# An Overview of LIFE

Hassan Aït-Kaci*

Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92563 Rueil-Malmaison Cedex
France
*email: hak@prl.dec.com*

**Abstract**

LIFE (Logic, Inheritance, Functions, Equations) is an experimental 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 which 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.

> Quelle est la vie du mathématicien ? Quels sentiments exprime son langage ? [...] Calembours, jeux de mots, associations fortuites, la piste est chaude pour l'analyste. Là où le rapport logique, conscient, est flottant, le rapport inconscient peut être fécond.
>
> Pierre Berloquin
> *Un souvenir d'enfance d'Evariste Galois*

## 1 Introduction

LIFE is the product to date of research meant to explore whether programming styles and conveniences evolved as part of Functional, Logic, and Object-Oriented Programming could be somehow brought together to coexist in a single programming language. Being aware that not everything associated to these three approaches to programming is either well-defined or even uncontroversial, we have been very careful laying out some clear foundations on which to build LIFE. Thus, LIFE emerged as the synthesis of three computational atomic components which we refer to as *function-oriented*, *relation-oriented*,

*This reports work done while the author was part of the Languages Group of the ACA Systems Technology Laboratory of MCC, in Austin, Texas.

and *structure-oriented*, each being an operational rendition of a well-defined underlying model.

LIFE is a trinity. The function-oriented component of LIFE is directly derived from functional programming languages standing on foundations in the $\lambda$-calculus like ML [HMT88], or Miranda [Tur85,PJ87]. The convenience offered by this style of programming is essentially 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 [CM84,SS86,O'K90], taking its origin in theorem-proving as Horn clause calculus with a specific and well-defined control strategy—SLD-resolution. To a large extent, this way of programming gives the programmer the power of expressing program declaratively using a logic of implication rules which are then procedurally interpreted with a simple built-in pattern-oriented search strategy. Unification of first-order patterns used as the argument-passing operation turns out to be the key of a quite unique and hitherto unheard of *generative* behavior of programs, which could 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 $\psi$-calculus [AK84,AK86]—and rigorously accounts for some of the (multiple) inheritance convenience typically found in so called object-oriented languages. An algebra of term structures adequate for the representation and formalization of frame-like objects is given a clear notion of subsumption interpretable as a subtype ordering, together with an efficient unification operation interpretable as type intersection. Disjunctive structures are accommodated as well, providing a rich and clean pattern calculus for both functional and logic programming.

Under these considerations, a natural coming to LIFE has consisted thus in first studying pairwise combinations of each of these three operational tools. Metaphorically, this means realizing edges of a triangle (see Figure 1) whose vertices would be some essential renditions of, respectively, $\lambda$-calculus, Horn clause resolution, and $\psi$-calculus. After informally sketching one of these three atoms pertaining with type inheritance, we shall describe how we achieve the pairwise bonding of these atoms in the molecule of LIFE. Lastly, we shall synthesize the full molecule of LIFE from the three atomic vertices and the pairwise bonds. For a detailed account of the formal semantics of LIFE, the reader is referred to [AKP90,AKP91].

## 2 $\psi$-Calculus: Computing with Types

This section gives a very brief and informal account of the calculus of type inheritance used in LIFE ($\psi$-calculus). The reader is assumed familiar with functional programming ($\lambda$-calculus) and logic programming ($\pi$-calculus *a.k.a.* Horn clause SLD-resolution).

The $\psi$-calculus consists of a syntax of structured types called $\psi$-terms together with subtyping and type intersection operations. Intuitively, as expounded in [AKN86], the $\psi$-calculus is an attempt at obtaining a convenience for representing record-like data structures in logic and functional programming more adequate than first-order terms without loss of the well-appreciated instantiation ordering and unification operation.

The natural interpretation of a $\psi$-term is that of a data structure built out of constructors, features functions, and subject possibly to equational constraints which reflect feature coreference—sharing of structure. Thus, the syntactic operations on $\psi$-terms which stand analogous to instantiation and unification for first-order terms simply de-

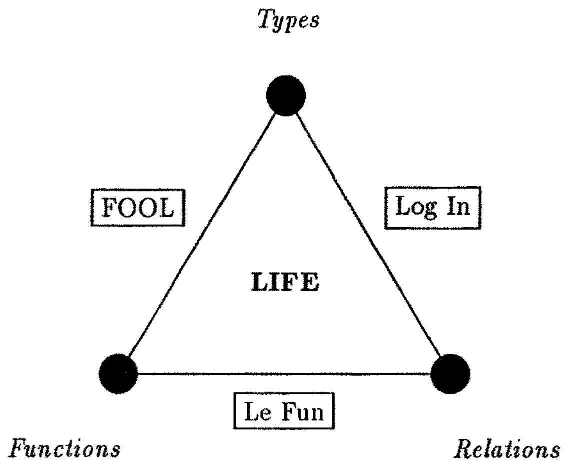


Figure 1: The LIFE molecule

note, respectively, sub-algebra ordering and algebra intersection, modulo type and equational constraints [AKP90]. This scheme even accommodates type constructors which are known to be partially-ordered with a given subtyping relation.[1] As a result, a rich calculus of structured subtypes is achieved formally without resorting to complex translation trickery. In essence, the $\psi$-calculus formalizes and operationalizes ***data structure inheritance***, all in a way which is quite faithful to a programmer's perception.

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 2, that a certain functional diagram commutes.

One way to specify this information algebraically would be to specify it as a *sorted equational theory* consisting of a *functional signature* giving the sorts of the functions involved, and an *equational presentation*. Namely,

$X$ : *person* $\underline{\textit{with}}$

$\underline{\textit{functions}}$

$$\begin{array}{lll} \textit{name} & : \textit{person} & \rightarrow \textit{id} \\ \textit{first} & : \textit{id} & \rightarrow \textit{string} \\ \textit{last} & : \textit{id} & \rightarrow \textit{string} \\ \textit{spouse} & : \textit{person} & \rightarrow \textit{person} \end{array}$$

$\underline{\textit{equations}}$

[1] We shall use "types" in reference to $\psi$-terms ans "sorts" in reference to the partially-ordered symbols. Thus, we may consistently refer to the latter as "types" or "sorts" interchangeably as a sort symbol is also an atomic $\psi$-term.

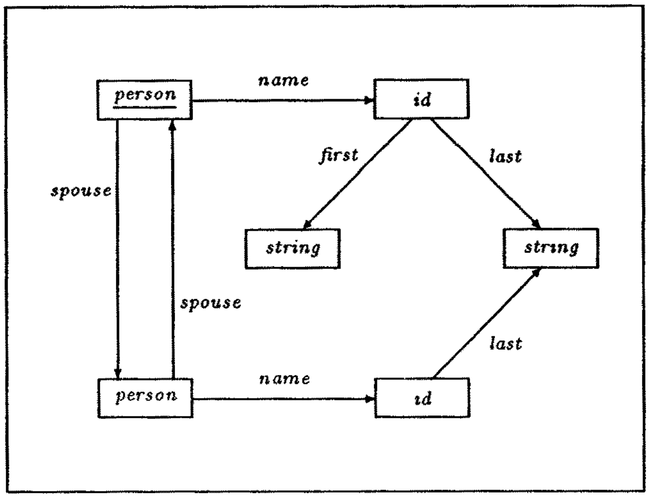


Figure 2: A functional diagram

$$
\begin{aligned}
&last(name(X)) = last(name(spouse(X)))\\
&spouse(spouse(X)) = X
\end{aligned}
$$

The syntax of $\psi$-terms is one simply tailored to express as a term this specific kind of sorted monadic algebraic equational presentations. Thus, in the $\psi$-calculus, this information of Figure 2 is unambiguously encoded into a formula, perspicuously expressed as the $\psi$-term:

$$
\begin{aligned}
X : person(&name \Rightarrow id(first \Rightarrow string,\\
&\qquad\qquad last \Rightarrow S : string),\\
&spouse \Rightarrow person(name \Rightarrow id(last \Rightarrow S),\\
&\qquad\qquad spouse \Rightarrow X)).
\end{aligned}
$$

We shall abstain in this summary from giving a complete formal definition of $\psi$-term syntax. (Such may be found elsewhere [AK86,AKN86].) Nevertheless, it is important to distinguish among the three kinds of symbols which participate in a $\psi$-term expression. Thus we assume given a set $\Sigma$ of *type constructor symbols*, a set $\mathcal{A}$ of *feature function symbols* (also called *attribute* symbols), and a set $\mathcal{R}$ of *reference tag symbols.* In the $\psi$-term above, for example, the symbols *person*, *id*, *string* are drawn from $\Sigma$, the symbols *name*, *first*, *last*, *spouse* from $\mathcal{A}$, and the symbols $X, S$ from $\mathcal{R}$.[2]

A $\psi$-term is either *tagged* or *untagged.* A tagged $\psi$-term is either a reference tag in $\mathcal{R}$ or an expression of the form $X : t$ where $X \in \mathcal{R}$ and $t$ is an untagged $\psi$-term. An untagged $\psi$-term is either *atomic* or *attributed.* An atomic $\psi$-term is a type symbol in $\Sigma$. An attributed $\psi$-term is an expression of the form $s(l_1 \Rightarrow t_1, \ldots, l_n \Rightarrow t_n)$ where $s \in \Sigma$ and the $\psi$-term principal type, the $l_i$'s are mutually distinct attribute symbols in $\mathcal{A}$, and the $t_i$'s are $\psi$-terms ($n \geq 1$).

[2]We shall use the lexical convention of using capitalized identifiers for reference tags.

Reference tags may be viewed as typed variables where the type expressions are untagged $\psi$-terms. Hence, as a condition to be well-formed, a $\psi$-term must have all occurrences of reference tags consistently refer to the same structure. For example, the reference tag $X$ in:

$$\begin{array}{l} \textit{person}(\textit{id} \Rightarrow \textit{name}(\textit{first} \Rightarrow \textit{string}, \\ \qquad\qquad\qquad\qquad \textit{last} \Rightarrow X : \textit{string}), \\ \qquad \textit{father} \Rightarrow \textit{person}(\textit{id} \Rightarrow \textit{name}(\textit{last} \Rightarrow X : \textit{string}))) \end{array}$$

refers consistently to the atomic $\psi$-term *string*. To simplify matters and avoid redundancy, we shall obey a simple convention of specifying the type of a reference tag at most once as in:

$$\begin{array}{l} \textit{person}(\textit{id} \Rightarrow \textit{name}(\textit{first} \Rightarrow \textit{string}, \\ \qquad\qquad\qquad\qquad \textit{last} \Rightarrow X : \textit{string}), \\ \qquad \textit{father} \Rightarrow \textit{person}(\textit{id} \Rightarrow \textit{name}(\textit{last} \Rightarrow X))) \end{array}$$

and understand that other occurrences are equally referring to the same structure. In fact, this convention is necessary if we have circular references as in:

$$X : \textit{person}(\textit{spouse} \Rightarrow \textit{person}(\textit{spouse} \Rightarrow X)).$$

Finally, a reference tag appearing nowhere typed, as in $\textit{junk}(\textit{kind} \Rightarrow X)$ is implicitly typed by a special universal type symbol $\top$ always present in $\Sigma$. This symbol will be left invisible (*i.e.*, not written explicitly as in $(\textit{age} \Rightarrow \textit{integer}, \textit{name} \Rightarrow \textit{string})$) or written as '@' (anything) as in $@(\textit{age} \Rightarrow \textit{integer}, \textit{name} \Rightarrow \textit{string})$. In the sequel, by $\psi$-term we shall always mean *well-formed* $\psi$-term.

Similarly to first-order terms, a subsumption preorder can be defined on $\psi$-terms which is an ordering up to reference tag renaming. Given that the set of sorts $\Sigma$ is partially-ordered (with a greatest element $\top$), its partial ordering is extended to the set of attributed $\psi$-terms. Informally, a $\psi$-term $t_1$ is subsumed by a $\psi$-term $t_2$ if (1) the principal type of $t_1$ is a subtype in $\Sigma$ of the principal type of $t_2$; (2) all attributes of $t_2$ are also attributes of $t_1$ with $\psi$-terms which subsume their homologues in $t_1$; and, (2) all coreference constraints binding in $t_2$ must also be binding in $t_1$.

For example, if $\textit{student} < \textit{person}$ and $\textit{paris} < \textit{cityname}$ in $\Sigma$ then the $\psi$-term:

$$\begin{array}{l} \textit{student}(\textit{id} \Rightarrow \textit{name}(\textit{first} \Rightarrow \textit{string}, \\ \qquad\qquad\qquad\qquad \textit{last} \Rightarrow X : \textit{string}), \\ \qquad \textit{lives\_at} \Rightarrow Y : \textit{address}(\textit{city} \Rightarrow \textit{paris}), \\ \qquad \textit{father} \Rightarrow \textit{person}(\textit{id} \Rightarrow \textit{name}(\textit{last} \Rightarrow X), \\ \qquad\qquad\qquad\qquad \textit{lives\_at} \Rightarrow Y)) \end{array}$$

is subsumed by the $\psi$-term:

$$\begin{array}{l} \textit{person}(\textit{id} \Rightarrow \textit{name}(\textit{last} \Rightarrow X : \textit{string}), \\ \qquad \textit{lives\_at} \Rightarrow \textit{address}(\textit{city} \Rightarrow \textit{cityname}), \\ \qquad \textit{father} \Rightarrow \textit{person}(\textit{id} \Rightarrow \textit{name}(\textit{last} \Rightarrow X))). \end{array}$$

In fact, if the set of sorts $\Sigma$ is such that *greatest lower bounds* (GLB's) exist for any pair of type symbols, then the subsumption ordering on $\psi$-term is also such that GLB's exist. Such are defined as the *unification* of two $\psi$-terms. A detailed unification algorithm for $\psi$-terms is given in [AKN86]. Consider for example the set of sorts displayed in Figure 3 and the two $\psi$-terms:

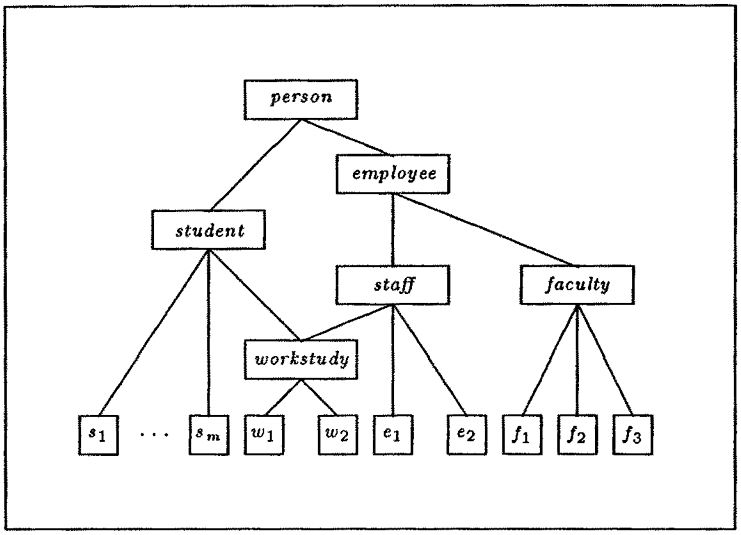


Figure 3: A partially-ordered set of sorts

$$
\begin{aligned}
X : \mathit{student}(\mathit{advisor} \Rightarrow \mathit{faculty}(&\mathit{secretary} \Rightarrow Y : \mathit{staff},\\
&\mathit{assistant} \Rightarrow X),\\
\mathit{roommate} \Rightarrow \mathit{employee}(&\mathit{representative} \Rightarrow Y))
\end{aligned}
$$

and:

$$
\begin{aligned}
\mathit{employee}(\mathit{advisor} \Rightarrow f_1(&\mathit{secretary} \Rightarrow \mathit{employee},\\
&\mathit{assistant} \Rightarrow U : \mathit{person}),\\
\mathit{roommate} &\Rightarrow V : \mathit{student}(\mathit{representative} \Rightarrow V),\\
\mathit{helper} &\Rightarrow w_1(\mathit{spouse} \Rightarrow U)).
\end{aligned}
$$

Their unification (up to tag renaming) yields the term:[3]

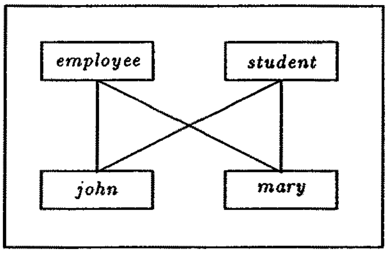
$$
\begin{aligned}
W : \mathit{workstudy}(\mathit{advisor} \Rightarrow f_1(&\mathit{secretary} \Rightarrow Z : \mathit{workstudy}(\mathit{representative} \Rightarrow Z),\\
&\mathit{assistant} \Rightarrow W),\\
\mathit{roommate} &\Rightarrow Z,\\
\mathit{helper} &\Rightarrow w_1(\mathit{spouse} \Rightarrow W)).
\end{aligned}
$$

A technicality arises if $\Sigma$ is not a lower semi-lattice. For example, given the (non-lattice) set of sorts:

[3]Incidentally, if *least upper bounds* (LUBs) are defined as well in $\Sigma$, so are they for $\psi$-terms. For example for these two $\psi$-terms, their LUB (most specific generalization) is

$$
\begin{aligned}
\mathit{person}(\mathit{advisor} \Rightarrow \mathit{faculty}(&\mathit{secretary} \Rightarrow \mathit{employee},\\
&\mathit{assistant} \Rightarrow \mathit{person}),\\
\mathit{roommate} \Rightarrow \mathit{person})).
\end{aligned}
$$

Thus, a lattice structure can be extended from $\Sigma$ to $\psi$-terms [AK84,AK86]. Although it may turn out useful in other contexts, we shall ignore this generalization operation here.



the GLB of ***student*** and ***employee*** is not uniquely defined, in that it could be ***john or mary***. That is, the set of their common lower bounds does not admit *one* greatest element. However, the set of their *maximal* common lower bounds offers the most general choice of candidates. Clearly, the *disjunctive* type $\{john; mary\}$ is an adequate interpretation.[4] Thus the $\psi$-term syntax may be enriched with disjunction denoting type union.

For a more complete formal treatment of disjunctive $\psi$-terms, the reader is referred to [AK86] and to [AKN86]. It will suffice to indicate here that a *disjunctive* $\psi$-term is a set of incomparable $\psi$-terms, written $\{t_1; \ldots; t_n\}$ where the $t_i$'s are basic $\psi$-terms. A *basic* $\psi$-term is one which is non-disjunctive. The subsumption ordering is extended to disjunctive (sets of) $\psi$-terms such that $D_1 \leq D_2$ *iff* $\forall t_1 \in D_1, \exists t_2 \in D_2$ such that $t_1 \leq t_2$. This justifies the convention that a singleton $\{t\}$ is the same as $t$, and that the empty set is identified with $\perp$. Unification of two disjunctive $\psi$-terms consists in the enumeration of the set of all maximal $\psi$-terms obtained from unification of all elements of one with all elements of the other. For example, limiting ourselves to disjunctions of atomic $\psi$-terms in the context of set of sorts in Figure 3, the unification of $\{employee; student\}$ with $\{faculty; staff\}$ is $\{faculty; staff\}$. It is the set of maximal elements of the set $\{faculty; staff; \perp; workstudy\}$ of pairwise GLB's.

In practice, it is convenient to allow nesting disjunctions in the structure of $\psi$-terms. For instance, to denote a type of person whose friend may be an astronaut with same first name, or a businessman with same last name, or a charlatan with first and last names inverted, we may write such expressions as:

$$
\begin{array}{l}
person(id \Rightarrow name(first \Rightarrow X : string, \\
\qquad\qquad\qquad\qquad last \Rightarrow Y : string), \\
\qquad friend \Rightarrow \{astronaut(id \Rightarrow name(first \Rightarrow X)) \\
\qquad\qquad\quad ; businessman(id \Rightarrow name(last \Rightarrow Y)) \\
\qquad\qquad\quad ; charlatan(id \Rightarrow name(first \Rightarrow Y, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad last \Rightarrow X))\})
\end{array}
$$

Tagging may even be chained or circular within disjunctions as in:

$$
\begin{array}{l}
P : \{charlatan \\
\quad ; person(id \Rightarrow name(first \Rightarrow X : \text{'john'}, \\
\qquad\qquad\qquad\qquad\qquad last \Rightarrow Y : \{\text{'doe'}; X\}), \\
\qquad\qquad friend \Rightarrow \{P; person(id \Rightarrow name(first \Rightarrow Y, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad last \Rightarrow X))\})\}
\end{array}
$$

which expresses the type of either a charlatan, or a person named either "John Doe" or "John John" and whose friend may be either a charlatan, or himself, or a person with his first and last names inverted. These are no longer graphs but hypergraphs.

[4]See [AKBLN89] for a description of an efficient method for computing such GLB's.

Of course, one can always expand out all nested disjunctions in such an expression, reducing it to a canonical form consisting of a set of non-disjunctive $\psi$-terms. The process is described in [AK84], and is akin to converting a non-deterministic finite-state automaton to its deterministic form, or a first-order logic formula to its disjunctive normal form. However, more for pragmatic efficiency than just notational convenience, it is both desirable to keep $\psi$-terms in their non-canonical form. It is feasible then to build a lazy expansion into the unification process, saving expansions in case of failure or unification against $\top$. Such an algorithm is more complicated and will not be detailed here for lack of space.

Last in this brief introduction to the $\psi$-calculus, we explain type definitions. The concept is analogous to what a global store of constant definitions is in a practical functional programming language based on the $\lambda$-calculus. The idea is that sorts may be specified to have attributes in addition to being partially-ordered. Inheritance of attributes of all supertypes to a type is done in accordance to $\psi$-term subsumption and unification. Unification in the context of such an inheritance hierarchy amounts to solving equations in an order-sorted algebra as explained in [SAK89], to which the reader is referred for a full formal account.

For example, given a simple signature for the specification of linear lists $\Sigma = \{list, cons, nil\}$[5] with $nil < list$ and $cons < list$, it is yet possible to specify that *cons* has an attribute $tail \Rightarrow list$. We shall specify this as:

$$list := \{nil; cons(tail \Rightarrow list)\}.$$

¿From which the partial-ordering above is inferred.

As in this *list* example, such type definitions may be recursive. Then, $\psi$-unification *modulo* such a type specification proceeds by unfolding type symbols according to their definitions. This is done by need as no expansion of symbols need be done in case of (1) failures due to order-theoretic clashes (*e.g.*, $cons(tail \Rightarrow list)$ unified with *nil* fails; *i.e.*, gives $\bot$); (2) symbol subsumption (*e.g.*, *cons* unified with *list* gives just *cons*), and (3) absence of attribute (*e.g.*, $cons(tail \Rightarrow cons)$ unified with *cons* gives $cons(tail \Rightarrow cons)$). Thus, attribute inheritance is done "lazily," saving much unnecessary expansions.

## 3 The Pairwise Bonds

In this section we indicate briefly how to achieve pairwise combination calculi from $\psi$, $\pi$, and $\lambda$, edges of the triangle of LIFE in Figure 1—the bonds between the atoms of the LIFE molecule. We shall keep an informal style, illustrating key points with examples.

### 3.1 $\pi\psi$-Calculus: Log In

Log In is simply Prolog where first-order constructor terms have been replaced by $\psi$-terms, with type definitions [AKN86]. Its operational semantics is the immediate adaptation of that of Prolog's SLD resolution. Thus, we may write a predicate for list concatenation as:[6]

[5] We shall always leave $\top$ and $\bot$ implicit.

[6] First-order terms being just a particular case of $\psi$-terms, an expression as $f(t_1, \ldots, t_n)$ is implicit syntax for $f(1 \Rightarrow t_1, \ldots, n \Rightarrow t_n)$. More flexibly, LIFE allows freely mixing position and keyword arguments. For instance, $f(a \Rightarrow X, g(X, b \Rightarrow c, Y), Z)$ is the same thing as

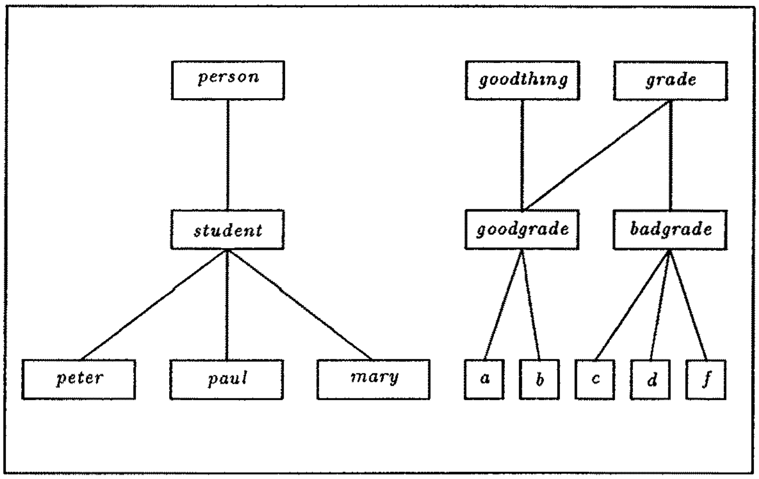


Figure 4: The "Peter-Paul-Mary" sort hierarchy

$list := \{[]; [@|list]\}.$
$append([], L : list, L).$
$append([H|T : list], L : list, [H|R : list]) \text{ :- } append(T, L, R).$

This definition, incidentally, is fully correct as opposed to Prolog's typeless version for which the query $append([], t, t)$ succeeds incorrectly for any non-list term $t$.

Naturally, advantage of the type partial-ordering can be taken as illustrated in the following simple example. We want to express the facts that a student is a person; Peter, Paul, and Mary are students; good grades and bad grades are grades; a good grade is also a good thing; 'A' and 'B' are good grades; and 'C', 'D', 'F' are bad grades. This information is depicted as the set of sorts of Figure 4. This taxonomic information is expressed in Log In as:

$student \lhd person.$
$student := \{peter; paul; mary\}.$
$grade := \{goodgrade; badgrade\}.$
$goodgrade \lhd goodthing.$
$goodgrade := \{a; b\}.$
$badgrade := \{c; d; f\}.$

In this context, we define the following facts and rules. It is known that all persons like themselves. Also, Peter likes Mary; and, all persons like all good things. As for grades, Peter got a 'C'; Paul got an 'F', and Mary an 'A'. Lastly, it is known that a person is happy if she got something which she likes. Alternatively, a person is happy if he likes something which got a good thing. Thus, in Log In,

$f(a \Rightarrow X, 1 \Rightarrow g(1 \Rightarrow X, b \Rightarrow c, 2 \Rightarrow Y), 2 \Rightarrow Z)$. Thus, Prolog's notation is fully subsumed. In particular, we adopt its notation for lists. Finally, recall that as well as using Prolog's notation for anonymous variables ("_"), LIFE uses the symbol @ for "don't-care" *a.k.a.* $\top$.

$likes(X : person, X).$
$likes(peter, mary).$
$likes(person, goodthing).$

$got(peter, c).$
$got(paul, f).$
$got(mary, a).$

$happy(X : person) :- likes(X, Y), got(X, Y).$
$happy(X : person) :- likes(X, Y), got(Y, goodthing).$

¿From this, it follows that Mary is happy because she likes good things, and she got an 'A'—which is a good thing. She is also happy because she likes herself, and she got a good thing. Peter is happy because he likes Mary, who got a good thing. Thus, a query asking for some "happy" object in the database will yield:

?- $happy(X).$

$X = mary;$

$X = mary;$

$X = peter;$

*No*

## 3.2 $\psi\lambda$-Calculus: FOOL

The basic paraphernalia the $\lambda$-calculus are not quite enough for even bare needs in symbolic computing as no provision is made for structuring data. The most primitive such facility is pairing (written as infix right-associative '.'). The pair constructor comes with two projection functions *fst* and *snd* such that the following equations hold:

$$\begin{aligned} fst(x.y) &= x \\ snd(x.y) &= y \\ fst(z).snd(z) &= z \end{aligned}$$

This allows the construction of binary tree structures and thus sufficient for representing any symbolic structure such as trees of any arity, as well-known to Lisp programmers. For these constructed pairs, a test of equality is implicitly defined as physical equality (*i.e.*, same address) as opposed to structure isomorphism. Thus, linear list structures may be built out of pairing and a nullary list terminator (written as [], as in 1.2.3.4.[]).

As an example, a function for concatenating two lists can be defined as:

$append(l1, l2) \Rightarrow if\ x = []\ then\ l2\ else\ fst(l1).append(snd(l1), l2).$

In fact, a pattern-directed syntax is preferable as it is expresses more perspicuous definitions of functions on list structures. Thus, the above list concatenation has the following pattern-directed definition:

$append([], l) \Rightarrow l.$
$append(h.t, l) \Rightarrow h.append(t, l).$

Again, this can be viewed as syntactic adornment as the previous form may be recovered in a single conditional expression covering each pattern case by explicitly introducing identifier arguments to which projection functions are applied to retrieve appropriate pattern occurrences. But again, this is for simplicity rather than efficiency. An efficient implementation will avoid the conditional by using the argument pattern as index key as well as using pattern-matching to bind the structure variables to their homologues in the actual argument patterns [PJ87].

Clearly, when it comes to programming convenience, linear lists as a universal symbolic construction facility can become quickly tedious and cumbersome. More flexible data structures such as first-order constructor terms can be used with the convenience and efficiency of pattern-directed definitions. Indeed, for each $n$-ary constructor symbol $c$, we associate $n$ projections $1_c, \ldots, n_c$ such that the following equations hold ($1 \leq i \leq n$):

$$\begin{aligned} i_c(c(x_1, \ldots, x_n)) &= x_i \\ c(1_c(z), \ldots, n_c(z)) &= z \end{aligned}$$

Pretty much as a linear list data structure could then be define as either [] or a pair $.(x, y)$ whose second projection $y$ is a linear list, one can then define any data structure as a disjoint sum of data constructors using recursive type equations as a definition facility. Then, a definition of a function on such data structures consists of an ordered sequence of pattern-directed equations such as *append* above which are invoked for application using term pattern-matching as argument binding.

A simple operational semantics of pattern-directed rewriting can thus be given. Given a program consisting as a set of function definitions. A function definition is a sequence of pattern-directed equations of the form:

$$\begin{aligned} &f(\vec{A}_1) = B_1. \\ &\quad\vdots \\ &f(\vec{A}_n) = B_n. \end{aligned}$$

which define a function $f$ over patterns $\vec{A}_i$, tuples of first-order constructor terms. Evaluating an expression $f(\vec{E})$ consists in (1) evaluating all arguments (components of $\vec{E}$); then, (2) finding the first successful matching substitution $\sigma$ in the order of the definitions; *i.e.*, the first $i$ in the definition of $f$ such that there is a substitution of the variables in the pattern $\vec{A}_i$ such that $f(\vec{E}) = f(\vec{A}_i)\sigma$ (if none exists, the expression is not defined); finally, (3) in evaluating in turn the expression $B_i\sigma$, which constitutes the result.

FOOL is simply a pattern-oriented functional language where first-order constructor terms have been replaced by $\psi$-terms, with type definitions. Its operational semantics is the immediate adaptation of that described above. Thus, we may write a function for list concatenation as:

$$\begin{aligned} &\mathit{list} := \{[\,]; [@|\mathit{list}]\}. \\ &\mathit{append}([\,], L : \mathit{list}) \Rightarrow L. \\ &\mathit{append}([H|T : \mathit{list}], L : \mathit{list}) \Rightarrow [H|\mathit{append}(T, L)]. \end{aligned}$$

Higher-order definition and currying are also naturally allowed in FOOL; *e.g.*,

$$\begin{aligned} &\mathit{map}([\,], @) \Rightarrow [\,]. \\ &\mathit{map}([H|T], F) \Rightarrow [F(H)|\mathit{map}(T, F)]. \end{aligned}$$

Thus, the expression $map([1,2,3],+1)$ evaluates to $[2,3,4]$.

The $\psi$-term subsumption ordering replaces the first-order matching ordering on constructor terms. In particular, disjunctive patterns may be used. The arbitrary richness of a user-defined partial-ordering on types allows highly generic functions to be written, thus capturing the flavor of code encapsulation offered by so called object-oriented languages. For example, referring back to Figure 3 on Page 6, the function:

$$age(person(dob \Rightarrow date(year \Rightarrow X)), ThisYear : integer) \Rightarrow ThisYear - X.$$

will apply generically to all subtypes and instances of persons with a birth year.

## 3.3 $\lambda\pi$-Calculus: Le Fun

Le Fun [AKLN87,AKN89] is a relational *and* functional programming language where first-order terms are generalized by the inclusion of *applicative expressions* as defined by Landin [Lan63] (atoms, abstractions, and applications) augmented with first-order constructor terms. Thus, *interpreted* functional expressions may participate as *bona fide* arguments in logical expressions just as conventional *constructor* terms do in Prolog.

Thus unification must consider unificands for which success or failure cannot be decided in a local context (*e.g.*, function applications may not be ready for reduction while expression components are still uninstantiated.) We propose to handle such situations by delaying unification until further variable instantiations make it possible to reduce unificands containing applicative expressions. In essence, such a unification may be seen as a residual equation which will have to be verified, as opposed to solved, in order to confirm eventual success—whence the name *residuation.* If verified, a residuation is simply discarded; if failing, it triggers chronological backtracking at the latest instantiation point which allowed its evaluation. This is very reminiscent of the process of asynchronous backpatching used in one-pass compilers to resolve forward references.

We shall merely illustrate Le Fun's operational semantics by giving very simple canonical examples.

A goal literal involving arithmetic variables may not be proven by Prolog, even if those variables were to be provided by proving a subsequent goal. This is why arithmetic expressions cannot be nested in literals other than the *is* predicate, a special one whose operation will force evaluation of such expressions, and whose success depends on its having no uninstantiated variables in its second argument. Consider the set of Horn clauses:

$$q(X,Y,Z) \text{ :- } p(X,Y,Z,Z), pick(X,Y).$$
$$p(X,Y,X+Y,X*Y).$$
$$p(X,Y,X+Y,(X*Y)-14).$$
$$pick(3,5).$$
$$pick(2,2).$$
$$pick(4,6).$$

and the following query:

$$?\text{- } q(A,B,C).$$

¿From the resolvent $q(A,B,C)$, one step of resolution yields as next goal to establish $p(A,B,C,C)$. Now, trying to prove the goal using the first of the two $p$ assertions is contingent on solving the equation $A+B=A*B$. At this point, Prolog would fail, regardless of the fact that the next goal in the resolvent, $pick(A,B)$ may provide instantiations for its variables which may verify that equation. Le Fun stays open-minded and proceeds with the computation as in the case of success, remembering however that eventual success of proving this resolvent must insist that the equation be verified. As it turns out in this case, the first choice for $pick(A,B)$ does not verify it, since $3+5 \neq 3*5$. However, the next choice instantiates both $A$ and $B$ to 2, and thus verifies the equation, confirming that success is at hand.

To emphasize the fact that such an equation as $A+B=A*B$ is a left-over granule of computation, we call it a *residual equation* or *equational residuation—E-residuation, for short.* We also coin the verb *"to residuate"* to describe the action of leaving some computation for later. We shall soon see that there are other kinds of residuations. Those variables whose instantiation is awaited by some residuations are called *residuation variables* (RV). Thus, an E-residuation may be seen as an *equational closure*—by analogy to a lexical closure—consisting of two functional expressions and a list of RV's.

There is a special type of E-residuation which arises from equations involving an uninstantiated variable on one hand, and a not yet reducible functional expression on the other hand (*e.g.*, $X=Y+1$). Clearly, these will never cause failure of a proof, since they are equations in solved form. Nevertheless, they may be reduced further pending instantiations of their RV's. Hence, these are called *solved residuations* or *S-residuations.* Unless explicitly specified otherwise, "E-residuation" will mean "equational residuations which are not S-residuations."

Going back to our example, if one were interested in further solutions to the original query, one could force backtracking at this point and thus, computation would go back eventually before the point of residuation. The alternative proof of the goal $p(A,B,C,C)$ would then create another residuation; namely, $A+B=(A*B)-14$. Again, one can check that this equation will be eventually verified by $A=4$ and $B=6$.

Since instantiations of variables may be non-ground, *i.e.*, may contain variables, residuations mutate. To see this, consider the following example:

$q(Z)$ :- $p(X,Y,Z), X=V-W, Y=V+W, pick(V,W)$.
$p(A,B,A*B)$.
$pick(9,3)$.

together with the query:

?- $q(Ans)$.

The goal literal $p(X,Y,Ans)$ creates the S-residuation $Ans = X*Y$. This S-residuation has RV's $X$ and $Y$. Next, the literal $X=V-W$ instantiates $X$ and creates a new S-residuation. But, since $X$ is an RV to some residuation, rather than proceeding as is, it makes better sense to substitute $X$ into that residuation and eliminate the new S-residuation. This leaves us with the *mutated* residuation $Ans=(V-W)*Y$. This mutation process has thus altered the RV set of the first residuation from $\{X,Y\}$ to $\{V,W,Y\}$. As computation proceeds, another S-residuation instantiates $Y$, another RV, and thus triggers another mutation of the original residuation into $Ans=(V-W)*(V+W)$, leaving it with the new RV set $\{V, W\}$. Finally, as $pick(9,3)$ instantiates $V$ to 9 and $W$ to 3, the residuation is left with an empty RV set, triggering evaluation, and releasing the residuation, and yielding final solution $Ans = 72$.

The last example illustrates how higher-order functional expressions and automatic currying are handled implicitly. Consider,

$sq(X) \Rightarrow X * X.$
$twice(F, X) \Rightarrow F(F(X)).$
$valid\_op(twice).$
$p(1).$
$pick(lambda(X, X)).$
$q(V) :- G = F(X), V = G(2 \Rightarrow 1), valid\_op(F), pick(X), p(sq(V)).$

with the query,

$?- \; q(Ans).$

The first goal literal $G = F(X)$ creates an S-residuation with the RV set $\{F, X\}$. Note that the "higher-order" variable $F$ poses no problem since no attempt is made to solve. Proceeding, a new S-residuation is obtained as $Ans = F(X)(2 \Rightarrow 1) = F(X, 1)$. One step further, $F$ is instantiated to the *twice* function. Thus, this mutates the previous S-residuation to $Ans = twice(X)(1)$. Next, $X$ becomes the identity function, thus releasing the residuation and instantiating $Ans$ to 1. Finally, the equation $sq(1) = 1$ is immediately verified, yielding success.

## 4 The $\lambda\pi\psi$ Molecule

Now that we have put together the pairwise bonds between the atoms; *i.e*, what constitutes the LIFE molecule as advertised in Figure 1 on Page 3. In LIFE one can specify types, functions, and relations. Rather than simply coexisting, these may be interwoven. Since the $\psi$-calculus is used in Log In and FOOL to provide a type inheritance systems of sorts to logic and functional programming, we can now enrich the expressiveness of the $\psi$-calculus with the power of computable functions and relations. More specifically, a basic $\psi$-term structure expresses only typed equational constraints on objects. Now, with FOOL and Log In, we can specify in addition ***arbitrary functional and relational constraints*** on $\psi$-terms.

In LIFE, a basic $\psi$-term denotes a functional application in FOOL's sense if its root symbol is a defined function. Thus, a *functional expression* is either a $\psi$-term or a conjunction of $\psi$-terms denoted by $t_1 : t_2 : \ldots : t_n$. An example of such is $append(list, L) : list$, where *append* is the FOOL function defined above. This is how functional dependency constraints are expressed in a $\psi$-term in LIFE. For example, in LIFE the $\psi$-term $foo(bar \Rightarrow X : list, baz \Rightarrow Y : list, fuz \Rightarrow append(X, Y) : list)$ is one in which the attribute *fuz* is derived as a list-valued function of the attributes *bar* and *baz*. Unifying such $\psi$-terms proceeds as before modulo residuation of functional expression whose arguments are not sufficiently refined to be subsumed by a function definition.

As for relational constraints on objects in LIFE, a $\psi$-term $t$ may be followed by a ***such-that*** clause consisting of the logical conjunction of literals $l_1, \ldots, l_n$. It is written as

$t \mid l_1, \ldots, l_n$. Unification of such relationally constrained terms is done modulo proving the conjoined constraints.

Let us take an example. We are to describe a LIFE rendition of a soap opera. Namely, a soap opera is a television show where a cast of characters is a list of persons. Persons in that strange world consist of alcoholics, drug-addicts, and gays. The husband character is always called "Dick" and his wife is always an alcoholic, who is in fact his long-lost sister. Another character is the mailman. The soap opera is such that the husband and mailman are lovers, and the wife and the mailman blackmail each other. Dick is gay, Jane is an alcoholic, and Harry is a drug-addict. In that world, it is invariably the case that the long-lost sister of gays are named "Jane" or "Cleopatra." Harry is a lover of every gay person. Also, Jane and a drug-addict blackmail one another if that drug-addict happens to be a lover of Dick. No wonder thus that it is a fact that this soap opera is terrible.

In LIFE, the above could look like:

$$cast := \{[\,]; [person|cast]\}.$$

$$\begin{aligned} soap\_opera := tv\_show(&characters \Rightarrow [H, W, M],\\ &husband \Rightarrow H : dick,\\ &wife \Rightarrow W : alcoholic : long\_lost\_sister(H),\\ &mailman \Rightarrow M)\\ \mid\ & lovers(M, H), blackmail(W, M). \end{aligned}$$

$$\begin{aligned} &person := \{alcoholic; drug\_addict; gay\}.\\ &dick \lhd gay.\\ &jane \lhd alcoholic.\\ &harry \lhd drug\_addict.\\ &long\_lost\_sister(gay) \Rightarrow \{jane; cleopatra\}.\\ &lovers(harry, gay).\\ &blackmail(jane, X : drug\_addict) :\!-\ lovers(X, dick).\\ &terrible(soap\_opera). \end{aligned}$$

Then, querying about a terrible TV show with its character cast is:

$$?\!-\ terrible(T : tv\_show(characters \Rightarrow cast)).$$

which unfolds from the above LIFE specification into:

$$\begin{aligned} T = soap\_opera(&characters \Rightarrow [H : dick, W : jane, M : harry],\\ &husband \Rightarrow H,\\ &wife \Rightarrow W,\\ &mailman \Rightarrow M) \end{aligned}$$

It is instructive as well as entertaining to convince oneself that somehow everything falls into place in this LIFE sentence.

## 5 Conclusion

We have overviewed some of the basic features of LIFE, a prototype programming language combining logic and functional programming, with a type system designed to accommodate multiple inheritance. Together, these features confer to LIFE a unique capability for AI applications like Natural Language Processing [AKL91], Computer-Aided Design, *etc.* We have illustrated LIFE's operations on various examples, and explained how the capabilities of each components may be combined. In fact, LIFE's conception as a composition of three calculi turns out to yield more power than intrinsic to each. Some of the examples we have shown already substantiate this claim, but there are even more pleasantly startling additional conveniences which have also come unexpectedly with our design such as (bounded) polymorphic types, infinite streams, deamonic constraints, and more. Examples of these may be found in [AKP90,AKM90,AKP91].

Finally, we must mention that quite a decent C implementation of a LIFE interpreter embodying all the concepts presented here has been realized by Richard Meyer. It is called *Wild_LIFE* [AKM90], and is in the process of being released as public domain software by Digital's Paris Research Laboratory. We hope to share it soon with the programming community at large so that LIFE may benefit from the popular wisdom of real life users, and hopefully contribute a few effective conveniences to computer programming, then perhaps evolve into *Real_LIFE.*

## References

[AK84] Hassan Aït-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures.* PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1984.

[AK86] Hassan Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351, 1986.

[AKBLN89] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989.

[AKL91] Hassan Aït-Kaci and Patrick Lincoln. LIFE, a natural language for natural language. *T. A. Informations*, 1991. (To appear).

[AKLN87] Hassan Aït-Kaci, Patrick Lincoln, and Roger Nasr. Le Fun: Logic, equations, and functions. In *Proceedings of the Symposium on Logic Programming*, pages 17–23, San Francisco, CA, USA, September 1987.

[AKM90] Hassan Aït-Kaci and Richard Meyer. Wild_LIFE, a user manual. PRL Technical Note 1, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, 1990.

[AKN86] Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[AKN89] Hassan Aït-Kaci and Roger Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[AKP90] Hassan Aït-Kaci and Andreas Podelski. Is there a meaning to LIFE? Research paper, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, 1990.

[AKP91] Hassan Aït-Kaci and Andreas Podelski. Functions as passive constraints in LIFE. Research paper, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, 1991.

[CM84] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, Germany, 2nd edition, 1984.

[HMT88] Robert Harper, Robin Milner, and Mads Tofte. The definition of standard ML - Version 2. Report LFCS-88-62, University of Edinburgh, Edinburgh, UK, 1988.

[Lan63] Peter Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1963.

[O'K90] Richard O'Keefe. *The Craft of Prolog.* Series on Logic Programming. MIT Press, Cambridge, MA, USA, 1990.

[PJ87] Samuel Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[SAK89] Gert Smolka and Hassan Aït-Kaci. Inheritance hierarchies: Semantics and unification. *Journal of Symbolic Computation*, 7:343–370, 1989.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* Series on Logic Programming. MIT Press, Cambridge, MA, USA, 1986.

[Tur85] David Turner. Miranda—Non-strict functional programming with polymophic types. In Jean-Pierre Jouannaud, editor, *Proceedings on the Conference on Functional Programming Languages and Computer Architecture (Nancy, France)*, pages 1–16, Berlin, Germany, 1985. Springer Verlag. (LNCS 201).