A Simple $\mathcal{LIFE}$ *Su Doku* Solver*

Hassan Aït-Kaci
hak@ca.ibm.com

IBM Canada Ltd.
Surrey, BC, Canada

August 14, 2009

Abstract

$\mathcal{LIFE}$ is a constraint-logic programming language over order-sorted graphs subject to functional dependency constraints. We use it to present a simple and purely declarative specification of the popular number puzzle *Su Doku.* This specification yields a surprisingly efficient *Su Doku* solver although the "*all-different*" constraint is not native to $\mathcal{LIFE}$. The trick is that this constraint can be realized efficiently though purely declaratively using $\mathcal{LIFE}$'s native data structure and adaptive control strategy. For added bonus and ease of interaction with the puzzle solver, we also use $\mathcal{LIFE}$ to specify a purely declarative GUI display.

Keywords: *Constraint-logic programming, $\mathcal{LIFE}$, Su Doku, "all-different" constraint, puzzle solving, declarative programming, declarative graphics*

1 Introduction

Life is *"trying things to see if they work…"*

RAY BRADBURY

The popular game *Su Doku* has been solved using a variety of techniques, in a plethora of programming idioms.[1] These range from (declaratively) obscure solutions—*e.g.*, in APL[2]—to (declaratively) elegant ones such as those using constraint-logic programming (CLP)—*e.g.*, in CHIP.[3] The CLP solution's declarativeness relies essentially on describing the problem in terms of the global '`alldiff`' constraint, which stipulates that a given set of n variables must always be assigned mutually distinct values [13, 15, 16]. $\mathcal{LIFE}$ is a constraint-logic programming language over order-sorted graphs subject to functional dependency constraints [3]. Our essential contribution in this paper is to show how $\mathcal{LIFE}$ enables a surprisingly efficient '`alldiff`' purely declaratively thanks to its:

1. built-in constrained data-structure (extensible records [3]), and
2. control strategy (constraint *residuation* [7]).

*A short version of this paper was published as [4].
[1]See: http://en.wikipedia.org/wiki/Sudoku
[2]See: http://www.vector.org.uk/archive/v214/sudoku2.htm
[3]See: http://uuu.enseirb.fr/~gloess/sudoku/CHIP/sudoku.pl

Residuation is the evaluation strategy whereby functional expressions are evaluated as far as possible, suspending upon unbound variables. A suspended evaluation is then awakened and resumed as soon as any variable it is waiting for gets further instantiated.

This paper's essential contribution is that of a Programming Pearl in $\mathcal{LIFE}$. It is in no way claimed that realizing the '`alldiff`' in the manner herein described is more efficient than the best known methods described in [13, 15, 16]—nor indeed that it comes even close. The point is simply to explain how $\mathcal{LIFE}$'s native features enable a more than decent '`alldiff`' *for free!*—*i.e.*, without having to make the investment of implementing such a specific solver. By "*more than decent*", we mean sufficient to solve a few reputedly hard *Su Doku* games.[4] Of course, for other applications involving more tricky configurations over many more variables than a *Su Doku* problem (such as, *e.g.*, large scale jobshop scheduling), this freebie '`alldiff`' is quite likely to run out of breath where native methods will hold sway. Still, for fast prototyping without access to a native '`alldiff`' solver, we found it instructive to realize that $\mathcal{LIFE}$'s idiosyncratic features could easily give it this particular capability.

In essence, under the guise of a ludic musing, we will demonstrate some subtle capabilities of $\mathcal{LIFE}$ and its adequacy for efficient declarative programming. We hope to illustrate how some of the innovating symbolic computation techniques that are behind $\mathcal{LIFE}$ (ψ-term unification, order-sorted feature constraint solving, Horn resolution, functional rewrite rules, residuation, *etc.*) can also naturally be combined with simple control to provide an adequate basis for efficient—nay, clever!—declarative programming. Indeed, our specific $\mathcal{LIFE}$ *Su Doku* solver is only meant as an exemplar, albeit entertaining, of such a serendipitous combination of features.

The remainder of this paper is organized as follows. In Section 2, background on $\mathcal{LIFE}$ is succinctly overviewed. In Section 3, what makes the *passive* constraint system of $\mathcal{LIFE}$ stand apart from *active* solvers is discussed. In Section 4, we summarize the essence of our contribution. In Section 5, we explain how the '`alldiff`' constraint may be realized purely declaratively in $\mathcal{LIFE}$ for free, and—to boot!—efficiently so. As a bonus showing the power of $\mathcal{LIFE}$'s passive constraint system for declarative programming, Section 6 explains how a purely declarative, though effective, graphical user interface (GUI) can be specified for interacting with the *Su Doku* solver. We conclude in Section 7 with a few remarks. The complete $\mathcal{LIFE}$ *Su Doku* program may be downloaded from the author's web site.[5] The programs therein run as claimed using the `WildLife 1.02` interpreter [5].

2 A quick look back on $\mathcal{LIFE}$

> Life can only be understood looking backwards but it must be lived forwards.
>
> SØREN KIERKEGAARD

$\mathcal{LIFE}$ [3] is a CLP language that may be loosely defined as Prolog over ψ-terms, which are order-sorted graphs, themselves possibly subject to functional and relational contraints. As Prolog uses first-order terms as its universal data structure, so does $\mathcal{LIFE}$ use ψ-terms. A ψ-term generalizes a first-order term (FOT) by allowing cycles (*à la* rational terms) and partial information (in the form of partially-ordered sorts). The sorts denote sets and the partial order on sorts denotes set containment.

Following is a summary of relevant features of $\mathcal{LIFE}$.

1. Like a FOT, a ψ-term may have arguments; these are specified implicitly as for a FOT using a parenthesized comma-separated sequence of ψ-terms, and/or they may be specified by

[4] See Appendix Section B.

[5] See: `http://wikix.ilog.fr/wiki/bin/view/Main/HassanAitKaci#3`

keywords (called *features*), including explicit out-of-order numerical positions. Examples of ψ-terms are:[6]

- `f(a,X,g(X))`—or, equivalently, `f(1=>a, 2=>X, 3=>g(1=>X))`,
- `person(name => "bozo", dob => date(year => 1980))`,
- `add(X,Y,result => X+Y)`,
- `X:person(spouse => person(spouse => X))`.

2. Unlike FOTS, ψ-terms do not impose a fixed arity: the number of possible arguments is not constrained, and can be zero or any, by implicit or explicit position, by keywords, or both, and in any order. For example, unifying the ψ-terms `f(a,3=>c)` and `f(a,b)` succeeds and results in `f(a,b,c)`. Similarly, unifying the ψ-term:

   ```
   person(P, dob => date(month => may))
   ```

   with the ψ-term:

   ```
   person(dob => date(year => 1980)),
   ```

   succeeds and results in the ψ-term:

   ```
   person(P, dob => date(month => may, year => 1980)).
   ```

3. $\mathcal{LIFE}$ predicates are defined by Horn rules over ψ-terms. In $\mathcal{LIFE}$, everything is a ψ-term, in the same way as everything is a FOT in Prolog. This provides a powerful convenience for metaprogramming.
4. Like Prolog, $\mathcal{LIFE}$ resolves a relational query using a top-down/left-right backtracking control strategy. The cut operator (`!`) may be used as in Prolog.
5. Invoking a predicate binds variables or refines sorts using ψ-term *unification* ($\mathcal{OSF}$ constraint conjunction).[7] In $\mathcal{LIFE}$, there is no conceptual difference between types and values. These are called *sorts* and are partialy ordered in a sort hierarchy. The topmost all-encompassing sort is written '`@`' and the bottommost all-excluding sort is '`{}`'—which causes failure in $\mathcal{LIFE}$. All $\mathcal{LIFE}$'s logical variables are sorted. A variable `X` bearing no sort is implicitly understood as being sorted by `@`—*i.e.*, '`X:@`'. The subsort ordering is declared using declarations such as '`apple <| fruit.`' and '`apple <| food.`', which make '`apple`' objects be also '`fruit`' and '`food`' objects. With such declarations, a query such as '`X = food, X = fruit?`' would succeed with the binding '`X = apple`'—*i.e.*, the intersection of sorts '`food`' and '`fruit`'. If in addition we had declared as well `banana <| fruit.`' and '`banana <| food.`', then the above query would first give '`X = apple`', then upon backtracking '`X = banana`'. The semicolon disjunction operation ('`;/2`') can also be used on sorts enclosed in curly braces (denoting their union)—*e.g.*, '`X = { breakfast ; lunch ; dinner }`'.
6. $\mathcal{LIFE}$ interprets functions that are defined by ordered rewrite rules transforming a ψ-term into another. The rules making up a function definition are tried in the given order, the next being tried only if the preceding's matching failed.

[6] We use Prolog's convention of capitalizing logical variables.

[7] $\mathcal{OSF}$ stands for *"Order-Sorted Feature."* Thus, an $\mathcal{OSF}$ term is a rooted graph whose nodes are labelled with (partially-***o****rdered*) ***s****ort* symbols, and whose arrows are labelled with ***f****eature* symbols. A ψ-term is an $\mathcal{OSF}$ term in normal (or canonical) form—see [6].

7. Evaluating a functional expression binds variables and verifies sort constraints using ψ-term *matching* ($\mathcal{OSF}$ constraint entailment). Upon a fully successful matching of a LHS, no further rules for this expression will be tried.[8]
8. Predicate *resolution* and function *evaluation* cooperate using *residuation*. For example, processing the resolvent '`X = Y+1, Y = 2?`' from left to right, the functional expression '`Y+1`' is a suspended functional expression because it needs to wait for the variable `Y` to take on a value. Thus, the equation '`X = Y+1`' is a residual constraint (or residuation). Thanks to commutativity of conjunction, $\mathcal{LIFE}$ proceeds to the right to the next atom to resolve (*i.e.*, '`Y = 2`'), binding `Y` to `2`. This automatically awakens the residuation '`X = Y+1`' whereupon the functional expression '`Y+1`' evaluates to `3`, binding `X` to this value. Should this instantiation violate any accumulated constraint, chronological backtracking to the last choice point would occur.[9]
9. A ψ-term's subterm may be extracted using *feature projection*, which is a dyadic function (written using the functor '`./2`'). This function takes two arguments: its first argument may evaluate to any ψ-term. Its second argument must evaluate to a feature symbol—*i.e.*, a natural number or a symbolic identifier. It returns the sub-ψ-term of the first argument located under the given position (or symbolic feature) specified as the second argument. In other words, the subterm T' of a ψ-term T at feature f is expressed as `T.f = T'` if and only if `T = s(..., f => T', ...)`, for some sort `s`. Being a function, feature projection residuates whenever its second argument is not ground—*i.e.*, whenever it is not bound to a natural number or a symbolic identifier.
10. New subterms may be added to a ψ-term by unification or feature projection. Indeed, when invoked on a ψ-term T as first argument and with a feature f as second argument, the '`./2`' function has as side-effect: its creates the feature f for T if T does not have the specified feature. For example, the resolvent '`X = foo(bar => buz), X.boo = fuz?`' will result in the augmented ψ-term '`X = foo(bar => buz, boo => fuz)`.'

3 How is $\mathcal{LIFE}$ (*all* that) *different*?

> Life is the sum of all your choices.
>
> ALBERT CAMUS

At first, $\mathcal{LIFE}$ feels very much like Prolog to a programmer. Lists are represented in the same manner as they are in Prolog—*viz.*., square-bracketed and comma-separated. Horn clauses are defined using the infix binary operator '`:-/2`', conjunction is the infix binary operator '`,/2`', disjunction is the infix binary operator '`;/2`'. For example, one can define the familiar '`append/3`' predicate *verbatim* as one would in Prolog. Namely:

```
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
```

Such a predicate can then be used exactly as in Prolog.

On the other hand, $\mathcal{LIFE}$ also differs from Prolog both in obvious and subtle ways. One essential difference is that arity is not constrained for $\mathcal{LIFE}$'s ψ-terms, whereas it is fixed per symbol in Prolog's FOTs. So what happens when one unifies two ψ-terms of different arities? Simply: if a

[8] This, in essence, is akin to a *systematic cut upon the first successful LHS match*. In other words, functional computation is always deterministic and never backtracks.

[9] Only predicate resolution may backtrack, whereas functional evaluation commits to the first successful match.

feature is present in both, then the corresponding subterms are unified; if a feature is present in one and missing in the other, it is simply kept with its subterm in the ψ-term resulting from the unification. For example:

```
A = foo(a => 1, b => 2), B = foo(b => X, c => 3), A = B ?
```

succeeds, resulting in the solved form:

```
A = foo(a => 1, b => X, c => 3), B = A, X = 2.
```

It can be thus said that ψ-term features are *"sticky!"* It is important to note that $\mathcal{LIFE}$ restores all unification side-effects upon bactracking. Therefore, features and subterms that materialize by ψ-term unification or feature projection will dematerialize upon backtracking to an earlier state.

Another difference is the use of interpreted functions. $\mathcal{LIFE}$ functions are defined as rewrite rules using the infix binary operator '`->/2`'. For example, one can define a function returning the length of a list as follows:

```
length([]) -> 0.
length([_|T]) -> 1 + length(T).
```

and use it to define a relation; say:

```
has_even_length(L:list) :- length(L) mod 2 = 0.
```

Then, proving the resolvent '`has_even_length([a,b])?`' succeeds as expected. On the other hand, the resolvent '`has_even_length([a,L:list])?`' will cause the evaluation of the equation '`(1 + length(L:list)) mod 2 = 0?`' to residuate. This is because the sort of `L` is `list`, a supersort of `[]` and `[_,_]`, and thus it cannot be decided which rule may apply.[10] $\mathcal{LIFE}$ deems this a conditional success, pending further instantiation of the residuation variable `L`, whose sort is then printed followed by as many tildas ('`~`') as there are residuations waiting for the variable to become instantiated in order to proceed, together with its current binding—*e.g.*, '`L = list~`' in our example.

Finally, note that metaprogramming can be used to reason about features, using feature projection. For instance, if '`A = foo(a => 1, b => 2, c => 2)`' then the query '`X = { a ; b ; c }, A.X = 2?`' will succeed first with the solution '`X = b`'; then, upon backtracking, with the second solution '`X = c`'.

4 Purely declarative *Su Doku*

The art of life is the art of avoiding pain.

THOMAS JEFFERSON

We now present our purely declarative and surprisingly efficient *Su Doku* solver. The solver itself is very easy—nay, trivial!—to express in $\mathcal{LIFE}$ as it should be in any CLP idiom: we simply write down literally the rules of the game and that's it! The complete code is given in Appendix Section A.1.

Of course, this declarative magic is made possible by the '`alldiff`' constraint, which stipulates that a set of objects must be globally and mutually distinct from one another. Of course, for effective solving of a real *Su Doku* game, this magic is only satisfactory if the '`alldiff`' constraint can be efficiently enforced. This is the case of most C(L)P systems such as ILOG Solver [12],

[10] Recall that, in $\mathcal{LIFE}$, function application uses ψ-term *matching*, not *unification*.

CLP($\mathbf{BNR}$) [10], CLP($\mathbb{R}$) [8], *etc.*, which have '`alldiff`' implemented as a *built-in constraint* [13, 16]. By contrast, although we do abide by the exact same straighforward and purely declarative CLP formulation using $\mathcal{LIFE}$, we do not however rely on a built-in '`alldiff`' constraint. Indeed, there is no such a primitive in the `WildLife 1.02` system.

On the other hand, $\mathcal{LIFE}$'s reasoning primitives can elegantly express '`alldiff`' purely declaratively, and yet achieve the high performance of a built-in '`alldiff`' constraint! *How*? Simply by combining two of its powerful principles:

1. *dynamic objects*, which can freely acquire new features as needed through ψ-term unification and feature projection; [3] and,
2. *automatic coroutining* of functional expressions—*i.e.*, suspension/resumption of functional evaluation pending on further instantiation of arguments (here, specifically, residuation of object feature projection) [7].

As a cherry on the cake, the actual $\mathcal{LIFE}$ code for '`alldiff`' is amazingly succint—one line of code! Last, but not least, it is surprisingly *efficient*. This is remarkable since this enables solving difficult puzzles using the `WildLife 1.02` *interpreter* running on a laptop under `cygwin/X`.[11] This is not so bad taking into account the disconcerting ease of the programming effort.

5 It's *all different* using graphs!

> If A equals success, then the formula is: A = X + Y + Z, where X is work, Y is play, and Z is keep your mouth shut.
>
> ALBERT EINSTEIN

We now explain how the well-known global constraint '`alldiff`' is efficiently enabled for free in $\mathcal{LIFE}$ thanks to its native combination of extensible cyclic graph unification (*i.e.*, ψ-term unification) and its automatically adaptable control behavior using *residuation* (*i.e.*, suspended functional evaluation, pending further instantiation) [7].

Recall that '`alldiff`' imposes that a finite set of variables taking values in some finite domains be each assigned to a distinct value. A *naïve* $\mathcal{O}(n^2)$ method consists in generating $n(n-1)/2$ disequalities among the variables. This is clearly expensive in space and time. Whereas, a much more efficient technique consists in ensuring that any assignment mapping the complete set of so-constrained variables to values is always a one-to-one mapping (*i.e.*, injective)—in effect realizing a maximal matching in a bipartite graph [13, 16, 14]. The good news is that enforcing this global constraint is achievable in *constant time and linear space!*[12]

This method can easily be made effective in $\mathcal{LIFE}$ by using residuation of the feature projection function ('`./2`') on a shared variable denoting a matching assignment for the variables. Namely, we can constrain each variable/value pair being assigned to be (globally) in mutual functional correspondence (*i.e.*, through a one-to-one mapping) by projecting the shared assignment-denoting variable using the unbound constrained variable as one of its features taking a unique id as value (an `int`, say). As long as it is unbound, the "feature" variable will cause the projection function to residuate. In this manner, *as soon as the variable gets bound to a value, the residuation fires, thus enforcing uniqueness of its assigned value thanks to that of the "feature" for the shared assignment variable*.

[11] See: `http://x.cygwin.com/`

[12] To be more precise, the time complexity is dominated by that of access into a ψ-term's feature table, which associates the term's root to its subterms. If access is hashed (as it is in the `WildLife 1.02` interpreter), these table accesses are of order $\mathcal{O}(\log n)$. Although for what concerns the subterms specified by numerical positions, the order of time complexity access is $\mathcal{O}(1)$; *i.e.*, constant. However, it is possible, by compilation, to eliminate all symbolic features to transform them into numerical positions [1].

Let us first illustrate the gist of this technique in $\mathcal{LIFE}$ by defining a predicate `'alldiff/3'` that imposes that its three argument variables `X1`, `X2`, and `X3` be each assigned mutually different values in their domains. We will generalize it later to any number of variables, not just three.

Thus, here is an '`alldiff`' constraint on three variables `X1`, `X2`, and `X3` using a predicate `'assign/3'` imposing the all-different assignment scheme using a shared logical variable denoting the global assignment (*viz.*, the variable `A`):

```
alldiff(X1,X2,X3) :-
     assign(A,X1,1), assign(A,X2,2), assign(A,X3,3).
```

In fact, we have used a predicate `'assign/3'` just to make notation more conspicuous since it is trivially defined by residuation of feature projection as follows:

```
assign(A,X,I) :- A.X = I.
```

Variable `A` denotes the global assignment, variable `X` the constrained variable, and variable `I` the assignment's unique id.

A simple example of using this limited *all-diff* constraint would be, for instance:

```
show(X1,X2,X3) :-
     alldiff(X1,X2,X3),
     X1 = { a ; b }, % domain of X1
     X2 = { b ; c }, % domain of X2
     X3 = { a ; d }. % domain of X3
```

Thus, invoking the query '`show(X1,X2,X3)?`' will give successively:

```
X1 = a, X2 = b, X3 = d.
X1 = a, X2 = c, X3 = d.
X1 = b, X2 = c, X3 = a.
X1 = b, X2 = c, X3 = d.
```

Now, let's see how this works. Fig. 1 shows the effects of executing each of the four lines making the body of the predicate '`show/3`' defined above. Each step results in a modification of the structure of the ψ-term rooted in variable `A`. The dashed arrow represents the residuated feature projections waiting for the feature variables `X1`, `X2`, and `X3`, to be instantiated. A solid arrow is obtained when the corresponding feature is actually materialized: the instantiated feature is then added as a *bona fide* feature to the structure rooted in `A`. Binding `X3` to `a` results in a clash due to the feature variable `X3` taking on the same value as `X1`. This is because functionality of features imposes then that the same individual be found under feature `a`. Instead, it is found that the existing values—*viz.*., `3` and `1`—are incompatible. This causes backtracking to the last choice point, giving as next choice `X3 = d`, which violates no constraint, and therefore succeeds with a legal all-different assigment to the three variables.

To obtain a predicate that would work not just on three variables, but on a set of any number of variables, we now simply generalize the above scheme, making it generic among an arbitrary number of variables instead of just three. We define the predicate '`alldiff/*`', that takes any number of arguments and imposes that they all be different, as follows:

```
C:alldiff :- assignment(features(C),C,A,1).
```

This rule uses the metaprogramming convenience made possible by seeing everything as a ψ-term in $\mathcal{LIFE}$. First, the head of the `alldiff` rule is tagged by variable `C`: it designates the ψ-term

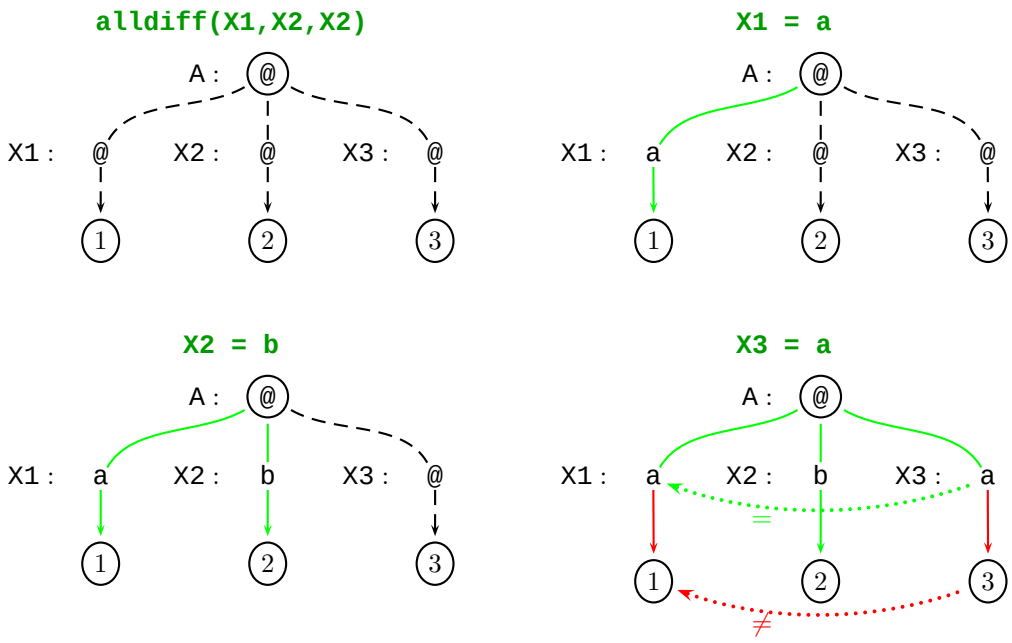

Figure 1: How `alldiff` works

that gets bound to the call; that is, the one that invokes `alldiff`. The one-argument (meta)function '`features/1`' returns the list of feature symbols currently attached as arguments to a given ψ-term. In other words, the body of the rule $\mathcal{LIFE}$ defining the `alldiff` predicate simply initiates a call to the predicate '`assignment/4`', which takes as arguments:

1. the list of features of the call to `alldiff`,
2. the ψ-term representing the call to `alldiff` itself needed to extract the actual subterms from,
3. the shared assignment variable, and
4. the rank in the specified list of features of the feature being currently extracted from the call.

Namely:[13]

```
assignment([]).
assignment([H|T],C,A,I) :-
     assign(A,C.H,I), assignment(T,C,A,I+1).
```

And that's it! Literally.

The *Su Doku* solver itself is defined as the predicate '`sudoku_solver/1`', which simply constrains a *Su Doku* grid `G` and specifies the numeric labels for the cells of `G`:

```
sudoku_solver(G) :- sudoku(G), labels(G).
```

where `sudoku/1` tests the all-different constraints on a 9×9 *Su Doku* grid, and `labels/1` generates labels between 1 and 9 for each cell in `G`.[14] Note that this strategy is simply infeasible in a

[13] Recall that the expression `assign(A,C.H,I)` is equivalent to `A.(C.H) = I`.

[14] See the code for '`sudoku/1`' and '`labels/1`' in Appendix Section A.1.

logic-programming language like Prolog because its operational semantics demands that a state be generated *before* it is tested whether or not it violates any constraint. However, using the reverse strategy (*viz.*, first setting up all the constraints tests as residuations, and then generating the states) makes all the difference! Indeed, in this way, efficient adaptive pruning of the search space takes place automatically since *most states need not be generated at all* due to immediate backtracking caused to any constraint violation as soon as one occurs. Therefore, the above line contains *the* innocuous key to $\mathcal{LIFE}$'s unique way for efficient constraint-handling. Indeed, thanks to residuation, $\mathcal{LIFE}$ allows any factors of a conjunction to commute. By contrast, the following predicate definition:

```
bad_sudoku_solver(G) :- labels(G), sudoku(G).
```

(obtained by simply reversing the order of the body goals) has identical model-theoretic semantics as the previous one's, but results in appallingly inefficient proof-theoretic performance. It is, however, perfectly correct from a model-theoretic point of view! Many model theorists miss this point.

6 $\mathcal{LIFE}$ bonus: a declarative *Su Doku* GUI

> Life is just a mirror, and what you see out there, you must first see inside of you.
>
> WALLY 'FAMOUS' AMOS

$\mathcal{LIFE}$, as a generic language, has a working instance called `WildLife` [5].[15] This system implements a constraint system based on:

- extensible records known as *feature structures* making up (possibly cyclic) labeled graphs; the arcs are the record field bearing labels called *features*; the nodes are the record constructor symbols called *sorts*;
- equality constraints among functional expressions involving parts of these graph feature structures;
- general predicative constraints among functional expressions involving parts of these graph feature structures; the predicates being either built-in constraints, or are defined in terms of one another and built-in constraints using Horn clauses (*i.e.*, *à la* Prolog).

Thus, $\mathcal{LIFE}$ finds solutions fitting functionally constrained order-sorted featured graph structures. In fact, $\mathcal{LIFE}$ does *not*, strictly speaking, *actively solve* these constraints—at least not in the classical sense of constraint-solving seen as the active search for a complete set of solutions. Rather, $\mathcal{LIFE}$'s residuation enables constraints to be used as passive demons acting as coroutined *filters* [7]. Indeed, this is deliberate and a key to its efficiency, since for declarative graphics, actual constraint-solving is rarely needed. This is because if one specifies, *e.g.*, a graphical interface panel containing several widgets such as buttons, text fields, drop-down menus, *etc.*, one has a specific unique feasible solution in mind (*e.g.*, by drawing it with pencil and paper). Active constraint-solving would try to *guess* values fit to accommodate positioning constraints specified for widgets making up a display panel. By contrast, $\mathcal{LIFE}$ uses a powerful constraint-postponement mechanism using a clever implementation technique whereby constraints act as incremental filters [11]. Hence, provided the GUI design is sound, $\mathcal{LIFE}$ simply uses basic information such as font size and relative spacing and alignment. Then, as soon as the needed information becomes available (either by default or explicit choice),

[15] See: http://wikix.ilog.fr/wiki/bin/view/Main/HassanAitKaci#3

every piece of the specified graphical set of objects eventually falls into place. As it turns out, this is all one needs for specifying constrained graphics fully declaratively—and this is indeed what the $\mathcal{LIFE}$ graphical toolkit does [2].

Fig. 2 shows the GUI display generated from the $\mathcal{LIFE}$ specification. In Appendix Section A.2, we give the complete actual code for the main predicate generating this GUI panel so that the reader may have an idea of the ease with which such sophisticated interactive controls can be specified in $\mathcal{LIFE}$.

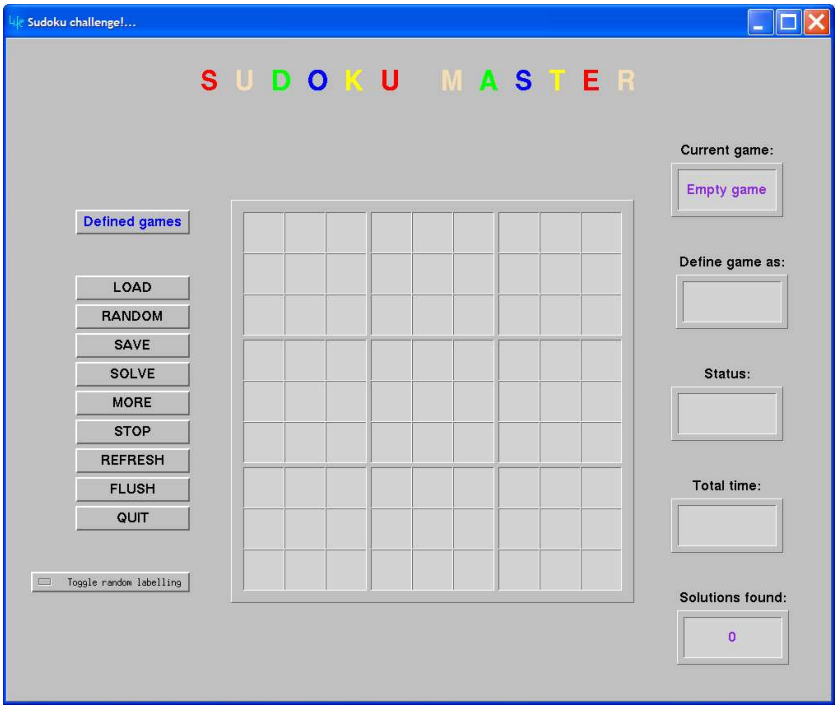

Figure 2: *Su Doku* game panel generated by $\mathcal{LIFE}$

Fig. 3 displays an easy *Su Doku* game to be solved. For the reader who wants to try solving this puzzle, we have postponed showing the solution found by our $\mathcal{LIFE}$ *Su Doku* solver in the display of Fig. 10 on Page 27 (also shown is the solving time).

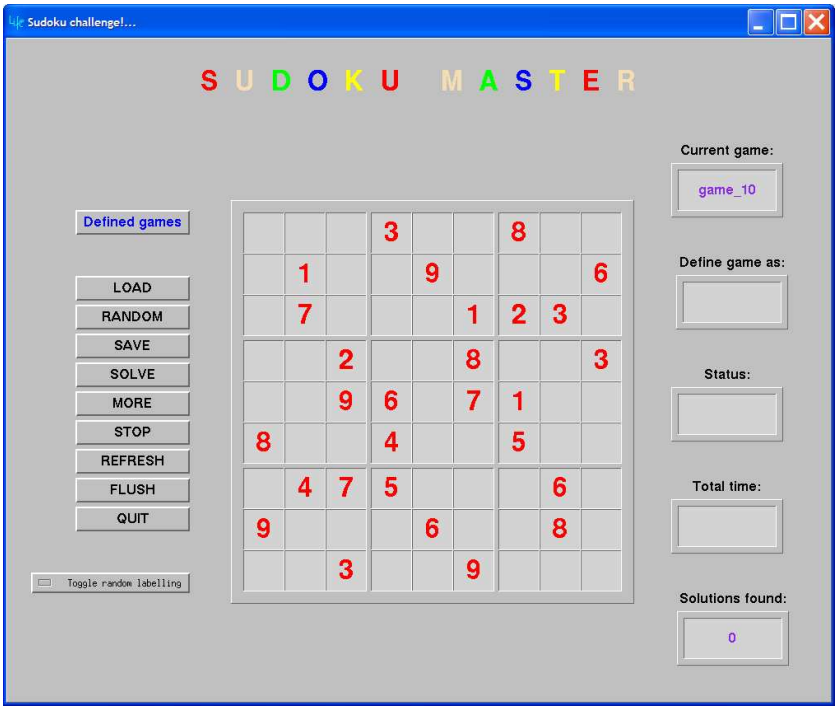

Figure 3: A defined *Su Doku* game display

7 Conclusion

> In life, the earlier one fails, the earlier one eventually succeeds!
>
> ALTAÏR EL-GHOUL

We have presented a purely declarative (yet duly executable) specification of the popular game puzzle *Su Doku* in the form of a disconcertingly simple (yet surprisingly effective—indeed, efficient!) $\mathcal{LIFE}$ program. The well-known key for solving this puzzle being the (in)famous '`alldiff`' global constraint and this constraint not being built into $\mathcal{LIFE}$, the contribution of interest is thus how $\mathcal{LIFE}$ is yet capable of realizing it efficiently thanks to its original data structure (the ψ-term) and control strategy (residuation). The former are extensible record structures and the latter is an automatic suspension/resumption scheme allowing suspended constraints to act as powerful search-tree pruners. We have also illustrated how $\mathcal{LIFE}$'s constraint system is amenable to specifying effective GUIs purely declaratively by specifying one for our $\mathcal{LIFE}$ *Su Doku* solver. The exercise is presented as an interesting, indeed entertaining, Programming Pearl illustrating the originality of $\mathcal{LIFE}$ as well as its adequacy for efficient declarative programming. One may retrieve this *Su Doku* solver and its GUI,[16] as well as the `WildLife 1.02` system itself.[17]

[16] See: `http://wikix.ilog.fr/wiki/pub/Main/HassanAitKaci/life_sudoku.tar.gz`
[17] See: `http://wikix.ilog.fr/wiki/bin/view/Main/HassanAitKaci#3`

References

[1] DUCHIER, D. Compiling the typed-polymorphic label-selective λ-calculus. Research Report ISG-RR-95-1, Intelligent Software Group, Simon Fraser University, Burnaby, BC, Canada, May 1995. [Available online[18]].

[2] DUMANT, B., AND HASSAN AÏT-KACI. A graphical toolkit in LIFE. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Graphics* (Maastricht, The Netherlands, September 1995), pp. 161–174.

[3] HASSAN AÏT-KACI. An introduction to LIFE—Programming with Logic, Inheritance, Functions, and Equations. In *Proceedings of the International Symposium on Logic Programming* (October 1993), D. Miller, Ed., MIT Press, pp. 52–68. [Available online[19]].

[4] HASSAN AÏT-KACI. *$\mathcal{LIFE}$ Su Doku*. In *Proceedings of the Tunisia–Japan Workshop on Symbolic Computation in Software Science* (Gammarth, Tunisia, September 2009), A. Bouhoula and T. Ida, Eds. [Available online[20]].

[5] HASSAN AÏT-KACI DUMANT, B., MEYER, R., PODELSKI, A., AND VAN ROY, P. The Wild_LIFE handbook. Technical report, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, March 1994. [Available online[21]].

[6] HASSAN AÏT-KACI AND PODELSKI, A. Towards a meaning of LIFE. *Journal of Logic Programming 16* (1993), 255–274.

[7] HASSAN AÏT-KACI AND PODELSKI, A. Functions as passive constraints in LIFE. *ACM Transactions on Languages and Systems 16*, 4 (1994), 1279–1318. [Available online[22]].

[8] JAFFAR, J., MICHAYLOV, S., STUCKEY, P., AND YAP, R. The CLP($\mathbb{R}$) language and system. *ACM Transactions on Programming Languages and Systems 14*, 3 (July 1992), 339–395.

[9] KNUTH, D. E. *The TEXbook*. Addison-Wesley, 1984.

[10] OLDER, W., AND BENHAMOU, F. Programming in CLP($\mathbf{BNR}$). In *First Workshop on Principles and Practice of Constraint Programming (PPCP'93)* (Providence, RI, 1993), P. Kanellakis, J.-L. Lassez, and V. Saraswat, Eds. [Available online[23]].

[11] PODELSKI, A., AND VAN ROY, P. A detailed algorithm testing guards over feature trees. In *Constraint Processing* (Heidelberg, Germany, 1995), Springer-Verlag, pp. 11–38. LNCS Volume 923.

[12] PUGET, J.-F. A C++ implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems* (Singapore, 1994). [Available online[24]].

[13] RÉGIN, J.-C. The symmetric `alldiff` constraint. In *Proceedings of the International Conference on Artificial Intelligence (IJCAI'99)* (Stockholm, Sweden, 1999), pp. 420–425. [Available online[25]].

[18] See: http://www.mozart-oz.org/users/duchier/papers/label-selective.ps.gz
[19] See: http://wikix.ilog.fr/wiki/pub/Main/HassanAitKaci/ilps93.ps.gz
[20] See: http://wikix.ilog.fr/wiki/pub/Main/HassanAitKaci/scss09.pdf
[21] See: http://citeseer.ist.psu.edu/134450.html
[22] See: http://doi.acm.org/10.1145/183432.183526
[23] See: http://citeseer.ist.psu.edu/older93programming.html
[24] See: http://www.ilog.com/products/optimization/tech/research/spicis94.pdf
[25] See: http://www.constraint-programming.com/people/regin/papers/symalldi.ps

[14] RÉGIN, J.-C. Filtering algorithms based on graph theory. Lecture note slides at International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'04), April 2004. [Available online[26]].

[15] SIMONIS, H. Sudou as a constraint problem. In *Proceedings of the Fourth Workshop on Modelling and Reformulating Constraint Satisfaction Problems* (October 2005), B. Hnich, P. Prosser, , and B. Smith, Eds., pp. 13–27. [Available online[27]].

[16] VAN HOEVE, W.-J. The `alldifferent` constraint: a survey. In *Proceedings of the Sixth Annual Workshop of the ERCIM Working Group on Constraints* (Prague, Czech Republic, June 2001), European Research Consortium for Informatics and Mathematics. [Available online[28]].

Acknowledgments

The author is indebted to the referees of SCSS 2009[29] for their helpful comments. Thanks also to Pierre-Etienne Moreau for judicious technical comments and sound editorial advice, to Eliès and Èta for enticing me to join the Su Doku craze—although I did it the lazy way: by letting $\mathcal{LIFE}$ deal with it! Many thanks also to Isaline, her family, and their two cats (dainty Charlotte and cool Touski), for kindly harboring me during the time I took composing this paper.[30]

[26] See: http://www.constraint-programming.com/people/regin/papers/GraphAndCP.pdf
[27] See: http://www.inf.tu-dresden.de/content/institutes/ki/cl/study/winter07/fcp/fcp/sudoku.pdf
[28] See: http://www.andrew.cmu.edu/user/vanhoeve/papers/ercim2001.pdf
[29] Tunisia–Japan Workshop on Symbolic Computation in Software Science, Gammarth, Tunisia, September 22-24, 2009—http://www2.score.cs.tsukuba.ac.jp/scssWorkshop/index.html.
[30] Sadly, *Touski* (so named because when he was a kitten he'd grab and played with *anything that moved*—in French, *tout ce qui bouge*) has died, the victim of a feral beast, while still in his prime youth.

Appendix

A The $\mathcal{LIFE}$ code

A.1 The *Su Doku* solver

The file "`sudoku.lf`" contains only the pure *Su Doku* solver in $\mathcal{LIFE}$. One can use this solver at the top level of `WildLife 1.02`. One can also interact with the solver using a constraint-driven $\mathcal{LIFE}$/`X` Window graphical interface: one must then import module "`x_sudoku`" defined in file "`x_sudoku.lf`" and then submit the query '`play_sudoku?`'.

```
module("sudoku"), public(sudoku,sudoku_solver)?
import("alldiff")?
```

A *Su Doku* game consists in enforcing the constraints making up the game's rules:

```
sudoku(@(@(X11,X12,X13,X14,X15,X16,X17,X18,X19),
         @(X21,X22,X23,X24,X25,X26,X27,X28,X29),
         @(X31,X32,X33,X34,X35,X36,X37,X38,X39),
         @(X41,X42,X43,X44,X45,X46,X47,X48,X49),
         @(X51,X52,X53,X54,X55,X56,X57,X58,X59),
         @(X61,X62,X63,X64,X65,X66,X67,X68,X69),
         @(X71,X72,X73,X74,X75,X76,X77,X78,X79),
         @(X81,X82,X83,X84,X85,X86,X87,X88,X89),
         @(X91,X92,X93,X94,X95,X96,X97,X98,X99))) :-
    % The rows constraints:
    alldiff(X11,X12,X13,X14,X15,X16,X17,X18,X19),
    alldiff(X21,X22,X23,X24,X25,X26,X27,X28,X29),
    alldiff(X31,X32,X33,X34,X35,X36,X37,X38,X39),
    alldiff(X41,X42,X43,X44,X45,X46,X47,X48,X49),
    alldiff(X51,X52,X53,X54,X55,X56,X57,X58,X59),
    alldiff(X61,X62,X63,X64,X65,X66,X67,X68,X69),
    alldiff(X71,X72,X73,X74,X75,X76,X77,X78,X79),
    alldiff(X81,X82,X83,X84,X85,X86,X87,X88,X89),
    alldiff(X91,X92,X93,X94,X95,X96,X97,X98,X99),
    % The columns constraints:
    alldiff(X11,X21,X31,X41,X51,X61,X71,X81,X91),
    alldiff(X12,X22,X32,X42,X52,X62,X72,X82,X92),
    alldiff(X13,X23,X33,X43,X53,X63,X73,X83,X93),
    alldiff(X14,X24,X34,X44,X54,X64,X74,X84,X94),
    alldiff(X15,X25,X35,X45,X55,X65,X75,X85,X95),
    alldiff(X16,X26,X36,X46,X56,X66,X76,X86,X96),
    alldiff(X17,X27,X37,X47,X57,X67,X77,X87,X97),
    alldiff(X18,X28,X38,X48,X58,X68,X78,X88,X98),
    alldiff(X19,X29,X39,X49,X59,X69,X79,X89,X99),
```

```
    % The square constraints:
    alldiff(X11,X12,X13,X21,X22,X23,X31,X32,X33),
    alldiff(X14,X15,X16,X24,X25,X26,X34,X35,X36),
    alldiff(X17,X18,X19,X27,X28,X29,X37,X38,X39),
    alldiff(X41,X42,X43,X51,X52,X53,X61,X62,X63),
    alldiff(X44,X45,X46,X54,X55,X56,X64,X65,X66),
    alldiff(X47,X48,X49,X57,X58,X59,X67,X68,X69),
    alldiff(X71,X72,X73,X81,X82,X83,X91,X92,X93),
    alldiff(X74,X75,X76,X84,X85,X86,X94,X95,X96),
    alldiff(X77,X78,X79,X87,X88,X89,X97,X98,X99).
```

The predicate '`labels`' specifies that the *Su Doku* cells may only be $1..9$ digits:

```
labels(@(@(X11,X12,X13,X14,X15,X16,X17,X18,X19),
         @(X21,X22,X23,X24,X25,X26,X27,X28,X29),
         @(X31,X32,X33,X34,X35,X36,X37,X38,X39),
         @(X41,X42,X43,X44,X45,X46,X47,X48,X49),
         @(X51,X52,X53,X54,X55,X56,X57,X58,X59),
         @(X61,X62,X63,X64,X65,X66,X67,X68,X69),
         @(X71,X72,X73,X74,X75,X76,X77,X78,X79),
         @(X81,X82,X83,X84,X85,X86,X87,X88,X89),
         @(X91,X92,X93,X94,X95,X96,X97,X98,X99))) :-
    % Specify the cell labels:
    X11=label, X12=label, X13=label, X14=label, X15=label,
    X16=label, X17=label, X18=label, X19=label, X21=label,
    X22=label, X23=label, X24=label, X25=label, X26=label,
    X27=label, X28=label, X29=label, X31=label, X32=label,
    X33=label, X34=label, X35=label, X36=label, X37=label,
    X38=label, X39=label, X41=label, X42=label, X43=label,
    X44=label, X45=label, X46=label, X47=label, X48=label,
    X49=label, X51=label, X52=label, X53=label, X54=label,
    X55=label, X56=label, X57=label, X58=label, X59=label,
    X61=label, X62=label, X63=label, X64=label, X65=label,
    X66=label, X67=label, X68=label, X69=label, X71=label,
    X72=label, X73=label, X74=label, X75=label, X76=label,
    X77=label, X78=label, X79=label, X81=label, X82=label,
    X83=label, X84=label, X85=label, X86=label, X87=label,
    X88=label, X89=label, X91=label, X92=label, X93=label,
    X94=label, X95=label, X96=label, X97=label, X98=label,
    X99=label.
```

The nullary function '`label`' returns a different digit in $1..9$ following the natural ordering (from 1 up to 9) each time it is backtracked over.

```
label -> { 1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9 }.
```

The *Su Doku* solver itself is defined as the predicate '`sudoku_solver`'. It simply constrains the *Su Doku* grid and specifies the cell labels:

```
sudoku_solver(G) :- sudoku(G), labels(G).
```

A.2 The *Su Doku* GUI

Row, row, row, your boat
Gently down the stream
Merrily, merrily, merrily
Life is but a dream...

ANNE O'NYMOUS

Here is is a purely declarative—although duly executable!—*LIFE* specification for a simple graphical interface to play *Su Doku* games. The main predicate to invoke is '`play_sudoku`': it specifies and creates the control panel display for the *Su Doku* graphical interface. Since it is a predicate controlling an interactive GUI panel that must be closed on exit, it consists of a disjunction whose first term builds and activates the GUI panel, and whose second term closes the panel and exits the interaction.

We only show the code for '`play_sudoku`'. This code uses help functions using and returning constructs of *LIFE*'s X Window toolkit, a modular library written in *LIFE* itself based on a raw X Window interface to C functions. Calls to the X Window C library functions are simply passed on to the X Window system or residuate according to whether or not they contain sufficiently instantiated terms as input parameters.

LIFE's X Window toolkit contains abstract predicates and functions that allow for easy relative positioning of graphical objects. The toolkit uses TEX's "box-and-glue" model [9, 2]. It consists of high-level functions and predicates, all written in *LIFE* on top of the raw X Window primitives, meant to ease graphical object construction. Their names have a mnemonic structure corresponding to the nature of the constructed objects and/or the constraints imposed thereon. For example, some such functions and constructs used in the code below are:

- '**`p_button`**' constructs and returns a push-button widget;
- '**`on_off_button`**' constructs and returns an on/off-button widget;
- '**`menu_button`**' constructs and returns a menu-button widget;
- '**`menu_panel`**' constructs and returns a menu panel;
- '**`menu_list`**' (prefix operator) uses its argument (a list of graphical frames or widgets) to construct and return a menu-list widget;
- '**`h_box`**' constructs and returns a horizontal box of given width (in pixels);
- '**`v_box`**' constructs and returns a vertical box of given height (in pixels);
- '**`ht_list`**' (prefix operator) uses its argument (a list of graphical frames or widgets) to construct and return a box containing the horizontal top-aligned sequence of boxes from the list;
- '**`hc_list`**' (prefix operator) uses its argument (a list of graphical frames or widgets) to construct and return a box containing the horizontal centered sequence of boxes from the list;
- '**`vc_list`**' (prefix operator) uses its argument (a list of graphical frames or widgets) to construct and return a box containing the vertical centered sequence of boxes from the list;
- '**`vr_list`**' (prefix operator) uses its argument (a list of graphical frames or widgets) to construct and return a box containing the vertical right-aligned sequence of boxes from the list;
- '**`vl_list`**' (prefix operator) uses its argument (a list of graphical frames or widgets) to construct and return a box containing the vertical left-aligned sequence of boxes from the list;
- '**`same_size`**' imposes that all the elements of the given list be widgets of equal size (*i.e.*, height and width);
- '**`containing`**' (infix operator) function returning its first argument (a graphical frame or widget) after including the second argument (a graphical frame or widget) in the first one;

- '**`create_boxes`**' takes a lists of graphical objects and creates them.

```
play_sudoku :-
   (
     % Save choice point for exit on QUIT:

     ExitPoint = get_choice,

     % A glitzy title box:

     Title = fancy_text_box(text   => "SUDOKU MASTER",
                            font   => title_font,
                            colors => [red,ivory,green,blue,yellow]),

     % The display is a list of boxes making up the Su Doku grid cells:

     Display = [ C11:cell(1,1) , C12:cell(1,2) , C13:cell(1,3)
               , C14:cell(1,4) , C15:cell(1,5) , C16:cell(1,6)
               , C17:cell(1,7) , C18:cell(1,8) , C19:cell(1,9)
               , C21:cell(2,1) , C22:cell(2,2) , C23:cell(2,3)
               , C24:cell(2,4) , C25:cell(2,5) , C26:cell(2,6)
               , C27:cell(2,7) , C28:cell(2,8) , C29:cell(2,9)
               , C31:cell(3,1) , C32:cell(3,2) , C33:cell(3,3)
               , C34:cell(3,4) , C35:cell(3,5) , C36:cell(3,6)
               , C37:cell(3,7) , C38:cell(3,8) , C39:cell(3,9)
               , C41:cell(4,1) , C42:cell(4,2) , C43:cell(4,3)
               , C44:cell(4,4) , C45:cell(4,5) , C46:cell(4,6)
               , C47:cell(4,7) , C48:cell(4,8) , C49:cell(4,9)
               , C51:cell(5,1) , C52:cell(5,2) , C53:cell(5,3)
               , C54:cell(5,4) , C55:cell(5,5) , C56:cell(5,6)
               , C57:cell(5,7) , C58:cell(5,8) , C59:cell(5,9)
               , C61:cell(6,1) , C62:cell(6,2) , C63:cell(6,3)
               , C64:cell(6,4) , C65:cell(6,5) , C66:cell(6,6)
               , C67:cell(6,7) , C68:cell(6,8) , C69:cell(6,9)
               , C71:cell(7,1) , C72:cell(7,2) , C73:cell(7,3)
               , C74:cell(7,4) , C75:cell(7,5) , C76:cell(7,6)
               , C77:cell(7,7) , C78:cell(7,8) , C79:cell(7,9)
               , C81:cell(8,1) , C82:cell(8,2) , C83:cell(8,3)
               , C84:cell(8,4) , C85:cell(8,5) , C86:cell(8,6)
               , C87:cell(8,7) , C88:cell(8,8) , C89:cell(8,9)
               , C91:cell(9,1) , C92:cell(9,2) , C93:cell(9,3)
               , C94:cell(9,4) , C95:cell(9,5) , C96:cell(9,6)
               , C97:cell(9,7) , C98:cell(9,8) , C99:cell(9,9)
               ],

     % The Su Doku grid's rows:

     Row1 = ht_list[ C11,C12,C13 , h_box(5) , C14,C15,C16 , h_box(5) , C17,C18,C19 ],
     Row2 = ht_list[ C21,C22,C23 , h_box(5) , C24,C25,C26 , h_box(5) , C27,C28,C29 ],
     Row3 = ht_list[ C31,C32,C33 , h_box(5) , C34,C35,C36 , h_box(5) , C37,C38,C39 ],
     Row4 = ht_list[ C41,C42,C43 , h_box(5) , C44,C45,C46 , h_box(5) , C47,C48,C49 ],
     Row5 = ht_list[ C51,C52,C53 , h_box(5) , C54,C55,C56 , h_box(5) , C57,C58,C59 ],
     Row6 = ht_list[ C61,C62,C63 , h_box(5) , C64,C65,C66 , h_box(5) , C67,C68,C69 ],
     Row7 = ht_list[ C71,C72,C73 , h_box(5) , C74,C75,C76 , h_box(5) , C77,C78,C79 ],
     Row8 = ht_list[ C81,C82,C83 , h_box(5) , C84,C85,C86 , h_box(5) , C87,C88,C89 ],
     Row9 = ht_list[ C91,C92,C93 , h_box(5) , C94,C95,C96 , h_box(5) , C97,C98,C99 ],
```

```
% The Su Doku grid:

Grid = frame_box(vl_list [ Row1, Row2, Row3
                         , v_box(5)
                         , Row4, Row5, Row6
                         , v_box(5)
                         , Row7, Row8, Row9
                         ],
                 padding => 10),

% The control buttons:

Load    = p_button(text   => "LOAD",
                   action => load_games(Display)),
Save    = p_button(text   => "SAVE",
                   action => save_all_games),
Solve   = p_button(text   => "SOLVE",
                   action => solve(Display)),
More    = p_button(text   => "MORE",
                   action => more(Display)),
Random  = p_button(text   => "RANDOM",
                   action => random_seeds(Display)),
Stop    = p_button(text   => "STOP",
                   action => stop),
Refresh = p_button(text   => "REFRESH",
                   action => (refresh(Display),
                              reset_state(Refresh,false))),
Flush   = p_button(text   => "FLUSH",
                   action => clear_all(Display)),
Quit    = p_button(text   => "QUIT",
                   action => (set_choice(ExitPoint),fail)),

% Imposing a same_size constraint on the control buttons:

same_size([Games,Load,Random,Save,Solve,More,Stop,Refresh,Flush,Quit]),

% A toggle button to switch to random labelling mode
% (negative offset means it's right-aligned):

Toggle  = on_off_button(text    => "Toggle random labelling",
                        font_id => button_font,
                        offset  => -10,
                        action  => toggle_random_labelling),

% The defined game menu:

Menu = menu_panel containing menu_list defined_games(Display),
Games = menu_button(text          => "Defined games",
                    font_id       => z_font,
                    text_color_id => blue,
                    menu          => Menu),
```

```
% We now define info boxes to display containing the
% game's name, status, time, etc., ...


% Binding CurrentFrame to a framed edit box (the current
% game name's edit box, which gets then bound to Current)
% captioned "Current game:", where the name of the current
% game is to be entered and displayed:


CurrentFrame = current_frame("Current game:",Current,Display),
reset_text(Current,"Empty game"),


% Binding DefineFrame to a framed edit box (which gets then
% bound to Define) captioned "Define game as:", where a name
% redefining the current game is to be entered and displayed:


DefineFrame = define_frame("Define game as:",Define,Display),


% Binding StatusFrame to a framed info box (which gets then
% bound to Status) captioned "Status:"  where solving status
% will be displayed:


StatusFrame = info("Status:",Status),


% Binding TimeFrame to a framed info box (which gets then
% bound to Time) captioned "Total Time:", where the total
% solving time will be displayed:


TimeFrame = info("Total time:",Time),


% Binding CountFrame to a framed info box (which gets
% bound to Count) captioned "Number of Solutions:",
% where the total number of solutions found so far will be
% displayed:


CountFrame = info("Solutions found:",Count),
reset_text(Count, "0"),


% Attaching some of the widgets to Display as extra features:


Display = @(current => Current, games  => Games,
            define  => Define,  status => Status,
            time    => Time,    count  => Count,
            more    => More,    random => Random,
            solve   => Solve,   stop   => Stop),
```

```
  % Putting together the main display panel:

  Panel = panel(title => "Sudoku challenge!...")
    containing
    padded_box(vc_list [ Title
                       , v_box(30)
                       , hc_list [ vr_list [ Games   , v_box(50)
                                           , Load    , v_box(5)
                                           , Random  , v_box(5)
                                           , Save    , v_box(5)
                                           , Solve   , v_box(5)
                                           , More    , v_box(5)
                                           , Stop    , v_box(5)
                                           , Refresh , v_box(5)
                                           , Flush   , v_box(5)
                                           , Quit    , v_box(50)
                                           , Toggle
                                           ]
                                 , h_box(50)
                                 , Grid
                                 , h_box(30)
                                 , vl_list [ CurrentFrame , v_box(10)
                                           , DefineFrame  , v_box(10)
                                           , StatusFrame  , v_box(10)
                                           , TimeFrame    , v_box(10)
                                           , CountFrame
                                           ]
                                 ]
                       ],
                padding => 20),

  % Finally, we create the Panel and Menu boxes and that's it:

  create_boxes([Panel,Menu])

;

  % This is the main backtrack point for graceful exit upon QUIT:

  write("Exiting Su Doku challenge ...\n"),
  succeed
).
```

B Some *Su Doku* challenges

Figures 4 to 9 contain some puzzles for the *Su Doku*-challenged reader's entertainment. They are given in order, ranging from easy (Fig. 4), to difficult (Figs. 5, 6), to nasty (Figs. 7, 8), to out-of-worldly diabolical (Fig. 9). These were collected from various Internet sites.

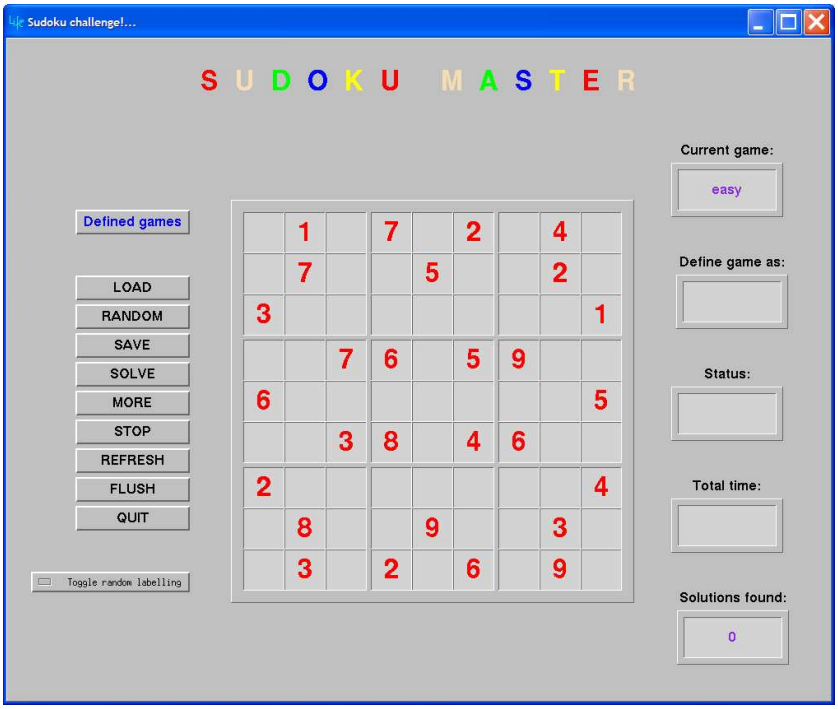

Figure 4: See solution displayed in Fig. 11.

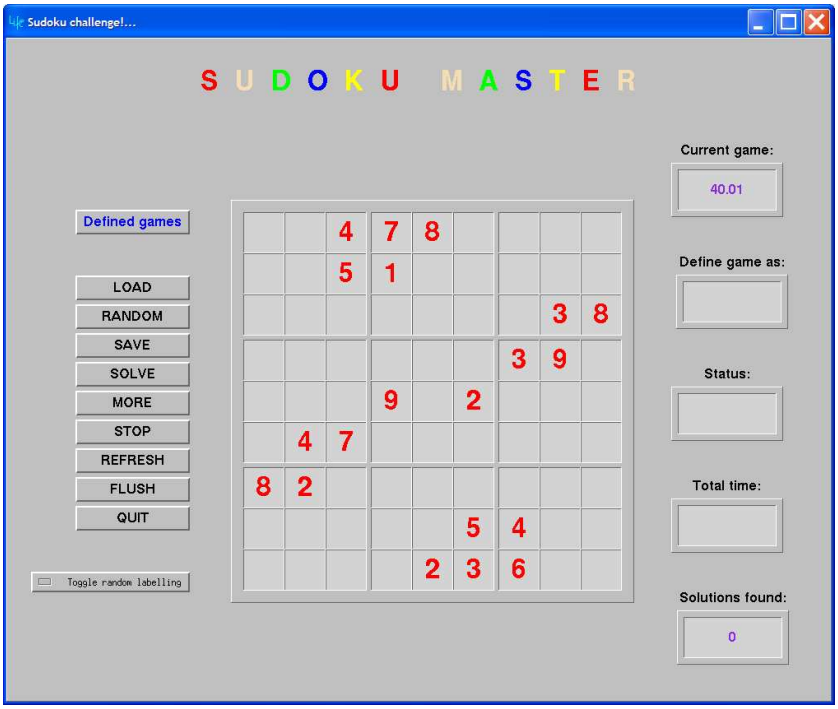

Figure 5: See solution displayed in Fig. 12.

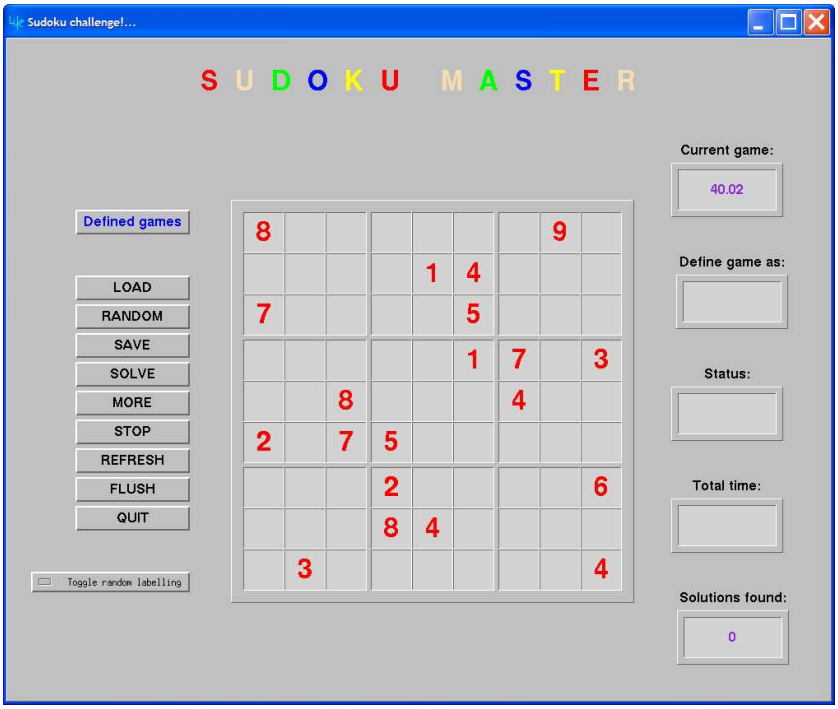

Figure 6: See solution displayed in Fig. 13.

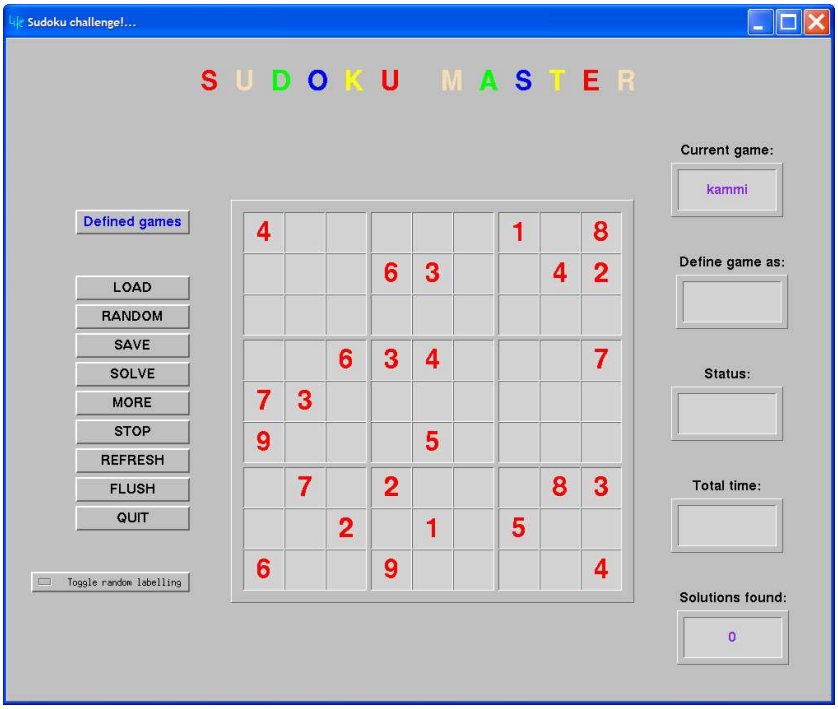

Figure 7: See solution displayed in Fig. 14.

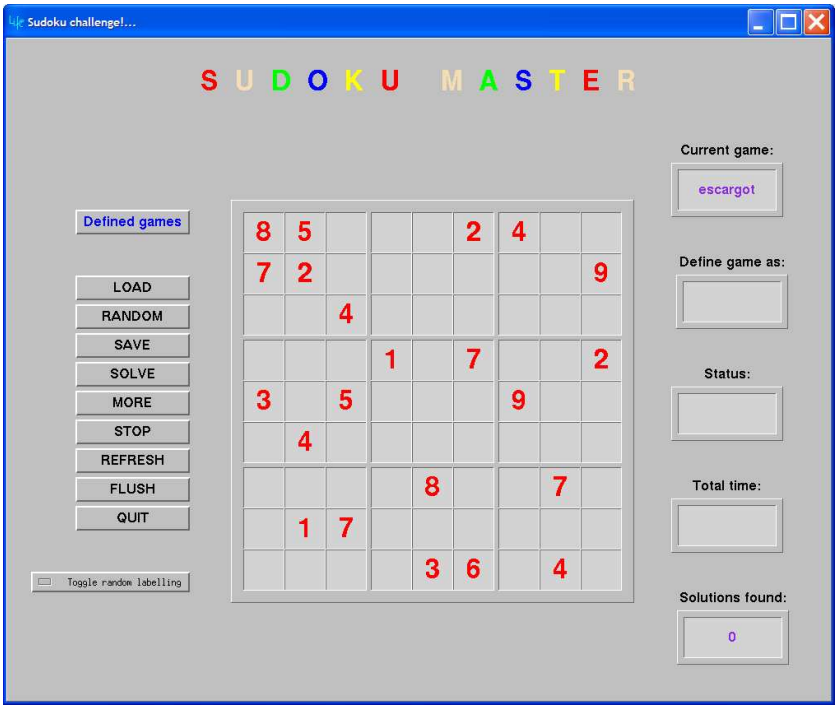

Figure 8: See solution displayed in Fig. 15.

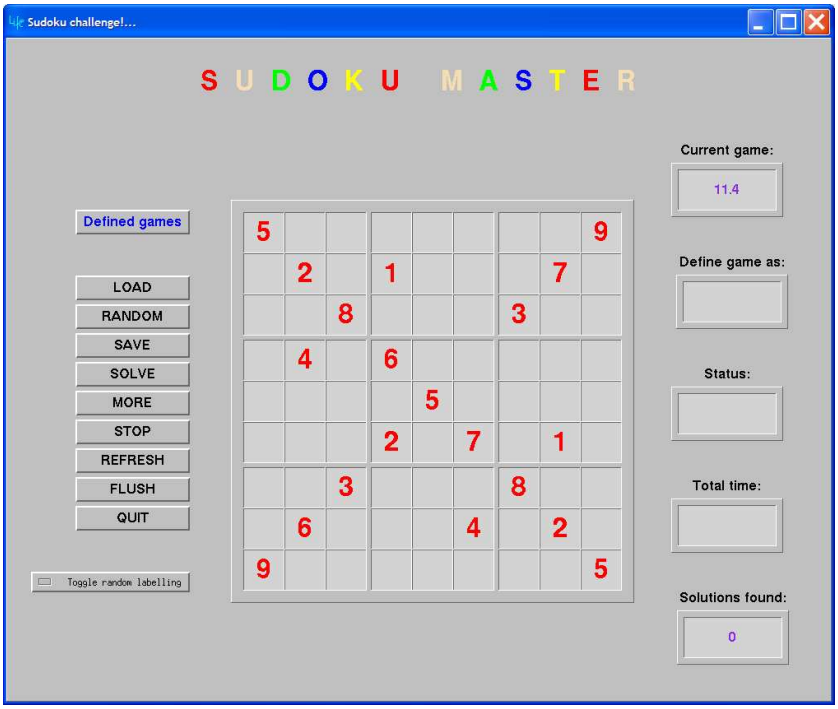

Figure 9: See solution displayed in Fig. 16.

C Solutions to the *Su Doku* challenges

Figures 10 to 16 contain the solutions to the seven puzzles proposed in this article in Figures 3 to 9. They are given here for the *Su Doku*-lazy reader's curiosity, as well as to illustrate $\mathcal{LIFE}$'s actual reaction to each puzzle. Note the solving times shown at the bottom right of the displays along each grid's solution, confirming the puzzles's estimated difficulty.

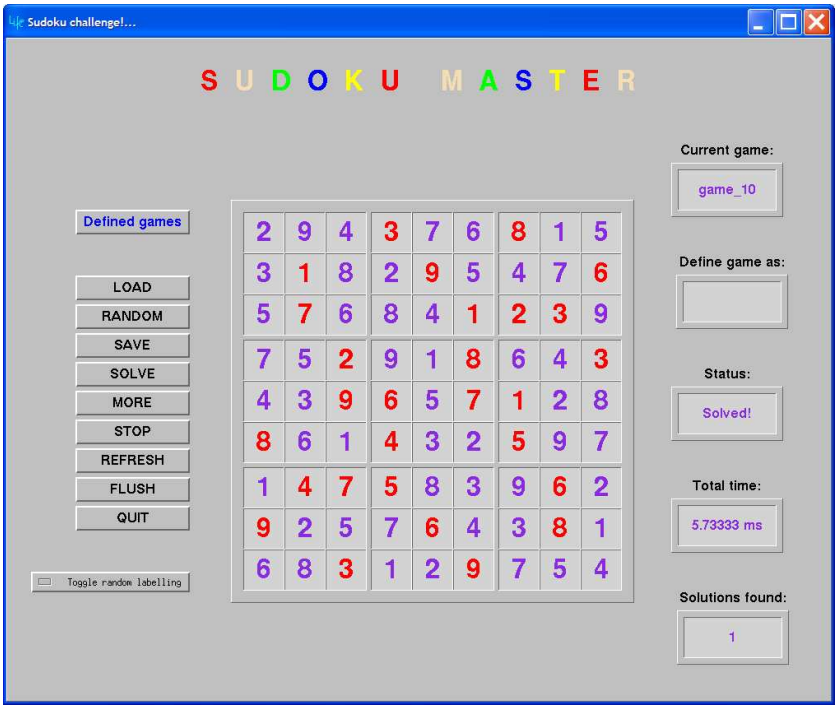

Figure 10: Solution display for the game in Fig. 3.

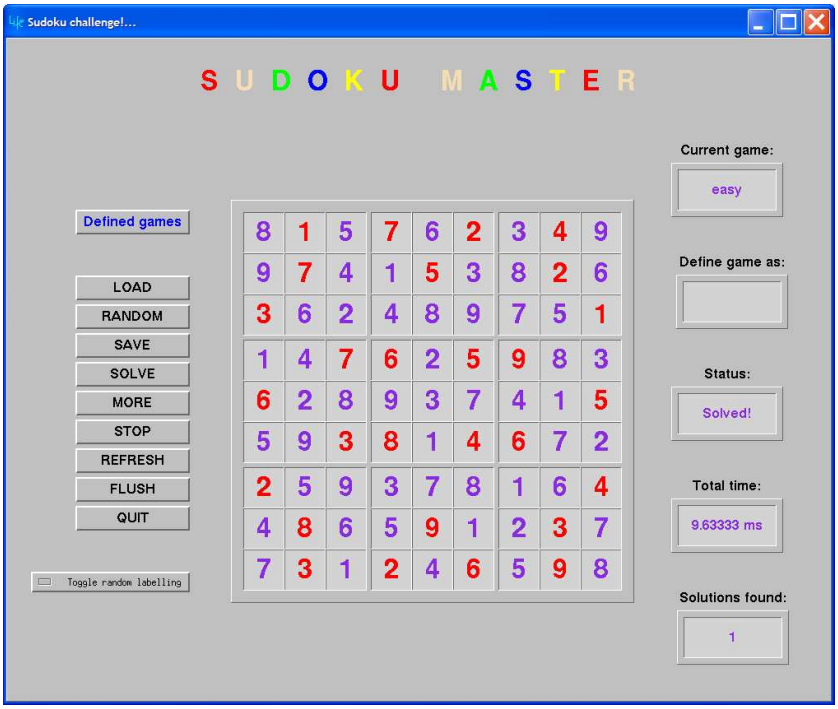

Figure 11: Solution display for the game in Fig. 4.

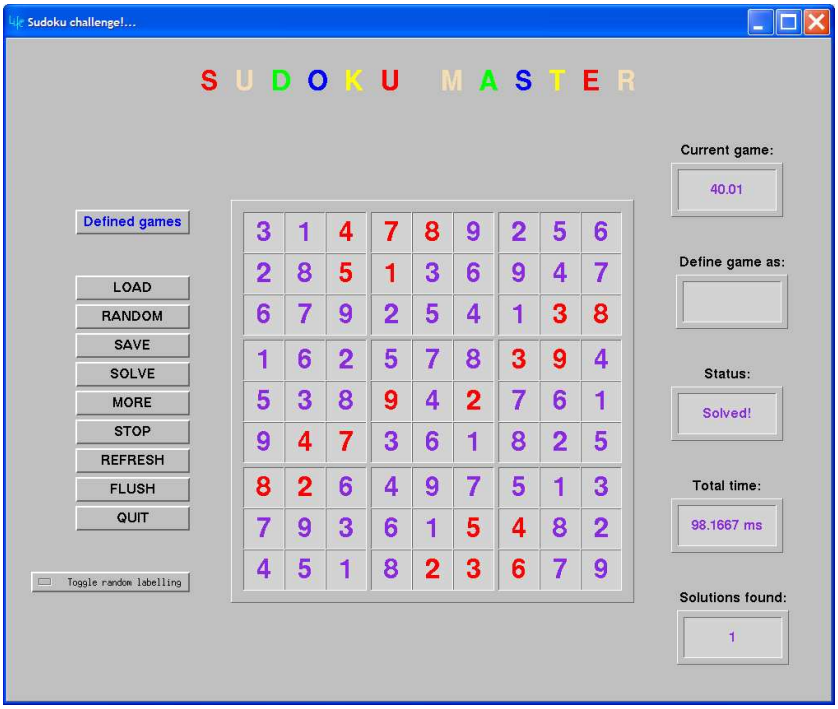

Figure 12: Solution display for the game in Fig. 5.

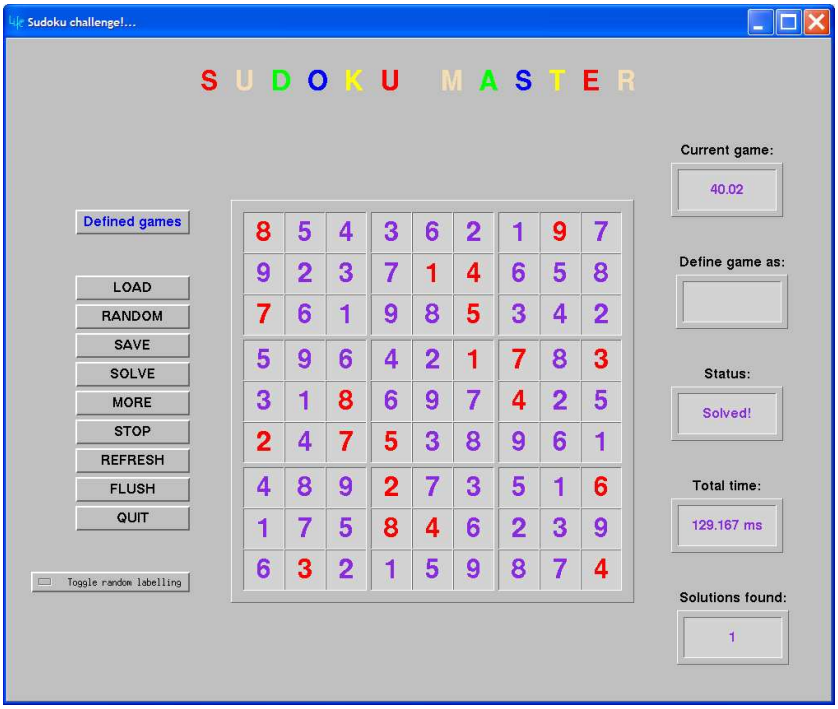

Figure 13: Solution display for the game in Fig. 6.

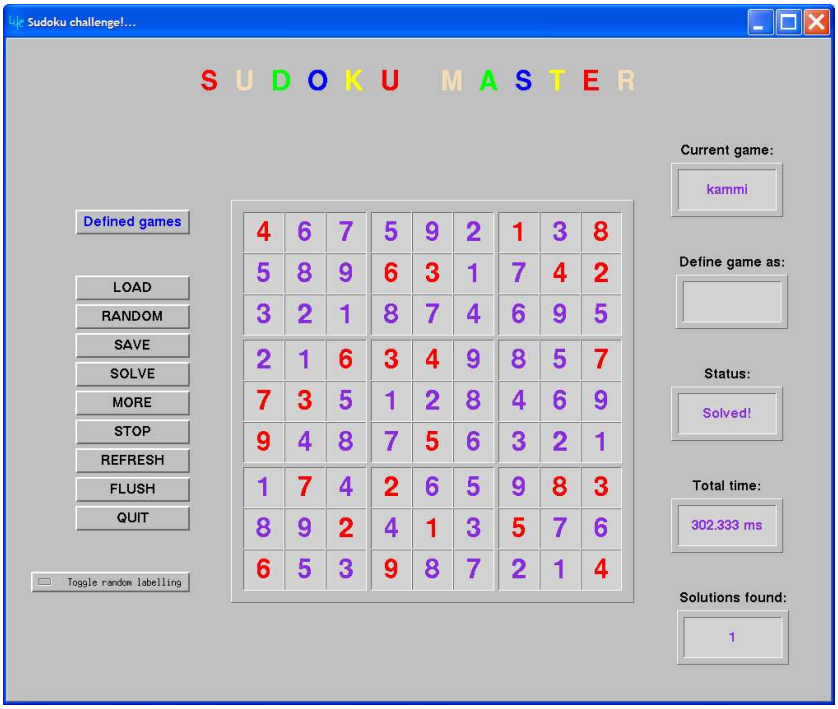

Figure 14: Solution display for the game in Fig. 7.

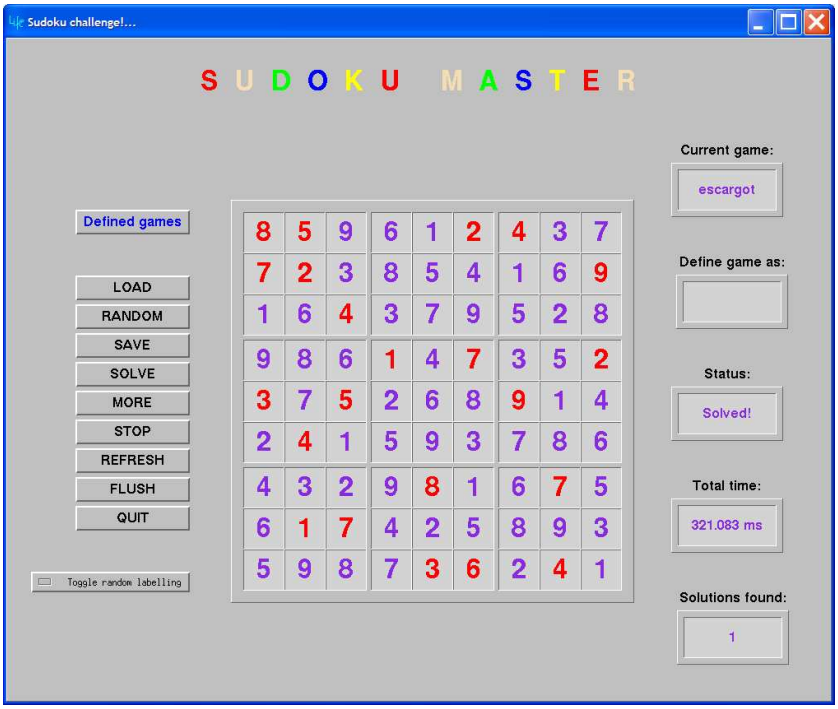

Figure 15: Solution display for the game in Fig. 8.

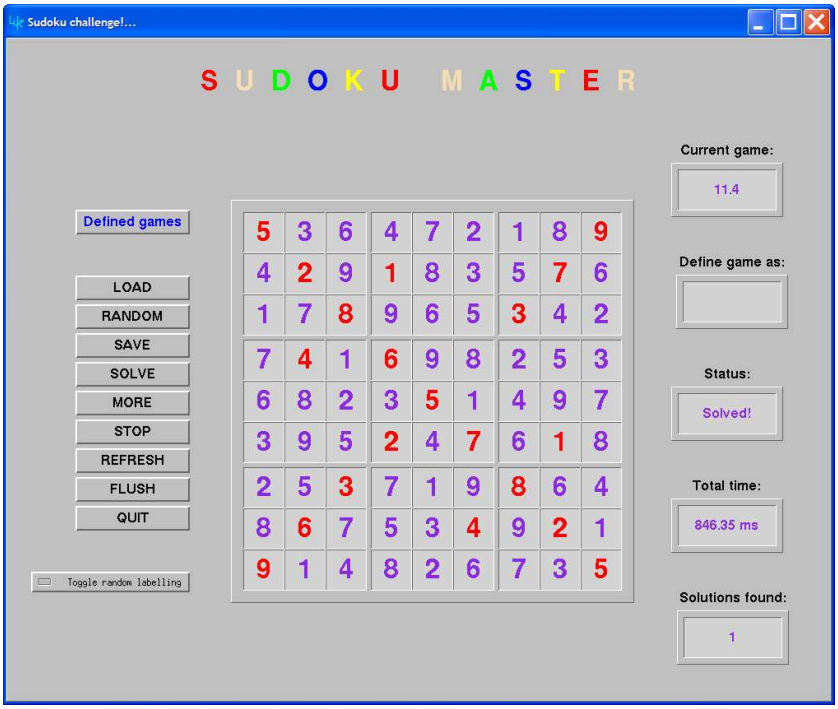

Figure 16: Solution display for the game in Fig. 9.