# An Introduction to LIFE—Programming with Logic, Inheritance, Functions, and Equations

**Hassan Aït-Kaci**
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92500 Rueil-Malmaison, France
hak@prl.dec.com

## Abstract

LIFE (Logic, Inheritance, Functions, Equations) is a programming language with a powerful facility for structured type inheritance. LIFE reconciles styles from functional programming and logic programming by implicitly delegating control to an automatic suspension mechanism. This allows interleaving interpretation of relational and functional expressions that specify abstract structural dependencies on objects. Together, these features provide a convenient and versatile power of abstraction for very high-level expression of constrained data structures.

... l'élément ne préexiste pas à l'ensemble, il n'est ni plus immédiat ni plus ancien, ce ne sont pas les éléments qui déterminent l'ensemble, mais l'ensemble qui détermine les éléments.
GEORGES PEREC *La vie, mode d'emploi.*[1]

## 1 Introduction

LIFE is a programming language originally conceived by Hassan Aït-Kaci and his colleagues at MCC, in Austin, Texas [5, 6]. It is a synthesis of three different programming paradigms: logic programming, functional programming and object-oriented programming.[2] At first sight it closely resembles Prolog, from which it partly derives its syntax and resolution method. However, the addition of functions, approximation structures and inheritance greatly enriches the language and allows one to formulate efficient programs more easily, more concisely, and—in our opinion—more naturally [9].

Still LIFE [17] was the first prototype of LIFE done at MCC by David Plummer in Quintus Prolog. As a first experimental implementation, Still LIFE was not a full embodiment of everything the designers had in mind. More problematically, it was slow, and plagued with a few chronic bugs. No further work on it being planned at MCC together with MCC's policy of withholding software as proprietary information have extinguished motivation to pursue its correction and completion.

[1]Life, a user manual.
[2]Or rather, a particular view of object-oriented programming dealing essentially with inheritance.

Wild LIFE is the successor of Still LIFE. Its implementation was carried out at Digital's Paris Research Laboratory (PRL). It is an independent, complete re-implementation, in C, of an interpreter for LIFE. In comparison with Still LIFE's Prolog implementation, Wild LIFE is much faster, even though, being just an interpreter, it has not been designed with performance in mind and therefore runs at interpreter's speed. More importantly, it is quite more complete, and more reliable, than Still LIFE with respect to the specified language. The interpreter represents roughly 28000 lines of highly portable code.[3] It was initially written, for its greatest part, by Richard Meyer for his Engineering Degree project under the supervision, and following the specifications, of Hassan Aït-Kaci. It was later corrected, completed, and extended by Peter Van Roy, also following Hassan Aït-Kaci's specifications. An interface to the X Window system was then added by Jean-Claude Hervé. Using Wild LIFE as a bootstrapper, a compiler for LIFE is currently being built at PRL by Peter Van Roy, Richard Meyer, and Bruno Dumant based on recent optimization technology [20], and using new algorithms for function coroutining [18] and sort unfolding [12].

This paper, extracted largely from [2], summarizes the essence of LIFE for someone interested in using it for actual programming. Of course, this is far from a complete tutorial on LIFE. However, it tries to give a faithful account of the main constructs of the language using Wild LIFE as a concrete representative. Therefore, in this paper, whenever we refer to LIFE as a concrete running language, we shall mean Wild LIFE.[4]

## 2 Generalities

LIFE is a generalization of Prolog. This is generally true to a point where little indeed need be changed in a Prolog program to run under LIFE. In fact, we shall tacitly assume that you, the reader, are familiar with Edinburgh-style syntax [14, 16, 19]. Thus, unless specifically indicated to be different, the same syntactic conventions will apply. In particular, variables are capitalized (or start with _) whereas all other identifiers start with a lower-case letter. The symbol `=` is the unification predicate, `:-` defines a rule, `!` is the cut predicate, *etc.* However, while most of what is known as Edinburgh-style syntax for Prolog has been retained in LIFE, a few minor but important departures from it do require some habit-breaking to the Prolog programmer.

When interacting with LIFE you may either make assertions, submit queries to be solved or exit by typing the key sequence CTRL-D which closes the input stream. The goal `halt`, as in (Edinburgh) Prolog can also be used to terminate LIFE.[5]

[3]Of which about 2000 lines are in LIFE.

[4]The Wild LIFE system is available as free public domain software distributed with source code. For retrieval information, send electronic mail to `life-request@prl.dec.com`. Use this same address to inquire about the LIFE's users mailing list.

[5]Whenever we refer to "Prolog" we mean a generic language of the family using "Edinburgh" style syntax; *e.g.*, DEC-10 Prolog, Quintus Prolog, SICStus Prolog, *etc.*

The following rule holds throughout:

- all program assertions are terminated by a period: `.`
- all queries (goals to be proved) are terminated by a question mark: `?`.

LIFE's interaction with the user is similar to Prolog's but it is in fact much more flexible as it extends it to allow the user to build a query in an incremental manner. Indeed, once a query has been proved and answered, the user is offered the possibility of extending it *using the resulting context*. Query levels in the form of increasingly indented and numbered prompts are thus printed to make one be aware of the depth in the query. Hence, upon success, LIFE will then prompt for further information and at this point you take either of the following actions:

- type $\langle$`CR`$\rangle$ to abandon the current query and go back to the previous level
- type `;` to force backtracking and look for another answer
- extend the query by typing a goal followed by `?`
- type `.` to pop up to the top-level prompt from any depth.

## 3 The basic data structure: $\psi$-terms

Just as Prolog is based on (first-order) terms, LIFE is based on $\psi$-terms [1]. If you simply substitute Prolog terms with LIFE's $\psi$-terms the resulting language is LOGIN [5]. In LIFE $\psi$-terms have been used to represent all data structures, including lists, clauses, functions and sorts.

### 3.1 Sorts

As a first approximation, LIFE's $\psi$-terms may be construed as record types. One could say that they are to first-order terms what flexible records are to static arrays. A $\psi$-term has a (principal, or root) *sort*. Sorts denote sets, and are partially-ordered with a subsort ordering denoting set inclusion. The partially ordered set of all sorts may be viewed as a so-called *inheritance hierarchy*. The subsorts of a sort inherit all the properties of the parent sort. Sorts may share common subsorts, thus allowing so-called *multiple inheritance*.

At the summit of the hierarchy of sorts, there is a greatest sort, denoted by $\top$, pronounced "top." It approximates everything as it denotes the universe of all objects in LIFE, and thus represents the union of all sorts. In LIFE, it is written as `@` (but still pronounced "top").[6] Likewise, at the base of this hierarchy we find the sort $\perp$, pronounced "bottom". This is the empty set and so any $\psi$-term containing $\perp$ denotes the empty set. In LIFE the symbol `{}` is used to represent $\perp$. Because of its collapsing effect on any structure that contains it, `{}` can appear explicitly in any useful way only in *non-strict* expressions like conditionals or so-declared definitions.

[6]Because $\top$ is not a standard ASCII character, we need an ASCII symbol to stand for it. Besides reminding a looped-around `a` that could stand for `anything`, the symbol `@` has a shape that is reminiscent of an embryo—a perfect ideogram to denote the most primeval sort in LIFE!

In LIFE, there is no conceptual difference made between values and sorts. Values are just singleton sorts. Thus, the value `1` is equivalent to the singleton set `{1}` and is a subsort of `int`, the built-in sort of all integers. In formal notation, we use the symbol $\lhd$ to denote the assertion that one sort is a subsort of another. In LIFE, this symbol is rendered as `<|` and used as a declaration.[7]

The built-in sorts available to the user are:

- All integers, real numbers, `int`, `real` with the declarations *n* `<| int` (for all integers *n*), *r* `<| real` (for all non integral real number *r*), and `int <| real`. For example, `0`, `-5`, and `3.0` are subsorts of `int` and thus of `real`, but `2.5` and `26.77` are not subsorts of `int` although they are subsorts of `real`.
- `list`, whose subsorts are the empty list `nil`, also written `[]`, and the non-empty list constructor `cons`, also written `[_|_]`. Non-empty lists may be written using all the flexibility of Prolog's syntax; *e.g.*:[8]
  `[a,b,c] = [a,b|[c]] = [a,b,c|[]] = [a|[b|[c|[]]]].`
- All strings *s* and `string`, with the declarations *s* `<| string`; specific strings appear within double quotes as in `"this is a string"`.
- `bool`, `true`, `false` with the declaration `true <| bool` and `false <| bool`.

### 3.1.1 Defining sorts

New sorts may be added by the user along with their relative position in the hierarchy.

Let us say that "*truck* $\lhd$ *vehicle*" in your intended model, which means that for the purpose of your program *"all trucks are vehicles, and any property pertaining to vehicles also applies to trucks."* You specify this in LIFE by typing:

```
> truck <| vehicle.

*** Yes
```

and then LIFE's unification will take that information into account:

```
> mobile(vehicle).

*** Yes
> useful(truck).

*** Yes
> mobile(X),useful(X)?

*** Yes
X = truck.
```

[7]This must not be confused with a predicate that would *test* whether to sorts are related.

[8]Note that the Prolog list notation is just syntactic sugaring. Internally, the sorts `nil` and `cons` are used and can also be used in a program indifferently for `[_|_]` and `[]`, respectively. For instance, `X = cons(2=>nil,1=>foo,3=>bar)` is the same as `X = [1](3=>bar)`, which is also the same as `X = cons(1,[],bar)`.

Besides their relative ordering, sorts may be specified to bear other information. Namely, they may be given attributes, coreference constraints among attributes, and more generally arbitrary relational and functional dependency constraints. This will be explained in the next sections.

### 3.1.2 Greatest Lower Bound

The intersection of two sorts is called the Greatest Lower Bound (GLB) of these sorts. It is their largest common subsort if it exists. However, in general such a unique common subsort does not exist. In that case, the intersection is the union of the maximal common subsorts. Such a union is seen as a *disjunction* of sorts. We write {$\texttt{a}_1$ ; ...; $\texttt{a}_n$} for the disjunction of the sorts $a_1, \ldots, a_n$.

Say we define `two_wheels` and `four_wheels` as the classes of objects having respectively 2 or 4 wheels, and the sort `vehicle` with the following hierarchy:

```
bike <| two_wheels.       bike <| vehicle.
truck <| four_wheels.     truck <| vehicle.
car <| four_wheels.       car <| vehicle.
toy_car <| four_wheels.   rolls_royce <| car.
```

Then we can establish the following values for the GLB of two pairs:

- GLB(`two_wheels`,`vehicle`) = `bike`
- GLB(`four_wheels`,`vehicle`) = {`car;truck`}
- GLB(`two_wheels`,`four_wheels`) = $\perp$
- GLB(`rolls_royce`,`car`) = `rolls_royce`
- GLB(`truck`,`@`) = `truck`

## 3.2 Attributes

A $\psi$-term represents a set of objects. All these objects may have certain attributes (or features). An attribute is a pair constituted by a label (or feature name) and an associated $\psi$-term. For example:

```
car(number_of_wheels => 4,
    manufacturer => string,
    maximum_speed => real).
```

An attribute label may be any unsigned natural number or any symbol, single-quoted if containing non-alphanumeric characters other that underscore. One may view a $\psi$-term as a generalization of conventional Prolog terms in that subterm positions are specified by explicit keywords rather then implicit numeric positions. Thus, for compatibility with Prolog, and also for ease of use, implicit positions may also be used in LIFE. More precisely, `thing(a,b,c)` is equivalent to `thing(1 => a, 2 => b, 3 => c)`. The order of attributes is completely irrelevant and so this $\psi$-term could also be entered as: `thing(2 => b, 3 => c, 1 => a)`.

Unlike Prolog terms, $\psi$-terms do not have fixed arities. This is an important deviation that causes a syntactic incompatibility with Prolog programs. Thus, in

Prolog, a functor is a pair `f/n` consisting of a symbol and its arity, such that two functors are considered equal if and only if they have both same symbol and same arity. Many Prolog programmers take advantage of this and use the same symbol with different arities to name distinct predicates. Clearly, this practice is no longer valid in LIFE.[9] Indeed, two $\psi$-terms with the same principal sort symbol but different numbers of arguments, or with different subterm attributes altogether, can very well unify. In fact, they may unify even with distinct root sorts, as long as these have a non-bottom GLB, and a $\psi$-term may acquire new attributes as a problem is solved (more information is learned about the object).

## 3.3 Variables and Tags

Variables start with _ or an upper case letter, as in Prolog. Unlike Prolog, in LIFE a variable is not restricted to appear only as the leaf of a term. Thus, variables can be used as (reference) *tags* within a $\psi$-term's structure and then used as explicit handles for referencing the part of $\psi$-term they tag. These references may be cyclic; *i.e.*, a variable may occur within a $\psi$-term tagged by it. As usual, variables are local to the clause or query in which they appear. The syntax expressing the tagging of a $\psi$-term `t` by a variable `X` is of the form `X:t`. It does not matter where a tagging occurs in a clause as it is global to the whole clause. In fact, even if a variable occurs not as a $\psi$-term's tag but as a simple isolated variable, it is implicitly considered to be tagging $\top$, exactly as if it had been written `X:@`. If the same variable needs to be constrained to be the conjunction of two terms, it is written using the `&` operator, as in `X:t1&t2`. This is equivalent to writing `X=t1, X=t2`.

## 3.4 Disjunctive terms

A disjunctive term is an expression of the form $\{t_1;\ldots;t_n\}$, $n \geq 0$, where each $t_i$ is either a $\psi$-term or a disjunctive term. A disjunctive term denotes the union of the various terms that compose it. Note that the empty disjunction is `{}` which naturally means the empty set, and explains why `{}` is used for $\perp$ in LIFE. Note also that a singleton disjunctive term `{t}` is equivalent to `t`, in natural accordance with the assimilation of a value to the singleton sort containing it.

In LIFE disjunctive terms are enumerated using a left-right depth-first backtracking strategy, exactly as Prolog's (and LIFE's!) predicate level resolution. That is, a disjunctive term takes the value of its first disjunct, then on backtracking successively takes the value of its second disjunct, *etc.*

- `A={1;2;3;4}?` is equivalent to `A=1;A=2;A=3;A=4?` where `;` signifies "or" in Edinburgh Prolog syntax.
- `p({a;b;c}).` is equivalent to asserting `p(a).  p(b).  p(c).`

[9] This is not a very serious limitation of compatibility as this practice is generally considered a bad one by serious Prolog programmers, and all Prolog programs where this is used can be systematically transformed.

- `write(vehicle&four_wheels)?` will first print `car` then on backtracking will print `truck`.

## 4 Predicate Definitions

As in Prolog, a predicate definition consists in one or several (definite) clauses. Definite clauses in LIFE are written in the same manner as they are in Prolog and behave in the same way, only $\psi$-terms replace terms. In many cases Prolog programs can run unaltered in LIFE. Clauses are stored in the assertion base in the same order as they are entered. This is not a tutorial on Prolog, so—for further information—please consult your local library.

Just as Prolog represents every syntactic construct as a first-order term (including definitions) so does LIFE use $\psi$-terms for the same purpose. This allows great flexibility for meta-programming in LIFE with added power thanks to the use of $\psi$-terms as opposed to simpler Prolog terms.

The general form of a clause is `Head :- Body.`, where Head and Body are two $\psi$-terms. If the sort of `Body` is a comma (`,`), the body is a conjunction: its first argument is its left conjunct and its second is its right conjunct. This form of a clause is called a *rule* and is read as "Head *if* Body". Another form of a clause is a *fact*, and is of the form: `Head`. A fact is really a special case of a rule as it is an abbreviation of `Head :- succeed.`[10]

A query (or resolvent) is a conjunction of atomic goals. An atomic goal in LIFE is a $\psi$-term whose root symbol is a predicate name. A query is proved using top-down/left-right SLD-resolution as in Prolog. As in Prolog the order in which the rules are entered is important as they will be tried in that order. In LIFE, attempting to prove a goal of an undefined predicate results in an error, unless this predicate has been declared `dynamic`, in which case it simply fails.

## 5 Functions

The availability of reducible functions is one of the major features that distinguish LIFE from Prolog [6]. They make programming much more flexible and more intuitive. In LIFE, functions of any functionality order may be used. They can be curryed, *i.e.*, called with missing arguments; and they can *"residuate"* (or suspend), *i.e.*, be called with insufficiently instantiated arguments. The latter feature allows programming with coroutines and constraints [8, 10, 11, 13].

[10]Prolog uses `true` rather than `succeed` as the predicate that admits any proof (and thus requires none), but `fail` instead of `false` as the predicate that has no proof. Thanks to the dual (declarative or procedural) interpretations of resolution, the confusion is mild. However, LIFE makes a more consistent choice by using `succeed/fail` for these predicates, reserving `true/false` for the boolean sorts returned as values of boolean functions.

## 5.1 Function Definitions

A function definition consists in one or several rules. These rules have a very simple syntax: `Head -> Body.`, where Head and Body are two $\psi$-terms. The first $\psi$-term is the head of the functions, the second is the resulting value. A functional expression is any $\psi$-term whose root symbol is a function symbol (*i.e.*, a symbol appearing as the root of the head of at least one function rule in the assertion base).[11] For example, one can define and use factorial as:

```
> fact(0) -> 1.
> fact(N:int) -> N*fact(N-1).
> write(fact(5))?
120
*** Yes
```

Since the head of a rule defining a function is a $\psi$-term, it is perfectly legal to define and use functional expressions using numerical positions as well as symbolic attributes to specify arguments.

Function rules are stored in the assertion base in the order they appear specified. As for clauses defining predicates, order of a function's rules matters as explained next.

As is the case with predicates, the rules defining a function are looked up in the order they are entered, but the important difference is that *functions are deterministic*. That is, there is no backtracking once a rule has fired, so the first rule to fire hides all those following it. Another way of seeing this is that functional evaluation does not allow argument guessing as would be non-deterministically possible by *narrowing* (*i.e.*, using unification instead of matching).

A $\psi$-term $U$ matches a $\psi$-term $V$ if $U \triangleleft V$. This means that $sort(U) \triangleleft sort(V)$ and that the $\psi$-terms associated to the labels of $U$, match the corresponding $\psi$-terms in $V$. There are two cases where $U$ and $V$ do not match: the first is if GLB($U$,$V$) = $\perp$, in which case matching fails and the next rule is used; the second is when $U$ is not included in $V$, in which case function evaluation is suspended: the function is *residuated*.

In simpler terms: residuation occurs when the arguments of a function are not sufficiently instantiated to determine the value of the result. If this occurs, the function value is given the temporary value $\top$, and as soon as the arguments are sufficiently instantiated the function is re-evaluated and the result is unified with the temporary result. In this way one can implement constraints.

Here is an example of residuation using the previously defined function `fact`. First, we impose the constraint $A = B!$.

```
> A=fact(B)?

*** Yes
A = @, B = @~.
```

[11] In LIFE, the sort name space, the predicate name space, and the function name space are mutually exclusive. For instance, it is not possible for the same symbol to be used simultaneously, within the scope of a same module, as a defined sort *and* a defined function. On the other hand, any (defined or undefined) symbol and any natural number may be used as an attribute.

The expression `fact(B:@)` residuated, yielding `@` as a temporary result. The tilde ~ after `@` means that B is a residuation variable, that is a variable that, if its sort is made more precise (more information is known), will cause the residuated function to be re-evaluated. We say that `B` is in the **RV-set** (set of residuation variables) of this constraint.

```
--1> B=real?

*** Yes
A = @, B = real~.
```

The function `fact` still residuates because *int* ◁ *real*.

```
----2> B=5?

*** Yes
```

5 ◁ *int* so `fact(B:5)` can be calculated.

```
A = 120, B = 5.
```

Let us now go back to the previous query level by typing ⟨`CR`⟩:

```
------3>

*** No
A = @, B = real~.
----2> A=123?

*** Yes
A = 123, B = real~.
------3> B=6?
```

We now have strengthened the constraint to $6! = 123$, and of course this constraint always fails.

```
*** No
A = 123, B = real~.
------3>
```

Functions are deterministic: there is no value guessing or backtracking involved. This means that a function, when invoked, may not alter its actual arguments to fit the definition. For example, if the definition `f(X,X) -> ...` is present, the call `f(foo,bar)` will skip it if `foo` and `bar` are non-unifiable, and it will residuate otherwise. In the latter case, the only way the definition may be eventually used is through explicit unification of both arguments.

To be precise, the scheme above concerns user-defined functions. This is also the case of built-in functions, but there are some important exceptions. For instance, all the binary arithmetic functions can infer one their arguments, if the result and the other argument is known, since this is compatible with our view that functions are deterministic. In fact, the built-in arithmetic functions perform all locally determined inversions; *e.g.*, the goal `0=B-C` causes `B` and `C` to be unified.

## 5.2 Currying

A functional expression is curryed if it has missing arguments. This is not the same as residuation, because the result of currying is a function, not $\top$. Currying in LIFE is not defined in quite the usual way because functions do not really have an arity. They have a list of required arguments, indexed by labels, not necessarily by integers.

The usual way of defining currying would be to say: $f(X,Y) = f(X)(Y)$. But the definition imposed by labels in LIFE is:
`f(a => X,b => Y) = f(a => X)&@(b => Y).`

Because labels are not ordered as are integers it is not possible to consume arguments by position, as in $\lambda$-calculus. Indeed, one can also write:
`f(a => X, b => Y) = f(b => Y)&@(a => X).`
This definition can be extended to any number of labels, basically the idea is that all the necessary argument positions have to be filled (order is irrelevant).[12]

Let us define a function that returns the list of its three arguments, and let us call it with only two arguments.

```
> f(X,Y,Z) -> [X,Y,Z].

*** Yes
> A=f(a,b)?

*** Yes
A = f(a,b).
```

The function `f` didn't find its third argument, so it returns in a curryed form. We can provide it the missing third argument and make its evaluation possible.

```
--1> A=f(3 => c)?

*** Yes
A = [a,b,c].
```

Several rules making up the definition of a function need not necessarily have their heads specify the same numbers or labels of arguments. Therefore, a functional expression is curryed as long as it misses arguments specified in the first eligible rule not yet dismissed, even if it already possesses all the arguments specified in a following rule.

```
> foo(a=>int) -> 1.

*** Yes
> foo(b=>int) -> 2.
```

[12] It should be noted that Wild LIFE's current handling of currying in the presence of explicit labels and implicit numeric positions is rather *ad hoc* and not quite satisfactory. This, like a few other things in LIFE, was done as a matter of temporary, if imperfect, solution to a yet unsolved issue. Only recently has a formal calculus been worked out that treats currying with label arguments correctly [3].

```
*** Yes
> X=foo(b=>0)?

*** Yes
X = foo(b => 0).
--1> X=foo(a => string)?

*** Yes
X = 2.
----2>
```

A rule defining a function may very well have a head without any arguments. This means necessarily that it will never residuate nor ever be curryed. It also means that it does not make any sense to define any more rules for that function following its first argumentless rule. This facility is in fact a nice way to define global constants or catch-all clauses.

Functional variables are allowed. That is, a functional expression may have a variable where a root symbol is expected.

```
map(F,[]) -> [].
map(F,[H|T]) -> [F(H)|map(F,T)].

> L=M(F,[1,2,3,4])?

*** Yes
F = @, L = @, M = @~.
--1> M=map?

*** Yes
F = @~~~~, L = [@,@,@,@], M = map.
----2> F= +(2=>1)?

*** Yes
F = +(2 => 1), L = [2,3,4,5], M = map.
------3>
```

This next example shows how residuation, currying, and functional variables can be combined together: a constraint is generated that binds the variable R to the result of applying F to A, where at that point both F and A are unknown, that is: $\top$. Later the argument A is chosen by the predicate `pick_arg` and the function F by the predicate `pick_function`. Note that `quadruple` is a function that returns a curryed function: `*(2 => 4)` that multiplies its first argument by 4.

So let us define, in addition to `fact` seen before, the following:

```
quadruple -> *(2=>4). % Here multiply is curryed

pick_arg({5;3;7}).

pick_func({quadruple;fact}).
```

```
test :- R=F(A),       % R results from applying a yet unknown
                      % function to a yet unknown argument.
        pick_arg(A),  % Here we pick an argument,
        pick_func(F), % and here a function.
        write("function ",F,
              " applied to argument ",A,
              " is ",R),
        nl,
        fail.         % This forces backtracking.
```

and let us try it:

```
> test?
function *(2 => 4) applied to argument 5 is 20
function fact applied to argument 5 is 120
function *(2 => 4) applied to argument 3 is 12
function fact applied to argument 3 is 6
function *(2 => 4) applied to argument 7 is 28
function fact applied to argument 7 is 5040

*** No
```

## 6 Sort definitions

We saw how to define a simple sort hierarchy. In practice, LIFE has more to offer: it is possible to attach properties (attributes or arbitrary constraints) to sorts. These properties will be verified during execution, and also inherited by subsorts.

### 6.1 Attributes

It is possible to attach attributes to sorts, using the following syntax:

```
::  sort(label => ψ-term, ..., label => ψ-term).
```

For example, to express that *“vehicles have a make that is a string, and a number of wheels that is an integer”* we may write:

```
:: vehicle(make => string, number_of_wheels => int).
```

And to say that *“cars are vehicles that have 4 wheels”*, we write:

```
car <| vehicle.
:: car(number_of_wheels => 4).
```

Obviously if the relation `car <| vehicle` is asserted then any properties attached to `car` must be compatible with those attached to `vehicle`, otherwise type `car` would be $\perp$.

It is also possible to use functions in attribute definitions, for example:

```
:: square(side => S:real,surface => S*S).
```

## 6.2 Constrained Sorts

One can also demand that certain constraints always hold about a sort. The syntax for this is:

```
:: sort(attributes) | constraint.
```

where `constraint` has the same form as that of a definite clause's body. The operator `|` is pronounced *"such that."* For example:

```
:: code(key => S:string) | pretty_complicated(S).
```

attaches a constraint to any object of sort `code` where `pretty_complicated` is a predicate expressing some property on strings. You might find it helpful to read this as *"all codes have a key that is a string, S, such that S is pretty complicated."*

Because the form of successive definitions:

```
sort_1 <| sort_2.
:: sort_1(attributes) | constraint.
```

is so frequent, LIFE allows a more convenient form with a little syntactic sugar. The alternative form is:

```
sort_1 := sort_2(attributes) | constraint.
```

which you can read as "`sort_1` *is a* `sort_2` *that has* `attributes` *such that* `constraint` *holds."*

This facility renders rather nicely the intuitive mathematical notation defining a set in terms of another one, of the form $A = \{x \in B \mid \ldots P(\ldots, x, \ldots) \ldots\}$.

In fact, the "such that" operator `|` may be used arbitrary terms in any expressions, even outside of sort definition. For example, the term conjunction operator & is defined as the following (infix) function:

```
X & Y -> X | X = Y.
```

## 6.3 Checking sort definitions

Sort definitions can specify arbitrarily constrained objects. This can cause a some overhead for LIFE, so checking definitions is done incrementally. Attribute inheritance can also be done lazily so that an attribute constrained in a sort definition is taken into consideration only if the corresponding attribute appears explicitly in a resolvent. A complete and consistent algorithm algorithm to do that exists and has been devised [12]. Such an algorithm is essentially a lazy unfolding of sort definitions and has as a specific advantage to prevent infinite loops in presence of recursive sort definitions like:

```
::  person(best_friend => P:person) | really_nice(P).
```

Unfortunately, this algorithm was not available at the time of Richard Meyer's original implementation of LIFE. Moreover, the data structure management and internal setting of the interpreter were not amenable to a simple adaptation of the interpreter to incorporate the lazy unfolding algorithm. Hence, what the current implementation uses is an incomplete (indeed, sometimes diverging) algorithm that consists in bringing in all the constrained attributes from a definition, whether or not they are explicitly used in a resolvent, and execute all constraints specified in the definition.

Therefore, in the current implementation, a recursive sort definition such as that of `person` above would systematically loop because of ever-expanding the definition of `person`. This is too bad (especially since we know how to handle it with the new algorithm alluded to before) and can currently be dealt with only thanks to an *ad hoc* means. Namely, in order to cope with such definitions, LIFE uses the following declaration to delay checking the attributes and constraints specified by the definition of a sort. For example, the foregoing `person` definition would also need the pragma declaration:

```
delay_check(person)?
```

This prevents LIFE from expanding its definition unless the sort in the resolvent has an attribute attached to it. This explains the following behavior:

```
> delay_check(person)?

*** Yes
> :: person(best_friend => P:person) | really_nice(P).

*** Yes
> cleopatra <| person.

*** Yes
> really_nice(cleopatra).

*** Yes
> A=person?

*** Yes
A = person.
```

Note that, at this point, the definition has not been checked because there were no attributes. Let us now continue and attach the attribute `nose => pretty`.

```
--1> A=@(nose => pretty)?

*** Yes
A = person(best_friend => cleopatra,nose => pretty).
```

We see that the definition was checked, the attributes where retrieved and the goal `really_nice(P)` proved. But `cleopatra` who is also a `person` has not been checked for the same reason.

Unfortunately, the `delay_check` trick is not sufficient to guarantee completeness and convergence in all cases. There are, however, easy guidelines for a general safe style of programming around these current limitations. Needless to say, all this hackery will have disappeared in the next release of LIFE for a clean, complete, and consistent handling of recursive sorts.

## 6.4 Constraints as daemons

Dynamic constraints that are attached to sort definitions are checked at run-time during unification. This is truly a novel programming feature offered by LIFE as they can be used very effectively as daemons—that is, code whose execution is triggered upon access to an object.

An interesting use of constraints as daemons is that one can use them to help debugging, for example by printing all $\psi$-terms of a given sort each time they have their constraints checked.

```
> :: I:int | write(I," ").
*** Warning: extending definition of sort 'int'.

*** Yes
> A=5*7?
5 7 35
*** Yes
A = 35.
--1> B=fact(5)?
5 1 4 1 3 1 2 1 1 1 0 1 1 2 6 24 120
*** Yes
A = 35, B = 120.
----2>
```

LIFE tries to be clever in dynamically checking constraints attached to sorts in that it remembers which constraints have already been solved. Hence, definitions are only retrieved the first time it becomes necessary, and not rechecked uselessly. In effect, one can bind arbitrary complicated proofs to a variable in LIFE.

## 7 Conclusion

We have overviewed the salient features of LIFE from a pragmatic standpoint. We have not covered all the programming conveniences offered by the systems that would not be relevant in a summary. Some of these omissions are standard facilities (*e.g.*, the module system, the graphical interface, the grammar preprocessor), while others are experimental, albeit quite useful, innovations (*e.g.*, global variables, persistent structures). The reader is referred to [2] for a more complete account.

LIFE is still under development, both conceptually and practically. Its main attraction is the convenience it offers with function and its more direct adequacy with recent formalisms for natural language processing [4] and database models [15]. It is bound to evolve in several respects so that some of the contents of this article may become out of date, or irrelevant, as some issues improve as ideas on which LIFE rests become better conceived and more efficiently implemented. Also, like Prolog, it may become a prisoner of its original aspect. However, its essential core will preserve a genetic stamp so that programming in LIFE will somehow always mean programming with Logic, Inheritance, Functions, and Equations.

## Acknowledgements

The author wishes to express his thanks and debt to the early LIFEers of his MCC days, Patrick Lincoln, Roger Nasr, and David Plummer, as well as the latter ones, all the members of the Paradise project at PRL, permanent and transient, for their contribution, big or small, to making LIFE a reality: Abder Aggoun, Christophe Bonnet, Bruno Dumant, Gerard Ellis, Jacques Garrigue, Seth Goldstein, Dinesh Katiyar, Richard Meyer, Aïssam Mezhoud, Kathleen Milsted, Andreas Podelski, Fadila Setbel, and Peter Van Roy.

## References

1. Hassan Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351 (1986).

2. Hassan Aït-Kaci, Bruno Dumant, Richard Meyer, and Peter Van Roy. Wild_LIFE, a user manual. PRL Research Report (forthcoming), Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1993).

3. Hassan Aït-Kaci and Jacques Garrigue. Label-selective $\lambda$-calculus. PRL Research Report 31, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison (May 1993).

4. Hassan Aït-Kaci and Patrick Lincoln. LIFE—A natural language for natural language. *T. A. Informations, revue internationale du traitement automatique du langage*, 30(1–2):37–67 (1989).

5. Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215 (1986).

6. Hassan Aït-Kaci and Roger Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89 (1989).

7. Hassan Aït-Kaci, Roger Nasr, and Patrick Lincoln. Le Fun: Logic, equations, and Functions. In *Proceedings of the Symposium on Logic Programming (San Francisco, CA)*, pages 17–23, Washington, DC (1987). IEEE, Computer Society Press.

8. Hassan Aït-Kaci and Andreas Podelski. Functions as passive constraints in LIFE. PRL Research Report 13, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (June 1991). (Revised, November 1992).

9. Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. In Jan Maluszyński and Martin Wirsing, editors, *Proceedings of the 3rd International*

*Symposium on Programming Language Implementation and Logic Programming (Passau, Germany)*, pages 255–274. Springer-Verlag, LNCS 528 (August 1991). (Full paper to appear in the *Journal of Logic Programming*).

10. Hassan Aït-Kaci and Andreas Podelski. Entailment and Disentailment of Order-Sorted Feature Constraints. In Andrei Voronkov, editor, *Proceedings of the Fourth International Conference on Logic Programming and Automated Reasoning (St. Petersburg, Russia)*. Springer-Verlag, LNCS (1993).

11. Hassan Aït-Kaci and Andreas Podelski. Logic Programming with Functions over Order-Sorted Feature Terms. In E. Lamma and P. Mello, editors, *Proceedings of the 3rd International Workshop on Extensions of Logic Programming (Bologna, Italy)*. Springer-Verlag, LNAI 660 (1992).

12. Hassan Aït-Kaci, Andreas Podelski, and Seth Copen Goldstein. Order-sorted feature theory unification. PRL Research Report 32, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (June 1993). (See also these proceedings.)

13. Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. In *Proceedings of the 5th International Conference on Fifth Generation Computer Systems*, pages 1012–1022, Tokyo, Japan (June 1992). ICOT. (Full paper to appear in *Theoretical Computer Science*).

14. William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, Germany, 2nd edition (1984).

15. Marcel Holsheimer, Rolf A. de By, and Hassan Aït-Kaci. A Database Interface for Complex Objects. PRL Research Report 31, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (March 1993).

16. Richard O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA (1990).

17. David Plummer. LIFE's rich tapestry. MCC Technical Report ACA-ST-409-88(P), Microelectronics and Computer Technology Corporation, Austin, Texas (1988).

18. Andreas Podelski and Peter Van Roy. The Beauty and the Beast Algorithm: Testing Entailment and Disentailment Incrementally. PRL Research Report (forthcoming), Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1993).

19. Leon Sterling and Ehud Shapiro. *The Art of Prolog*. Series in Logic Programming. MIT Press, Cambridge, MA (1986).

20. Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. In *IEEE Computer* 25 (1), pages 54-68, Jan. 1992.