A Graphical Toolkit in LIFE

Bruno Dumant* Hassan Aït-Kaci†

April 12, 1995

Abstract

LIFE is a programming language integrating different programming paradigms in the same framework, such as object orientation, concurrency and passive constraints. We have used this language to implement the prototype of a very basic and simple graphical toolkit for LIFE applications. This prototype is built on top of a basic interface between LIFE and the X Windows library It demonstrates that the combination of the different paradigms supported by LIFE is most adequate for designing such applications. Because the toolkit is all realized in LIFE, it could be easily ported to another window system (*e.g.*, MS-Windows) provided the basic calls to the underlying window system are given as interface.

1 Introduction

LIFE, an acronym for **L**ogic, **I**nheritance, **F**unctions, and **E**quations, is an experimental programming language designed after these four precepts for specifying structures and computations([3]. LIFE can also be seen as a specific instantiation of a Constraint Logic Programming (CLP) scheme with a particular constraint language; in its most primitive form, this constraint language constitutes a logic of record structures.

In this paper, we mean to overview informally the functionality of LIFE and the conveniences that it offers for programming, and describe a simple toolkit to build interactive window-based applications in LIFE. It provides the user with some basic functionalities of bigger toolkits, in short the ability to use buttons, text fields, menus, and sliders. Composite objects containing these primitives can be created arbitrarily at run-time. The prootype toolkit is built on top of a basic X interface that consists essentially in calls to the X library.

The toolkit is organized around three concepts: boxes, looks, and constructors.

- *boxes* are used to compute the sizes and positions of objects on the screen.

*INRIA (Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France). *Bruno.Dumant@inria.fr*

†Simon Fraser University (School of computing science, Burnaby, British Columbia V5A 1S6, Canada) *hak@cs.sfu.ca*

- *constructors* are used to build and initialize screen objects. All objects that have a behavior (i.e. not simple graphical objects, but real widgets) inherit from one constructor type.
- *looks* are used to describe the appearance of screen objects. An object may be a subtype of several look types and will inherit the appearance of these ''looks''.

In the next section, we give an overview of the LIFE language, then we shall focus on the toolkit and describe its implementation.

2 The LIFE language

LIFE expressions are of three kinds: functional, relational, and structural. The function-oriented component of LIFE is directly derived from functional programming languages with higher-order functions as first-class objects, data constructors, and algebraic pattern-matching for parameter-passing. The convenience offered by this style of programming is one in which expressions of any order are first-class objects and computation is determinate. The relation-oriented component of LIFE is essentially one inspired by the Prolog language [5, 6]. Unification of first-order patterns used as the argument-passing operation turns out to be the key of a quite unique and hitherto unusual *generative* behavior of programs, which can construct missing information as needed to accommodate success. Finally, the most original part of LIFE is the structure-oriented component which consists of a calculus of type structures---the ψ-calculus [1, 2]---and accounts for some of the (multiple) inheritance convenience typically found in so-called object-oriented languages.

The next subsection gives a very brief and informal account of the calculus of type inheritance used in LIFE (ψ-calculus). The reader is assumed familiar with logic programming.

2.1 ψ-Calculus

In this section, we give an informal but informative introduction of the notation, operations, and terminology of the data structures of LIFE. It is necessary to understand the programming examples to follow, and the description of the toolkit.

The ψ-calculus consists of a syntax of structured types called ψ-terms together with subtyping and type intersection operations. Intuitively, as expounded in [4], the ψ-calculus is a convenience for representing record-like data structures in logic and functional programming more adequately than first-order terms do, without loss of the well-appreciated instantiation ordering and unification operation.

Let us take an example to illustrate. Let us say that one has in mind to express syntactically a type structure for a *person* with the property, as expressed for the underlined symbol in Figure 1, that a certain functional diagram commutes.

The syntax of ψ-terms is one simply tailored to express as a term this kind of approximate

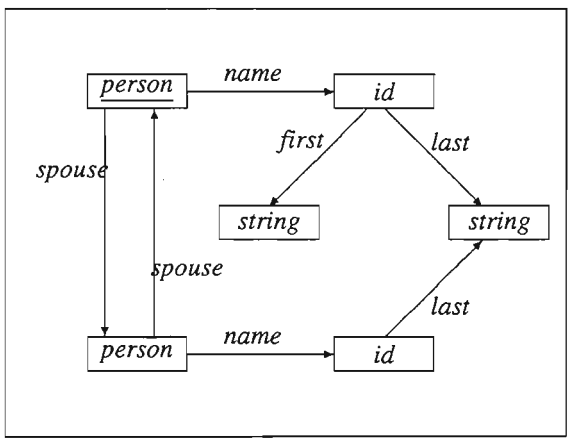


Figure 1: A commutative functional diagram

description. Thus, in the ψ-calculus, the information of Figure 1 is unambiguously encoded into a formula, perspicuously expressed as the ψ-term:

$$X : person\big(name \Rightarrow id\big(first \Rightarrow string,\\ last \Rightarrow S : string\big),\\ spouse \Rightarrow person\big(name \Rightarrow id\big(last \Rightarrow S\big),\\ spouse \Rightarrow X\big)\big).$$

It is important to distinguish among the three kinds of symbols participating in a ψ-term. We assume given a set $\mathcal{S}$ of sorts or *type constructor symbols*, a set $\mathcal{F}$ of *features, or attributes* symbols, and a set V of *variables* (or *coreference tags*). In the ψ-term above, for example, the symbols *person*, *id*, *string* are drawn from $\mathcal{S}$, the symbols *name*, *first*, *last*, *spouse* from $\mathcal{F}$, and the symbols X, S from V. (We capitalize variables, as in Prolog.)

A ψ-term is either *tagged* or *untagged*. A tagged ψ-term is either a variable in V or an expression of the form $X : t$ where $X \in V$ is called the term's *root variable* and t is an untagged ψ-term. An untagged ψ-term is either *atomic* or *attributed*. An atomic ψ-term is a sort symbol in $\mathcal{S}$. An attributed ψ-term is an expression of the form $s\big(\ell_1 \Rightarrow t_1, \ldots, \ell_n \Rightarrow t_n\big)$ where the root variable's sort symbol $s \in \mathcal{S}$ and is called the ψ-term's *principal* type, the ℓ_i's are mutually distinct attribute symbols in $\mathcal{F}$, and the t_i's are ψ-terms $\big(n \geq 0\big)$.

Variables capture coreference in a precise sense. They are coreference tags and may be viewed as typed variables where the type expressions are untagged ψ-terms. Hence, as a condition to be *well-formed*, a ψ-term must have all occurrences of each coreference tag consistently refer to the same structure. For example, the variable X in:

$$person\big(id \Rightarrow name\big(first \Rightarrow string,\\ last \Rightarrow X : string\big),\\ father \Rightarrow person\big(id \Rightarrow name\big(last \Rightarrow X : string\big)\big)\big)$$

refers consistently to the atomic ψ-term *string*. To simplify matters and avoid redundancy, we shall obey a simple convention of specifying the sort of a variable at most once and understand that other occurrences are equally referring to the same structure, as in:

$$person(id \Rightarrow name(first \Rightarrow string, \\ last \Rightarrow X : string), \\ father \Rightarrow person(id \Rightarrow name(last \Rightarrow X)))$$

In fact, since there may be circular references as in:

$$X : person(spouse \Rightarrow person(spouse \Rightarrow X))$$

this convention is necessary. Finally, a variable appearing nowhere typed, as in $junk(kind \Rightarrow X)$ is implicitly typed by a special greatest initial sort symbol $\top$ always present in $\mathcal{S}$. This symbol will be written as the symbol @ as in $@(age \Rightarrow integer, name \Rightarrow string)$. In the sequel, by ψ-term we shall always mean well-formed ψ-term.

Generalizing first-order terms,[1] ψ-terms are ordered up to variable renaming. Given that the set $\mathcal{S}$ is partially-ordered (with a greatest element $\top$), its partial ordering is extended to the set of attributed ψ-terms. Informally, a ψ-term t_1 is subsumed by a ψ-term t_2 if (1) the principal type of t_1 is a subtype in $\mathcal{S}$ of the principal type of t_2; (2) all attributes of t_2 are also attributes of t_1 with ψ-terms which subsume their homologues in t_1; and, (3) all coreference constraints binding in t_2 must also be binding in t_1.

For example, if $student < person$ and $paris < cityname$ in $\mathcal{S}$ then the ψ-term:

$$student(id \Rightarrow name(first \Rightarrow string, \\ last \Rightarrow X : string), \\ lives\_at \Rightarrow Y : address(city \Rightarrow paris), \\ father \Rightarrow person(id \Rightarrow name(last \Rightarrow X), \\ lives\_at \Rightarrow Y))$$

is subsumed by the ψ-term:

$$person(id \Rightarrow name(last \Rightarrow X : string), \\ lives\_at \Rightarrow address(city \Rightarrow cityname), \\ father \Rightarrow person(id \Rightarrow name(last \Rightarrow X))).$$

In fact, if the set $\mathcal{S}$ is such that *greatest lower bounds* (GLB's) exist for any pair of type symbols, then the subsumption ordering on ψ-term is also such that GLB's exist. Such are defined as the *unification* of two ψ-terms. A detailed unification algorithm for ψ-terms is given in [4].

Consider for example the poset displayed in Figure 2 and the two ψ-terms:

[1] In fact, if a first-order term is written $f(t_1, \ldots, t_n)$, it is nothing other than syntactic sugar for the ψ-term $f(1 \Rightarrow t_1, \ldots, n \Rightarrow t_n)$.

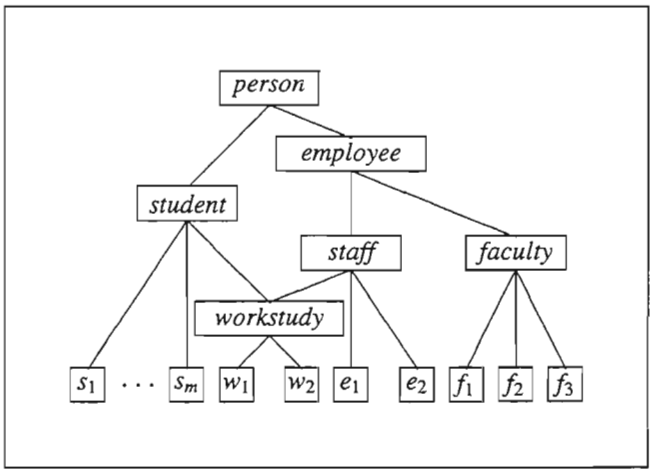


Figure 2: A lower semi-lattice of sorts

$$X : student\big(advisor \Rightarrow faculty\big(secretary \Rightarrow Y : staff,\ assistant \Rightarrow X\big),\ roommate \Rightarrow employee\big(representative \Rightarrow Y\big)\big)$$

and:

$$employee\big(advisor \Rightarrow f_1\big(secretary \Rightarrow employee,\ assistant \Rightarrow U : person\big),\ roommate \Rightarrow V : student\big(representative \Rightarrow V\big),\ helper \Rightarrow w_1\big(spouse \Rightarrow U\big)\big).$$

Their unification (up to tag renaming) yields the term:

$$W : workstudy\big(advisor \Rightarrow f_1\big(secretary \Rightarrow Z : workstudy\big(representative \Rightarrow Z\big),\ assistant \Rightarrow W\big),\ roommate \Rightarrow Z,\ helper \Rightarrow w_1\big(spouse \Rightarrow W\big)\big).$$

Last in this brief introduction to the ψ-calculus, we explain type definitions. The idea is that types in the signature may be specified to have attributes in addition to being partially-ordered. Inheritance of attributes from all supertypes to a subtype is done in accordance with ψ-term subsumption and unification. For example, given a simple signature for the specification of linear lists $\mathcal{S} = \{list, cons, []\}$ with $[] < list$ and $cons < list$, this is expressed as:

```
[] <| list.
cons <| list.
```

In addition, it is possible to specify that *cons* has an attribute $2 \Rightarrow list$. This is expressed as:

```
:: cons(2 => list).
```

In fact, it is possible to attach complex properties (attributes or arbitrary constraints) to sorts. These properties will be verified during execution, and also inherited by subsorts. A definition attaching attributes to a sort is expressed 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).
```

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."*

In fact, the "such that" operator | may be used not only in sort definitions but as with arbitrary terms in functions and predicate definitions. It is a function in that the expression `E|G` (pronounced "E *such that* G") evaluates the expression `E`, then proves the goal `G`, and finally returns the value `E` in the `G`'s solution context. For example, the ψ-term conjunction operator `&` is defined as the following (infix) function:

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

2.2 Functions and suspensions in LIFE

In LIFE, a basic ψ-term denotes a functional application if its root symbol is a defined function. An example of such is $append(list, L)$, where *append* is the function defined as:[2]

$$append([], L : list) \rightarrow L.$$
$$append([H|T : list], L : list) \rightarrow [H|append(T, L)].$$

The ψ-term $foo(bar \Rightarrow X : list, baz \Rightarrow Y : list, fuz \Rightarrow append(X, Y))$ is one in which the attribute *fuz* is derived as a function of the attributes *bar* and *baz*. Unifying such ψ-terms proceeds modulo suspension of functional expressions whose arguments are not sufficiently refined to be provably subsumed by patterns of function definitions. For instance, the equation *X = append(Y,Z)* will be suspended until *Y* is bound to a ψ-term subsumed by [] or [@|*list*], and *Z* by a ψ-term subsumed by *list*.

Arithmetic functions in particular are implemented as functions that will suspend until their arguments are bound to numbers. This gives the power of passive constraints to the language, which will be extensively used in the following to express placement constraints between objects.

Another important application of this suspension mechanism is the ability to mimic concurrent processes that may communicate thanks to the logical variables. Consider for instance the two functions:

```
process_1([1|NX],Y) -> process_1(NX,NY) | Y = [2|NY].
process_2([2|NY],X) -> process_2(NY,NX) | X = [1|NX].
```

with the query:

```
process_1(X,Y), process_2([2|Y],X) ?
```

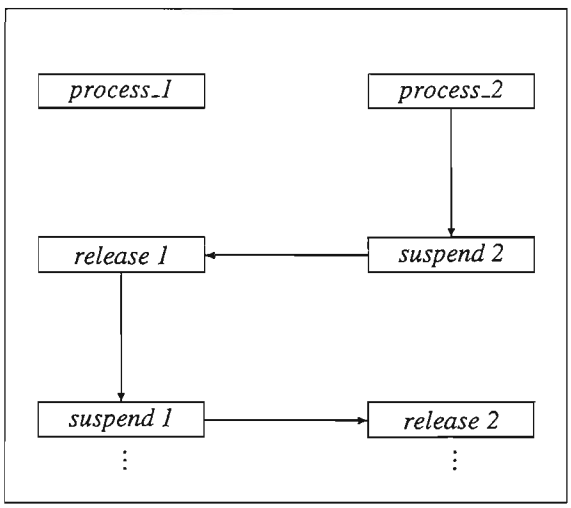
The execution first suspends `process_1` since its arguments are not sufficiently instantiated. Then it calls `process_2`, which can be executed. Its execution first suspends the recursive call to `process_2`, then binds `X` to `[1|NX]`. By doing this, the call to `process_1` will be released, and its execution will suspend a new call to `process_1`, and release the suspended call to `process_2`. Thus, we have two processes synchronized thanks to their arguments.

This mechanism is used in the toolkit to deal with events. An event handler is a recursive function that waits for events, is released when an event occur, and reinstalls itself after having performed the right actions.

3 A Graphical Toolkit in LIFE

As we have seen in the introduction, the toolkit we describe now is organised around three concepts: boxes, constructors, and looks. These three concepts are represented by sorts in the

[2]For convenience, LIFE adopts the list syntax of Prolog. The notation `[A|B]` is syntactic sugar for the term `cons(A,B)`, which is itself a shorthand for the ψ-term `cons(1=>A, 2=>B)`.



implementation, which we detail in the next subsections.

3.1 Boxes and placement constraints

3.1.1 Definition of boxes

The basic concept used in this toolkit is that of a *box*. A box resembles very much the box concept used by TeX. They have been introduced in the toolkit to let the developper of an application describe simply the position constraints between the objects of the interface. The passive constraints of LIFE are then used to compute the actual position parameters of the boxes satisfying these constraints.

All the objects manipulated by the toolkit are boxes. A box is defined by the following type declaration:

```
:: box(X,Y,width => DX,height => DY,border => B).
```

`X` and `Y` are integers giving the coordinates of the box, `DX` and `DY` are integers giving the dimensions of the box. The `border` feature is the width of reserved space on each side of a box.

When a box is declared, the user need not specify all these parameters. Some of them have (customizable) default values, others may be computed thanks to constraints defined over boxes. We describe in the following the basic constraints offered by the toolkit. These primitives just set and try to resolve the placement constraints.

3.1.2 Constraints on boxes

Relative positioning The toolkit offers a number of primitives to place boxes. They are functions taking two boxes as arguments and returning the smallest box containing these arguments placed in a certain way. For instance

```
Box1 l_above Box2
```

returns the smallest box containing `Box1` and `Box2`,such that:

- Box1 is above Box2, and
- their left sides are aligned.

Containment "`Box1 contains Box2`" expresses that `Box1` contains `Box2`. If no size is specified for Box1, it will be given the same size as that of Box2. If `Box1` has a `border` feature worth `Border`, it will be used to reserve a space of that width around the box. In this case, `Box1` will be larger than `Box2`.

Refined positioning There are also some primitives that set finer constraints. For instance:

```
Box1 lr_aligned Box2
```

will force the left side of `Box1` to be aligned with the right side of `Box2`.

```
Box1 cc_v_aligned Box2
```

will force the centers of `Box1` and `Box2` to be vertically aligned.

Sizes of boxes It is very useful to be able to constrain some boxes to have the same size. `same_size(List_of_Boxes)` will force all the boxes of the list to have the same height and width.

When boxes contain text, their sizes are computed on the fly, using a subsort of box: `t_box`. If no size is already given for a box, and if it is a subtype of `t_box`, then its size is computed according to the text, the font used, and the space to be reserved around it (which are declared as features of `t_box`).

These constraints may be cumulated and imposed in any order. The local constraint propagation of LIFE guarantees that if the constraints are consistent and enough information exists to determine a placing, it will be determined. If the constraints are inconsistent, then they will fail and cause backtracking.

3.2 Constructors: Describing the behavior of widgets

3.2.1 Constructor types

A constructor type defines the behavior of a family of widgets, and defines the methods to initialize them by creating the windows and setting up the event handlers.

Let us take the example of the constructor of text field widgets. It is defined as follows:

```
text_field_c <| box.
:: A:text_field_c( on => bool,
                   text => string,
                   action => Action,
                   constructor => build_text_field(A)).
```

The `on` feature determines the state of the button, `text` its text, `action` the action to be performed when a new text is entered, and `constructor` the method used to actually create a text field. Creating a text field consists in initializing the first features, creating a window, and setting up an event handler for this window.

```
build_text_field(Button) :-
    create_subwindow(Button),
    Button.on <<- false,
    initialize_action(Button),
    initialize_text(Button),
    catch_text_field_events(Button).
```

`Button.on` designates the `on` feature of `Button`. The `<<-` sign is a permanent assignment, which means that this value won't be modified on backtracking.

3.2.2 Handling events

Events are handled thanks to the ability of LIFE to express implicitly concurrent processes. Communication with X is handled by the `xGetEvent` function. This function takes two arguments, a window and a mask, and suspends until an event matching the mask occurs in the window.

```
catch_events(Button) -> true |
    Event = get_event(...),
    handle_event(Event).

handle_event(mouse_event) -> true |
    (actions)
    Event = get_event(...),
    handle_event(Event).
```

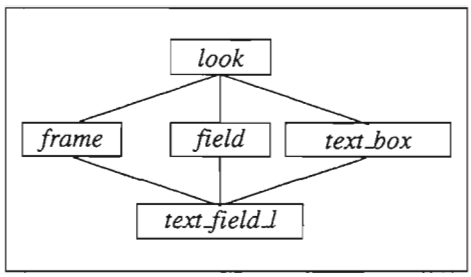


Figure 3: text field look

Events are implemented as sorts (`mouse_event`, `expose_event`, ...) An event handler is implemented as a recursive function that performs some actions when an event of the proper sort occurs, and reinstalls itself afterwards.

3.3 Looks

The look of a box is also handled through inheritance. For instance, the sort describing the appearance of a text field --- `text_field_l` --- is a subsort of `text_box`, `field` and `frame`, each of these sorts describing a special look feature. The idea is to have specific types to which some specific drawing methods are attached. As there is no such thing as overriding in LIFE, inheritance is not handled as usual.

Drawing methods are described thanks to two predicates: `draw_static` and `draw_dynamic`. `draw_static` is just used once, when the box is drawn for the first time. `draw_dynamic` is used each time the state of the widget changes, to redraw it according to its new state.

A special control mechanism is used to draw widgets. It is important here to make sure that all the drawing methods defined for a look will be used each time it is necessary. To do this, the drawing predicate calls `draw_static` and `draw_dynamic` with matching (and not unification) and backtracks until no method can anymore be used.

3.4 Screen objects

The screen objects manipulated by the X toolkit are subsorts of looks and/or constructors. They usually have an additional feature that stipulates how the states of the look and that of the constructor are linked (`change_state`).

All these mixed paradigms make it very easy to design interfaces. It is also very easy to add new objects to the toolkit. For instance, a user that wants to define a slider that contains text can do it easily this way:

```
my_slider <| h_slider_c.
my_slider <| frame.
my_slider <| field.
```

```
my_slider <| text_box.
```

These declarations just stipulate that `my_slider` has the behavior of an horizontal slider, and the look of a frame, a field, and a text box. Now, imagine the user wants to use it in an interface where the value of a text field is related to that of the slider, and the text "DANGER" must appear on the slider if the value is too big. This will be turned into the following code:

```
make_interface(Panel) :-
        Button = text_field_button(text => "100",
                                   action => set_slider(Button,Slider)),
        SlideBar : slide_bar(width => 200)
          contains
          Slider : my_slider(min => 0, max => 100,
                             text => "DANGER",
                             action => set_text(Slider,Button)),
        Panel : panel
          contains
          Button c_left_of SlideBar.

set_slider(Button,Slider) :-
        Value = parse(Button.text),
        cond(Value :< real,
             cond(Value >= Slider.min and Value =< Slider.max,
                  (move_slider(Slider,Value), check_danger(Slider)))).

set_text(Slider,Button) :-
        Button.text <<- int2str(floor(Slider.value)),
        refresh_look(Button).

check_danger(Slider) :-
        cond( Slider.value >= 80,
              Slider.text <<- "DANGER",
              Slider.text <<- ""),
        refresh_look(Slider).
```

`make_interface` defines the layout of the interface and its objects. The text in the text field (resp. the position of the slider) is modified by `set_text` (resp. `set_slider`) when the position of the slider (the text) is modified. Both call `check_danger` that changes the text of the slider according to its value.

4 Conclusion

We have shown in this paper how the different programming paradigms of LIFE may be used to simplify the design of the basic mechanisms of a graphical toolkit. Constraints are used to deal with the placement of boxes, inheritance to deal with the looks of widgets, suspension to easily design event handlers. The central data structure of LIFE, the ψ-term seems to be really

well-suited for the modelling of objects such as those needed in graphical interfaces. What has made this toolkit so easy to build is the simultaneous presence of these paradigms in the same language. To give an idea of the conciseness of the implementation, this toolkit is only 2200 lines long.

The main limitations of this prototype come from the speed limitations of the LIFE interpreter itself, and the number of available functionalities of the basic X libraries interfaced with LIFE. Nevertheless, it should be very easy to extend. This toolkit has been used with great ease to build the X interfaces of some of the example programs of the LIFE system. One can retrieve the system and use it from the World Wide Web URL `http://www.isg.sfu.ca/life/`.

References

[1] Hassan Aït-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Types*. PhD thesis, University of Pennsylvania, Philadelphia, PA, 1984.

[2] Hassan Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293--351, 1986.

[3] Hassan Aït-Kaci, Bruno Dumant, Richard Meyer, Andreas Podelski, and Peter Van Roy. The wild life handbook, a user manual. Technical report, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, 1994. (prepublication edition).

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

[5] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, Germany, 2nd edition, 1984.

[6] Richard O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA, 1990.