LIFE, a Natural Language for Natural Language

Hassan Aït-Kaci
Patrick Lincoln

1989

Publication Notes

This article is a revised version of MCC Technical Report Number ACA-ST-074-88, Microelectronics and Computer Technology Corporation, 3500 West Balcones Center Drive, Austin, TX 78759, USA. This present version has been published in a special issue on Unification Grammars of *T. A. Informations, revue internationale du traitement automatique du langage* (volume 30, number 1-2, 1989 pages 37–67), an international journal on computational linguistics (ISSN 0039-8217) published by the *Centre Nationational de la Recherche Scientifique* and the *Ecole Normale Supérieure de Saint-Cloud*, France.

The authors' current addresses are:

Hassan Aït-Kaci
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92563 Rueil-Malmaison Cedex
France
hak@prl.dec.com

Patrick Lincoln
Computer Science Department
Margaret Jacks Hall
Stanford University
Stanford, CA 94305
USA
lincoln@polya.stanford.edu

Abstract

Experimenting with formalisms for Natural Language Processing involves costly programming overhead in conventional computing idioms, even as "advanced" as Lisp or Prolog. LIFE (Logic, Inheritance, Functions, and Equations) is a programming language which incorporates an elegant type system which supports a powerful facility for structured type inheritance. Also, 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. Computational linguistics is a discipline where such abstractions are particularly useful. Therefore, obvious convenience is offered by LIFE for experimentation to the computational linguist, who becomes relieved from burdensome yet extrinsic programming complications. We presently attempt to show how LIFE may be a natural computer language for processing natural human languages.

Keywords

Natural Language Processing, Type Inheritance, Logic Programming, Functional Programming, Delayed Evaluation.

Acknowledgements

We owe much to Roger Nasr for his momentous contribution to the conception and architecture of LIFE. Many thanks also to Raymonde Guindon and Jon Schlossberg for their kind tutorial help in computational linguistics while implementing the LIFE NL parser.

We would like to thank David Plummer for *Still Life*, his original partial implementation of LIFE in Prolog which allowed us to experiment with many ideas presented in this paper. We also thank Richard Meyer for implementing *Wild Life* a complete, portable, and efficient LIFE interpreter realized in C [16]. Richard's implementation was done at Digital, Paris Research Lab., and is available as public domain software.

Contents

> We modern Europeans (...) have lost the ability to think in large dimensions. We need a change in *Lebensgefuhl* [our feeling for life]. It is my hope that the enormous perspective of human growth which has been opened to us by [this] research (...) may serve to contribute in some small measure to its development.
>
> LEO FROBENIUS, *Volksmärchen und Volksdichtungen Afrikas.*

1 Introduction

LIFE, so-denominated for **L**ogic, **I**nheritance, **F**unctions, and **E**quations, is a prototype programming language. It 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*, and *structure-oriented*, each being an operational rendition of a well-defined underlying model.

Formalisms for linguistic analysis have emerged, based on Horn clause logic [20], frame unification [23], λ-calculus [25], each proving itself adequate for particular aspects of Natural Language Processing (NLP). LIFE happens to reconcile all these approaches, therefore offering a unique experimental tool for the computational linguist. To be sure, there are other efforts attempting to tailor programming languages, typically logic programming, for linguistic analysis. (As has been pointed out in [12], order-sorted logic is quite convenient for parsing.) Among those known to us CIL [17, 18] is one that comes close to LIFE's spirit in that it combines partial features of Log In [3] (see Section 2.2.2) with delayed evaluation handled with an explicit *freeze* meta-predicate borrowed from Prolog-II [11]. CIL's constructs are called Partially Specified Terms (PST's) which are exactly the same as feature matrices used in Unification Grammars [23], and are a strict particular case of Log In's ψ-terms. To our knowledge PST's do not accommodate disjunctive constructs, nor do they use a type hierarchy, let alone type definitions. In addition, judging from the literature, we find CIL constructs rather unnecessarily convoluted as opposed to our simple LIFE style, although the reader is encouraged to make an opinion for herself. On the programming language side, there is a growing multitude dealing with integrating logic and functional programming. However, none of them worries about bringing in frame-like unification or inheritance, and few have higher-order functions. We refer the reader to [6] for a survey of prominent approaches. LIFE stands apart as the only formalism we know which encompasses such a breadth of functionality.

This document consists essentially of two parts: an informal overview of LIFE (Section 2) and a particular experiment applying LIFE to linguistic analysis meant as an illustration of its adequacy (Section 3). For a formal semantics of LIFE, the reader is referred to [9] where all aspects of LIFE are given a rigorous mathematical meaning.

2 The Chemistry of LIFE

LIFE is a trinity. The function-oriented component of LIFE is directly derived from functional programming languages standing on foundations in the λ-calculus like HOPE [10], SASL [26], ML [13], or Miranda [27]. 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 [24] language, 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 heretofore 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 ψ-calculus [2, 4]—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 2) whose vertices would be some essential operational renditions of, respectively, λ-calculus, Horn clause resolution, and ψ-calculus. (It is assumed that the reader is familiar with the essential terminology and notions of functional and logic programming.) Therefore, we shall first very briefly and informally describe what we understand to be the canonical functionality found in each vertex. Then, we shall describe how we achieve pairwise bonding. Lastly, we shall synthesize the molecule of LIFE from the three atomic vertices and the pairwise bonds.

2.1 The Atoms

This section gives a very brief and informal operational account of functional programming, logic programming, and type inheritance.

2.1.1 λ-Calculus: Computing with Functions

The view taken by this way of computing is to formulate every computational object as a functional expression. There are essentially two sorts of such expressions—constants and reducible expressions. Constants may be of any order of type. Ground objects are null-order constants (values) and abstractions are higher-order constants (functions). Typically, these

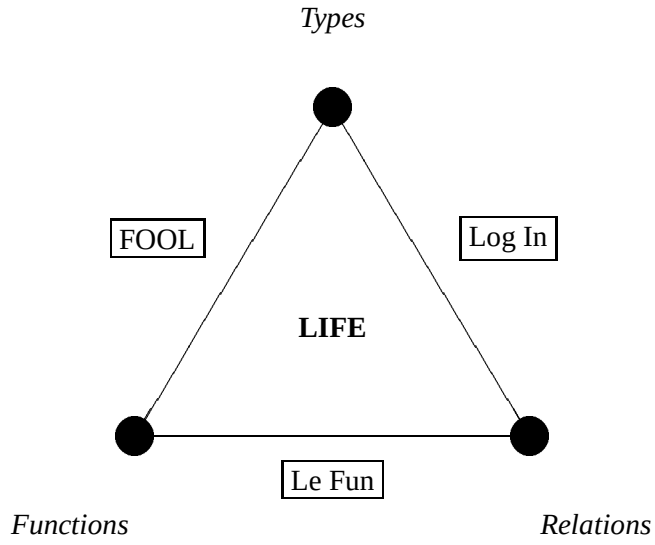


Figure 1: The LIFE Molecule

evaluate to themselves. Reducible expressions are essentially applications. Indeed, the only rule of computation is β-reduction in the context of a global store of defined constants. Strategies of reduction consisting of whether arguments in a function application are reduced first or last fall into pragmatic considerations, and are irrelevant to this particular description. For the sake of choice, we shall assume *applicative order* of reduction, although *normal order*, lazy or otherwise, could be as well considered.

Although the "pure" λ-calculus is computationally complete, and therefore theoretically sufficient to express all general recursive functions, a "real-life" functional programming language will typically have a built-in store of constants of which the user's definitions may be seen as an extension. At the very least, the usual integer arithmetic constants and functions would be assumed defined, as well as boolean constants and null-order equality—*i.e.*, equality on ground values. Notably, and regardless of the chosen evaluation strategy, an exceptional constant function will also assumed defined for conditional expressions. The simplest conditional function is a three argument *if-then-else* whose infix form usually is *if* e_1 *then* e_2 *else* e_3, and whose evaluation consists in first evaluating e_1 whose boolean value, upon termination, will determine evaluation of either e_1 or e_2, yielding the result of the whole conditional expression. Thus, as an archetypical example, the factorial function may be defined as:[1]

[1]We shall use $\Rightarrow$ to express global definitions; *i.e.*, the facility which installs a constant in the global constant store.

$$\mathit{fact}(n) \Rightarrow \mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n * \mathit{fact}(n-1).$$

Some functional programming languages make recursion syntactically explicit by differentiating two defining facilities. For instance a definition announced by a reserved word (*rec*, say) as in

$$\mathit{rec}\ \mathit{fact}(n) \Rightarrow \mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n * \mathit{fact}(n-1).$$

would explicitly specify that occurrences of the constant being defined in its own body are recursive occurrences as opposed to namesakes which are presumably defined in the global store. The reason has its roots in the simple way these definitions can be parsed by left-recursive descent and readily translated into a form stripped of syntactic adornments which requires the explicit use of the recursion combinator Y. However, this is not strictly required as LL(1) parsing or even implementation of recursion with Y are necessary, especially when efficiency rather than simplicity of implementation is sought [21]. Thus, we shall dispense from such explicit *rec* mentions, (mutual) recursion being systematically implicit when and only when free occurring constants appear in definitions (as in the first of the two foregoing definitions).

These basic paraphernalia are yet 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:

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

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:

$$\mathit{append}(l1, l2) \Rightarrow \mathit{if}\ l_1 = []\ \mathit{then}\ l_2\ \mathit{else}\ \mathit{fst}(l_1).\mathit{append}(\mathit{snd}(l_1), l_2).$$

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:

$$\begin{aligned} \mathit{append}([], l) &\Rightarrow l. \\ \mathit{append}(h.t, l) &\Rightarrow h.\mathit{append}(t, l). \end{aligned}$$

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. Not only are pattern-directed definitions more perspicuous, they also lead to more efficient code generation. 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 [21].

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 defined 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:

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

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 σ 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.

2.1.2 π-Calculus: Computing with Relations

Logic programming, of which Prolog is *the* canonical language, expresses programs as relational rules of the form:

$$r_0(\vec{t}_0) \leftarrow r_1(\vec{t}_1), \ldots, r_n(\vec{t}_n).$$

where the r_i's are relationals symbols and the $\vec{t_i}$'s are tuples of first-order terms. One reads such a rule as: *"For all bindings of their variables, the terms $\vec{t_0}$ are in relation r_0 if the terms $\vec{t_1}$ are in relation r_1 and ... the terms $\vec{t_n}$ are in relation r_n."* In the case where $n = 0$, the rule reduces to the simple unconditional assertion $r_0(\vec{t_0})$ that the terms $\vec{t_0}$ are in relation r_0. These rules are called *definite clauses*; expressions such as $r_i(\vec{t_i})$ are called *literals*; the *head* of a definite clause is the literal on the left of the arrow, and its *body* is the conjunction of literals on the right of the arrow.

Given a set of such definite clauses, linear resolution is the non-deterministic computation rule by which such rules are giving interpretations to *query* expressions of the form:

$$\leftarrow q_1(\vec{s}_1), \ldots, q_m(\vec{s}_m).$$

which may be read: *"Does there exist some binding of variables such that the terms $\vec{s}_1$ are in relation q_1 and ... $\vec{s}_m$ are in relation q_m?"* The linear resolution rule is a transformation rule applied to a query. It consists in choosing a literal $q_i(\vec{s}_i)$ in the query's body and a definite clause in the given set whose head $r_0(\vec{t_0})$ *unifies* with $q_i(\vec{s}_i)$ thanks to a variable substitution σ (*i.e.*, $q_i(\vec{s}_i)\sigma = r_0(\vec{t_0})\sigma$), then replacing it by the body of that clause in the query, applying substitution σ to all the new query. That is,

$$\leftarrow q_1(\vec{s}_1)\sigma, \ldots, q_{i-1}(\vec{s}_{i-1})\sigma, r_1(\vec{t}_1)\sigma, \ldots, r_n(\vec{t}_n)\sigma, q_{i+1}(\vec{s}_{i+1})\sigma, \ldots, q_m(\vec{s}_m)\sigma.$$

The process is repeated and stops when and if the query's body is empty (success) or no rule head unifies with the selected literal (failure). There are two non-deterministic choices made in the process: one of a literal to rewrite in the query and one among the potentially many rules whose head unify with this literal.

Prolog's computation rule is called SLD-resolution. It is a deterministic flattening of linear resolution; that is, it is a particular deterministic approximation implementing the above non-deterministic computation rule. It consists in seeing a program as an *ordered* set of definite clause, and a definite clause body as an *ordered* set of literals. These orders are meant as a rigid guide for the two choices made by the linear resolution rule. Thus, Prolog's particular computation strategy transforms a query by rewriting the query literals in their order, attempting to unify against heads of rules in the order of the rules. If failure is encountered, a backtracking step to the latest choice point is made, and computation resumed there with the next alternative.

In exactly the same spirit as β-reduction is for the λ-calculus, strategies of choice of where to apply the linear resolution computation rule are all theoretically consistent in the sense that if computation terminates, the variable binding exhibited is a legitimate solution to the original query. However, not all possible linear resolution strategies are complete. Indeed, pretty much as applicative order reduction in the λ-calculus may diverge on an expression which does have a normal form, Prolog's particular strategy of doing linear resolution may diverge although finite solutions to a query may exist.

Central to logic programming is the presence of first-order constructor terms as well as unification.

2.1.3 ψ-Calculus: Computing with Types

The ψ-calculus consists of a syntax of structured types called ψ-terms together with subtyping and type intersection operations. Intuitively, as expounded in [3], the ψ-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 ψ-term is that of a data structure built out of constructors, access functions, and subject possibly to equational constraints which reflect access coreference—sharing of structure. Thus, the syntactic operations on ψ-terms which stand analogous to instantiation and unification for first-order terms simply denote, respectively, sub-algebra ordering and algebra intersection, modulo type and equational constraints. This scheme even accommodates type constructors which are known to be partially-ordered with a given subtyping relation. As a result, a powerful operational calculus of structured subtypes is achieved formally without resorting to complex translation trickery. In essence, the ψ-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* *with*

functions

name : *person* $\rightarrow$ *id*
first : *id* $\rightarrow$ *string*
last : *id* $\rightarrow$ *string*
spouse : *person* $\rightarrow$ *person*

equations

last(*name*(X)) = *last*(*name*(*spouse*(X)))
spouse(*spouse*(X)) = X

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

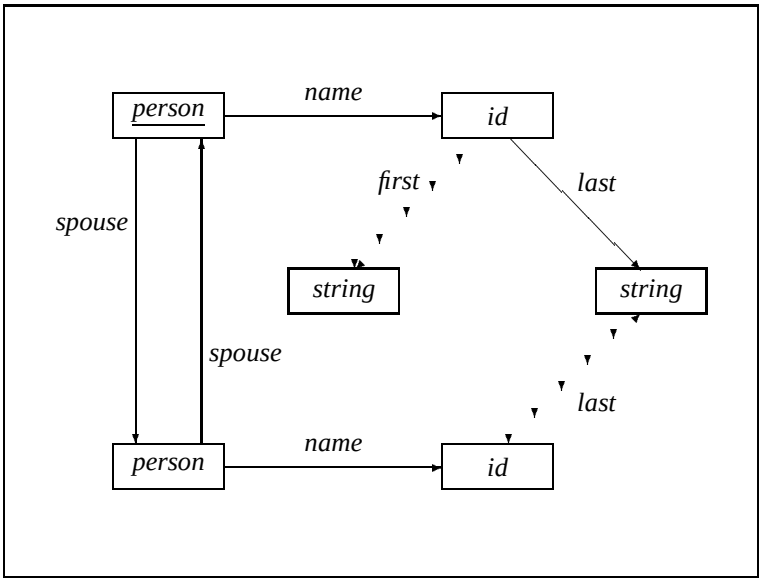


Figure 2: A Functional Diagram

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

Since it is beyond the informal scope of this paper, we shall abstain from giving a complete formal definition of ψ-term syntax. (Such may be found elsewhere [4, 3].) Nevertheless, it is important to distinguish among the three kinds of symbols which participate in a ψ-term expression. Thus we assume given a signature of *type constructor symbols*, a set $\mathcal{A}$ of *access function symbols* (also called *attribute* symbols), and a set $\mathcal{R}$ of *reference tag symbols*. In the ψ-term above, for example, the symbols *person*, *id*, *string* are drawn from , the symbols *name*, *first*, *last*, *spouse* from $\mathcal{A}$, and the symbols X, S from $\mathcal{R}$.[2]

A ψ-term is either *tagged* or *untagged*. A tagged ψ-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 ψ-term. An untagged ψ-term is either *atomic* or *attributed*. An atomic ψ-term is a type symbol in . An attributed ψ-term is an expression of the form $s(l_1 \Rightarrow t_1, \ldots, l_n \Rightarrow t_n)$ where $s \in$ and the ψ-term principal type, the l_i's are mutually distinct attribute symbols in $\mathcal{A}$, and the t_i's are ψ-terms ($n \geq 1$).

Reference tags 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 reference tags consistently refer to the same structure. For example, the reference tag X in

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

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

refers consistently to the atomic ψ-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

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

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

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

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

Similarly to first-order terms, a subsumption preorder can be defined on ψ-terms which is an ordering up to reference tag renaming. Given that the signature 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 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 $\mathit{student} < \mathit{person}$ and $\mathit{austin} < \mathit{cityname}$ in then the ψ-term

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

is subsumed by the ψ-term

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

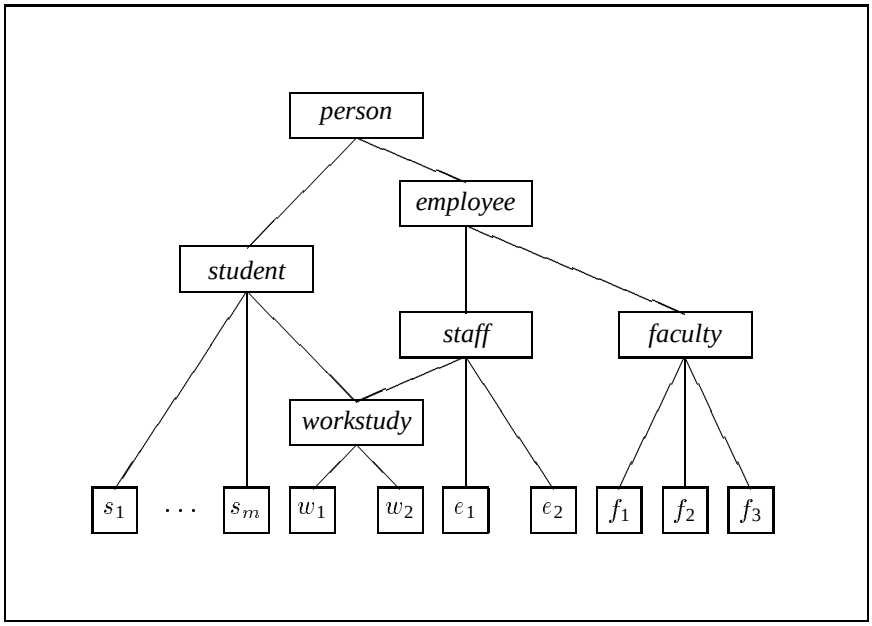


Figure 3: A Signature with Well-Defined GLB's

In fact, if the signature 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. Consider for example the signature displayed in Figure 3 and the two ψ-terms:

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

and

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

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

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

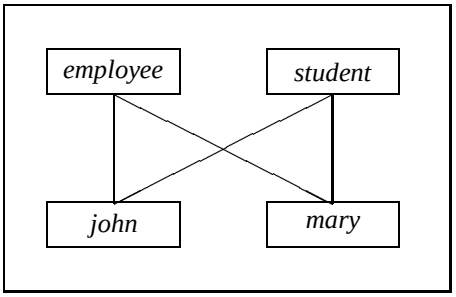
A detailed unification algorithm for ψ-terms is given in [3]. This algorithm is an adaptation of an efficient unification algorithm based on a rooted labeled (directed) graph representation of ψ-terms, such as is illustrated in Figure 2. The nodes are labeled with type symbols from , and the arcs are labeled with attribute symbols. The root node is one from which every other is reachable and is labeled with the principal type of the ψ-term (underlined in Figure 2). Nodes which are shared in the graph correspond to tagged subterms. Such graphs are quite like finite-state automata with -sorted nodes (Moore machines) and where the transitions are attribute symbols. In fact, the ψ-term unification algorithm is an immediate adaptation of the algorithm deciding equivalence of finite-state automata [1]. This algorithm merges nodes which are reached by equal transition paths into coreference classes, starting from the roots and following all reachable strings of attributes from them. Each merged class is assigned the type symbol in which is the GLB of the types of all nodes in the class. The inconsistent type $\perp$ (the least element in) may result which makes the whole unification fail.

Incidentally, if *least upper bounds* (LUBs) are defined as well in , so are they for ψ-terms. Thus, a lattice structure can be extended from to ψ-terms [2, 4]. For example, for these two ψ-terms, their LUB (denoting their most specific generalization) is:

person(*advisor* $\Rightarrow$ *faculty*(*secretary* $\Rightarrow$ *employee*,
assistant $\Rightarrow$ *person*),
roommate $\Rightarrow$ *person*)).

Although it may turn out interesting in other contexts, we shall not use this generalization operation here.

A technicality arises if is not a lower semi-lattice. For example, given the (non-lattice) type signature:



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.[3] Thus the ψ-term syntax may be enriched with disjunction denoting type union.

[3] See [7] for a description of an efficient method for computing such GLB's.

For a more complete formal treatment of disjunctive ψ-terms, the reader is referred to [4] and to [3]. It will suffice to indicate here that a *disjunctive* ψ-term is a set of incomparable ψ-terms, written $\{t_1; \ldots; t_n\}$ where the t_i's are basic ψ-terms. A *basic* ψ-term is one which is non-disjunctive. The subsumption ordering is extended to disjunctive (sets of) ψ-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 $\bot$. Unification of two disjunctive ψ-terms consists in the enumeration of the set of all maximal ψ-terms obtained from unification of all elements of one with all elements of the other. For example, limiting ourselves to disjunctions of atomic ψ-terms in the context of signature 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*; $\bot$; *workstudy*$\}$ of pairwise GLB's.

In practice, it is convenient to allow nesting disjunctions in the structure of ψ-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:

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

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

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

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.

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 ψ-terms. The process is described in [2], 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 ψ-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.

Last in this brief introduction to the ψ-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 λ-calculus. The idea is that types in the signature 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 ψ-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 [22], to which the reader is referred for a full formal account.

For example, given a signature for the specification of linear lists $= \{list, cons, nil\}$[4] 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, ψ-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.*, $cons(tail \Rightarrow list)$ unified with *cons* gives $cons(tail \Rightarrow list)$). Thus, attribute inheritance is done "lazily," saving much unnecessary expansions.

2.2 The Bonds

In this section we indicate briefly how to operationalize pairwise combination calculi from λ, π, and ψ computation models. That is, we describe the edges of the triangle of LIFE in Figure 2 on Page 3—the bonds between the atoms of the LIFE molecule. We shall keep an informal style, illustrating key points with examples.

2.2.1 $\lambda\pi$-Calculus: Le Fun

We now introduce a relational and functional programming language called Le Fun [5, 6] where first-order terms are generalized by the inclusion of *applicative expressions* as defined by Landin [15] (atoms, abstractions, and applications) augmented with first-order constructor terms. Thus, *interpreted* functional expressions may participate as *bona fide* arguments in logical expressions.

A unification algorithm generalized along these lines 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 the operands are ready. That is, until

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

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:

$q(X, Y, Z) :\!- p(X, Y, Z, Z), pick(X, Y).$

$p(X, Y, X + Y, X * Y).$
$p(X, Y, X + Y, (X * Y) \perp 14).$

$pick(3, 5).$
$pick(2, 2).$
$pick(4, 6).$

and the following query:

?– $q(A, B, C).$

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, $A + B = (A * B) \perp 14$. 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:

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

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 $X = V \perp W$ 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 $Ans = (V \perp W) * Y$. 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 $Ans = (V \perp W) * (V + W)$, 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*) ⇒ $X * X$.
twice(*F*, *X*) ⇒ *F*(*F*(*X*)).

valid_op(*twice*).
$p(1)$.
pick(*lambda*(X, X)).
$q(V)$:– $G = F(X), V = G(1)$, *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 $Ans = F(X)(1)$. 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.

2.2.2 $\pi\psi$-Calculus: Log In

Log In is simply Prolog where first-order constructor terms have been replaced by ψ-terms, with type definitions [3]. Its operational semantics is the immediate adaptation of that of Prolog's SLD resolution described in Section 2.1.2. Thus, we may write a predicate for list concatenation as:[5]

list := {[]; [_|*list*]}.
append([], *L* : *list*, *L*).
append([*H*|*T* : *list*], *L* : *list*, [*H*|*R* : *list*]) :– *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 signature of Figure 4. This taxonomic information is expressed in Log In as:

student ◁ *person*.
student := {*peter*; *paul*; *mary*}.
grade := {*goodgrade*; *badgrade*}.

[5] First-order terms being just a particular case of ψ-terms, we use such an expression as $f(t_1, \ldots, t_n)$ them as implicit syntax for $f(1 \Rightarrow t_1, \ldots, n \Rightarrow t_n)$. Thus, pure Prolog is fully subsumed. In particular, we adopt its notation for lists, and _ for "don't-care" *a.k.a.* $\top$.

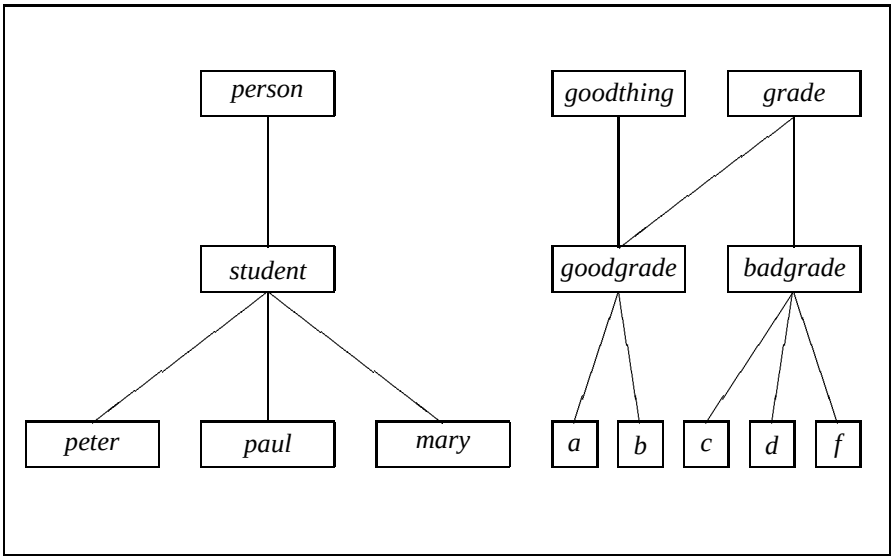


Figure 4: The Peter-Paul-Mary Signature

$goodgrade \triangleleft goodthing.$
$goodgrade := \{a; b\}.$
$badgrade := \{c; d; f\}.$

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,

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

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

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

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

2.2.3 $\psi\lambda$-Calculus: FOOL

FOOL is simply a pattern-oriented functional language where first-order constructor terms have been replaced by ψ-terms, with type definitions. Its operational semantics is the immediate adaptation of that described in Section 2.1.1. Thus, we may write a function for list concatenation as:

$$list := \{[]; [\_|list]\}.$$
$$append([], L : list) \Rightarrow L.$$
$$append([H|T : list], L : list) \Rightarrow [H|append(T, L)].$$

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

$$map([], \_) \Rightarrow [].$$
$$map([H|T], F) \Rightarrow [F(H)|map(T, F)].$$

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

The ψ-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 the signature in Figure 3 on Page 10, 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.

2.3 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 2 on Page 3. In LIFE one can specify types,

functions, and relations. Rather than simply coexisting, these may be interwoven. Since the ψ-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 ψ-calculus with the power of computable functions and relations. More specifically, a basic ψ-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 ψ-terms.

In LIFE, a basic ψ-term denotes a functional application in FOOL's sense if its root symbol is a defined function. Thus, a *functional expression* is either a ψ-term or a conjunction of ψ-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 ψ-term in LIFE. For example, in LIFE the ψ-term

$$\mathit{foo}(\mathit{bar} \Rightarrow X : \mathit{list}, \mathit{baz} \Rightarrow Y : \mathit{list}, \mathit{fuz} \Rightarrow \mathit{append}(X, Y) : \mathit{list})$$

is one in which the attribute *fuz* is derived as a list-valued function of the attributes *bar* and *baz*. Unifying such ψ-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 ψ-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:

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

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

$$\mathit{person} := \{\mathit{alcoholic}; \mathit{drug\_addict}; \mathit{gay}\}.$$
$$\mathit{dick} \lhd \mathit{gay}.$$

jane ◁ *alcoholic*.
harry ◁ *drug_addict*.

long_lost_sister(*gay*) ⇒ {*jane*; *cleopatra*}.

lovers(*harry*, *gay*).

blackmail(*jane*, *X* : *drug_addict*) :– *lovers*(*X*, *dick*).

terrible(*soap_opera*).

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

?– *terrible*(*T* : *tv_show*(*characters* ⇒ *cast*)).

which unfolds from the above LIFE specification into:

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

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

3 Natural Language

This section is a description of a specific parser of a very small subset of English where syntactic, semantic, and pragmatic constraints are expressed all at once. This alleviates the need to pipeline many invalid forms from syntax to semantics, then to pragmatics. This example is by no means to imply that its parsing scheme is what we recommend: We are, indeed, mere neophytes in computational linguistics. We nonetheless hope to convince the computational linguist that we, as programming language designers, did put together in LIFE a unique functionality for NLP.

3.1 Traditional NLP

Natural language understanding systems are notoriously large, inefficient systems with slow response times. Thus, optimizing the parsing process is an important task if natural language is to be used in realistic user interfaces.

Traditional natural language processing systems work in three phases. The first, syntactic analysis, phase determines the surface structure of the input—looking up words in a dictionary, checking word order, etc. The second phase determines some of the semantic content of the input—checking semantic agreement and enforcing selectional restrictions. Finally, the

third phase determines the deepest meaning behind an input—binding anaphora to their referents, analyzing paragraph or discourse structure, and checking the consistency of the guessed meaning of the input with world knowledge. Although quite standard even in state of the art natural language processing systems, this sequential method contains some inherent inefficiencies.

Obviously, if there is a deep semantic clash at the beginning of a long input, one would hope that a system would not waste too much time processing the entire input before noticing the clash. More commonly, there will be many readings of an input, and most of them will be be semantically flawed. It is desirable that the semantic flaws be found as soon as possible, eliminating the wasted work of doing even the surface analysis of the rest of the input under the bad readings. However, this is very difficult to achieve using the traditional approach of three phase processing. Only by doing all levels of processing simultaneously can a system achieve the desired behavior.

By processing input syntax, semantics, and pragmatics at the same time a system has the further opportunity to use the semantics to drive the syntax. For instance, if the semantics of the first part of an input have been discovered, and the topic is known, any lexical ambiguities (multiple word definitions, etc) may be correctly interpreted immediately. In traditionally constructed systems, the lexical forms of all ambiguous readings would be fully fleshed out, and only upon semantic checking would they be thrown out.

Pushing the semantics and pragmatics through to the initial grammar seems daunting to those familiar with implementations of natural language systems. Efficiently handling all the constraints on language is very difficult, even in such high level languages as Lisp or Prolog. However, LIFE's formalism is one in which complex constraints are easily and cleanly incorporated in declarative programs through the intermingling of relational and functional expressions.

3.2 NLP in LIFE

A simplified natural language processing system was built in LIFE as an experiment in using LIFE's full functionality on a complex problem. First, a simple best first chart parser was constructed using a standard logic programming cliche. Second, constraints were added to the categories and objects in the parser, in order to enforce number agreement and similar trivial conditions. Then semantic functions which expressed the meanings of certain words were included in the dictionary definitions of those words. The grammar was also modified to use those functions, when present, to enforce semantic agreement and selectional restrictions. Finally, the grammar was modified slightly to force unification of pronouns with referents, resolving all anaphora into coreferences.

Thus when an input is presented, all the constraints come to bear immediately. As soon as a verb with a semantic function is looked up in the dictionary, its entire semantics are enforced. LIFE automatically handles the delayment of functional expressions until certain arguments are sufficiently bound, firing the function at the earliest possible time. Using this

functionality, powerful semantic functions can be posted as constraints as soon as they apply. Also, LIFE's unification routine supports partially ordered partially specified types, which is useful in capturing semantic information.

3.2.1 Syntax—The Grammar

The initial parsing routine was encoded as a set of facts and relations, broken up into three main categories; a set of grammar rules expressing English word order and basic grammar, a dictionary relation from words to categories, and a parser, which relates lists of words to categories such as noun phrase or sentence.

Each grammar rule was encoded as a LIFE fact, relating some number of constituent categories to a single result category. For example,

grammar_rule(*np*, *art*, *n*).
grammar_rule(*s*, *np*, *vp*).

could be read as "an article followed by a noun can be a noun phrase, and a noun phrase followed by a verb phrase can be a sentence." Then a small dictionary was constructed which related words to their definition. For example,

dictionary(*compilers*, *n*).
dictionary(*john*, *pn*).
dictionary(*the*, *art*).
dictionary(*runs*, *iv*).

could be read as "'*compilers*' is a noun, '*john*' is a proper noun, '*the*' is an article, and '*runs*' is an intransitive verb." A particular word may have multiple definitions in the dictionary, which are chosen nondeterministically. To complete the base system, a simple parser was constructed which attempts to find a reading of the input list of words which satisfies the given category.

parse(*List*, *null*, *List*).

parse([*Word*|*Rest*], *Cat*, *S1*) :–
    dictionary(*Word*, *Def*),
    grammar_rule(*Cat*, *Def*, *Needing*),
    parse(*Rest*, *Needing*, *S1*).

parse([*Word*|*Rest*], *Cat*, *S1*) :–
    dictionary(*Word*, *Def*),
    grammar_rule(*Cat*, *Cat1*, *Cat2*),
    parse([*Def*|*Rest*], *Cat1*, *S2*),
    parse(*S2*, *Cat2*, *S1*).

One might call the parser with

parse([*john*, *runs*], *s*, []).

In order to prevent useless search in a chart parser, it is necessary to precompute a "next word" attribute for each grammar rule. As given above, parsing would progress bottom-up. In order to enforce a left-to-right strategy, one needs to restrict which grammatical rules can be used based on the next word in the sequence. For instance, the first grammar-rule given above describing noun phrases should only be applied when the next word in the sequence is an article. The next word for the second grammar rule above is $\{art; n; adj; pn; pron\}$ which stands for article or noun or adjective or proper noun or pronoun. (Other rules for sentences which start with interjections, adverbs, *etc*, and rules for noun phrases starting with nouns, adjectives, proper nouns and pronouns do exist.) In order to take advantage of the precomputed next word, an extra argument was added to each grammar rule. Thus, given a list of words, the category of the first word is looked up in the dictionary. The two rules above have become:

grammar_rule(*np*, *art*, *art*, *n*).
grammar_rule(*s*, $\{art; n; adj; pn; pron\}$, *np*, *vp*).

Then, only those grammar rules which have the proper category as a possible first word are tried.

Thus extended, these grammar rules can now be read as "a string of words starting with an article, if made up of an article followed by a noun, can be seen as a noun phrase, and a string of words starting with either an article, noun, adjective, proper noun, or pronoun, if made up of a noun phrase followed by a verb phrase, can be seen as a sentence." Thus, if given a string of words beginning with an adverb, neither rule would fire, since the precomputed "next word" attributes both fail to unify with *adverb*. However, given a list of words starting with a proper noun, the second rule could fire.

The operation of this parser is simply to find the first reading of an input form that satisfies the user specified category. If there is a choice at any point, for instance the choice of which definition of a word, or which grammar rule to use, a non-deterministic choice is made. If there is a failure along the way, due to some category not unifying with another, or the precomputed next word disallowing the use of a grammar rule, then backtracking ensues. Control returns to the last nondeterministic choice made, where a new choice is demanded. If there are no choices left to make, control returns to the previous choice. If there is no previous choice, then failure has occurred, and it is reported that the input form cannot be parsed into the given category.

3.2.2 Semantics—The Constraints

Using the above as a base, additional rules of proper English were encoded by modifying dictionary entries and grammar rules. For instance, number agreement is enforced by adding a number field to certain dictionary entries:

$dictionary(compilers, n(number \Rightarrow plural)).$
$dictionary(john, pn(number \Rightarrow singular)).$
$dictionary(the, art).$
$dictionary(runs, iv(number \Rightarrow singular)).$

Since '*the*' can be either singular or plural, the number field is left out, and is implicitly anything. Also, certain grammar rules were restricted to constituents that agreed:

$$grammar\_rule(\{art; n; adj; pn; pron\}, s, \\ np(number \Rightarrow N), vp(number \Rightarrow N)).$$

So long as the number of the noun phrase can be coerced to be the same as the number of the verb phrase, the two together can be read as a sentence. Thus, '*john runs*' is accepted, since the number of '*john*' and '*runs*' agree. However, '*compilers runs*' is rejected, since '*compilers*' is plural, and '*runs*' is singular.

Gender agreement was added in precisely the same manner, although less words and rules have an explicit gender. The gender attribute is also useful in anaphora resolution. Moreover, selectional restrictions were added. Fields of dictionary definitions were added to enforce these constraints:

$$dictionary(throws, tv(number \Rightarrow singular, \\ object \Rightarrow projectile, \\ subject \Rightarrow animate)).$$
$dictionary(john, pn(number \Rightarrow singular, class \Rightarrow human)).$
$dictionary(frisbee, n(number \Rightarrow singular, class \Rightarrow projectile)).$

The first entry above describes the transitive verb '*throw*'. Its object must be a projectile, and its subject mush be animate. The second entry is a slight modification of the above definition of '*john*', the added information is that '*john*' is human. The corresponding grammar rules were then modified to use this information:

$$grammar\_rule(\{art; n; adj; pn; pron\}, s, \\ np(number \Rightarrow N, class \Rightarrow X), \\ vp(number \Rightarrow N, subject \Rightarrow X)).$$

Here the sentence rule is modified to ensure that the verb phrase's subject can be unified with the class of the noun phrase. In this way, semantic information is gathered at the same time that syntactic constraints are met.

A semantic hierarchy was constructed to account for the meanings of the classes mentioned in the dictionary; *e.g.*,

$human \triangleleft animate.$

which can be read as "anything which is human is also animate." This semantic hierarchy can be very rich. Using the powerful inheritance mechanisms of LIFE, complex semantic domains can be described very economically. Rules of the type shown above express simple type subsumption, but if the type *animate* has any attributes, those are inherited by the type *human*. Thus, the list of words '*John throws the frisbee*' can be parsed only if john is animate, which he is since he is human, and if the frisbee is a projectile, which it is. This kind of constraint is very simple to enforce, since it has been translated into type checking during unification.

Simple constraints like "only projectiles can be thrown" are thus simple to implement. In order to express more complex constraints, like "carnivores eat meat," functions were used. Each definition of a verb can have a semantic function which can express constraints on its subject or object that are dependent on something else. Consider the verb '*eat.*'

$$\begin{aligned} &\mathit{dictionary}(\mathit{eat}, \mathit{tv}(\mathit{subject} \Rightarrow \mathit{animate}(\mathit{eating\_habit} \Rightarrow \mathit{EH}),\\ &\qquad\qquad \mathit{object} \Rightarrow \mathit{food} : \mathit{eaten\_by}(\mathit{EH}),\\ &\qquad\qquad \mathit{number} \Rightarrow \mathit{plural})). \end{aligned}$$

The word '*eat*' is a plural transitive verb whose subject must be animate, and whose object must be food. Further, if the subject has an eating habit, then the object must be edible by something with that eating habit. The pattern-directed function *eaten_by* expresses the relationship between an eating habit and eaten objects:

$$\begin{aligned} &\mathit{eaten\_by}(\mathit{vegetarian}) \Rightarrow \mathit{vegetable}.\\ &\mathit{eaten\_by}(\mathit{carnivore}) \Rightarrow \mathit{meat}.\\ &\mathit{eaten\_by}(\mathit{omnivore}) \Rightarrow \mathit{food}. \end{aligned}$$

As a consequence, certain nouns were modified with further dictionary information:

$$\begin{aligned} &\mathit{dictionary}(\mathit{john}, \mathit{pn}(\mathit{number} \Rightarrow \mathit{singular},\\ &\qquad\qquad \mathit{class} \Rightarrow \mathit{human}(\mathit{eating\_habit} \Rightarrow \mathit{omnivore}))).\\ &\mathit{dictionary}(\mathit{monk}, \mathit{n}(\mathit{number} \Rightarrow \mathit{singular},\\ &\qquad\qquad \mathit{class} \Rightarrow \mathit{human}(\mathit{eating\_habit} \Rightarrow \mathit{vegetarian}))). \end{aligned}$$

The semantic functions can take advantage of any of the information available, using higher-order functions and complex type attributes. Arbitrary amounts of computation can be done in order to determine the proper category of the subject or object.

3.2.3 Pragmatics—Anaphora

Not surprisingly, the most difficult aspect of natural language processing to push into the syntactic parser was pragmatics. However, even this was fairly simple to encode in LIFE.

The approach to anaphora and their referents is demonstrative. As a list of words is parsed, each possible referent of an anaphora is pushed onto a list. Whenever an anaphora

is encountered it has to unify with some element of the list. The unification would ensure that all the information known about the anaphora in place matched all the information known about the referent in its place. Thus in '*John runs and he walks*' unifying '*john*' and '*he*' is correct, where in '*John runs and she walks*' unifying '*john*' and '*she*' is not, due to the fact that the pronoun '*he*' has the attribute (*gender* $\Rightarrow$ *male*) and so does '*john*'. Thus '*he*' and '*john*' are unifiable. However, '*she*' has the attribute (*gender* $\Rightarrow$ *female*) which does not unify with '*john*'s attribute (*gender* $\Rightarrow$ *male*).

Even paragraphs such as '*The computer compiled my file. It then deleted it.*' are parsed correctly using this scheme. There are four possible meanings of "It then deleted it" in this context:[6] '*the computer then deleted the computer*', '*the computer then deleted my file*', '*my file then deleted the computer*', and '*my file then deleted my file*'. However, only one of these is semantically coherent. Computers can not be deleted, and files are not animate, and thus can not delete anything. Thus '*The computer compiled my file. The computer then deleted my file.*' is the only reading generated.

Interestingly, even anaphora resolution is performed at the same time as syntactic and semantic checking. As a list of words is parsed, as soon as an anaphora is encountered, its referent is identified before any following words are even looked up in the dictionary.

The result of all this is a parse graph which represents the syntactic, semantic, and pragmatic information corresponding to the input list of words. In the parse graph anaphora and their referents corefer; that is, they point to the same data structure. The parse graph contains the surface structure of the input in much the same way a traditional natural language processing system represents parse trees. Semantic information is represented as additional information on the parse graph. Embedded in the graph are also pointers to the dictionary definitions of words, reduced functional constraints, and complex objects.

4 Conclusion

4.1 Why LIFE?

There are several unique aspects of LIFE which make it suitable for natural language processing. The first and most useful is the logic programming base. Declarative encoding of grammar rules and parsing strategies is extremely elegant and is not inherently inefficient. LIFE's powerful type system allows an even more declarative style with added programming convenience. Addressing LIFE term's by label instead of position allows quick extensions to existing code without significant rewriting. Partially ordered types allow an unprecedented economy of expression, and extremely powerful constructions obtainable in standard logic programming languages only through large amounts of redundant code. Higher-order functions allow one to express extremely complex and powerful semantic notions in simple

[6] In fact, there could be more interpretations of '*it*' than just '*the computer*' or '*my file*'. The act of compiling could be referred to as '*it*', or if any previous sentence had a neuter referent those objects could be referred to as '*it*'. For simplicity, we will only discuss the two obvious meanings.

ways. Complex meaning dependencies can be encoded with little effort.

4.2 Categorial Grammars

LIFE is suitable for the examination of alternative grammars such as originally inspired by Lambek [14]—so called Categorial Grammars. We have in mind, especially, the combination of a unification-based formalism and the categorial parsing paradigm. As we have demonstrated, a unification-based grammar could easily be encoded in LIFE thanks to its native structured type calculus. Higher-order functions being also a basic feature in LIFE, they ought to come as handy to formulate a Categorial Unification Grammars such as proposed by Wittenburg [28].

We are in the process of implementing a Categorial Grammar in LIFE. The lexical category functions on which these grammars are based, with their attendant type raising and function composition, can be easily encoded as higher-order functions in LIFE. In Categorial Grammars, each word has an associated function. For example, in the sentence *'John ate lunch'*, *'john'* has the categorial function 'S/VP' which can be roughly read as "sentence looking for verb phrase". The word *'ate'* has the category 'VP/NP', and *'lunch'* has the category 'NP.' Through function composition and function application, the whole string can be reduced to 'S.' In order to capture syntactic issues such as long distance dependencies (as in the sentence *'What did john eat'*) higher-order functions are necessary. The search strategy using such a grammar could be encoded, as above, using the nondeterministic mechanism of logic programming. In the near future we hope to build a Categorial Unification Grammar following Wittenburg's approach closely [28].

4.3 Limitations of Current System

The current system was constructed in less than two weeks for demonstration purposes. Although improvements have been made, the scope of this project has been limited by time more than technical difficulties. Thus, the user must type perfect English. This sort of fragility makes the system impractical, but this limitation may be surmountable. The most important extension seems to be the addition of some sort of scoring mechanism for failures—if every reading of an input fails, the system should go back and try to find a reading which "almost" succeeded.

Also, certain constructs demand that the semantic checking be turned off, or at least altered. '*I dreamt...*' is one such phrase. '*I dreamt I ate my frisbee*' is an acceptable English sentence. In order to accomplish this some sort of flag needs to be set which determines how much analysis to perform.

Idiomatic phrases are often rejected. '*John threw up*' should be acceptable, but is not, since '*up*' is not a projectile. Some sort of intermediate idiom processing needs to be done to accomplish this.

The dictionary and grammar are fairly small, and thus the language accepted by the system is a very small subset of English. This project was a proof of concept, not construction of

realistic system.

Finally, performance of the system can and must be improved. Using the original implementation (*Still Life*), on a Sun-3, simple sentences take just a few seconds, but large sentences can take over a minute to process. Paragraphs (with ever-lengthening lists of possible anaphora referents, and other deep semantic information) take several minutes to process. On the other hand, with the C version of the LIFE interpreter (*Wild Life*) which is about 50 times faster than *Still Life*, performance becomes quite acceptable. On a fast machine like a DECstation-3100, performance matches that of compiled Common Lisp on a Sun 3.

Even then, independently of which platform is used, there is room for improvement. The most obvious is that we have constructed a natural language interpreter on top of a LIFE interpreter. Furthermore, the grammar is not well tuned, and often searches less frequently successful branches before searching the most often successful. (Although properly called a "best first chart parser" what is implemented is a parser that could perform best first search if we had the empirical evidence from English text about which forms are truly most often successful.) It is our belief that once a LIFE compiler is implemented with, in particular, indexing facilities as in Prolog's [8], the parser will perform in much more reasonable, even competitive time.

In conclusion, with this paper, we hope to have achieved our goal to illustrate the unique adequacy of LIFE's functionality for the specific purpose of Natural Language Processing. The point of our statement is that it is our responsibility as symbolic programming language designers to cater to the needs of high-level applications. Although much work has yet to be done to make the idea a competitive one, we nonetheless have the conviction that LIFE is a natural choice of computer language for the computational linguist.

References

1. Aho, V.A., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA. 1974.

2. Aït-Kaci, H., *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures*. Ph.D. Thesis. Computer and Information Science, University of Pennsylvania. Philadelphia, PA. 1984.

3. Aït-Kaci, H. and Nasr, R., "LOGIN: A Logic Programming Language with Built-in Inheritance." *Journal of Logic Programming* **3**(3), pp. 187–215. 1986.

4. Aït-Kaci, H., "An Algebraic Semantics Approach to the Effective Resolution of Type Equations." *Journal of Theoretical Computer Science* **45**, pp. 293–351. 1986.

5. Aït-Kaci, H., Lincoln, P. and Nasr, R., "Le Fun: Logic, equations, and Functions." *Proceedings of the ACM Symposium on Logic Programming*, pp. 17–23. San Francisco, September 1987.

6. Aït-Kaci, H. and Nasr, R., "Integrating Logic and Functional Programming." *Lisp and Symbolic Computation* **2**, pp. 51–89. 1989.

7. Aït-Kaci, H., Boyer, R., Lincoln, P., and Nasr, R., "Efficient Implementation of Lattice Operations." *ACM Transactions on Programming Languages and Systems* **11**(1), pp. 115–146. January, 1989.

8. Aït-Kaci, H., *The WAM: A (Real) Tutorial*. PRL Research Report No. 5, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France. January 1990.

9. Aït-Kaci, H., and Podelski, A., *The Meaning of LIFE*. Forthcoming PRL Research Report, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France. 1990.

10. Burstall, R., MacQueen, D., and Sanella, D., *Hope: an Experimental Applicative Language*. Technical Report No. CSR-62-80, Department of Computer Science, University of Edinburgh, Edinburgh, UK. May 1980.

11. Colmerauer, A., *et al*, *Prolog-II: Reference Manual and Theoretical Model*. Groupe d'Intelligence Artificielle, Faculté des Sciences d'Aix-Luminy. Marseille, France. 1982.

12. Frisch, A., *Parsing with Restricted Quantification*. Cognitive Science Research Paper No. CSRP-043, School of Social Sciences. University of Sussex, Brighton, UK. February 1985.

13. Gordon, M., Milner, A., Wadsworth, C., *Edinburgh LCF*. LNCS **78**. Springer Verlag, Berlin, FRG. 1979.

14. Lambek, J., "The Mathematics of Sentence Structure," *American Mathematical Monthly* **65**, pp. 154–169. 1958.

15. Landin, P.J., "The Mechanical Evaluation of Expressions." *Computer Journal* **6**(4), pp. 308–320. 1963.

16. Meyer, R., and Aït-Kaci, H., *Wild Life, a User Manual*. PRL Technical Note No. 1, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France. August 1990.

17. Mukai, K., "Anadic Tuples in Prolog," Report draft from lecture presented at the Workshop on Foundations of Deductive Databases and Logic Programming organized by J. Minker. Washington, DC. August 1986.

18. Mukai, K., "A System of Logic Programming for Linguistic Analysis," Report draft from lecture presented at the Workshop on Semantic Issues on Human and Computer Languages organized by J. Barwise, D. Israel, and S. Peters. Half-Moon Bay, CA. March 1987.

19. Partee, B. (Ed.), *Montague Grammars*. Academic Press, New York, NY. 1976.

20. Pereira, F., and Warren, D.H.D.,"Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks," *Artificial Intelligence* **13**, pp. 231–278. 1980.

21. Peyton Jones, S.L., *The Implementation of Functional Programming Languages.* Prentice-Hall. 1987.

22. Smolka, G., and Aït-Kaci, H., "Inheritance Hierarchies: Semantics and Unification." *Journal of Symbolic Computation* **7**, pp. 343–370. 1989.

23. Shieber, S., *An Introduction to Unification-Based Approaches to Grammar.* CSLI Lecture Notes **4**, Center for the Study of Language and Information, Stanford University. 1986.

24. Sterling, L., and Shapiro, E., *The Art of Prolog*. The MIT Press, Cambridge, MA. 1986.

25. Thomason, R. (Ed.), *Formal Philosophy, Selected Papers of Richard Montague.* Yale University Press, New Haven, CT. 1973.

26. Turner, D., "Recursion Equations as a Programming Language," in J. Darlington *et al* (Eds.), *Functional Programming and its Applications*, pp. 1–28. Cambridge University Press, Cambridge, UK. 1982.

27. Turner, D., "Miranda—Non-Strict Functional Language with Polymorphic Types," in J.P. Jouannaud (Ed.), *Proceedings on the Conference on Functional Programming Languages and Computer Architecture (Nancy, France)*. LNCS **201**, pp. 1–16. Springer Verlag, Berlin, FRG. 1985.

28. Wittenburg, K., *Natural Language Parsing with Combinatory Categorial Grammar in a Graph-Unification-Based Formalism.* Ph.D. Thesis, Linguistics, University of Texas. Austin, TX. 1986.