SWI-Prolog version 7 extensions

Jan Wielemaker

Web and Media group, VU University Amsterdam,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands,
J.Wielemaker@vu.nl

Abstract. SWI-Prolog version 7 extends the Prolog language as a general purpose programming language that can be used as 'glue' between components written in different languages. Taking this role rather than that of a domain specific language (DSL) *inside* other IT components has always been the design objective of SWI-Prolog as illustrated by XPCE (its object oriented communication to the OS and graphics), the HTTP server library and the many interfaces to external systems and file formats. In recent years, we started extending the language itself, notably to accommodate expressing syntactic constructs of other languages such a HTML and JavaScript. This resulted in an extended notion of operators and quasi quotations. SWI-Prolog version 7 takes this one step further by extending the primitive data types of Prolog. This article describes and motivates these extensions.

1 Introduction

Prolog is often considered a *DSL*, a *Domain Specific Language*. This puts the language in a position similar to e.g., SQL, acting as a component in a larger application which takes care of most of the application and notably the interfaces to the outside world (be it a graphical interface targeting at humans or a machine-to-machine interface). This point of view is illustrated by vendors selling their system as e.g., *Prolog + Logic Server* (Amzi!), the interest in Prolog-in-some-language implementations as well as by the many questions about embedding interfaces that appear on mailing lists and forums such as stackoverflow.[1]

SWI-Prolog always had the opposite viewpoint, proposing Prolog as a 'glue' (or scripting language) suitable for the overall implementation of applications. As a consequence, SWI-Prolog has always provided extensive libraries to communicate to other IT infrastructure, such as graphics (XPCE), databases, networking, programming languages and document formats. We believe this is a productive approach for the following reasons:

– Many external entities can easily be wrapped in a Prolog API that provides a neat relational query interface.
– Given a uniform relational model to access the world makes reasoning about this world simple. Unlike classical relational query languages such as SQL though, Prolog rules can be *composed* from more elementary rules.

[1] http://stackoverflow.com/

– Prolog is one of the few languages that can integrate application logic without suffering from the *Object-Relational impedance mismatch*.[2] [5]
– Prolog naturally blends with constraint systems, either written in Prolog or external ones.
– Many applications have lots of little bits of logic that is way more concisely and readably expressed in terms of Prolog rules than in imperative if-then-else rules.
– Prolog's ability to write application-specific DSLs is highly valuable for developing larger applications.
– Prolog's reflective capabilities simplify many application specific rewriting, validation and optimisation requirements.
– Prolog's dynamic compilation provides a productive development environment.
– Prolog's simple execution order allows for writing simple sequences of actions.

For a long time, we have facilitated this architecture within the limitations of classical Prolog, although we lifted several limits that are common to Prolog implementations. For example, SWI-Prolog offers atoms that can hold arbitrary long Unicode strings, including the code point '0', which allows applications to represent text as well as 'binary blobs' as atoms. Atom garbage collections ensures that such applications can process unbounded amounts of data without running out of memory. It offers unbounded arity of compound terms to accommodate arrays and it offers multi-threading to allow for operation in threaded server environments. SWI-Prolog's support of data types and syntax was considered satisfactory for application development. Over the past (say) five years, our opinion started to shift for the following reasons:

– With the uptake of SWI-Prolog as a web-server platform, more languages came into the picture, notably HTML, JavaScript and JSON. While HTML can be represented relatively straightforward by Prolog terms, this is not feasible for JavaScript. Representing JSON requires wrapping terms in compounds to achieve an unambiguous representation. For example, JavaScript `null` is represented as @(null) to avoid confusing it with the string `"null"`. SWI-Prolog version 7 allows for an alternative JSON representation where Prolog strings are mapped to JSON strings and atoms are used for the JSON constants `null`, `true` and `false`.
– The primary application domain at SWI-Prolog's home base, the computer science institute at the VU University Amsterdam, in particular the 'web and media' and 'knowledge representation and reasoning' groups, is RDF and web applications. This domain fits naturally with Prolog and especially with SWI-Prolog. Nevertheless, we experienced great difficulty motivating our PhD students to try this platform, often because it looked too 'alien' to them.

In [7] we proposed extensions to the Prolog syntax to accommodate languages such as JavaScript and R using 'extended operators', which allows using $\{\ldots\}$ and $[\ldots]$ as operators. In [8] we brought the notion of quasi quotations to Prolog, providing an elegant way for dealing with external languages such as HTML, JavaScript, SQL and SPARQL as well as long strings. In this article we concentrate on extending Prolog's data types with two goals in mind:

[2] `http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch`

- Facilitate the interaction with other IT systems by incorporating their data types. In particular, we wish to represent data from today's dynamic data exchange languages such as JSON naturally and unambiguously.
- Provide access to structured data elements using widely accepted (functional) syntax.

This paper is organised as follows. In section 2 we identify the missing pieces. In section 3 we describe how these are realised in SWI-Prolog version 7, followed by an evaluation of the impact on compatibility, a preliminary evaluation of the new features, and our conclusions.

2 The missing pieces of the puzzle

2.1 Representing text

ISO Prolog defines two solutions for representing text: atoms (e.g., 'ab') and lists of characters, where the characters are either represented as code points, i.e., integers, such as [97,98] or atoms of one character ([a,b]). Representing text using atoms is often considered inadequate for several reasons:

- It hides the conceptual difference between text and program symbols. Where content of text often matters because it is used in I/O, program symbols are merely identifiers that match with the same symbol elsewhere in the program. Program symbols can often be consistently replaced, for example to obfuscate or compact a program.
- Atoms are globally unique identifiers. They are stored in a shared table. Volatile strings represented as atoms come at a significant price due to the required cooperation between threads for creating atoms. Reclaiming temporary atoms using *Atom garbage collection* is a costly process that requires significant synchronisation.
- Many Prolog systems (not SWI-Prolog) put severe restrictions on the length of atoms, the characters that can be used in atoms or the maximum number of atoms.

Representing text as a list of character codes or 1-character atoms also comes at a price:

- It is not possible to distinguish (at run time) a list of integers or atoms from a string. Sometimes this information can be derived from (implicit) typing. In other cases the list must be embedded in a compound term to distinguish the two types. For example, `s("hello world")` could be used to indicate that we are dealing with a string.
  Lacking run time information, debuggers and the top level can only use heuristics to decide whether to print a list of integers as such or as a string (see **portray_text/1**). While experienced Prolog programmers have learned to cope with this, we still consider this an unfortunate situation.
- Lists are expensive structures, taking 2 cells per character (3 for SWI-Prolog, which threads lists as arbitrary compound terms, represented as the 'functor' (**./2**) and an array of arguments). This stresses memory consumption on the stacks while pushing them on the stack and dealing with them during garbage collection is unnecessarily expensive.

2.2 Representing structured data

Structured data is represented in Prolog using compound terms, which identify the arguments by position. While that is perfectly natural for e.g., `point`(*X,Y*), it becomes cumbersome if there is no (low) natural number of arguments or if there is no commonly accepted order of the arguments. The Prolog community has invented many workarounds for this problem:

- Use lists of *Name*=*Value* or `Name`(*Value*) terms. While readable, this representation wastes space while accessing elements is inefficient.
- Use compound terms and some form of symbolic access. Alternatives seen here are SWI-Prolog's library(record), which generates access predicates from a declaration, the Ciao solution [4], which provides access using functional notation using *Term*$*field*, the ECLiPSe solution mapping terms `name{key1:value1,key2:value2,...}` to a term `name(value2,_,value1,_,...)` using expansion called by **read_term/3** based on a 'struct' declaration.[3]
- Using binary trees (e.g., the classical DEC10 library(assoc)). This provides fast access, but uses a lot of space while the structures are hard to write and read.

2.3 Ambiguous data

We have already seen one example of ambiguous data: the list [97,98] can be the string `"ab"` or a list with two integers. Using characters does not solve this. Defining a string as a list of elements of a new type 'character' still does not help as it fails to distinguish the empty list (`[]`) from the empty string (`""`). Normally, ambiguity is resolved in one of two ways: the data is passed between predicates that interpret the ambiguous terms in a predefined way (e.g., `atom_codes(A,[])` interprets the `[]` as an empty string) or the data is wrapped in a compound, e.g., `s([97,98])`. The first requires an interpretation context, which may not be present. The latter (known as *non-defaulty* representation) is well suited for internal processing, but hard to read and write and requires removing and wrapping the data frequently.

3 SWI-Prolog 7

With SWI-Prolog version 7, we decided to solve the above problems, accepting that version 7 would not be completely backward compatible with version 6 and the ISO standard. As we will see in section 4 though, the compatibility of SWI-Prolog version 7 to its predecessors can be considered fair. Most of the changes are also available in some other Prolog.

[3] `http://www.eclipseclp.org/doc/userman/umsroot022.html`

3.1 Double quoted strings

Strings, and their syntax have been under heavy debate in the Prolog community, but did not make it into the ISO standard. It is out of the scope of this paper to provide a historical overview of this debate. Richard O'Keefe changed his opinion on strings during the debate on the SWI-Prolog mailinglist. His current opinion can be found in "An Elementary Prolog Library", section 10[4]

SWI-Prolog 7 reads `"..."` as an object of type *string*. Strings are atomic objects that are distinct from atoms. They can represent arbitrary long sequences of Unicode text. Unlike atoms, they are not combined in a central table, live on the term-stack (global stack or heap) and their life time is the same as for compound terms (i.e., they are discarded on backtracking or garbage collection).

Strings as a distinct data type are present in various Prolog systems, e.g., SWI-Prolog, Amzi! and YAP. ECLiPSe [6] and hProlog[5] are the only system we are aware of that uses the common double-quoted syntax for strings. The set of predicates operating on strings has been synchronised with ECLiPSe.

3.2 Modified representation of lists

Representing lists the conventional way using **./2** as cons-cell and the atom '[]' as list terminator both (independently) poses difficulties, while these difficulties can be avoided easily. These difficulties are:

- Using **./2** prevents using this commonly used symbol as an operator because `a.B` cannot be distinguished from `[a|B]`. Changing the functor used for lists has little impact on compatibility because it is (almost) always written as [...]. It does imply that we can use this symbol to introduce widely used syntax to Prolog, as described in section 3.3.
- Using `'[]'` as list terminator prevents dynamic distinction between atoms and lists. As a result, we cannot use polymorphism that involve both atoms and lists. For example, we cannot use *multi lists* (arbitrary deeply nested lists) of atoms. Multi lists of atoms are in some situations a good representation of a flat list that is assembled from sub sequences. The alternative, using difference lists or Definite Clause Grammars (DCGs) is often less natural and sometimes demands for 'opening' proper lists (i.e., copying the list while replacing the terminating empty list with a variable) that have to be added to the sequence. The ambiguity of atom and list is particularly painful when mapping external data representations that do not suffer from this ambiguity.
  At the same time, avoiding `'[]'` as a list terminator avoids the ambiguity of text representations described in section 2.3, i.e., `'[]'` unambiguously represents two characters and [] unambiguously represents the empty string.

[4] `http://www.cs.otago.ac.nz/staffpriv/ok/pllib.htm#strs`
[5] `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW366.abs.html`

We changed the cons-cell functor name from '.' to '[|]', inspired by Mercury.[6] We turned the empty list ([]) into a new data type, i.e., [] has the properties demonstrated by the queries below. This extension is also part of CxProlog using `--flags nil_is_special=true`.[7]

```
?- atom([]).                      % [] is not an atom
false.
?- atomic([]).                    % [] is atomic
true.
?- is_list('[]').                 % '[]' is not a list
false.
?- [] == '[]'.                    % our goal
false.
?- [] == [].                      % obviously
true.
?- [] == [/*empty list*/].        % also
true.
?- '[]' == '[   ]'.               % two different atoms
false.
```

3.3 Introducing dicts: named key-value associations

Dicts are a new data type in SWI-Prolog version 7, which represents a key-value association. The keys in a dict are unique and are either atoms or integers. Dictionaries are represented by a canonical term, which implies that two dicts that represent the same set of key-value associations compare equal using **==/2**. Dicts are natively supported by **read/1** and **write/1**. The basic syntax of a dict is described below. Similar to compound terms, there cannot be a space between the *Tag* and the {...} term. The *Tag* is either an atom or a variable, notably _{...} is used as 'anonymous' dict.

Tag{Key1:Value1, Key2:Value2, ...}

Below are some examples, where the second example illustrates that the order is not maintained and the third illustrates an anonymous dict.

```
?- A = point{x:1, y:2}.
A = point{x:1, y:2}.

?- A = point{y:2, x:1}.
A = point{x:1, y:2}.

?- A = _{first_name:"Mel", last_name:"Smith"}.
A = _G1476{first_name:"Mel", last_name:"Smith"}.
```

[6] http://www.mercurylang.org/information/doc-latest/transition_guide.pdf

[7] http://ctp.di.fct.unl.pt/~amd/cxprolog/MANUAL.txt

Note that our dict notation looks similar, but is fundamentally different from ECLiPSe structs. The ECLiPSe struct notation $\{\ldots\}$ is a sparse notation for a declared normal compound term. The ECLiPSe notation can only be used for *input*, it is not possible to dynamically add new keys and the resulting term can be used anywhere where a compound term is allowed. For example, ECLiPSe allows us to write predicates with named arguments like this:

```
person{last_name:"Smith"}.
```

In contrast, our dicts are dynamic, but can only appear as a data term, i.e., not as the head of a predicate. This allows for using them to represent dynamic data from e.g., JSON objects or a set of XML attributes.

Dicts also differ from LIFE PSI-Terms [1], which are basically feature vectors that can unify as long as there are no conflicting key-value pairs in both PSI terms. Dicts are ground if all values are ground and it is thus impossible to add keys using unification. PSI terms can be simulated by associating a dict to an attributed variable. The code to unify two dicts with non-conflicting key-value pairs is given below, where **>:</2** succeeds after all values associated with common keys are unified. For example, `T{a:1,b:B} >:< d{a:A,b:2,c:C}` succeeds with *T*=d, *B*=2, leaving *C* unbound.

```
psi_unify(Dict1, Dict2, Dict) :-
        Dict1 >:< Dict2,
        put_dict(Dict1, Dict2, Dict).
```

Functional notation on dicts Dicts aim first of all at the representation of dynamic structured data. As they are similar to the traditional library(assoc), a predicate `get_dict`(*+Key,+Dict,-Value*) is obvious. However, code processing dicts will become long sequences of **get_dict/3** and **put_dict/4** calls in which the actual logic is buried. This at least is the response we get from users using library(record) and library(assoc) and is also illustrated by the aforementioned solutions in Ciao and ECLiPSe. We believe that a functional notation is the most natural way out of this.

The previously described replacement of '.'[8] with '`[|]`' as list cons-term provides the ideal way out of this because (1), the a.b notation is widely used for this purpose and (2) **./2** terms are extremely rare in Prolog. Therefore, SWI-Prolog transforms **./2** terms appearing in goals into a sequence of calls to **./3**,[9] followed by the original goal. Below are two examples, where the left code is the source and the right is the transformed version.

[8] Note that '.' and a plain . are the same, also in SWI-Prolog version 7. We use '.' in running text to avoid confusion with the end of a sentence.

[9] We called the helper predicate to evaluate **./2** terms **./3** to make the link immediate. It could also have been named e.g., **evaluate/3**.

```
                              .(D, last_name, LN),
   writeln(D.last_name)       writeln(LN)
```

```
last_name(D, D.last_name).    last_name(D, LN) :-
                                  .(D, last_name, LN).
```

Functions on dicts In the previous section we described the functional notation used to access keys on dicts. In addition to that, we allow for user defined functions on dicts. Such functions are invoked using the notation *Dict.Compound*, e.g., `Point.offset(X,Y)` may evaluate to a new *Point* dict at offset (*X*,*Y*) from the original. User defined functions are realised by means of the dict *tag*, which associates the dict with a Prolog module (inspired by attribute names of attributed variables). The offset function is defined as:

```
:- module(point, []).

Pt.offset(OX,OY) := point{x:X,y:Y} :-
     X is Pt.x + OX,
     Y is Pt.y + OY.
```

The above function definition is rewritten using term_expansion rules into the code below. The predicate **./3**, handling functional notation based on **./2**, translates *Dict.Compound* into `call`(*Compound, Dict, Result*). For example, the expression `point{x:1,y:2}.offset(2,4)` is translated into `call(offset(2,4),point{x:1,y:2},Result)`, which in term calls the predicate below.

```
offset(OX, OY, Pt, point{x:X, y:Y}) :-
         '.'(Pt, x, X0),
         X is X0+OX,
         '.'(Pt, y, Y0),
         Y is Y0+OY.
```

As *Dict.Atom* accesses a key and *Dict.Compound* calls a user-defined function, we have no way to express functions without arguments. In [7] we already concluded that *name()* is a valuable syntactic building block for DSL construction and therefore we decided to add support for zero-argument compound terms, such as *name()*. Zero-argument compounds are supported by the following predicates:

compound_name_arity(*Compound, Name, Arity***)**
compound_name_arguments(*Compound, Name, Arguments***)**
These predicates operate on compounds with any number of arguments, including zero.

functor(*Callable, Name, Arity*)
Callable **=..** *List*

These predicates operate on atoms or compounds. They raise an error if the first argument is a zero-arity compound.

4 Compatibility

SWI-Prolog version 7 is not fully backward compatible with older versions and drifts further away from the ISO standard. We believe that a programming language must evolve over time to match changing demands and expectations. In other words, there should be a balance between compatibility with older versions of the language and fulfilling evolving demands and expectations. As far as we understand, the ISO standardisation process only allows for strict *extensions* of a language, guaranteeing full backward compatibility with ISO standard. The ISO model allows for none of the described changes because all of them have at least corner cases where they break compatibility. Such a restrictive view does not allow for gradual language *evolution* and forces a *revolution*, replacing the language with a new one. We believe that the evolutionary route is more promising.

Nevertheless, SWI-Prolog version 7 introduces significant changes. We have evaluated the practical consequences of these changes as soon as we had a prototype implementation by porting our locally developed software as well as large systems for which we have the source code. A particularly interesting code base is Alpino [3], a parser for the Dutch language. Alpino has been developed for over 15 years, counts approximately 500K lines of Prolog code and contains many double quoted strings. Porting took two days, including implementing **list_strings/0** described below.

We received several evaluations from users about porting their program from version 6 to version 7, two of which concern code basis between 500K and 1M lines. This section summarise our key findings.

Double quoted strings This is the main source of incompatibilities. However, migrating programs proves to be fairly easy. First, note that string literals in DCGs can be mapped to lists of character codes by the DCG compiler, which implies that code such as `det --> "the"` remains valid. The double-quoted notation is commonly used to represent e.g., the format argument for **format/3**. This is, also conceptually, correct usage of strings and does not require any modification. We developed a program analyzer (**list_strings/0**) which examines the compiled program for instances of the new string objects. The analyzer has a user extensible list of predicates that accept string arguments, which causes a goal `format("Hello World~n")` to be considered safe. In practise, this reveals compatibility issues accurately. There are three syntactic measures to adapt your code:

- Rewrite the code. For example, change `[X] = "a"` into `X = 0'a`.
- If a particular module relies heavily on representing strings as lists of character code, consider adding the following directive to the module. Note that this flag only applies to the module in which it appears.

```
:- set_prolog_flag(double_quotes, codes).
```

– Use a back quoted string (e.g., `` `text` ``). Note that using `` `text` `` ties the code to SWI-Prolog version 7. There is no portable syntax that produces a list of characters. Such a list can be obtained using portable code using one of the constructs below.

  - `phrase("text", List)`
  - `atom_codes("text", List)`

Using the new [] as list terminator This change has very few consequences to Prolog programs. We encountered four issues, caused by calls such as `atom_concat`(*[], . . .*). Typically, these result from using [] as 'nil' or 'null', i.e., *no value*, but using them as a real value.

Using `[|]` *as cons-cell* We encountered a few cases where **./2** terms were handled explicitly or where the functor-name of a term was requested and **.** was expected. We also found lists written as e1.e2.e3.[] and [H|T] written as H.T. Such problems can typically be found by examining the compiled code for **./2** terms.

Together with [], the only major problem encountered is JPL, the SWI-Prolog Java interface. This interface represents Prolog terms as Java objects, assuming there are variables, atoms, numbers and compound terms. I.e., lists are treated as **./2** terms ending in the atom '[]'. We think JPL should be extended with a list representation. This will also cure the frequent stack overflows caused by long lists represented as deeply nested **./2** terms that are traversed by a recursive Java method.

Dicts, functions and zero-arity compounds The dict syntax infrequently clashes with a prefix operator followed by {. . . }. Cases we found include the use of `@{null}` to represent 'null' values, `::{...}` used (rarely) in Logtalk and `\+{...}` which can appear in DCGs to express negation of a native Prolog goal. The **./2** functional notation causes no issues after **./2**-terms that should have used list notation were fixed. The introduction of zero-arity compounds has no consequences for applications that do not use these terms. The fact that such terms can exist exposed many issues in the development tools.

5 Evaluation

The discussions on SWI-Prolog version 7 took place mostly in October to December 2013. The implementation of the above is now considered almost stable. Migrating towards effective and consistent use of these new features is not easy because most existing code and libraries use atoms to represent text and one of the described alternatives to represent key-value sets.

Our evaluation consists of two parts. First, we investigate uptake inside SWI-Prolog's libraries and by users. The second part is a performance analysis.

Dicts Currently, dicts can be used in the following areas:

– As an alternative option representation. All built-in predicates as well as library(option) accept dicts as an alternative specification for options.
– The JSON support library provides alternative predicates to parse input into JSON represented using dicts. The JSON write predicates accept both the old representation and the new dict representation.

We have been using dicts internally for new code. The mailing list had a brief discussion on dicts.[10] There two contributed SWI-Prolog add-ons[11] that target lists: *dict_schema* and *sort_dict* address type checking and sorting lists of dictionaries by key. Other remarks:

– *Michael Hendricks*: "We're using V7. Most of our new code uses dicts extensively. I've found them especially compelling for working with JSON and for replacing option lists."
– *Wouter Beek*: Uses dictionaries for WebQR[12] for dealing with JSON messages. Claims that the code becomes more homogeneous, readable and shorter. Also uses the *real* add-on, claiming that the dot-notation, functions without argument and double quoted strings gives the mapping a more 'native' flavour and makes the code more readable.

Strings Many applications would both conceptually and performance-wise benefit from using strings. As long as most libraries return their textual data using atoms, we cannot expect serious uptake. Strings (and zero-argument compounds) are currently used by the R-interface package *real* [2].

5.1 Dict performance

The dict representation uses exactly twice the memory of a compound term, given the current implementation which uses an array of consecutive (key,value) pairs sorted by the key. Future implementation may share the key locations between dicts with the same keys. We compared the performance using a tight loop. The full implementation using functional notation is given in figure 1.

We give only the recursive clause for the other test cases in figure 2. Thus, **t0/2** provides the base case without extracting data, **t1/2** is the same as **tf/2** in figure 1, but without using functional notation and thus avoiding **./3**. The predicate **t2/2** uses **arg/3** to extract a field from the structured data represented as a compound term, **t3/2** uses a plain list of *Name*=*Value* and **t4/2** uses library(assoc), the SWI-Prolog implemented of which uses an AVL tree. Table 1 shows the performance of the four loops for 1,000,000 iterations, averaged over 10 runs on an Intel i7-3770 running Ubuntu 13.10 and SWI-Prolog 7.1.12.

[10] `http://swi-prolog.996271.n3.nabble.com/Some-thoughts-on-dicts-in-SWIPL-7-tt14327.html`
[11] `http://www.swi-prolog.org/pack/list`
[12] `https://github.com/wouterbeek/WebQR`

```
tf(Size, N) :-
        data(Size, D),
        tf2(N, Size, D).

tf2(0, _, _) :- !.
tf2(N, Q, D) :-
        Q1 is (N mod Q)+1,
        a(D.Q1),
        N2 is N - 1,
        tf2(N2, Q, D).

data(Size, D) :-
        numlist(1, Size, L),
        maplist(pair, L, Pairs),
        dict_pairs(D, _, Pairs).

pair(X, X-X).
```

Fig. 1. Benchmark program to evaluate dict lookup performance

Table 1 shows that the overhead of using dicts compared to compound terms is low (t1 vs. t2). The overhead of the functional notation is caused by type checking and checking for accessing a field vs. accessing a function on the dict in the predicate **./3**. This overhead could be removed if we had type inference. The last two predicates (t3, t4) show the performance of two classical Prolog solutions.

```
t0(N,Q,D) :- Q1 is (N mod Q)+1,                     a(x), ...
t1(N,Q,D) :- Q1 is (N mod Q)+1, get_dict(b,D,A),    a(A), ...
t2(N,Q,D) :- Q1 is (N mod Q)+1, arg(Q1,D,A),        a(A), ...
t3(N,Q,D) :- Q1 is (N mod Q)+1, memberchk(Q1=A,D), a(A), ...
t4(N,Q,D) :- Q1 is (N mod Q)+1, get_assoc(Q1,D,A), a(A), ...
```

Fig. 2. Alternative body (second clause of **tf2/3** in figure 1). The predicate a/1 is the dummy consumer of the data, defines as `a(_)`.

5.2 String performance

We evaluated the performance of text processing with the task to create the texts "test"*N* for *N* in 1..1,000,000. The results of these tests are presented in table 2. The core of the 4 loops is shown in figure 3. The predicate **tl2/1** has been added on request by one of the reviewers who claimed that **tl1/1** is unfair because it requires pushing the

		GC		
Test	CPUTime	Times	AvgSize	GCTime
t0(1000,1000000)	0.111	0	0	0.000
tf(1000,1000000)	0.271	259	18,229	0.014
t1(1000,1000000)	0.218	129	18,229	0.007
t2(1000,1000000)	0.165	129	10,221	0.006
t3(1000,1000000)	20.420	305	50,213	0.031
t4(1000,1000000)	3.968	1,299	50,213	0.149
t0(3,1000000)	0.113	0	0	0.000
tf(3,1000000)	0.232	259	2,277	0.011
t1(3,1000000)	0.181	129	2,277	0.005
t2(3,1000000)	0.166	129	2,245	0.005
t3(3,1000000)	0.277	189	2,357	0.009
t4(3,1000000)	0.859	346	2,397	0.014

Table 1. Performance test for accessing structured data using various representations. The first argument is the number of key-value pairs in the data and the second is the number of iterations in each test. **CPUTime** is the CPU time in seconds. The **GC** columns represent heap garbage collection statistics, showing the number of garbage collections, the average size of the heap *after* GC and the time spent on GC in seconds.

list `` `list` `` onto the stacks on each iteration and the SWI-Prolog implementation of **append/2** performs a type-check on the first argument, making it relatively slow. We claim that **tl1/1** is a more natural translation of the other tests. The test `c`(*Threads,Goal*) runs *Threads* copies of *Goal*, each in their own threads, while the main thread waits for the completion of all threads. The tests were executed on a (quad core) Intel i7-3770 machine running Ubuntu 13.10 and SWI-Prolog 7.1.15.

```
t0(N)    :- dummy([test,N],_), N2 is N-1, t0(N2).
ta(N)    :- atomic_list_concat([test,N],_), N2 is N-1, ta(N2).
ts(N)    :- atomics_to_string(["test",N],_), N2 is N-1, ts(N2).
tl1(N)   :- number_codes(N,S), append([`test`,S],_),
            N2 is N-1, tl(N2).
tl2(N)   :- tl2(N,`test`).
tl2(N,P) :- number_codes(N,S), append(P,S,_),
            N2 is N-1, tl(N2).
dummy(_,_).
```

Fig. 3. Benchmarks for comparing concatenation of text in various formats.

We realise that the above performance analysis is artificial and limited. The tests only analyse *construction*, not processing text in the various representations. Notably atoms

	Time			GC			Atom GC		
Test	Process	Thread	Wall	Times	AvgWorkSet	GCTime	Times	Reclaimed	AGCTime
t0(1000000)	0.058	0.058	0.058	786	2,186	0.009	0	0	0.000
ta(1000000)	0.316	0.316	0.316	785	2,146	0.013	99	10,884,136	0.042
ts(1000000)	0.214	0.214	0.214	1,703	2,190	0.023	0	0	0.000
tl1(1000000)	1.051	1.051	1.051	8,570	2,267	0.108	0	0	0.000
tl2(1000000)	0.437	0.437	0.437	3,893	2,231	0.077	0	0	0.000
c(4,t0(1000000))	0.252	0.000	0.065	0	0	0.000	0	0	0.000
c(4,ta(1000000))	6.300	0.000	1.924	0	0	0.000	332	36,442,981	0.227
c(4,ts(1000000))	0.886	0.000	0.232	0	0	0.000	0	0	0.000
c(4,tl1(1000000))	4.463	0.000	1.143	0	0	0.000	0	0	0.000
c(4,tl2(1000000))	1.731	0.000	0.441	0	0	0.000	0	0	0.000

Table 2. Comparing atom and string handling performance. The **Time** columns represent the time spent by the process, calling thread and the wall time in seconds. The **GC** columns are described with table 1. The **AtomGC** columns represent the atom garbage collector, showing the number of invocations, number of reclaimed atoms and the time spent in seconds. Note that the GC values for the concurrent tests are all zero because GC is monitored in the main thread, which just waits for the others to complete.

and strings are internally represented as arrays and thus provide O(1) access to the i-th character, but lists allow splitting in head and tail cheaply and can exploit DCGs.

Table 2 makes us draw the following conclusions regarding construction of a short text from short pieces:

– To our surprise, constructing text as atoms is faster than using lists of codes.
– Strings are constructed considerably faster than atoms.
– Atom handling significantly harms performance of multi-threaded application. Disabling atom garbage collection and looking at the internal contention statistics indicate that the slowdown is caused by contention on the mutex that guards the atom table.

6 Conclusions

With SWI-Prolog version 7 we have decided to narrow the gap between Prolog and other key components in the IT infrastructure by introducing commonly found data types and harmonizing the syntax with modern languages. Besides more transparent interfacing, these changes are also aimed at simplifying the transition from other languages. The most important changes are the introduction of dicts (key-value sets) as primary citizens with access to keys using functional notation, the introduction of strings, including the common double-quoted notation and an unambiguous representation of lists. Previous changes added [. . .] and {. . . } as operators and introduced quasi quotations. These extensions aim at smooth exchange of data with other IT infrastructure, a natural syntax for accessing structured data and the ability to define syntax for DSLs that is more natural to those not familiar with Prolog's history.

Acknowledgements

The described extensions result for a large part from interaction with users and other Prolog developers. Nicos Angelopoulos pointed at the issues interacting with the R-project. Michiel Hildebrand and Jacco van Ossenbruggen helped formulating the issues with JavaScript and JSON. Jose Morales suggested the 'block operators' [...] and {...}. Michael Hendricks brought quasi quotations to our attention. Manuel Hermenegildo and Bart Demoen supported the idea to start moving during the ICLP in Istanbul. Joachim Schimpf helped achieving a shared set of string operations with ECLiPSe and provided a test suite for them. Vitor Santos Costa commented on several of the ideas and is likely to implement most of them for YAP after the dust has settled. Other people that commented on the drafts include Richard O'Keefe, Ulrich Neumerkel, Paulo Moura, Feliks Kluzniak, Alan Baljeu, Kilian Evang, Shon Feder, Jeff Schultz and Carlo Capelli.

References

1. Hassan Ait-Kaci. An overview of LIFE. In *Next generation information system technology*, pages 42–58. Springer, 1991.
2. Nicos Angelopoulos, Vítor Santos Costa, João Azevedo, Jan Wielemaker, Rui Camacho, and Lodewyk Wessels. Integrative functional statistics in logic programming. In Konstantinos F. Sagonas, editor, *PADL*, volume 7752 of *Lecture Notes in Computer Science*, pages 190–205. Springer, 2013.
3. Gosse Bouma, Gertjan Van Noord, and Robert Malouf. Alpino: Wide-coverage computational analysis of dutch. *Language and Computers*, 37(1):45–59, 2001.
4. Amadeo Casas, Daniel Cabeza, and Manuel V Hermenegildo. A syntactic approach to combining functional notation, lazy evaluation, and higher-order in lp systems. In *Functional and Logic Programming*, pages 146–162. Springer, 2006.
5. Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on*, pages 36–43. IEEE, 2009.
6. Micha Meier and Joachim Schimpf. An architecture for prolog extensions. In *Extensions of Logic Programming*, pages 319–338. Springer, 1993.
7. Jan Wielemaker and Nicos Angelopoulos. Syntactic integration of external languages in Prolog. In *Proceedings of WLPE 2012*, 2012.
8. Jan Wielemaker and Michael Hendricks. Why It's Nice to be Quoted: Quasiquoting for Prolog. *CoRR*, abs/1308.3941, 2013.