cannot in general be physically removed right away, as it might be in the scope of some dynamic choicepoint.

Dead clause reclamation. It is only safe to physically reclaim a clause when it is dead with respect to the global clock as well as all dynamic choicepoints. It would be logically correct to leave them around, but that would, of course, have a disastrous effect on performance. It is a non-trivial problem how to efficiently detect them and organize their reclamation. In O'Keefe and Lindholm (1987), the authors describe how to scan for and reclaim clauses in time linear in the number of the retracted clauses plus the number of choicepoints, but the question is when to do it. If it's done too often, the choicepoint stack will be scanned over and over again for nothing. If it's done too seldom, dead clauses accrete, degrading performance of dynamic code accesses. Our implementation is a variant of this scheme. To make it really work, we also found it necessary:

- to register retracted clauses in some data structure so that they can be found in $O(1)$ time,
- to recognize and speed up the case where there are *no* dynamic choicepoints, and
- to recognize cases when a retracted clause can be reclaimed immediately.

Clause references. Although not in the ISO standard, many Prologs provide a way to directly access a dynamic clause with a term known as a *db_reference*. This is provided by at least Ciao, Quintus, SICStus, SWI, and Yap Prologs. In SICStus Prolog, a db_reference has the form `'$ref'(`*i*`,`*j*`)` where *i* is an integer denoting the address of the clause, and *j* is an integer for validation purposes; see below. The built-in predicate `instance(+Ref,-Clause)` will take a db_reference and unify *Clause* with a brand new copy of the clause referred to. The built-in predicate `erase(+Ref)` will retract the clause, and so on. This feature, however, suffers from a dangling pointer problem. What to do if the clause has already been retracted? What if its memory has been reclaimed? We now outline how we address this problem.

- We maintain a global counter of asserted clauses and a global hash table that maps the address of a clause, i, to the value that the counter had when the clause was created, j.
- Db_references are validated by checking that the hash table still maps i to j.
- Hash table entries are removed when the corresponding clause is reclaimed.

This scheme ensures that db_references are unique, even if the memory used by one clause happens to be reused later by another one.

4.6 General memory management

The Prolog runtime system needs to dynamically allocate and free a huge amount of memory blocks of sizes varying from a few bytes to potentially several gigabytes. The natural choice would be to use the POSIX primitives `malloc()` and `free()`, and if code development had started today, that would have been the likely choice. But in the 1980s, the quality of their implementations left much to be desired. Worse, the quality and performance varied dramatically from platform to platform. Also, SICStus Prolog 3's requirement that certain memory areas be allocated in a certain region of the address space is incompatible with the standard `malloc()` and `free()`. So for historical and other reasons, SICStus Prolog has its own memory manager, the main features of which are the following:

- A two-layer architecture. The bottom layer requests memory from the operating system (O/S) and returns memory to it. Such requests are relatively infrequent and deal with *bigmems*, i.e. relatively large chunks of memory. The behavior of the bottom layer is subject to several tunables that the user can set. The top layer is the runtime system interface. It chops up the *bigmems* into smaller *mems* and keeps tracks of all free *mems*.
- When in use, a *mem* has no header or other memory overhead.
- The top layer keeps free *mems* in multiple unsorted chains, each chain corresponding to a specific range of sizes. This allows *mems* to be allocated in almost constant time.
- *Mems* are freed in constant time—no attempt is made to eagerly congeal adjacent free *mems*.
- From time to time, an $O(n \log n)$ algorithm to congeal all adjacent free *mems* is run, where n is the number of free *mems*.
- The built-in predicate `trimcore` orders the bottom layer to endeavor to return *bigmems* that are totally unused to the O/S.

The Prolog stacks tend to be the largest memory blocks by a wide margin. So the question arises, should a Prolog stack correspond to a *mem* or a *bigmem*? It was found that treating Prolog stacks as *mems* could cause severe memory fragmentation, so our current policy is to reserve a *bigmem* for each Prolog stack.

4.7 Interfacing foreign code

SICStus Prolog provides multiple interfaces for calling foreign code and vice versa. This is not the place to describe them all, but a few points are worth mentioning,

Instruction	**TYPE**
`push_`*TYPE* y	`float`
`push_result_`*TYPE*	`integer`
`receive_`*TYPE* y	`term`
`pop_`*TYPE* y	`atom`
	`string`
	`codes`
`open_foreign_call ...`	
`call_foreign` f, a	
`close_foreign_call`	

Fig. 8. The SICStus Prolog 4 instruction set for the Prolog-to-C interface. Top left: instructions for arguments and return values. `push_`*TYPE* y (for input arguments) and `push_result_`*TYPE* (for output arguments) populate the call frame. `pop_`*TYPE* y receives an output argument. `receive_`*TYPE* y receives the return value. Top right: the types handled by the API. Bottom left: instructions to manage the actual call. `open_foreign_call` allocates the call frame, `call_foreign` executes the call, and `close_foreign_call` deallocates the call frame. y denotes a permanent variable; f is the address of the foreign function, and a is the arity.

in particular, the fact that none of them exposes the internal Prolog data structures to the foreign code. A comparison of such interfaces for several implementations of Prolog can be found in Bagnara and Carro (2002).

Prolog-to-C interface. The interface provides a linking of Prolog predicates to C functions, which can succeed, fail, and raise exceptions. The interface does not allow to define non-deterministic predicates. The mapping of predicate and function names, as well as type conversions, is declared in Prolog facts.

In SICStus Prolog 3, a piece of C code is compiled from such facts for each such procedure. This piece of code implements all necessary checks and conversions on input arguments, calls the target functions, and converts and unifies the output arguments as necessary. Such code tends to have large chunks in common from one predicate to another.

In SICStus Prolog 4, the VM has instructions for such checks and conversions (see Figure 8). Foreign predicates are compiled to VM code instead of C code. This avoids the need to use a C compiler and allows more code to be shared. The only difficulty is the actual call to the foreign function, which expects its arguments to be passed in a way compliant with the platform ABI. In the presence of floating-point arguments, all call patterns cannot be precoded in the VM emulator. The `call_foreign` instruction, whose job is to do this call, is the only part of the system that is implemented in assembly code. Figure 9 shows an example of this compilation.

The basic interface handles simple C types only. In addition, the `structs` module provides a way to declare C structs in Prolog with name-based access to their fields and to pass struct pointers to C code (see Figure 10). The `objects` module is built on top of this feature.

```
extern long
ixkeys(SP_term_ref spec, SP_term_ref term, SP_term_ref list);
```

```
foreign(ixkeys, c_index_keys(+term, +term, -term, [-integer])).
```

```
open_foreign_call 4,3,c_index_keys/4,0
push_term y(0)
push_term y(1)
push_result_term
call_foreign ixkeys,4
pop_term y(2)
receive_integer y(3)
close_foreign_call
```

Fig. 9. Prolog-to-C interface example: binding the predicate `c_index_keys/4` to the `ixkeys()` function. Top: the header of the C function `ixkeys`. The type *SP_term_ref* provides a safe reference from C to a Prolog term. Middle: the foreign declaration, from which the VM instruction sequence is generated. Bottom: the SICStus Prolog 4 VM instruction sequence for `c_index_keys/4`: the first four instructions allocate and populate a call frame, one instruction executes the call, two instructions receive the output argument and the function value, and the last instruction deallocates the call frame.

C-to-Prolog interface. This interface provides services to start a query to a Prolog goal, request the next solution to a query, commit to the current solution of a query, and close a query. Exceptions can be raised in Prolog and inspected in C. Type check and conversion functions from Prolog to C and vice versa are available. C code accesses Prolog terms only via *SP_term_refs*, which are handles under the control of the memory manager so that, e.g. the garbage collector can function correctly with this interface. The C-to-Prolog and Prolog-to-C interfaces are re-entrant to arbitrary depth.

4.8 Source-linked debugging

The ability to step through program execution with the current line of code being highlighted is a crucial piece of debugger functionality, witness, e.g. `gdb` for C, and Prolog is no exception. This functionality was designed and implemented for SICStus Prolog around 1997 by Péter Szeredi. Using the same infrastructure, when an error exception is raised, SICStus Prolog tries to precisely pinpoint the responsible line of code. To support this functionality, an essential service is a way to read a Prolog term so that every subterm gets annotated with the line number on which it occurs. Another essential service is a data structure that can map a program location to a filename and a line number. We use one mechanism for interpreted code and another one for compiled (native or virtual) code.

Interpreted code. Having read a clause annotated as mentioned above, the clause is first asserted, obtaining a unique db_reference. We then create a *layout table* associated with this db_reference and store the filename in it. Treating the annotated clause as a tree, every path from its root to a leaf or internal node is stored in the

```
:- foreign_type
    intgr     = integer_32,
    bool      = enum([
                      false,
                      true
                ]),

    position = struct([
                      x:integer_32,
                      y:integer_32
                  ]),

    size      = struct([
                      width:integer_16,
                      height:integer_16
                  ]),

    mongo     = struct([
                      a:intgr,
                      b:integer_16,
                      c:integer_8,
                      d:unsigned_16,
                      e:unsigned_8,
                      f:float_32,
                      g:float,
                      h:atom,
                      i:string,
                      j:address,
                      k:array(81,integer_8),
                      l:size,
                      m:pointer(position),
                      n:pointer(belch),
                      o:bool,
                      p:integer,
                      q:pointer(mongo)
                  ]),

    uex       = union([
                      a:integer_32,
                      b:integer,
                      c:float
                 ]).
```

```
typedef int intgr;
typedef enum _bool {
        false,
        true
} bool;
typedef struct _position position;
struct _position {
        int x;
        int y;
};
typedef struct _size size;
struct _size {
        short width;
        short height;
};
typedef struct _mongo mongo;
struct _mongo {
        intgr a;
        short b;
        char c;
        unsigned short d;
        unsigned char e;
        float f;
        double g;
        SP_atom h;
        char *i;
        void *j;
        char (k)[81];
        size l;
        position *(m);
        belch *(n);
        bool o;
        long p;
        mongo *(q);
};
typedef union _uex uex;
union _uex {
        int a;
        long b;
        double c;
};
```

```
make_size(Width, Height, SizeStr) :-
    new(size, SizeStr),
    put_contents(SizeStr, width, Width),
    put_contents(SizeStr, height, Height).
```

Fig. 10. Left: a *foreign_type* declaration, a feature of the **structs** module. Right: the corresponding, automatically generated C header file containing type declarations. Bottom: a predicate that creates a **size** struct with given *Height* and *Width*.

layout table, together with its line number. A path is simply a list of numbers, e.g. $[3, 1, 2]$ means "take the 3rd argument of the 1st argument of the 2nd argument of the body". A custom compressed format is used so as to minimize space.

During execution of an interpreted clause, it maintains a virtual program counter, consisting of the db_reference of the clause plus the path to the current goal. This can be maintained very cheaply. To identify the line of code in the source, we just look up the associated layout table, retrieve the filename, and map the path to a line number.

Compiled code. For compiled code, we use a global B-tree that maps call sites to filenames and line numbers. Having read a clause annotated as mentioned above, the line number information is threaded through the compiler to its back-end, which actually stores the virtual code in memory. When the back-end is about to store a `call` or `execute` instruction, it adds the call site and associated filename and line number to the B-tree.

The VM emulator has a register holding the most recent call site. During tracing of compiled code, the emulator escapes to an entry-point of the debugger, passing the value of this register. Using the value, the associated filename and line number are looked up in the B-tree.

4.9 Operating system interface

Interfacing with the underlying O/S and with the file system is inherently a low-level activity. There are a lot of platform specific details and many operations that can report permanent or temporary failures. In addition, every O/S to which SICStus Prolog has been ported has idiosyncrasies, like operations that do not work for all types of streams or for streams but not process handles, or vice versa.

Prolog programming, on the other hand, is a high-level activity and we want to hide as much as possible of the underlying complexity and provide an interface to the O/S that "just works" and is portable across major platforms such as UNIX and Windows as well as to more exotic platforms where SICStus Prolog is sometimes used, such as mobile phones.

SICStus Prolog 3 interfaced to the O/S using the mechanism provided by the standard `stdio` library and its I/O operations. This design made sense at a time when characters were 7-bit ASCII, Microsoft Windows was irrelevant, threads did not exist, and (standardized) UNIX was not widely adopted. This lowest common denominator strategy eased portability but also severely limited the features that could be offered to the Prolog programmer.

With SICStus Prolog 4, we took the opportunity to redesign the interface to the underlying O/S and its I/O operations in a way that directly uses the native capabilities of the underlying O/S. This new interface was code named the *SICStus Prolog I/O library* (SPIO).

Non-blocking and interruptible operations. Some operations, especially I/O related, can take a long time or even block indefinitely. In threaded languages, like Java, it is common to handle this by simply spawning a new worker thread that handles the blocking operation, while the main program can either wait for the spawned thread to complete or can continue to run while the operation completes in the worker thread. Non-blocking and interruptable operations are crucial for multiple reasons:

- During development, the programmer must be able to interrupt a debugged program without terminating the process or otherwise corrupting its state.
- Server applications that need to keep responding to clients while at the same time performing I/O. They must be able to wait for either of several I/O operations to complete.
- SICStus Prolog has a feature called *asynchronous events*. Such events can be posted from C by an arbitrary thread of the process and will cause some

associated procedure, which can call Prolog, to be called by the Prolog main thread. When such an event is posted, any blocking I/O must be interrupted so that the event can be processed. Internally, asynchronous events are used for signal handling, the timeout facility, etc.

The standard C library provides no non-blocking operations and no way to wait for I/O to complete. In SICStus Prolog 3, some low-level routines were used together with `stdio` streams to provide waitable I/O. However, mixing `stdio` stream operations and O/S-level stream operations does not always work well or even correctly and does not work at all for some types of streams.

SICStus Prolog 4 does not use `stdio` for I/O. Instead, the use of native O/S routines allows us to wait on, and to do non-blocking I/O to, many kinds of O/S streams. Unfortunately, not all streams can be handled in this way. In fact, neither UNIX nor Windows provides non-blocking primitives that works for all, or even for most, I/O operations. Instead, SPIO uses worker threads in C, when needed, to provide the appearance of non-blocking and interruptible blocking operations. SPIO also provides the necessary operations for symbolic streams that do not use an underlying O/S stream, e.g. streams used for reading from a string. Thus, in Prolog code, and code that uses our C API, the high-level I/O functionality “just works”, regardless of the type of stream.

The availability of non-blocking streams makes it possible to wait for multiple streams to become readable or writable, thus enabling server applications to be written in Prolog. It also allows a debugged Prolog program to be interrupted, even if it is waiting for I/O to complete, without disturbing the I/O operation.

File system. File names with non-ASCII characters are handled differently by different operating and file systems. SPIO ensures that such file names behave correctly on systems like Mac OS X and Windows, which use Unicode file names. The standard UNIX way of handling file name encoding, based on a process-specific *locale*, is arguably broken by design and is largely ignored by SPIO. Instead, SPIO falls back on UTF-8 on such systems. SPIO permits file names and file paths to be as long as the underlying O/S can handle. Thus, the Prolog programmer is not restricted by the limited length supported by `stdio`.

Processes. SPIO handles all command line quoting and argument encoding necessary to launch processes on any supported O/S. SPIO also provides a common abstraction for process handles. The Prolog programmer does not need to care about its details, e.g. when passing a non-ASCII file name, with embedded spaces, as an argument to a launched program and then waiting for the subprocess to terminate.

Unicode and character encodings. A number of character encodings are provided for encoding and decoding file and stream contents. In many cases, SPIO can automatically detect the encoding used when reading data from a file.

Non-trivial character sets, such as Unicode, and non-trivial encodings, such as UTF-8, place special requirements on the implementation. For instance, it is possible to get an error when writing a character code that cannot be represented in the encoding used by the stream being written to. Such write errors raise an I/O exception. Similarly, an exception is raised if the file contains byte sequences that are invalid in the given encoding.

4.10 Attributed variables and constraint solvers

SICStus Prolog was possibly the first Prolog implementation to incorporate Holzbaur's seminal idea about attributed variables as a way to extend unification (Holzbaur 1992). Attributed variables are involved in two related mechanisms: (i) suspending a goal on a variable, i.e. until that variable has been bound, and (ii) a means of associating data with a variable while that variable is not yet bound. The first mechanism is implemented by the `freeze/2` predicate (Carlsson 1987) together with the generic overflow mechanism: binding the variable will set the generic overflow flag, and running the suspended goal will be handled by the generic overflow handler, as described earlier.

The second mechanism allows Prolog code to refer to attributes by names, which are declared per module. Once the attributes have been declared, attribute values can be attached to, modified, and detached from any variable. On backtracking, such changes are undone. A module that has declared some attributes may also define several local "hook" predicates, which add extra functionality, needed by constraint solvers in particular. The most important such predicate is `verify_attributes(`*`AVar`*`,`*`Value`*`,`*`Goals`*`)`, which extends default unification as follows. The predicate is called by the generic overflow handler whenever a variable *AVar* with attributes in the given module is about to be bound to a non-variable term or another attributed variable *Value*. It is expected to return in *Goals* a list of goals. The suspended unification resumes after the call to `verify_attributes/3`. Finally, the goals in *Goals* are called.

Figure 11 shows the internal representation of attributed variables, as used by the CLPFD solver. References to attributes by name in the Prolog code are translated by macro expansion to more direct accesses into this representation. When attribute values are attached, modified, or detached, destructive updates are used if they are safe. Otherwise, the internal representation is partly copied, and the value cell is bound to the copy. Once the value cell has been bound, the extra data structures are no longer reachable and so are subject to normal garbage collection.

Attributed variables are a crucial mechanism for constraint solvers in at least B, Ciao, ECLiPSe, GNU, SICStus, SWI, and Yap Prologs. SICStus Prolog has constraint solvers over Booleans (Carlsson 1991), rationals and reals (Holzbaur 1995), finite domains (Carlsson *et al.* 1997), and CHR (Schrijvers and Demoen 2004).

The finite domain solver has grown into a significant subsystem, comprising some 60,000 lines of C and 9,500 lines of Prolog code. The code is dominated by implementations of propagators for global constraints. Two attributes are used for a given domain variable x, as shown in Figure 11. Constraint propagation is driven by domain changes as opposed to variable bindings, and so the solver uses its own propagation loop instead of the `freeze/2` mechanism. The solver resides in the `clpfd` Prolog module, which also exploits some extensions to the Prolog system:

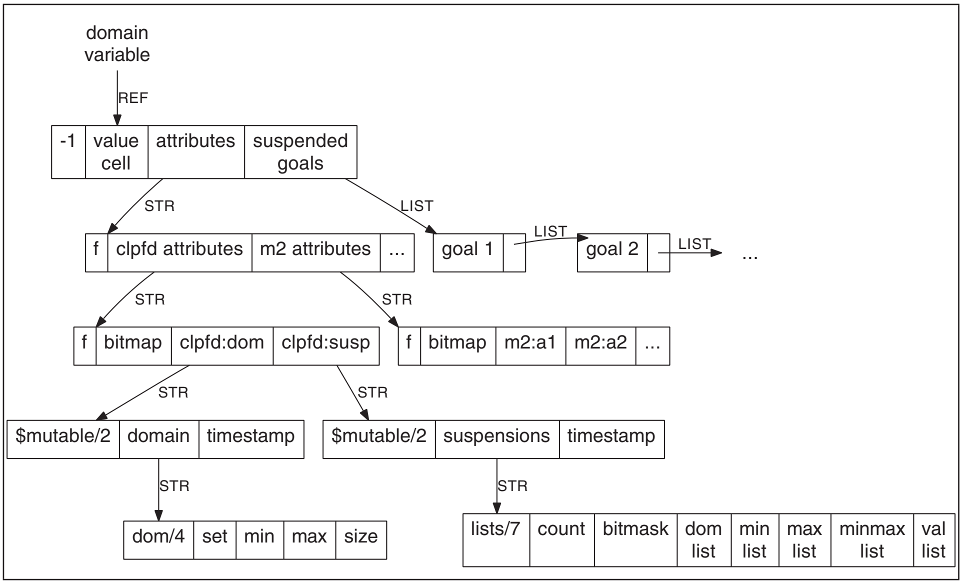


Fig. 11. Internal representation of domain variables, as a special case of attributed variables. The root is a reference to a value cell extended with an attributes slot and a suspended goals slot. The value cell is a self-reference while the variable is unbound. SICStus Prolog 3 used a dedicated tag for attributed variables, represented as three consecutive words (value cell, attribute slot, suspension list). SICStus Prolog 4 uses a generic variable tag, but the three words are preceded by a word containing −1. Together with an address comparison, this suffices to distinguish attributed variables from normal variables. This distinction needs to be made mainly when a variable is bound: if it is attributed, the generic overflow flag is set. The attributes slot contains a structure with one component per module (`m1`, `m2`, ...) that has declared attributes. Each such component is a structure with the actual attribute values, plus a bitmap indicating whether or not each given value is present. The suspended goals slot contains a plain list of goals, i.e. the `freeze/2` mechanism can suspend more than one goal on the same variable. The CLFPD solver uses two attributes, both holding a mutable, for a given domain variable x. `dom/4` stores its domain, while `lists/7` encodes the dependency lists, i.e. the set of constraints mentioning x as well as what kind of domain change should schedule each given constraint.

New predicate type. So-called indexical propagators (Van Hentenryck *et al.* 1991) for smallish constraints can be expressed in a special stack machine language. The solver provides a compiler into this language as well as an "assembly code" notation. Such propagators are seen by Prolog as predicates of a specific type—the constraint is posted simply by calling the predicate. Whenever the VM emulator encounters such a call, it escapes to `clpfd:solve/2`, the relevant solver entrypoint. The binary file format also needed to be extended to accommodate these predicates.

Global term references. The global constraint propagators are stateful. They maintain the constraint arguments as well as auxiliary data structures in a block of memory. This requires a way to store a persistent reference to a Prolog term in a C variable. The SP_term_ref mechanism mentioned earlier is, however, not persistent—an SP_term_ref becomes invalid as soon as control returns from C to Prolog. So a persistent variant of term references needed to be introduced.

Memory management. The solver C code has a license to penetrate the normal memory barriers, i.e. it can directly manipulate the internal term representation, bypassing the normal interface functions. In addition to global term references, the solver has other data structures that the Prolog memory manager needs to be aware of. Thus, when, e.g. a heap overflow occurs, the memory manager calls certain `clpfd` interface functions to ensure that the solver data structures are processed as need be.

4.11 Miscellaneous

SICStus Prolog uses a large number of implementation techniques that are shared with other implementations, Prolog, or otherwise. Some of these features can be traced back to a source; others are folklore. We now list a few of these points.

Cyclic term unifier. Without special care, the unification algorithm may not terminate on cyclic terms. In Colmerauer (1982), a simple method to avoid this problem is described. Briefly, before recursively unifying the ith argument of two compound terms p and q, the unifier temporarily sets the memory cell holding $p[i]$ to $q[i]$ (or vice versa). If the unifier later encounters the same pair of memory cells, it will see two identical terms instead of falling into infinite recursion. Before returning, the unifier restores all such modifications. We use the same technique in the term comparison algorithm that determines the relation between two given terms in the standard order of terms.

Mutable terms. SICStus Prolog used to have a non-logical feature called `setarg(`I`,`P`,`X`)`. The effect is to set the Ith argument of the compound term P to X, restoring the old value on backtracking. To support restoring, the trail must be generalized to accommodate such old values and their destinations. This feature exists in at least B, Bin, Ciao, ECLiPSe, GNU, SWI, and Yap Prologs. Around 1995, we replaced `setarg/3` by a new abstract datatype *mutable term* with operations to create such a term and to get and update its value. The implementation is based on Aggoun and Beldiceanu (1990): each mutable term has a time-stamp, which indicates when the value was last updated. The point is, if no choicepoint has been pushed between two updates, the second update does not need to be trailed. We also extended the variable shunting algorithm (Carlsson and Sahlin 1990) to compress reset chains for mutables. We treat mutable terms as non-ground, no matter what the current value is. Subsequently, mutable terms have been adopted by Yap Prolog.

Bignums. Bignums are available in at least Ciao, ECLiPSe, SICStus, SWI, and Yap Prologs. We do not use any publicly available multi-precision libraries, since when our code was developed, none of the available libraries was compatible with our particular memory management requirements.

Asserting clauses and copying terms. Internally, these two operations are very similar and share much of the code. Both use variants of Cheney's algorithm (Cheney 1970). The main difference is in the output: the assert operation creates an interpreted clause, i.e. a kind of blue-print from which a brand new clause copy can be built in linear time, whereas the copy operation creates a new term directly.

Object-oriented programming. Although the combination of logic programming and object-oriented programming was never a research topic at SICS, SICStus Prolog does provide such modules. The SICStus Prolog 3 `objects` module was designed with an emphasis on knowledge representation. It was based on the notions of prototypes, inheritance, and delegation. The implementation piggybacked on the module system: a named object was represented by the Prolog module with the same name, resulting in an obvious risk for name clashes. Furthermore, the module data structures and primitives had to be extended in order to provide all the services that the object system needed.

The SICStus Prolog 4 `objects` module is based on the notions of classes and inheritance. The emphasis is on efficiency. The implementation is 100% based on source-to-source compilation and does not rely on or extend the module system. A detailed description can be found in Saab and Schachte (1995).

Exceptions, or `catch` and `throw`. We use the implementation proposed in Demoen (1989).

Cleaning up, or `call_cleanup`. A very common situation in programming is the following. Some algorithm needs to run, holding some resources. Those resources must be freed afterwards, no matter whether or not the algorithm terminates normally. Common Lisp provides a primitive for this purpose:

`(unwind-protect protected cleanup)`

which evaluates the form *protected* in a context where the form *cleanup* is guaranteed to be executed when and if control leaves the form *protected* by *any* means. Finally, the value of *protected* is returned from the `unwind-protect` form.

Around 1997, the first author introduced an analogous construct into SICStus Prolog, naming it `call(Goal,Cleanup)`. Richard O'Keefe criticized him for this choice of name, which clashes with the multiple argument generalization of `call/1`. Richard was absolutely right, of course, and the construct was later renamed to `call_cleanup/2`, its present name. Subsequently, it has found its way into at least B, ECLiPSe, SWI, XSB, and Yap Prologs.

`call_cleanup/2` guarantees the execution of *Cleanup* if *Goal* succeeds determinately, fails, or raises an exception. Also, if *Goal* succeeds with some alternatives outstanding, and those alternatives are removed by a cut or an exception, *Cleanup* is executed. The implementation is composed of the following elements:

- *Cleanup* goals are placed on the trail. The general backtracking mechanism simply executes such goals as they are encountered on failure or exception.
- A bit $c(b)$ is reserved in every choicepoint b, denoting the fact that there may be a pending *Cleanup* goal when b equals the current choicepoint B.
- When `call_cleanup` is called, $b_0 \leftarrow B$, $c(b_0)$ is set, and *Cleanup* is pushed on the trail.

- On non-deterministic exit from `call_cleanup`, $c(b)$ is set for all choicepoints b that predate b_0, so as to ensure that *Cleanup* is run if and when a cut back to b_0 or beyond occurs.
- On deterministic exit from `call_cleanup`, and upon execution of a cut, if $c(B)$ is set, the generic overflow flag is set.
- If the generic overflow handler finds a cleanup goal in the current trail segment, it arranges for it to be run. It clears $c(B)$ if appropriate.

5 Development environment

5.1 *Background*

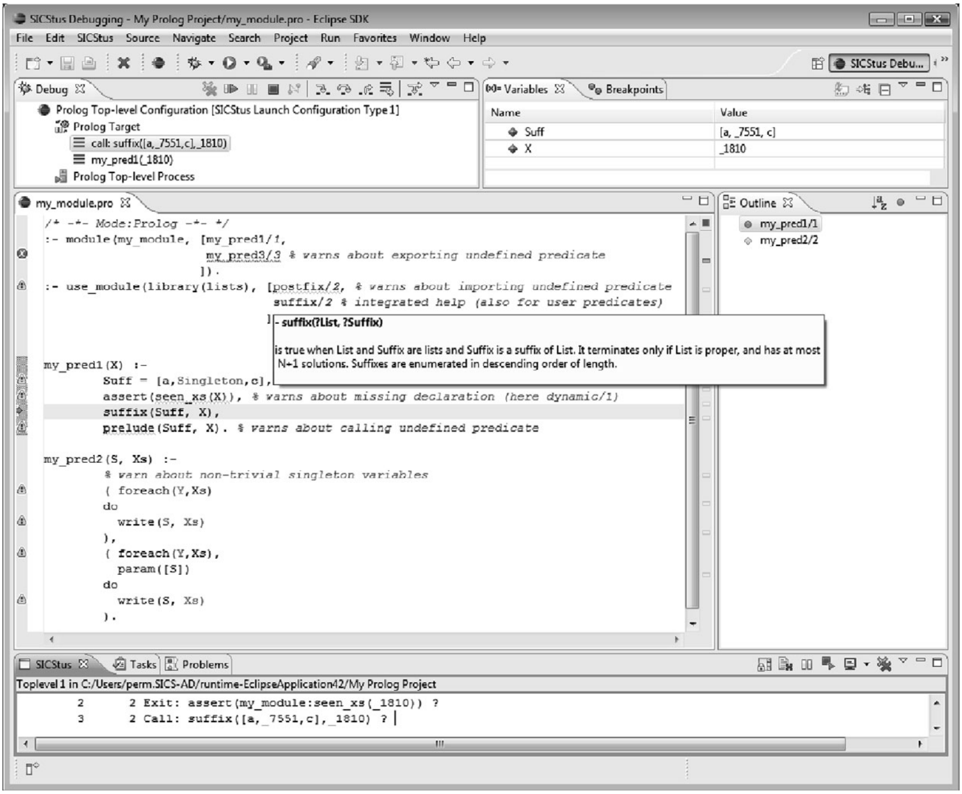
Since early on, SICStus Prolog has had an Emacs-based development environment, with syntax highlighting, source-linked debugging, links to the documentation, and more. However, both our Emacs-based development environment and Emacs itself lacks many of the features that users have come to expect from a modern integrated development environment (IDE), such as:

Parser. Anything but the most trivial language support requires a proper parser, including support for operator directives. Without a parser, it is not possible to get much more advanced than showing variables in italics. The parser must be part of the IDE, as running it in a separate process would likely cause intolerable response times.

Semantic analysis. The dynamic nature of Prolog is an advantage for the developer but makes it difficult for the compiler to provide diagnostics. Traditionally, like most other Prolog implementations, SICStus Prolog warns about syntax errors but provides little in terms of semantic diagnostics. Semantic diagnostics are mostly limited to local issues such as singleton variables and discontiguous clauses. While SICStus Prolog comes with several useful tools that provide more advanced diagnostics, e.g. for determinacy checking and cross referencing, these tools must be run separately, which is inconvenient. On the other hand, an IDE, especially if it has knowledge about the set of files that makes up a Prolog program, can provide the same and more functionality than the existing tools, while the user edits or browses the program files. An IDE can also give feedback from syntactic and semantic analysis in a more useful way than what is possible with separate tools, e.g. by highlighting undefined predicate calls or incorrect predicate arguments directly in the source code editor.

Code refactoring. Code refactoring means automatic and usually global changes to a program, preserving the semantics of the program. Typical examples for Prolog are: renaming a predicate, reordering the arguments of a predicate, or adding arguments to a predicate, automatically updating all callers.

Scalability. Our commercial customers have applications comprising hundreds of modules adding up to several hundred thousand lines of code. This fact stresses the importance that our IDE be scalable to such code sizes.



Prolog Top-level Configuration [SICStus Launch Configuration Type 1]
Prolog Target
call: suffix([a,_7551,c],_1810)
my_pred1(_1810)
Prolog Top-level Process

Name	Value
Suff	[a, _7551, c]
X	_1810

```
/* -*- Mode:Prolog -*- */
:- module(my_module, [my_pred1/1,
                      my_pred3/3 % warns about exporting undefined predicate
                     ]).
:- use_module(library(lists), [postfix/2, % warns about importing undefined predicate
                               suffix/2 % integrated help (also for user predicates)
                              ]


my_pred1(X) :-
        Suff = [a,Singleton,c],
        assert(seen_xs(X)), % warns about missing declaration (here dynamic/1)
        suffix(Suff, X),
        prelude(Suff, X). % warns about calling undefined predicate

my_pred2(S, Xs) :-
        % warn about non-trivial singleton variables
        ( foreach(Y,Xs)
        do
          write(S, Xs)
        ),
        ( foreach(Y,Xs),
          param([S])
        do
          write(S, Xs)
        ).
```

- suffix(?List, ?Suffix)

is true when List and Suffix are lists and Suffix is a suffix of List. It terminates only if List is proper, and has at most N+1 solutions. Suffixes are enumerated in descending order of length.

my_pred1/1
my_pred2/2

Toplevel 1 in C:/Users/perm.SICS-AD/runtime-EclipseApplication42/My Prolog Project

```
       2      2 Exit: assert(my_module:seen_xs(_1810)) ?
       3      2 Call: suffix([a,_7551,c],_1810) ? |
```

Fig. 12. SICStus Prolog IDE window. Top left: debugger pane. Top right: variable pane. Middle left: source code pane with highlighting and pop-ups. Middle right: outline pane. Bottom: toplevel pane.

Implementation. We have implemented our IDE in Eclipse, an application and IDE framework written in Java. Eclipse has already proved itself as a foundation for powerful IDEs for many programming languages. Using Eclipse as the basis for an IDE also gives many features for free, such as portability, integration with common revision control systems and support for multiple programming languages in the same IDE. Using Eclipse will also make it possible to integrate other tools such as profiler and constraint visualizers into the IDE. In addition, Eclipse makes it possible for us to package our IDE as a standalone product with a completely Prolog-centric appearance, if needed.

A first version of the IDE, with working name SPIDER, was released together with SICStus Prolog 4.1, in December 2009. It is still in beta and lacks some of the planned features but it is already quite useful and its analysis functionality has helped us identify and fix several defects in our own code.

5.2 SPIDER in action

Figure 12 shows some of the features of SPIDER in action. We now discuss some of its central features:

Editor. While editing, SPIDER continuously re-parses the code and annotates the text with warnings and semantic highlightings. Warnings include: calls to undefined predicates, import or export of predicates not defined in module, assert of predicate not declared dynamic, not using `use_module/[1,2]` when loading a module file, singleton variables.

Semantic highlightings include a special appearance of first and single occurrences of a variable. This is done also in the context of disjunction and logical loops (Schimpf 2002), when the variable may have more than one *semantically* first or singleton occurrence.

Calls to undefined predicates are highlighted, including when they appear as arguments to meta predicates.

The editor provides completion of predicate names and documentation pop-up when the mouse pointer is hovering over a predicate name. The documentation is formatted on the fly for user written code and there is an integrated browser for the SICStus Prolog product documentation.

The definition of a user-defined or built-in library predicate or module can be opened with a single click or keyboard command.

Toplevel. The toplevel implements the traditional terminal interface and provides a familiar interface, including the traditional debugger.

Debugger. The debugger shows an ancestor stack, local variable bindings, and direct access to some common debugger control commands, like step into, step over, and redo. The traditional terminal-based debugger interface is active at all times in the toplevel, so the power user is free to use that, if desired.

The debugger and editor together provide a point and click interface for setting line breakpoints and spypoints. It is also possible to temporarily disable all breakpoints and to save breakpoints across debugging sessions.

The debugger and toplevel can attach to a running SICStus Prolog process that may be running on another machine (and platform) than the IDE. This is useful for those that embed SICStus Prolog as part of a larger program or system.

Future features. A prerequisite of many types of program analysis is complete information about all source code in a program. This requires not only knowing which files make up the program but also how these files load each other, especially when modules are distributed among multiple non-module files. SPIDER, like many other Eclipse-based language environments, delegates this task to a separate *indexer*, which updates the information as files are modified. The indexer functionality of SPIDER is currently being implemented. When this work is completed, we plan to add features such as call hierarchy and determinacy analysis, providing similar functionality as that of our current `spxref` and `spdet` tools, but with immediate feedback as the program is modified. The indexer is also a requirement for refactoring and other planned features that currently have no counterpart among the existing SICStus Prolog tools.

6 Applications

SICStus Prolog is being used on a 24/7 basis in major applications comprising hundreds of modules adding up to several hundred thousand lines of code. It is a pity, but for reasons of customer confidentiality, we are not at liberty to describe some of the most impressive ones. Anyway, we now briefly describe some applications for which permission has been generously granted, or where the information is publicly available.

Speech recognition. Clarissa[2], a fully voice-operated procedure browser was developed by the NASA Intelligent Systems Division. On the International Space Station (ISS), astronauts execute thousands of complex procedures to maintain life support systems, check out space suits, and conduct science experiments, among their many tasks. Today, when carrying out these procedures, an astronaut usually reads from a PDF viewer on a laptop computer, which requires them to shift attention from the task to scroll pages. Clarissa enables astronauts to be more efficient and to give full attention to the task while they navigate through complex procedures using spoken commands.

Clarissa was implemented mainly using SICStus Prolog and a speech recognition toolkit provided by Nuance Communications. Application-specific spoken command grammars were constructed using the SICStus Prolog based Regulus platform (Rayner *et al.* 2006).

Telecom. Ericsson Network Resource Manager (NRM) provides the capabilities for configuring and managing complex multi-vendor IP Backbone networks. NRM assists the operator in making decisions when planning, configuring, and making configuration changes.

The modeling part of the NRM software, an expert tool assisting the network operator, was implemented in SICStus Prolog. The constructed network model, created by analyzing the actual router configurations, is used both for showing a graphical representation and for validating the network.

Biotech. A dispensation order generation algorithm for Pyrosequencing's sequence analysis instruments, using constraint programming in SICStus Prolog (Carlsson and Beldiceanu 2004a, 2004b). The algorithm can be described as a compiler, which calculates an instruction sequence based on an input specification. Applications include genetics, drug discovery, microbiology, SNP and mutation analysis, forensic identification using mtDNA, pharmacogenomics, and bacterial and viral typing.

Logistics. One of the products of RedPrairie Corporation, a leading provider of real-time logistics solutions, is a real-time optimization engine, COPLEX. The kernel of the engine is written in SICStus Prolog using its finite domain constraint solver library.

Data mining. Compumine AB's data mining software Rule Discovery System (RDS$^{\text{TM}}$) is a tool for generation of reliable, accurate, and interpretable rule based

---

[2] `http://ti.arc.nasa.gov/project/clarissa/`

prediction models by automatically searching databases for significant patterns and relationships. RDS^{TM} was implemented in SICStus Prolog and has been successfully applied to problems in a large number of data intensive areas such as pharmaceutical research, language technology, and engineering.

Business rules: The 360^o *Fares system.* The paper (Wilson 2005) describes an application running the 360^o Fares System. It is one of the largest and most profitable Prolog applications written. Prolog is the business-rule component in a multi-component application that includes network, user interface, and security data access tiers.

Biomedical text search. MetaMap[3] was developed by Alan Aronson at the National Library of Medicine (NLM) to map biomedical text to the UMLS Metathesaurus or, equivalently, to discover Metathesaurus concepts referred to in text. MetaMap uses a knowledge intensive approach based on symbolic, natural language processing (NLP) and computational linguistic techniques. MetaMap is one of the foundations of NLM's Medical Text Indexer (MTI), which is being applied to both semiautomatic and fully automatic indexing of biomedical literature at NLM. MetaMap was first implemented in Quintus Prolog and is being ported to SICStus Prolog.

Safety-critical applications. SPARK[4] (Barnes 2003) is a high-level programming language and toolset designed for writing software for high integrity applications. SPARK enables the application of formal verification techniques in a segregated monitor architecture, ensuring rapid compliance. The SPARK toolset comes in a GPL version and includes a theorem prover implemented in SICStus Prolog.

7 Conclusion

Now that the system has been around for nearly 25 years, a relevant question to ask is: what are the key good and less good design decisions? We now try to give some answers.

First of all, there hardly were any truly bad decisions. Some decisions, like endeavoring into compiling to native code, meant huge amounts of work for platforms that eventually went extinct. But at the same time, good research was done, important lessons were learned, and pieces of technology were developed that could be reused in a JIT compiler, for example.

One questionable decision was the fact that SICStus Prolog 3 supported two dialects, “classic” and ISO, in the same system, and even let the user dynamically switch between the two. This made it awkward to document certain built-in predicates, like `atom_chars/2`, whose semantics differs from dialect to dialect, as well as all the other, subtler differences. It also made it quite a challenge to ensure that all library modules would run in both dialects. We are not aware of any other programming system, Prolog or otherwise, that provides this degree of freedom. Of course, this situation stemmed from the fact that the ISO standard was published

---

[3] `http://metamap.nlm.nih.gov/`

[4] `http://www.praxis-his.com/spark.aspx`

quite late, when a lot of application code had already been written by users as well as implementers. We wanted to promote the ISO standard, but at the same time, we had no right to break people's existing code. Our solution to this dilemma was a dual dialect system.

A lesson that keeps getting reiterated is the importance of backward compatibility. For obvious reasons, users are very unforgiving to changes in behavior of the programming system, even if it concerns minor points that are not necessarily specified in detail in the documentation. For example, at one time, we were flamed by a customer for changing the printed appearance of certain floating-point numbers although the old and new appearances were both legal syntax. There is no escape from this issue, and the Prolog standardization committee is well advised to bear it in mind. The first author knows from first hand experience as a committee member how tempting it is to start "cleaning up" or "redesigning" parts of the language. Such ambitions can be commendable, but at this stage they are only viable if full backward compatibility can be preserved.

Finally, the quality of the POSIX primitives `malloc()` and `free()` in today's operating systems is probably high enough to make a dedicated memory manager redundant. However, we do have customers that depend on the ability to control memory allocation with tunables, and it is not clear whether their applications would run with tolerable performance without a tunable, dedicated memory manager.

But by and large, *je ne regrette rien.*

Acknowledgements

A large number of people have contributed to the development of SICStus Prolog, including: Jonas Almgren, Johan Andersson, Stefan Andersson, Nicolas Beldiceanu, Tamás Benkő, Kent Boortz, Per Brand, Göran Båge, Per Danielsson, Joakim Eriksson, Jesper Eskilson, Niklas Finne, Lena Flood, György Gyaraki, Dávid Hanák, Seif Haridi, Ralph Haygood, Christian Holzbaur, Key Hyckenberg, Carl Kesselman, Péter László, Carl Nettelblad, Greger Ottosson, Dan Sahlin, Peter Schachte, Rob Scott, Thomas Sjöland, Péter Szeredi, Tamás Szeredi, Johan Widén, Magnus Ågren, Emil Åström, and the authors. The Industrialization of SICStus Prolog (1988-1991) was funded by Ericsson Telecom AB, NobelTech Systems AB, Infologics AB, and Televerket, under the National Swedish Information Technology Program IT4. The development of SICStus Prolog 3 (1991-1995) was funded in part by Ellemtel Utvecklings AB. We thank Magnus Ågren and the anonymous referees for their constructive comments.

References

Aggoun, A. and Beldiceanu, N. 1990. Time stamps techniques for the trailed data in constraint logic programming systems. In *Proc. of the* $8^{ème}$ *Séminaire de Programmation en Logique (SPLT)*, S. Bourgault and M. Dincbas, Eds, 487–510.

Aho, A. V., Sethi, R. and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading. ISBN 0-201-10088-6.

Ali, K. and Karlsson, R. 1990a. The Muse or-parallel Prolog model and its performance. In *Proc. of the North American Conference on Logic Programming (NACLP '90)*. MIT Press, Cambridge, 757–776.

Ali, K. and Karlsson, R. 1990b. The Muse approach to or-parallel Prolog. *International Journal of Parallel Programming 19*, 2, 129–162, Springer.

Amdahl, G. 1967. Validity of the single-processor approach to achieving large scale computing capabilities. In *Proc. of the AFIPS Spring Joint Computer Conference (AFIPS '67)*, vol. 30. AFIPS Press, Arlington, 483–485.

Appleby, K., Carlsson, M., Haridi, S. and Sahlin, D. June 1988. Garbage collection for Prolog based on WAM. *Communications of the ACM 31*, 6, 719–740.

Bagnara, R. and Carro, M. 2002. *Foreign Language Interfaces for Prolog: A Terse Survey*. Quaderno 283, Dipartimento di Matematica, Università di Parma, Italy.

Barnes, J. 2003. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, Reading. ISBN 0-321-13616-0.

Bevemyr, J. and Lindgren, T. 1994. A simple and efficient copying garbage collector for Prolog. In *Proc. of the International Symposium on Programming Language Implementation and Logic Programming (PLILP '94)*, M. Hermenegildo and J. Penjam, Eds. Lecture Notes in Computer Science, vol. 844. Springer, Berlin, 88–101.

Brady, M. 2005. *Open Prolog: A Structure-Sharing Prolog for the Macintosh*. Ph D Thesis. Trinity College Dublin, Dublin.

Carlsson, M. 1987. Freeze, indexing, and other implementation issues in the WAM. In *Proc. of the International Conference on Logic Programming (ICLP '87)*, Melbourne, J.-L. Lassez, Ed. MIT Press Series in Logic Programming. MIT Press, Cambridge, 40–58.

Carlsson, M. 1989. On the efficiency of optimising shallow backtracking in compiled Prolog. In *Proc. of the International Conference on Logic Programming (ICLP '89)*, Lisbon, G. Levi and M. Martelli, Eds. MIT Press Series in Logic Programming. MIT Press, Cambridge, 3–16.

Carlsson, M. 1991. *Boolean Constraints in SICStus Prolog*. SICS Research Report T91:09. Swedish Institute of Computer Science.

Carlsson, M. and Beldiceanu, N. 2004a. Dispensation order generation for pyrosequencing. In *Proc. of the Asia Pacific Bioinformatics Conference (APBC '04)*, Y.-P. P. Chen, Ed. Conferences in Research and Practice in Information technology, vol. 29. Australian Computer Society, Dunedin, New Zealand.

Carlsson, M. and Beldiceanu, N. 2004b. Multiplex dispensation order generation for pyrosequencing. In *Proc. of the Workshop on CSP Techniques with Immediate Application (CP '04)*.

Carlsson, M., Ottosson, G. and Carlson, B. 1997. An open-ended finite domain constraint solver. In *Proc. of the International Symposium on Programming Language Implementation and Logic Programming (PLILP '97)*, H. Glaser, P. Hartel, and H. Kuchen, Eds. Lecture Notes in Computer Science, vol. 1292. Springer, Berlin, 191–206.

Carlsson, M. and Sahlin, D. 1990. Variable shunting for the WAM. In *Proc. of the NACLP Post-Conference Workshop on Logic Programming Architectures and Implementations (NACLP '90)*, J. Mills, Ed. Also available as SICS Research Report R91:07.

Cheney, C. 1970. A nonrecursive list compacting algorithm. *Communications of the ACM 13*, 11, 677–678.

Chung, Y., Moon, S.-M., Ebcioğlu, K. and Sahlin, D. 2000. Reducing sweep time for a nearly empty heap. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. ACM, New York, NY, USA, 378–389.

COLMERAUER, A. 1982. Prolog and infinite trees. In *Logic Programming*, K. Clark and S.-Å. Tärnlund, Eds. Academic Press, London, 231–251.

COLTON, C. 1825. *The Lacon*, vol. 1. Longman, Hurst, Rees, Orme, Brown and Green. Paternoster-row, London.

DA SILVA, A. F. AND COSTA, V. S. 2007. Design, implementation, and evaluation of a dynamic compilation framework for the YAP system. In *Proc. of the International Conference on Logic Programming (ICLP '07)*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, Berlin, 410–424.

DEMOEN, B. 1989. *A 20' Implementation of Catch and Throw in WAM*. Report CW 96. Department of Computer Science, K. U. Leuven, Leuven, Belgium.

DEMOEN, B., MARIËN, A. AND CALLEBAUT, A. 1989. Indexing Prolog clauses. In *Proc. of the North American Conference on Logic Programming (NACLP '89)*, E. Lusk and R. Overbeek, Eds. MIT Press, Cambridge, 1001–1012.

ESKILSON, J. AND CARLSSON, M. 1998. SICStus MT—A multithreaded execution environment for SICStus Prolog. In *Proc. of the International Symposium on Programming Language Implementation and Logic Programming (PLILP '98)*, C. Palamidessi, H. Glaser, and K. Meinke, Eds. Lecture Notes in Computer Science, vol. 1490. Springer, Berlin, 36–53.

FARKAS, Z., KÖVES, P. AND SZEREDI, P. 1994. MProlog: an implementation overview. In *Implementations of Logic Programming Systems*, E. Tick and G. Succi, Eds. Kluwer Academic Publishers, Boston, 103–117.

GORLICK, M. AND KESSELMAN, C. 1987. Timing Prolog programs without clocks. In *Proc. of the Symposium on Logic Programming (SLP '87)*. IEEE Computer Society, Los Angeles, 426–432.

GUPTA, G., PONTELLI, E., ALI, K., CARLSSON, M. AND HERMENEGILDO, M. 2001. Parallel execution of Prolog programs. *ACM Transaction on Programming Languages and Systems 23*, 4, 1–131.

HAYGOOD, R. 1994. Native code compilation in SICStus Prolog. In *Proc. of the International Conference on Logic Programming (ICLP '94)*. MIT Press, Cambridge, MA, USA, 190–204.

HOLZBAUR, C. 1992. Metastructures versus attributed variables in the context of extensible unification. In *Proc. of the International Symposium on Programming Language Implementation and Logic Programming (PLILP '92)*, M. Bruynooghe and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 631. Springer, Berlin, 260–268.

HOLZBAUR, C. 1995. *ÖFAI clp(q,r) Manual*, 1.3.3 ed. Austrian Research Institute for Artificial Intelligence, Vienna. TR-95-09.

LUSK, E., BUTLER, R., DISZ, T., OLSON, R., OVERBEEK, R., STEVENS, R., WARREN, D., CALDERWOOD, A., SZEREDI, P., BRAND, P., CARLSSON, M., CIEPIELEWSKI, A. AND HAUSMAN, B. 1990. The Aurora or-parallel Prolog system. *New Generation Computing 7*, 2–3, 243–271.

NÄSSÉN, H., CARLSSON, M. AND SAGONAS, K. 2001. Instruction merging and specializing in the SICStus Prolog virtual machine. In *Proc. of the International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '01)*. ACM, New York.

O'KEEFE, R. AND LINDHOLM, T. 1987. Efficient implementation of a defensible semantics for dynamic Prolog code. In *Proc. of the International Conference on Logic Programming (ICLP '87)*. MIT Press, Cambridge, 21–39.

RAYNER, M., HOCKEY, B. A. AND BOUILLON, P. 2006. *Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler*. CSLI studies in computational linguistics. Stanford University Center for the Study of language and information, Stanford, California.

SAAB, G. AND SCHACHTE, P. 1995. Efficient object-oriented programming in Prolog. In *Logic Programming: Formal Methods and Practical Applications*. Studies in Computer Science and Artificial Intelligence, vol. 11. Elsevier Science, Amsterdam, 205–243.

Schimpf, J. 2002. Logical loops. In *Proc. of the International Conference on Logic Programming (ICLP' 02)*, P. J. Stuckey, Ed. Lecture Notes in Computer Science, vol. 2401. Springer, Berlin, 224–238.

Schrijvers, T. and Demoen, B. 2004. The K.U. Leuven CHR system: Implementation and application. In *Proc. First Workshop on CHR: Selected Contributions*, T. Fruehwirth and M. Meister, Eds. University of Ulm, Ulm, 1–5.

Tarau, P. and Neumerkel, U. 1994. A novel term compression scheme and data representation in the BinWAM. In *Proc. of the International Symposium on Programming Language Implementation and Logic Programming (PLILP '94)*, M. V. Hermenegildo and J. Penjam, Eds. Lecture Notes in Computer Science, vol. 844. Springer, Berlin, 73–87.

Taylor, A. 1991. High performance Prolog on a RISC. *New Generation Comput. 9*, 3–4, 221–232.

Van Hentenryck, P., Saraswat, V. and Deville, Y. 1991. *Constraint processing in cc(FD)*. Technical report, Brown University, Providence.

Van Roy, P. and Despain, A. M. 1992. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer 25*, 1, 54–68.

Warren, D. 1979. Prolog on the DECsystem-10. In *Expert Systems in the Micro-Electronic Age*, D. Mitchie, Ed. Edinburgh University Press, Edinburgh.

Warren, D. 1983. *An Abstract Prolog Instruction Set*. Technical note 309. Artificial Intelligence Center, SRI International, Palo Alto.

Wilson, W. G. 2005. Use of logic programming for complex business rules. In *Proc. of the International Conference on Logic Programming (ICLP '05)*. Lecture Notes in Computer Science, vol. 3668. Springer, Berlin, 14–20.