Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research Report
RR-91-32

Towards the Integration of Functions, Relations and Types in an AI Programming Language

Rolf Backofen, Lutz Euler, Günther Görz

August 1991

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Towards the Integration of Functions, Relations and Types in an AI Programming Language

Rolf Backofen, Lutz Euler, Günther Görz

DFKI-RR-91-32

A short version of this report has been published in Heinz Marburger (ed.), Proceedings of the 14th German Workshop on Artificial Intelligence (GWAI-90), Springer, Berlin 1990.

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITW-9002 0).

Towards the Integration of Functions, Relations and Types in an AI Programming Language

Rolf Backofen
DFKI GmbH
Saarbrücken

Lutz Euler
Günther Görz
Universität Hamburg
Fachbereich Informatik — AB NatS

Abstract

This paper describes the design and implementation of the programming language PC-Life. This language integrates the functional and the logic-oriented programming style and feature types supporting inheritance. This combination yields a language particularly suited to knowledge representation, especially for application in computational linguistics.

Keywords: Knowledge representation, AI software, inferences, natural language processing

---

Contents

1 Introduction

Different programming styles have proved to be interesting for AI programming. The most important ones are the functional, the logic-oriented and the object-oriented style. The functional programming style is defined by deterministic computations and first-classness of functional expressions of any order. A logic-oriented language like Prolog contains constructor terms with an unification operation defined on them and uses a resolution-based theorem prover. The object-oriented style allows to specify a hierarchy of classes containing objects. The properties of these objects can be inherited through the hierarchy.

In [4] Aït-Kaci describes the language "LIFE", which was developed as an attempt to integrate these three programming styles into a single language. The most interesting new ideas in his paper are the conception of feature types and the treatment of function evaluation in a logic-oriented programming language.

LIFE has in its type concept the core of a knowledge representation language and its other concepts can serve as blocks out of which to build the remaining part of such a language. In particular it can be applied to computational linguistics. Here we find functional formalisms, like Montague-grammar, logic-based parsing, e. g. in definite clause grammar, and the use of complex types with inheritance for unification-based grammar formalisms and for representation of semantic knowledge. The use of a language integrating all of these has obvious advantages in that the same formalism can be used from syntactic processing up to semantics and pragmatics. Conventional hierarchically organized systems cannot avoid to apply the constraints of these different levels sequentially whereas such a language can account for them simultaneously.

The design of PC-Life[1], which has been developed in Backofen's and Euler's master's theses [6, 9], aims towards the same goals. Our main interest was to explore the difficulties that occur in designing a language that combines the abovementioned programming styles. The aim was definitely not to build a knowledge representation language that could immediately be used in an AI application.

We chose Scheme as an implementation language for its simplicity and versatility in dealing with complex control structures. This leads to some differences to Aït-Kaci's LIFE: Firstly, it was natural to use a Lisp-like syntax and user interface as opposed to LIFE's Prolog-like toplevel. Secondly, the functional part of the language is more like Scheme than like any other "pure" functional language (e. g. ML [14]). This concerns questions of whether functions are of fixed arity, automatic currying is possible, arguments are passed by pattern matching and so on.

More important differences to Aït-Kaci's work in the definition of the language are: PC-Life contains closed types, atoms and atomic types (see below). With respect to disjunctions it has a considerably larger expressive power, because Aït-Kaci's LIFE admits only type disjunctions. With this restriction an appropriate type-as-set semantics cannot be given (see below).

[1]The name is derived from "LIFE" and from the implementation language "PC-Scheme".

The suitability of PC-Life for natural language processing has been demonstrated by implementing a small system for the interpretation of a fragment of German.

This paper describes first the overall design of the language. Then we give a formal description of feature terms together with implementation issues, especially for disjunctive feature terms. At last the design decisions concerning the implementation of functions and relations are detailed.

2 Design of the Language

A program in PC-Life consists of the definition of a type hierarchy and definitions of functions and relations. These can be loaded into an interpreter which then evaluates functional expressions interactively.

The data types of PC-Life consist of the types of the type hierarchy and feature terms. The type hierarchy is a partial order of the elementary types of the Scheme system (`number`, `string`, ...), which are called here "atomic types", arbitrary user-defined types, a least element $\bot$ and a greatest element $\top$. The "values" of the Scheme system (e. g. `42`, `"Deep Thought"`) are called "atomic values" and are also part of the type hierarchy.[2] Atomic types are the only elements of the hierarchy that can semantically[3] be represented as a union of other types. This sort of types is also known as *disjunctive types.*

A feature term can be regarded as an extension of first-order constructor terms with variable arity and fields labeled by name instead of place. A feature term consists of a type entry, which is a type symbol, and any number of attributes or features, which are pairs of an attribute name and an attribute value which again is a feature term. These terms are called subterms of the first feature term. Any subterm (including the outermost term) can be labeled with a variable. Using the same variable at different places expresses a coreference constraint between the corresponding subterms.

At any place where a variable can occur any number of functional expressions may be given too. These may contain references to any variables in the feature term and so express functional constraints between subterms. For a more elaborate description of feature terms see Aït-Kaci [4]. A formal definition of feature terms and their semantics is given in section 4.1.

"Closed types" are a special kind of types. A feature term of such a type may have only attributes whose names are taken from a fixed list defined with the type. So these types are used to model constructor terms of fixed arity. A feature term of atomic type — whose type is an atomic type or an atomic value — cannot contain attributes. It may be seen as a constructor term of arity 0.

A problem occurs with the structured types of the Scheme system, especially with `pair`, the values of which are cons cells, but also with `vector`. It was tempting

[2]Considering only the partial order the distinction between types and values, common in other programming languages, is no longer meaningful.

[3]In a set-theoretic semantics, cf. section 4.1

to use these as built-in constructors and allow any feature term as part of these structures. But this turned out to be impossible because the implementation of any feature term must contain extra information for internal management (e. g. to allow undoing of unification effects in the case of backtracking). So only atomic types and values are allowed as parts of structured Scheme types. If the user needs lists of feature terms, the only solution is to define a closed type `cons` with the attributes `head` and `tail`.

An important extension of the concept of feature terms is the introduction of disjunctions, because they allow to express ambiguous information. To that end at any place where a feature term can occur also a set of feature terms is allowed.

Thus the integration of functions with types provides that functions may have feature terms as arguments and value and that feature terms may contain functional constraints between subterms. To fully benefit of complex types it is necessary to have functions that pass their arguments by pattern matching. Such a function can be applied if all of its actual arguments are subsumed by their corresponding formal parameters. The idea is that the formal parameters contain variables at arbitrary places whose values are then used to build the result of the function. The values are derived by an unification operation on the actual and formal parameters that does not modify the actual parameters.

To explain the integration of relations we begin with ordinary Prolog. Here relations are defined over first order terms. Since the function symbols in these terms are only used as constructors for complex data types and are never applied, the syntax of terms is too much restricted. They can be extended up to feature terms without losing the possibility of using unification and a resolution-based theorem-prover on them. This is described in full detail in [5].

As already mentioned, PC-Life uses a functional top-level. The function `prove` is provided to enable the use of the relational part. It takes a relation application as an argument. Calling `prove` starts a resolution prover on this relation that delivers the solutions one by one.

A second way by which relations may be used is the following: The user specifies the partial order of types in the type hierarchy by entering "<"-relations of types. Additionally it is possible to define a type as being a feature term of another type which further obeys relational restrictions. If a feature term of such a type is used in an unification it must normally be *expanded*, i. e. the definition of the type is unified with its feature term and the relational restrictions are added to the list of goals that remain to be proven.

An important advantage of integration is the treatment of the evaluation of functional expressions that occur inside of feature terms. They must be evaluated when a feature term is defined or unified to check if the functional constraints can be met. Here the problem is that arguments of function applications may be not sufficiently specified to allow evaluation.[4] The solution proposed by Aït-Kaci is what

[4]This problem occurs in simpler form in Prolog when variables on the right hand side of an **is** relation are uninstantiated.

he calls *residuation.* Evaluation is interrupted and delayed until the arguments that caused the break are specified more exactly. This may happen when they are further unified in the course of the resolution process. It is then tested again whether the evaluation can proceed. If disjunctions are used an evaluation may even be restarted several times from the same point but with different values for the arguments.

With respect to residuation we can differentiate between three classes of functions in PC-Life :

1. *System functions,* i. e. functions of the underlying Scheme system. These require that all their arguments are atomic values and residuate on all other feature terms.
2. *Normal functions* accept all values as arguments and pass them using lambda binding. They cannot cause residuation.
3. *Pattern matching* functions pass arguments by pattern matching. They residuate if any argument is not sufficiently specified to decide if it is subsumed by the corresponding pattern.

For a full description of the language see [9].

3 Representation of the Type Hierarchy

It should be clear now that unification and the test for subsumption of feature terms are important operations in PC-Life. These operations require to calculate the infimum (glb) of types or to check for "$\leq$"-relations in the type hierarchy respectively. Straightforward implementations of the latter operations require exponential time (in the size of the type hierarchy). In [3] Aït-Kaci describes a coding approach that allows a much more efficient execution of these operations. The basic idea is to embed the partial order of types into a boolean lattice which is implemented by bitvectors. The above mentioned operations are then implemented as bitwise logical operations. A coding function maps each type onto its bitvector. This function can be precomputed in polynomial time and its value for each type can be stored as the code of this type.

Aït-Kaci describes three related coding methods that preserve existent glbs. We have corrected and implemented the algorithm for "compact encoding". This yields an embedding with the following properties:

- The size of the code bitvectors lies between $\log_2 N$ (where N is the number of types) in case the hierarchy is already a boolean lattice and $N - 1$ in the worst case. The important case of the hierarchy being a binary tree leads to a code size of $N/2$.
- At least one lub is preserved, namely $\top$. Only in the case that the hierarchy is a boolean lattice all lubs are preserved. (This last property must hold for all embeddings that preserve glbs.)

Aït-Kaci says that type disjunction and negation can be implemented with such an encoding. Indeed one can implement a lub operation as bitwise or and negation as bitwise negation. However because of the above mentioned restrictions on using lub operations this leads to an incorrect semantics.

4 Feature terms with distributed disjunctions

4.1 The ψ-term calculus

As already mentioned, feature terms consist of a type entry, features and coreference constraints. A string of features is called a *path* or an *address*. Formally a feature term is a triple $\langle \Delta, \psi, \tau \rangle$ with Δ as the *prefix-closed* set of all addresses, a type function $\psi : \Delta \rightarrow \mathcal{T}$, which assigns a type to each address, and a tag function τ, which associates a variable with each address. A feature term is called *inconsistent* iff its denotation is the empty set in all interpretations.

Feature terms come with a set-theoretic semantics, which is described in detail in Aït-Kaci [2]. A similar system is introduced in Smolka [13] and Nebel and Smolka [12]. Feature terms can be understood as expressions of an attributive representation language that is basically an instance of feature logic. Features are interpreted as partial functions whereas in languages of the KL-ONE family they generalize to roles. It has been shown (mentioned in [12]) that this causes undecidability of subsumption.

Feature terms defined so far allow too much redundancy. E. g., one can get equivalent feature terms by consistent variable renaming. Therefore Aït-Kaci introduced abstract objects, ψ-terms, as representatives for equivalence classes of feature terms which denote the same set of objects in each interpretation.[5] In ψ-terms coreferences are expressed by a coreference relation $\mathcal{K}$. Two addresses are coreferent iff they are assigned the same variable. A ψ-term is therefore a triple $\langle \Delta, \psi, \mathcal{K} \rangle$ with a *right-invariant* coreference relation. A ψ-term must be *referentially consistent* which means that any address of a coreference class of $\mathcal{K}$ carries the same subterm. A ψ-term is consistent iff $\perp \notin \mathbf{Im}(\psi)$.

There is a partial order defined on the set of ψ-terms, the *subsumption order.* A ψ-term t_1 is subsumed by a term t_2 ($t_1 \sqsubseteq t_2$) iff in any interpretation $[\![t_1]\!]^{\mathbf{I}}$ is a subset of $[\![t_2]\!]^{\mathbf{I}}$. The subsumption relation can be calculated easily using the following syntactic conditions:

$$t_1 \sqsubseteq t_2 \iff (\Delta_2 \subseteq \Delta_1) \ \wedge \ (\mathcal{K}_2 \subseteq \mathcal{K}_1) \ \wedge \ \forall a \in \Delta_2 : [\psi_1(a) \leq \psi_2(a)].$$

The basic operation on ψ-terms is *unification.* The unification of two ψ-terms t_1 and t_2 combines the information contained in both terms yielding a term $t = t_1 \sqcap t_2$ with $[\![t]\!]^{\mathbf{I}} = [\![t_1]\!]^{\mathbf{I}} \cap [\![t_2]\!]^{\mathbf{I}}$. Syntactically unification is the process of computing the greatest lower bound (glb) of the terms t_1 and t_2. The most difficult part of this

[5] Because there is a unique translation from feature terms to ψ-terms we don't distinguish them terminologically.

computation is to determine the resulting coreference relation $\mathcal{K}$. The resulting term domain Δ is simply the union of all equivalence classes of $\mathcal{K}$, and the type of an address $a \in \Delta$ is the glb of all types of all addresses in $a/\mathcal{K}$ in both terms.

Because t is the glb of t_1 and t_2, $\mathcal{K}$ must be the smallest coreference relation containing $\mathcal{K}_1$ and $\mathcal{K}_2$. This is the right-invariant completion of

$$\mathcal{K}' = \bigcup_{n \in \mathbb{N}} (\mathcal{K}_1 \circ \mathcal{K}_2)^n$$

Taking the transitive closure $\mathcal{K}'$ of the composition of $\mathcal{K}_1$ and $\mathcal{K}_2$ means to join all equivalence classes of $\mathcal{K}_1$ and $\mathcal{K}_2$ which have an address in common.

ψ-terms are represented as structures built up of nodes. A node is a data structure with three entries: a type entry, a subnode entry which is a list of pairs consisting of features and corresponding values, and a coreference entry. The unification algorithm presented by Aït-Kaci in [5] descends recursively through both ψ-term structures. Nodes with the same address in both structures are merged by dereferencing them to a new node carrying the joined information. Dereferencing uses the coreference entry. The unification fails if $\perp$ results as a type entry for any node.

4.2 Including disjunctions

The unification of terms corresponds semantically to their intersection. To express the union of ψ-terms we introduce *disjunctions* which are sets of ψ-terms. The unification of disjunctions of ψ-terms is done by unifying all possible combinations of ψ-terms of the two disjunctions.

Because of the problem of global coreferences one cannot simply admit disjunctions as feature values. Global coreferences are generated during unification if a coreference in one term involves an address which is not in the scope of the disjunctions actually looked up in the other term. For example, the coreference between l_1 and l_2 is global in the unification of

$$\begin{array}{l} top(l_1 \Rightarrow X \\ \qquad l_2 \Rightarrow X) \end{array} \sqcap \; top(l_1 \Rightarrow \{+;-\})\;. \tag{1}$$

The straightforward result of the unification

$$\begin{array}{l} top(l_1 \Rightarrow X : \{+;-\} \\ \qquad l_2 \Rightarrow X : \{+;-\}) \end{array}$$

is incorrect, since it contains the term $top(l_1 \Rightarrow +\;,\; l_2 \Rightarrow -)$ as one possible extension, which is contradictory to the first unificand (see also Eisele/Dörre [8]). One possible solution is to expand the disjunction to the greatest common prefix of all addresses of an global coreference (cf. [8]). Because we want ψ-terms to be able to share the same subterm this method cannot be employed. Instead we decided to use *distributed disjunctions.* The fundamental idea behind distributed disjunctions is that the problem of global coreferences can be solved by naming disjunctions. So the result of (1) can be calculated straightforwardly, because now both occurring disjunctions carry the same symbol. It must only be guaranteed that in later unifications the same alternative of a named disjunction is chosen wherever the disjunction symbol occurs.

Because disjunctions can be nested, one has not only to remember a pair consisting of a disjunction symbol and the number of the resp. alternative, but a whole set of those pairs. This leads to the notion of *context.* A context *con* is a set of pairs $\langle disj.symbol, alt.number\rangle$ satisfying the following condition: For any disjunction symbol d if there is a pair $\langle d, a\rangle \in con$ then there is no pair $\langle d, b\rangle \in con$ with $a \neq b$. Each node in a disjunctive ψ-term structure has a unique context, which can be defined inductively:

- each subnode of a node n has the same context as n;
- each alternative of a disjunction named with d has a context which is extended by $\langle d, alt.number\rangle$.

We define a partial order on contexts and compatibility of contexts:

- A context con_1 is *smaller* than a context con_2 iff $con_1 \subseteq con_2$.
- con_1 is *compatible* with con_2 iff $con_1 \cup con_2$ is a context.
  iff there is no disjunction symbol d with $\langle d, a\rangle \in con_1$, $\langle d, b\rangle \in con_2$ and $a \neq b$.

The fact that each node has a unique context can be translated into the formal definition of ψ-terms by associating a context to each address in the term domain. A disjunctive term domain is a family of domains $[\Delta_{con}]_{con \in \mathbf{Kon}}$ indexed by contexts, where **Kon** is the set of all possible contexts, and a family of type functions and coreference relations on these term domains. A disjunctive ψ-term, which we call a δ-term, is therefore a triple $\langle[\Delta_{con}], [\psi_{con}], [\mathcal{K}_{con}]\rangle$. The conditions a ψ-term has to satisfy must be slightly modified for δ-terms :

- $[\Delta_{con}]$ must be *weakly prefix-closed*: Every prefix of an address $a \in \Delta_{con}$ must be contained by a term domain $\Delta_{con'}$ with a smaller context con'. This can be motivated using the following example:

  In the term $top(l_1 \Rightarrow \{_{d_1} top\ (l_2 \Rightarrow t_1)\ ;\ t_2\})$ the address $l_1.l_2$ is an element of $\Delta_{\{\langle d_1, 1\rangle\}}$. The prefixes ϵ and l_1 are naturally in a term domain with smaller context.

- $[\mathcal{K}_{con}]$ has to obey *strong right-invariance*: For every context *con* the coreference relation $\mathcal{K}_{con}$ has to be right-invariant and any coreference of $\mathcal{K}_{con}$ has to be continued right-invariantly to all contexts con' with $con \subseteq con'$:

$$[a \in \Delta_{con} \wedge \langle a, b\rangle \in \mathcal{K}_{con}] \Rightarrow$$
$$\forall con', \forall v : [con \subseteq con' \wedge av \in \Delta_{con'} \Rightarrow \langle av, bv\rangle \in \mathcal{K}_{con'}]$$

For example, the coreference $\langle l_1, l_2 \rangle$ in the term

$$\begin{array}{l} top(l_1 \Rightarrow X : \{_{d_1} top(l_3 \Rightarrow t_1) \; ; \; t_2\} \\ \quad\;\; l_2 \Rightarrow X) \end{array}$$

must influence all coreference relations in all other contexts, e. g. $\langle l_1.l_3, l_2.l_3 \rangle \in \mathcal{K}_{\{\langle d_1,1 \rangle\}}$.

- There are additional conditions for $[\Delta_{con}]$ and $[\psi_{con}]$ which have more technical reasons and are left out here for simplicity.

As in the ψ-term calculus, the unification of δ-terms has primarily to compute the resulting family of coreference relations. This again is done by composition of coreference relations of the involved δ-terms. But this time the contexts the relations are indexed by have to be considered, so that a composition sequence has the form

$$\mathcal{K}^1_{con_{i_1}} \Box \mathcal{K}^2_{con_{j_1}} \Box \cdots \Box \mathcal{K}^1_{con_{i_n}} \Box \mathcal{K}^2_{con_{j_n}}$$

As already mentioned, the object of naming disjunctions was to use the same alternative wherever the disjunction symbol occurs. The contexts of coreference relations within any sequence must be pairwise compatible. The sequence itself has a context *con* which is simply the union of all used contexts. A coreference relation $\mathcal{K}_{con}$ of the resulting δ-term is the union of all sequences with context *con*.

There are two kinds of disjunctions that can occur in δ-terms. *Value disjunctions* occur if disjunctions are allowed as feature values. A simple variant of these are disjunctions of atomic values. *Attribute disjunctions* constitute the second kind, as in the term

$$\begin{array}{l} t = top(\quad a \Rightarrow + \\ \qquad\qquad \{_{d_1} \; b \Rightarrow -, \; c \Rightarrow + \; \}). \end{array}$$

Our formalism supports both value and attribute disjunctions. Although the current implementation of the δ-term unification algorithm handles only value disjunctions, it can easily be extended to attribute disjunctions. In this case the management of the set of all defined features for every node — which is required to process closed types — is more complicated.

A δ-term is represented by a structure which is built up of δ-term nodes. A δ-term node is either a ψ-term node or a named disjunction whose list of alternatives internally consists of δ-term nodes. For possibly nested disjunctions the notion of a *disjunction tree* is introduced. An alternative in a disjunction tree is a non-disjunction which can be reached by traversing the tree. The context of an alternative a relative to its root disjunction r is defined as $relcon(a, r) = con(a) \backslash con(r)$. It describes the path from r to the alternative a.

The unification algorithm for δ-terms is an extension of the unification algorithm for ψ-terms (see Aït-Kaci [5]). Again both δ-term structures are passed through all possible paths and all nodes reached under the same address are merged. ψ-term

$$\{_{d_1}\ t_1; t_2\ \} \quad \sqcap \quad \{_{d_2}\ \{_{d_3}\ \{_{d_1} t_1'; t_2'\}\ ;\ t_3'\}\ ;\ t_4'\ \}$$

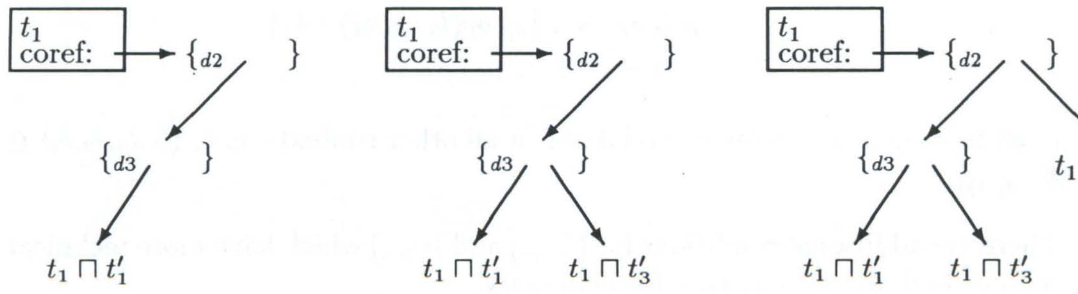

Figure 1: This example shows the successive construction of b_{t_1}. The nodes $t_1 \sqcap t_i'$ are also elements of $b_{ti'}$, e. g. t_1' is bound to $t_1 \sqcap t_1'$ thus completing the merging of t_1 and t_1' in the context $\{\langle d_1, 1\rangle; \langle d_2, 1\rangle; \langle d_3, 1\rangle\}$.

nodes are unified as before. In order to unify two disjunction trees, the algorithm determines all alternatives of the first tree together with their relative context. Every alternative is then unified with all alternatives of the second disjunction tree whose relative contexts are compatible to its relative context. In addition, every alternative a of both disjunction trees is bound to a disjunction tree b_a containing all results of unifications involving a. Therefore every alternative of b_a is an element $a \sqcap a'$ with an appropriate alternative a'. The relative context of $a \sqcap a'$ is $relcon(a \sqcap a', a) = con(a')\backslash con(a)$, such that the equation $con(a \sqcap a') = con(a) \cup con(a')$ holds.

Although contexts are only partially ordered by set inclusion, a disjunction tree defines a total ordering on the relative contexts of its nodes. The unification algorithm processes all alternatives in ascending order. This holds also for every alternative of a binding tree b_a. So we can build up the binding tree successively: at the first unification involving the alternative a b_a is built up to the relative context of $a \sqcap a'$. During all further unifications b_a is always extended at the leaves (see also Fig. 1).

Determining consistency is somewhat more costly for δ-terms than for ψ-term structures. A ψ-term node is inconsistent if its type entry is $\perp$ or if there exists a subterm which is inconsistent. A disjunction is inconsistent if all alternatives are inconsistent. Because disjunctions may be distributed, there may be inconsistencies which cannot be detected locally. For example the following term is inconsistent:

$$\begin{aligned} top(l_1 \Rightarrow\ & \{_{d_1} \perp; \{_{d_2} \perp; t_1\}\} \\ l_2 \Rightarrow\ & \{_{d_2} t_2; \{_{d_1} t_3; \perp\}\}) \end{aligned}$$

Therefore all inconsistent contexts must be stored globally.

A detailed description of the δ-term-calculus is given in [6]. Similar systems using distributed disjunctions can be found in Dörre and Eisele [7] and Maxwell and Kaplan [11].

We see two advantages of our approach compared to Dörre and Eisele's: (1) Their formalism does not treat attribute disjunctions. (2) In our system the unification of disjunctions is defined more abstractly, in that the method of finding appropriate disjunction alternatives is left unspecified, whereas in their system it is not expressed separately but is part of the rewriting rules.

Maxwell and Kaplan provide a general method of extending feature systems into systems with named disjunctions. Each part of the resulting feature structure carries its own context, and rewriting rules used by the original system are translated into a contexted version, which rewrites both the context and this part. Because there is no explicit representation of the relation between a context and its corresponding feature structures, the following inefficiencies arise: During unification of feature structures parts are unified although they carry incompatible contexts, and when components with the same context are rewritten a new context is calculated unneccessarily. Therefore efficient unification algorithms for non-disjunctive feature structures cannot be used.

A remark on negation

The implementation of PC-Life allows only negation of atomic values. For the treatment of negated complex feature terms additionally negation of types, undefined feature entries and inequality constraints[6] are required. The latter will cause problems if used in conjunction with closed types: Because of the interpretation of closed types as constructors two nodes carrying the same closed type are unequal iff they are not dereferenced to the same node *and* if there exists a feature for which the corresponding subnodes are unequal. This transfer of inequality constraints to subnodes possibly has to be iterated if nodes with closed terms are nested, therefore producing a lot of conditions that have to be tested during each unification.

5 The Functional Part: A Variant of Scheme

The implementation of PC-Life in Scheme leads naturally to the use of a modified Scheme as the functional part of the language. So we immediately get the advantages of first-class functions and binding environments, lexical binding within a block structure and full tail recursion optimization. But without modifications Scheme can not handle argument passing by pattern matching and operations with feature terms.

Regarding efficiency it is desirable to use the already existent Scheme evaluator as an evaluator for functional expressions in PC-Life. However this turned out to be very difficult if not impossible, considering the need to enable the interruption of evaluation at any point where a function is applied to insufficiently specified arguments.

[6]which are sometimes also called disagreements or negations of path equivalences.

For this reason we decided to implement a new evaluator whose design follows Abelson/Sussman [1, p. 293ff]. It is necessary to provide the central functions **`eval`** and **`apply`**, the core special forms like **`lambda`** and **`if`**, and to define representations for environments and procedures. All other parts can be implemented by using the existing evaluator. The non-core special forms like **`cond`** and **`let`** in PC-Scheme are defined as macros and are automatically provided by enabling the new evaluator to handle macros. All other functions of the Scheme system are handled as primitives of the new evaluator.

Feature terms are implemented as a new datatype that is checked for in places where residuation may occur.

Functions passing their arguments by pattern matching are defined using the special form `match`. They are treated specially in the evaluator by **`apply`**.

An assignment in a functional expression will lead to incorrect results if this expression is evaluated from inside a disjunctive feature term. This happens because the state of execution that is saved in the case of a residuation contains only the continuation and not the binding environments. Thus in case of multiple execution the same environment will be affected incorrectly. Because of this side-effects must be avoided in all places where a residuation can occur.

6 The Relational Part: A Variant of PROLOG

The relational part of the language is an elaboration of PROLOG in which first order terms are replaced by δ-terms. The role of the logical variable is taken by term-nodes. This means that coreferences do not only occur between different addresses of one term but also within different parts of a clause. The bound/unbound effect of logical variables is replaced by a gradual refinement of nodes.

The resolution prover we implemented is an extension of the one described by Haynes [10]. Its control structure is based on so called "upward-failure-continuations": the theorem prover returns a failure continuation which is invoked when backtracking is necessary. The failure continuations are implemented as Scheme continuations.

The theorem prover works with a structure copying technique. Normally it is necessary to copy the δ-term structure of the clause before unifying it with the argument the relation is invoked with. Our method avoids superfluous copying by simultaneously doing the two steps. This has the advantage that in case the unification fails not the whole recorded structure has been copied.

Using Haynes' taxonomy [10, p. 673] the integration of the relational into the functional part is an *environment embedding.* This means that both share a common environment, therefore providing efficient information transfer. If an embedding additionally allows the sharing of control contexts, it is called *complete.* Although failure continuations which store a specific control context can be obtained at the functional top-level, our embedding is not yet complete, because an arbitrary invocation of failure continuations can violate PROLOG's semantics. But the embedding can be completed by incorporating Haynes' *state-space* model (see [10]).

Type expansion

A type of the type hierarchy can be defined as a δ-term with additional relational constraints. If a δ-term is of such a type, the type has to be *expanded.* This is done by unifying the δ-term with the defined term and evaluating the relational constraints.

In a system with a relational top level like Aït-Kaci's Life, type expansion is easy, because evaluating the relational constraints is done by adding them as additional goals. But with a functional top level this causes problems, because the user can not control the nondeterminism that occurs when expanding a type with relational constraints. This is contradictory to the deterministic behavior of the top level. Therefore we suppress type expansion at the top level.

References

[1] Harold Abelson and Gerald Jay Sussmann. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

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

[3] Hassan Aït-Kaci et al. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

[4] Hassan Aït-Kaci and Patrick Lincoln. LIFE — a natural language for natural language. Technical report, Microelectronics and Computer Technology Corporation, Austin (TX), February 1988.

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

[6] Rolf Backofen. Integration von Funktionen, Relationen und Typen beim Sprachentwurf. Teil II: Attributterme und Relationen. Diplomarbeit, Universität Erlangen-Nürnberg, 1989.

[7] Jochen Dörre and Andreas Eisele. Determining consistency of feature terms with distributed disjunctions. In D[ieter] Metzing, editor, *Proc. of the 13th German Workshop on Artificial Intelligence*, volume 216 of *Informatik Fachberichte*, pages 270–279. Springer, Berlin, 1989.

[8] Andreas Eisele and Jochen Dörre. Unification of disjunctive feature descriptions. In *26th Annual Meeting of the Association for Computational Linguistics*, pages 186–194, Buffalo (NY), 1988.

[9] Lutz Euler. Integration von Funktionen, Relationen und Typen beim Sprachentwurf. Teil I: Konzeption, Typhierarchie und Funktionen. Diplomarbeit, Universität Erlangen-Nürnberg, 1989.

[10] Christopher T. Haynes. Logic continuations. *Journal of Logic Programming*, 4:157–176, 1987.

[11] John Maxwell and Ronald Kaplan. An overview of disjunctive constraint satisfaction. In *Proceedings of the International Parsing Workshop 1989*, pages 18–27, 1989.

[12] Bernhard Nebel and Gert Smolka. Representation and reasoning with attributive descriptions. IWBS-Report 81, IBM Deutschland GmbH, Stuttgart, 1989.

[13] Gert Smolka. A feature logic with subsorts. LILOG-Report 33, IBM Deutschland GmbH, Stuttgart, May 1988.

[14] Åke Wikström. *Functional Programming Using Standard ML.* Prentice Hall, London, 1987.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen oder die aktuelle Liste von erhältlichen Publikationen können bezogen werden von der oben angegebenen Adresse.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of currently available publications can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-90-01
Franz Baader: Terminological Cycles in KL-ONE-based Knowledge Representation Languages
33 pages

RR-90-02
Hans-Jürgen Bürckert: A Resolution Principle for Clauses with Constraints
25 pages

RR-90-03
Andreas Dengel, Nelson M. Mattos: Integration of Document Representation, Processing and Management
18 pages

RR-90-04
Bernhard Hollunder, Werner Nutt: Subsumption Algorithms for Concept Languages
34 pages

RR-90-05
Franz Baader: A Formal Definition for the Expressive Power of Knowledge Representation Languages
22 pages

RR-90-06
Bernhard Hollunder: Hybrid Inferences in KL-ONE-based Knowledge Representation Systems
21 pages

RR-90-07
Elisabeth André, Thomas Rist: Wissensbasierte Informationspräsentation:
Zwei Beiträge zum Fachgespräch Graphik und KI:
1. Ein planbasierter Ansatz zur Synthese illustrierter Dokumente
2. Wissensbasierte Perspektivenwahl für die automatische Erzeugung von 3D-Objektdarstellungen

24 Seiten

RR-90-08
Andreas Dengel: A Step Towards Understanding Paper Documents
25 pages

RR-90-09
Susanne Biundo: Plan Generation Using a Method of Deductive Program Synthesis
17 pages

RR-90-10
Franz Baader, Hans-Jürgen Bürckert, Bernhard Hollunder, Werner Nutt, Jörg H. Siekmann: Concept Logics
26 pages

RR-90-11
Elisabeth André, Thomas Rist: Towards a Plan-Based Synthesis of Illustrated Documents
14 pages

RR-90-12
Harold Boley: Declarative Operations on Nets
43 pages

RR-90-13
Franz Baader: Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles
40 pages

RR-90-14
Franz Schmalhofer, Otto Kühn, Gabriele Schmidt: Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories
20 pages

RR-90-15
Harald Trost: The Application of Two-level Morphology to Non-concatenative German Morphology
13 pages

RR-90-16
Franz Baader, Werner Nutt: Adding Homomorphisms to Commutative/Monoidal Theories, or: How Algebra Can Help in Equational Unification
25 pages

RR-90-17
Stephan Busemann: Generalisierte Phasenstrukturgrammatiken und ihre Verwendung zur maschinellen Sprachverarbeitung
114 Seiten

RR-91-01
Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, Gert Smolka: On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations
20 pages

RR-91-02
Francesco Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, Werner Nutt: The Complexity of Existential Quantification in Concept Languages
22 pages

RR-91-03
B.Hollunder, Franz Baader: Qualifying Number Restrictions in Concept Languages
34 pages

RR-91-04
Harald Trost: X2MORF: A Morphological Component Based on Augmented Two-Level Morphology
19 pages

RR-91-05
Wolfgang Wahlster, Elisabeth André, Winfried Graf, Thomas Rist: Designing Illustrated Texts: How Language Production is Influenced by Graphics Generation.
17 pages

RR-91-06
Elisabeth André, Thomas Rist: Synthesizing Illustrated Documents A Plan-Based Approach
11 pages

RR-91-07
Günter Neumann, Wolfgang Finkler: A Head-Driven Approach to Incremental and Parallel Generation of Syntactic Structures
13 pages

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist: WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

RR-91-09
Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta: RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

RR-91-10
Franz Baader, Philipp Hanschke: A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

RR-91-11
Bernhard Nebel: Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

RR-91-12
J.Mark Gawron, John Nerbonne, Stanley Peters: The Absorption Principle and E-Type Anaphora
33 pages

RR-91-13
Gert Smolka: Residuation and Guarded Rules for Constraint Logic Programming
17 pages

RR-91-14
Peter Breuer, Jürgen Müller: A Two Level Representation for Spatial Relations, Part I
27 pages

RR-91-15
Bernhard Nebel, Gert Smolka: Attributive Description Formalisms ... and the Rest of the World
20 pages

RR-91-16
Stephan Busemann: Using Pattern-Action Rules for the Generation of GPSG Structures from Separate Semantic Representations
18 pages

RR-91-17
Andreas Dengel, Nelson M. Mattos:
The Use of Abstraction Concepts for Representing and Structuring Documents
17 pages

RR-91-18
John Nerbonne, Klaus Netter, Abdel Kader Diagne, Ludwig Dickmann, Judith Klein:
A Diagnostic Tool for German Syntax
20 pages

RR-91-19
Munindar P. Singh: On the Commitments and Precommitments of Limited Agents
15 pages

RR-91-20
Christoph Klauck, Ansgar Bernardi, Ralf Legleitner
FEAT-Rep: Representing Features in CAD/CAM
48 pages

RR-91-21
Klaus Netter: Clause Union and Verb Raising Phenomena in German
38 pages

RR-91-22
Andreas Dengel: Self-Adapting Structuring and Representation of Space
27 pages

RR-91-23
Michael Richter, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Akquisition und Repräsentation von technischem Wissen für Planungsaufgaben im Bereich der Fertigungstechnik
24 Seiten

RR-91-24
Jochen Heinsohn: A Hybrid Approach for Modeling Uncertainty in Terminological Logics
22 pages

RR-91-25
Karin Harbusch, Wolfgang Finkler, Anne Schauder: Incremental Syntax Generation with Tree Adjoining Grammars
16 pages

RR-91-26
M. Bauer, S. Biundo, D. Dengler, M. Hecking, J. Koehler, G. Merziger:
Integrated Plan Generation and Recognition
- A Logic-Based Approach -
17 pages

RR-91-27
A. Bernardi, H. Boley, Ph. Hanschke, K. Hinkelmann, Ch. Klauck, O. Kühn, R. Legleitner, M. Meyer, M. M. Richter, F. Schmalhofer, G. Schmidt, W. Sommer:
ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge
18 pages

RR-91-28
Rolf Backofen, Harald Trost, Hans Uszkoreit:
Linking Typed Feature Formalisms and Terminological Knowledge Representation Languages in Natural Language Front-Ends
11 pages

RR-91-30
Dan Flickinger, John Nerbonne:
Inheritance and Complementation: A Case Study of *Easy* Adjectives and Related Nouns
39pages

RR-91-31
H.-U. Krieger , J. Nerbonne:
Feature-Based Inheritance Networks for Computational Lexicons
11 pages

RR-91-32
Rolf Backofen, Lutz Euler, Günther Görz:
Towards the Integration of Functions, Relations and Types in an AI Programming Language
14 pages

DFKI Technical Memos

TM-90-02
Jay C. Weber: The Myth of Domain-Independent Persistence
18 pages

TM-90-03
Franz Baader, Bernhard Hollunder: KRIS: Knowledge Representation and Inference System -System Description-
15 pages

TM-90-04
Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, Hans-Jürgen Profitlich: Terminological Knowledge Representation: A Proposal for a Terminological Logic
7 pages

TM-91-01
Jana Köhler: Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

TM-91-02
Knut Hinkelmann: Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 pages

TM-91-03
Otto Kühn, Marc Linster, Gabriele Schmidt:
Clamping, COKAM, KADS, and OMOS:
The Construction and Operationalization of a KADS Conceptual Model
20 pages

TM-91-04
Harold Boley:
A sampler of Relational/Functional Definitions
12 pages

TM-91-05
Jay C. Weber, Andreas Dengel , Rainer Bleisinger:
Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

TM-91-06
Johannes Stein: Aspects of Cooperating Agents
22 pages

TM-91-08
Munindar P. Singh: Social and Psychological Commitments in Multiagent Systems
11 pages

TM-91-09
Munindar P. Singh: On the Semantics of Protocols Among Distributed Intelligent Agents
18 pages

TM-91-10
Béla Buschauer, Peter Poller, Anne Schauder, Karin Harbusch: Tree Adjoining Grammars mit Unifikation
149 pages

TM-91-11
Peter Wazinski: Generating Spatial Descriptions for Cross-modal References
21 pages

DFKI Documents

D-90-03
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Abschlußbericht des Arbeitspaketes PROD
36 Seiten

D-90-04
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: STEP: Überblick über eine zukünftige Schnittstelle zum Produktdatenaustausch
69 Seiten

D-90-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Formalismus zur Repräsentation von Geo-metrie- und Technologieinformationen als Teil eines Wissensbasierten Produktmodells
66 Seiten

D-90-06
Andreas Becker: The Window Tool Kit
66 Seiten

D-91-01
Werner Stein , Michael Sintek: Relfun/X - An Experimental Prolog Implementation of Relfun
48 pages

D-91-03
Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause: RFM Manual: Compiling RELFUN into the Relational/Functional Machine
43 pages

D-91-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1990
93 Seiten

D-91-06
Gerd Kamp: Entwurf, vergleichende Beschreibung und Integration eines Arbeitsplanerstellungssystems für Drehteile
130 Seiten

D-91-07
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
TEC-REP: Repräsentation von Geometrie- und Technologieinformationen
70 Seiten

D-91-08
Thomas Krause: Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN
137 pages

D-91-09
David Powers and Lary Reeker (Eds):
Proceedings MLNLO'91 - Machine Learning of Natural Language and Ontology
211 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-10
Donald R. Steiner, Jürgen Müller (Eds.):
MAAMAW'91: Pre-Proceedings of the 3rd European Workshop on „Modeling Autonomous Agents and Multi-Agent Worlds"
246 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-11
*Thilo C. Horstmann:*Distributed Truth Maintenance
61 pages

D-91-12
Bernd Bachmann:
$^{\text{Hiera}}\text{C}_{\text{on}}$ - a Knowledge Representation System with Typed Hierarchies and Constraints
75 pages

D-91-13
International Workshop on Terminological Logics
Organizers: Bernhard Nebel, Christof Peltason, Kai von Luck
131 pages

D-91-14
Erich Achilles, Bernhard Hollunder, Armin Laux, Jörg-Peter Mohren: *KRIS* : *K*nowledge *R*epresentation and *I*nference *S*ystem
- Benutzerhandbuch -
28 Seiten

Towards the Integration of Functions, Relations and Types in an AI Programming Language

Rolf Backofen, Lutz Euler, Günther Görz

RR-91-32
Research Report