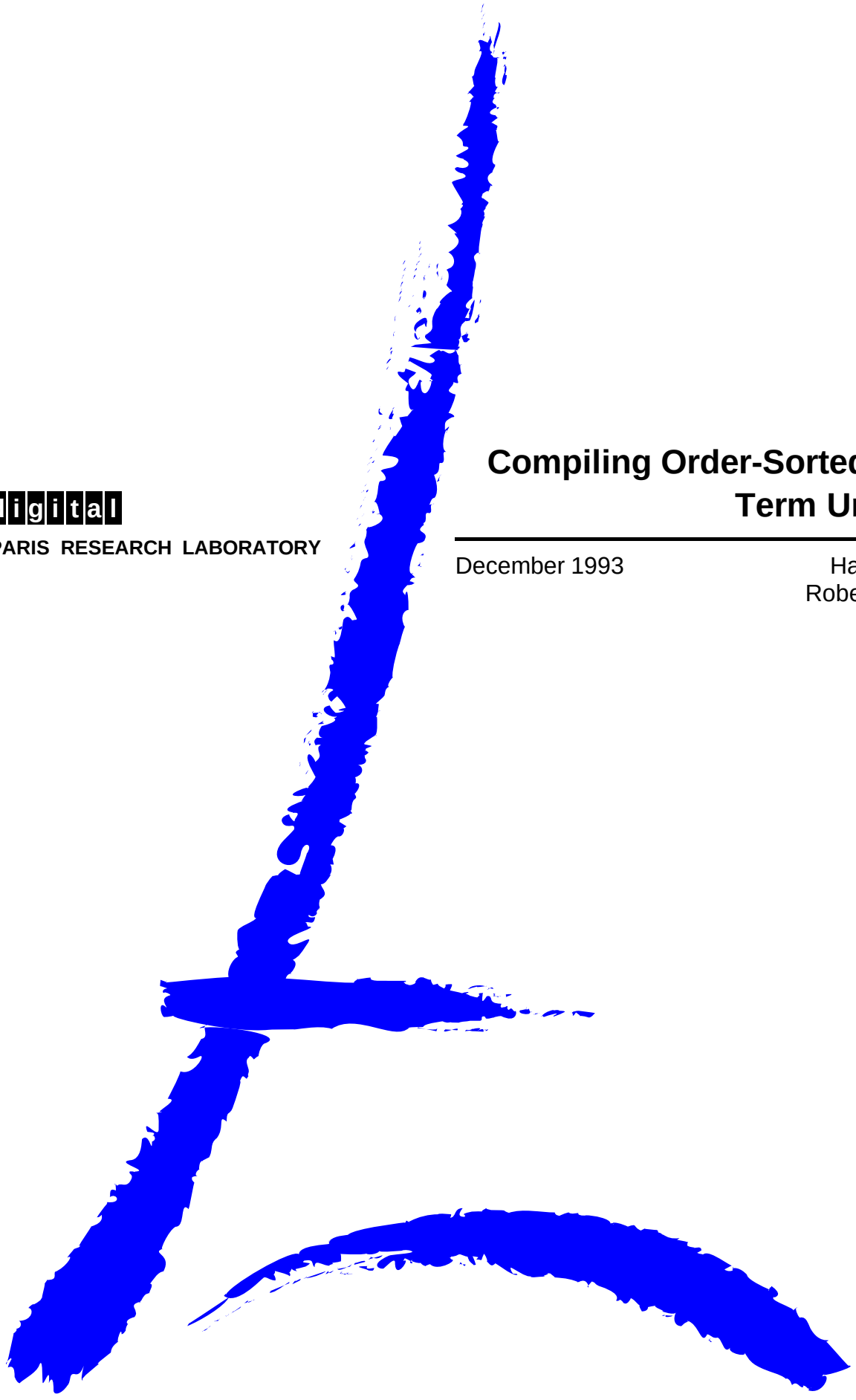
7

Compiling Order-Sorted Feature Term Unification

digital
PARIS RESEARCH LABORATORY

December 1993

Hassan Aït-Kaci
Roberto Di Cosmo



PRL TECHNICAL NOTE

7

Compiling Order-Sorted Feature Term Unification

Hassan Aït-Kaci
Roberto Di Cosmo

December 1993

Publication Notes

The work reported, belatedly, in this technical note is the result of the authors' collaboration from September 1992 until December 1992, while Roberto Di Cosmo was visiting PRL from ENS to work as a student intern under the supervision of Hassan Aït-Kaci.

Current contact addresses of authors:

Hassan Aït-Kaci
hak@cs.sfu.ca

School of Computing Science
Simon Fraser University
Burnaby, British Columbia
V5A 1S6, Canada

Roberto Di Cosmo
dicosmo@ens.ens.fr

École Normale Supérieure
Laboratoire d'Informatique
45 rue d'Ulm
75005 Paris, France

Abstract

Order-sorted feature (OSF) terms generalize first-order rational terms: functors become partially ordered sorts, arity is unconstrained, and subterms are unordered, indicated by explicit feature symbols rather than implicit positions. Thus, OSF terms provide a handy data structure to represent objects in symbolic programming languages. LIFE is such an experimental language extending Prolog by replacing its term data structure and unification operation with OSF term and unification. In this paper, we present an abstract machine design for OSF term unification. This compiling scheme consists of an alteration of recent refinements of WAM technology for compiling Prolog's unification. Our modifications account for order-sortedness, symbolic features, and absence of arity. Then, we improve that design by incorporating several immediate optimizations.

Résumé

Les termes à traits et à sortes ordonnées généralisent les termes rationnels de premier ordre : les symboles fonctionnels deviennent des sortes partiellement ordonnées. Ils n'ont pas de contrainte d'arité, et leurs sous-termes sont dans le désordre, indiqués par des symboles de traits explicites plutot que par des positions implicites. Ainsi, ces termes offrent une structure de données adéquate pour représenter des objets dans des langages de programmation symbolique. LIFE est un tel langage, étendant Prolog en remplaçant sa structure de données de terme et son opération d'unification par les termes à traits et sortes ordonnées et leur unification. Dans ce papier, nous présentons une maquette de machine abstraite pour l'unification de ces termes. Ce schéma de compilation consiste en une altération de la technologie WAM pour compiler l'unification de Prolog. Nos modifications prennent en compte l'ordre sur les sortes, les traits symboliques, et l'absence d'arité. Puis, nous améliorons cette maquette en y incorporant plusieurs optimisations immédiates.

Keywords

Unification, Prolog, LIFE, Warren’s abstract machine, Feature structures.

Contents

> On the other hand, if it is true that regions gradually merge into one another, and this remains to be proved, then I may well have left mine many times, thinking I was still within it.
>
> SAMUEL BECKETT, *Molloy*

This document describes an abstract instruction set for compiling order-sorted feature (OSF) term unification [4] as it is used in the LIFE language [2]. This set of instructions is being used as the basis for the compiler of LIFE currently under development at PRL [8].

For an introduction to OSF terminology and the unification rules, the reader is referred to [4, 3].

1 Heap representation

We consider a simple language consisting of two kinds of entities, both OSF terms: a *query* and a *program*. The operational semantics of this language amounts to unifying the query term with the program term. As in the case for Prolog, instructions compiling the query will build a representation for it in a memory area (called HEAP). However, the heap representation of OSF terms, will be different from that used in the WAM, because we will need to take into account the following additional elements that are specific of OSF terms:

- there is no difference between a *structure* and a *variable* (in particular each node may be dereferenced);
- every node in an OSF term has a sort information in lieu of the *functor* information of Prolog terms, and this information *can be modified* during unification;
- the arity of functors is no longer fixed, and subterms are accessed by feature name, and not just by position, as for the argument of a Prolog structure.

A consequence of the first and second points is that the distinction between structure and variable made by the WAM heap representation is no longer appropriate: variables are just sorted coreference tags virtually attached to *any* subterm of a term and no longer restricted as leaves only. In particular, variables such as Prolog's are implicitly sorted with $\top$, the greatest sort. Also, order-sorted unification can refine structures; for example, if the sort ordering is such that $zero < zeropos$ and $zero < zeroneg$, them the terms *zeropos* and *zeroneg* can be unified, refining their sorts to *zero*. This means that in our heap representation, everything must be encoded uniformly.

The third point confronts us with the only real difficulty: during OSF unification, subterms identified by the same feature name must be unified. The problem is to perform this efficiently. This issue does not arise in Prolog because, there, subterms come with an associated natural order: the *n*-th subterm is the one coming in position *n*, so unification of two terms proceeds by structural decomposition: $foo(first(a), second(int))$ and $foo(second(int), first(a))$ are not unifiable.

For OSF terms, on the contrary, subterms are identified by features names. So, for example, the two OSF terms $foo(first \Rightarrow a, second \Rightarrow int)$ and $foo(second \Rightarrow int, first \Rightarrow a)$ are not only unifiable, but actually identical. Thus, subterms must be unified not in the order they come, but as indicated by common feature names. What is worse, arity of OSF terms being unconstrained, it can vary dynamically through the unification process. So it is not possible to use at compile time, as the WAM does for Prolog, a fixed order for the feature names.

This fact causes a major departure from the WAM heap term representation. It is necessary to abandon the convention of having the references to the *n* arguments of a functor stored exactly after it in *n* consecutive locations. What is needed is an alternative representation of the set of featured subterms that enjoys the following properties:

- fast associative access to, whether retrieval or insertion of, subterms via feature names;
- fast iteration over a set of feature names and their associated reference subterms;
- eventually for backtracking purposes, a small overhead for trailing operations and undoing them.

One immediate drawback of associative access to subterms is that it is slower (even if not by much) than in the WAM where access is direct thanks to fixed arity constraints.[1] On the other hand, the decoupling between the functor representation and the argument representation will allow us to adopt more flexible code generation schemes. Hence, for simplicity and generality, we will use abstract operations dealing with associative feature access to subterms.

As an illustration of our OSF term representation, Figure 1 shows a heap representation of the OSF term:

$$\begin{array}{l} X : person(name \Rightarrow id(first \Rightarrow string, \\ \qquad\qquad last \Rightarrow Y : string), \\ \quad spouse \Rightarrow person(name \Rightarrow id(last \Rightarrow Y), \\ \qquad\qquad spouse \Rightarrow X)). \end{array} \tag{1}$$

This representation is explained as follows. An OSF term is essentially a labeled sorted graph: the nodes contain sort and structure-sharing information and the edges are labeled with feature names. This justifies separating the conventional single heap area in memory into two: HEAP and FHEAP. The area HEAP is where nodes are stored and FHEAP contains tables associating feature names to nodes. Therefore a HEAP cell consists of three fields:

- CREF: the coreference field, an index into HEAP. This determines whether this term is unbound or bound to another. If it is unbound, the value of this field is the index of its own HEAP cell.

[1] However, many optimizations are possible, that will not be discussed here, whereby global analysis, explicit pragmas, together with clever indexing techniques such as perfect hashing and/or cached feature access can provide for Prolog terms to be compiled virtually as efficiently as in Prolog [8].

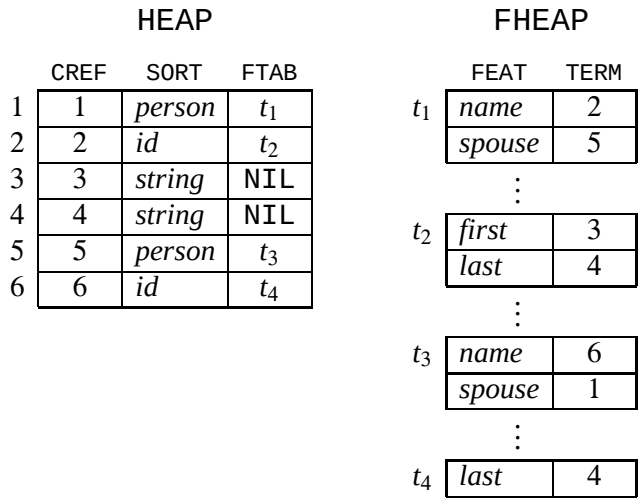


Figure 1 Heap representation of OSF term (1)

- SORT: the sort field, a (representation of) the sort symbol of the root of this term.
- FTAB: the feature table field, an index into FHEAP containing the association table between feature symbols and the node address in HEAP of the subterms. If there are no subterms, this field is set to NIL.

Similarly, the feature heap FHEAP consists of tables whose entries are cells made out of two fields:

- FEAT: the feature field.
- TERM: the term field. This is a index into HEAP.

In the following sections we will first see how to generate efficient code to build such representations and to perform unification on them, and then we will discuss some simple immediate improvements.

2 Compiling query terms

Compiling a query term consists of generating a sequence of instructions that, when executed, will build its representation on the heap, ready to be used later for unification with a program term.

As in Prolog, we start by flattening an OSF term into a list of simple tokens. Then, we choose an appropriate ordering for the tokens from which well-founded code may be generated. The tokens will no longer refer to the user level names of the variables, but only to internal variables

known as *registers*. Those will play exactly the same role as the WAM's so-called *temporary* registers and similarly denoted as X_1, X_2, *etc.* As in the WAM, X_1 is always used for the root of the outermost term.

The tokens making up an OSF term's flattened form are of two kinds:

- $\mathsf{X}_i : s$ (X_i has sort s);
- $\mathsf{X}_i.\ell = \mathsf{X}_j$ (X_i has feature ℓ and it corresponds to X_j).

For example, the flattened form of OSF term (1) is given by the following sequence of tokens:

$$\mathsf{X}_1 : \mathit{person},\ \mathsf{X}_1.\mathit{name} = \mathsf{X}_2,\ \mathsf{X}_2 : \mathit{id},\ \mathsf{X}_2.\mathit{first} = \mathsf{X}_3,\ \mathsf{X}_3 : \mathit{string},$$
$$\mathsf{X}_2.\mathit{last} = \mathsf{X}_4,\ \mathsf{X}_4 : \mathit{string},\ \mathsf{X}_1.\mathit{spouse} = \mathsf{X}_5,\ \mathsf{X}_5 : \mathit{person},\ \mathsf{X}_5.\mathit{name} = \mathsf{X}_6,$$
$$\mathsf{X}_6 : \mathit{id},\ \mathsf{X}_6.\mathit{last} = \mathsf{X}_4,\ \mathsf{X}_5.\mathit{spouse} = \mathsf{X}_1.$$

If one follows the WAM approach, each token yields an abstract machine instruction building the corresponding piece of term on the heap. Namely:

- `set_cell` X_i, s
  This corresponds to the token $\mathsf{X}_i : s$ and has for effect to create a cell on the heap, set register X_i to its address, and set the sort field of this newly created cell to s.
- `set_feature` $\mathsf{X}_i, \ell, \mathsf{X}_j$
  This corresponds to the token $\mathsf{X}_i.\ell = \mathsf{X}_j$ and has for effect to create an entry in the feature table of X_i for feature ℓ and set it to point to the address contained in register X_j.

Note that `set_cell` X_i, s corresponds in fact to the combination of two WAM instructions: `put_structure` X_i, s and `set_variable` X_i.[2] This is due to the fact that structures and variables are now just one same thing. Similarly, `set_feature` $\mathsf{X}_i, \ell, \mathsf{X}_j$ corresponds to `set_value` X_j for the WAM, but with the need to declare explicitly *which* feature of *which* register we are setting, since the contiguous positional convention of WAM term representation is no longer sound.

However, it is important to recognize that the `set_cell` X_i, s instruction is actually built out of two simpler instructions, which can used to produce better code. The `set_cell` X_i, s instruction really does two different things:

- `push_cell` X_i: it pushes on the heap a new cell and reserves it to register X_i
- `set_sort` X_i, s: it initializes the sort field of this new cell to s.

Now, if we keep the `set_cell` X_i, s instruction, this means that new heap cells are generated only in correspondence with a token $\mathsf{X}_i : s$, while if we decompose it into `push_cell` X_i and

[2] Strictly speaking, using original WAM instructions, we should say `unify_variable` X_i in `write` mode, rather than `set_variable` X_i. However, we follow [1] using `set` instructions for queries rather than `unify` instructions with a systematic useless `write` mode test as done in [9].

`set_sort` X_i, s, we have the possibility to create new cells at a different time.

This is not only more natural, it simplifies code generation as no particular order must be imposed on the token sequence. Moreover, this gives the additional freedom to commute instructions, a technique that is often yielding relevant optimizations---we shall see later some examples.

Our abstract instructions will be generated as follows from the token sequence:

- `push_cell` X_i
  This corresponds to a first seen X_i in a token (either an $\mathsf{X}_i : s$ or a $\mathsf{X}_i.\ell = \mathsf{X}_j$ or a $\mathsf{X}_j.\ell = \mathsf{X}_i$) and has for effect to push on the heap a new cell and stored its address in X_i. It is generated before any other instruction coming from the same token.
- `set_sort` X_i, s
  This corresponds to the token X_i and has for effect to set the sort field of the cell pointed to by X_i to s.
- `set_feature` $\mathsf{X}_i, \ell, \mathsf{X}_j$
  This corresponds to the token $\mathsf{X}_i.\ell = \mathsf{X}_j$ and has for effect to create an entry in the feature table of X_i for feature ℓ and set it to point to the address contained in register X_j.

The main advantage of this code generation schema is that it will produce correct code *irrespective of the order of the tokens*: indeed, it ensures that a `set_feature` $\mathsf{X}_i, \ell, \mathsf{X}_j$ istruction will be generated only after a `push_cell` X_i and a `push_cell` X_j have been generated, and similarly for `set_sort` X_i, s instructions. Furthermore, this advantage is obtained at no cost: the operations performed by a `push_cell` X_i and a `set_sort` X_i, s instructions amount exactly to the same work done by one `set_cell` X_i, s instruction. We are now free to choose the token ordering that best suits our compilation needs.

Let us now consider the advantages of this compilation schema on the simple example of the OSF term $X : \mathit{loop}\big(a(X)\big)$. Its flattened form is:

$$\mathsf{X}_1 : \mathit{loop},\ \mathsf{X}_1.1 = \mathsf{X}_2,\ \mathsf{X}_2 : a,\ \mathsf{X}_2.1 = \mathsf{X}_1.$$

Let us suppose that we want to follow the WAM compilation schema: it is clear that we must first generate the code to build the subterms of X_1, and then the code to build X_1. But doing so, we are forced to generate the code for X_2 first, which requires in turn that X_1 be already available. So we are apparently stuck here: it seems impossible to generate the code for a cyclic OSF term in one pass.

Suppose now that we adopt the decoupling of representation of nodes and edges and we take the `set_cell` X_i, s instruction (and not its components `push_cell` X_i and `set_sort` X_i, s) as primitive: it is then possible to generate the code in one single pass. Indeed, when features (arguments) of a sort (functor) are no longer required to be contiguous following that sort in the heap, one can reorganize the code generation for an OSF term of the form $X : s\big(t_1, \ldots, t_n\big)$ as follows:

```
set_cell X1, loop        %  X1 : loop,
set_cell X2, a           %  X2 : a,
set_feature X2, 1, X1    %  X2.1 = X1,
set_feature X1, 1, X2    %  X1.1 = X2.
```

Figure 2 Code with `set_cell` for query OSF term $X : loop(a(X))$

```
push_cell X1             %  X1 : loop,
set_sort X1, loop        %  X1 : loop,
push_cell X2             %  X2 : a,
set_feature X2, 1, X1    %  X2.1 = X1,
set_sort X2, a           %  X2 : a,
set_feature X1, 1, X2    %  X1.1 = X2.
```

Figure 3 Code with `push_cell` and `set_sort` for query OSF term $X : loop(a(X))$

- first, build the structure for $X : s$ at address H in the heap and save it in a register X_i;[3]
- then, build the representation of the subterms t_k, already knowing the address X_i of $X : s$ (which can be used even with cyclic references);
- finally, build the feature table of $X : s$, already knowing the addresses of each of the subterms.

Hence, it is necessary to use the flattened token sequence in a specific order. This order must generate a `set_feature` $\mathsf{X}_i, \ell, \mathsf{X}_j$ instruction only *after* the pair of instructions `set_sort` X_i, s_i and `set_sort` X_j, s_j have been generated. For instance, the tokens for query in our previous example must be reordered thus:

$$\mathsf{X}_1 : loop,\ \mathsf{X}_2 : a,\ \mathsf{X}_2.1 = \mathsf{X}_1,\ \mathsf{X}_1.1 = \mathsf{X}_2$$

and the corresponding abstract machine code for it is then shown in Figure 2.

Finally, if we take `push_cell` X_i and `set_sort` X_i, s as primitive instructions, the code generation proceeds flawlessly without any need to reorder the token stream. For instance, the code generated for the original token stream of the previous example is shown in Figure 3, and the code for the reordered token stream is shown in Figure 4: they are both valid and they both build the same structure on the heap. Notice that this last abstract instruction sequence will be translated in the very same pseudo--code as the sequence in Figure 2.

[3] As in the WAM, we shall use a global register H to indicate the first free cell in HEAP. Similarly, a global register FH will indicate the first free address in FHEAP.

```
push_cell X1            %  X1 : loop,
set_sort X1, loop       %  X1 : loop,
push_cell X2            %  X2 : a,
set_sort X2, a          %  X2 : a,
set_feature X2, 1, X1   %  X2.1 = X1,
set_feature X1, 1, X2   %  X1.1 = X2.
```

Figure 4 Code for query OSF term $X : loop(a(X))$

Going back to our OSF term 1, its flattened form can be directly translated into the abstract machine code shown in Figure 5.

The effect of these instruction is described in Figure 6, so that, when executed, the code of Figure 5 will build the heap representation shown in Figure 1. The *add_feature* operation (Figure 7) installs a new feature in a table, allocating a new table if necessary.

3 Compiling program terms

Now that we know how to compile an OSF term query into executable code that will build its correct representation in memory, we can tackle the problem of generating code for a *program*; *i.e.*, another OSF term to be unified with the query already built. Let us recall briefly the instructions used in the WAM to compile a program term. For simplicity, we will just focus on the general purpose instructions `get_structure` $f/n, \mathtt{X}_i$, `unify_variable` $\mathtt{X}_i$ and `unify_value` $\mathtt{X}_i$, and not worry about their specialized versions for constants and lists.

These basic instructions can work essentially in two modes:

- `read` mode---this is the default mode, in which the instructions check that the term on the heap matches the program term;
- `write` mode---this is the mode in which the instructions build on the heap a copy of (a portion of) the program term, binding a variable in the query term.

Taking advantage of the sequential and contiguous representation of the argument of a functor in memory, these instructions use a specialized register S (next subterm) together with a global mode flag to switch back and forth from one mode to the other. The `get_structure` $f/n, \mathtt{X}_i$ instruction is in charge of this switching: it *always* checks if the memory location indicated by $\mathtt{X}_i$ contains a structure or a variable. If it finds a structure, it sets the mode flag to `read` and initializes S to the first subterm in the heap. The instructions following it in sequence, one `unify` instruction per subterm in the order they come, will then unify what S points to in the heap with the portion of the program term encoded by the instruction, and then increment S. If, on the other hand, `get_structure` $f/n, \mathtt{X}_i$ finds a variable on the heap, it switches the

```
push_cell X1                % X1 : person
set_sort X1, person         % X1 : person
push_cell X2                % X2 : id
set_feature X1, name, X2    % X1.name = X2
set_sort X2, id             % X2 : id
push_cell X3                % X3 : string
set_feature X2, first, X3   % X2.first = X3
set_sort X3, string         % X3 : string
push_cell X4                % X4 : string
set_feature X2, last, X4    % X2.last = X4
set_sort X4, string         % X4 : string
push_cell X5                % X5 : person
set_feature X1, spouse, X5  % X1.spouse = X5
set_sort X5, person         % X5 : person
push_cell X6                % X6 : id
set_feature X5, name, X6    % X5.name = X6
set_sort X6, id             % X6 : id
set_feature X6, last, X4    % X6.last = X4
set_feature X5, spouse, X1  % X5.spouse = X1.
```

Figure 5 Code for query OSF term (1)

```
push_cell Xi          ≡ HEAP[H].CREF ← H;
                        HEAP[H].FTAB ← NIL;
                        Xi ← H;
                        increment(H)

set_sort Xi, s        ≡ HEAP[Xi].SORT ← s;

set_feature Xi, ℓ, Xj ≡ add_feature(HEAP[Xi].FTAB, ℓ, Xj)
```

Figure 6 Instructions for query terms

```
procedure add_feature(table : address, feature : symbol, value : address);
  begin
    if table = NIL
      then
        begin
          FHEAP[FH] ← new_table;
          table ← FH;
          increment(FH)
        end;
    put_feature(table, feature, value)
  end add_feature;
```

Figure 7 The *add_feature* operation

mode flag to `write`, and the same `unify` instructions will then build on the heap the portion of the program term that must occur at that corresponding position (*i.e.*, behave exactly as for a query term). Hence, these `unify` instructions test the mode flag to see if they have to match or build their bit of information on the heap.

It is clear that for this to work, two conventions must be obeyed: (1) subterm cells must follow contiguously a functor cell on the heap and in the right order; and (2) exactly one `unify` instruction per subterm must be generated, contiguously and in the same order. Neither of these two conventions are observed in our representation and code generation. Nevertheless, it would not be difficult to adapt the basic idea to our setting, although at a cost.

Recently, another scheme for compiling unification was devised which does not rely on contiguity conventions, and has the advantage of being more efficient than the original contiguity scheme. The gist of this method relies on the fact, as described in [7],[4] that the original WAM contiguity scheme generates redundant work and, therefore, inefficiency. Indeed, *every* execution of a `get_structure` $f/n, \mathtt{X}_i$ performs a complex test to determine if the machine must proceed in `read` or `write` mode. Namely, it *always* dereferences $\mathtt{X}_i$ and tests whether it is a variable or a structure, no matter where and when it is executed. As a result, this means that if the machine enters `write` mode to build a large term on the heap, it will continually and uselessly test the mode flag and dereference newly built variables just to discover that it has to keep on writing.

To avoid this unnecessary overhead an optimization can remove redundant mode tests from the instructions. Rather than processing terms *breadth-first* as done by the WAM contiguity conventions, it is advantageous to proceed through a term *depth-first*. In this way, the mode flag need not be reset for a term once set to `write` at its root. Two streams of instructions, a

[4] This method was in fact originally discovered as early as 1988 by Mohamed Amraoui, a doctoral student at Rennes, France [5]. Amraoui's idea, since unpublished, seems to have gone unnoticed until it was apparently rediscovered independently by Micha Meier [7] and later optimized further by André Mariën and Bart Demoen [6]. It is the latter scheme that we shall adapt to our layout.

`write_test` *Level*, *Label* $\equiv$ **if** `D` $\geq$ *Level* **then goto** *Label*

Figure 8 Conditional jump out of `write` mode

`read` stream and a `write` stream are thus generated, each adequately peppered by conditional jumps from one to the other depending upon whether the term since the last jump into the stream has been completely processed. A consequence of this is that different instructions must be used for different (`read` or `write`) modes. This entails using more, albeit lighter-weight, instructions. Moreover, these can further be arranged in such a way as to reduce tests for mode switching to the minimum necessary. This approach can be smoothly adapted to our case, and we will follow it in our design of the abstract machine.

A program term will be compiled into two sequences of instructions: a `write` sequence consisting of exactly the same instructions used to compile a query term, and a `read` sequence consisting of instruction to perform appropriate unification with the heap. The problem posed by the depth-first organization of instructions is to insert tests and jumps properly in each sequence. Clearly, these must be placed at the boundaries marking the code for each subterm. They are logically where it is necessary to decide whether to carry on in the current mode (`read` or `write`), or jump to the other stream.

Micha Meier's instructions keep track of term boundaries using a rather complicated scheme. André Mariën and Bart Demoen proposed a simpler and better means to do the same. Their idea uses information given by the *nesting depth* of a term inside the outermost term being compiled. A global register `D` will be used to indicate, at execution time, the least depth encountered since the last stream jump. Thus, instructions ''recognize'' term boundaries by a simple comparison test of their own (statically known) depth and the depth register `D`.

3.1 The `write` sequence

The `write` sequence is the simpler of the two; and this, for two reasons. Firstly, its writing instructions are exactly the same as the code produced for compiling a query. Secondly, once `write` mode is set, for a subterm this holds for all subterms underneath it. The idea is that execution of a program term starts in `read` mode. Whenever, at a given depth, it is found there that a subterm must be built, the register `D` is set to that depth and control jumps to the `write` subsequence corresponding to that term. Control will know when to return to the `read` sequence as soon as it reaches `write` instructions pertaining to a lesser level than the one indicated by `D`. Therefore, the appropriate place to test for a jump out of a `write` sequence is immediately after the last instruction corresponding to a subterm. This is achieved by placing the instruction `write_test` *Level*, *Label* shown in Figure 8 at the end of the code sequence for each subterm of depth *Level*. The label *Label* is the address of the first instruction in the code of the `read` stream which corresponds to the next appropriate subterm where to resume in `read` mode. Namely, that of the next subterm of depth lower or equal to that of the

subterm just written.

3.2 The `read` sequence

Now, regarding the `read` sequence of a program term, its instructions will need to perform the following operations on the representation on the term built on the heap, depending on the information found on the heap:

- refine the sort of a HEAP cell;
- access a feature in the feature table of a HEAP cell; or,
- perform unification on two terms.

In addition, it will need to test when and where to jump to the `write` sequence. Jumping out of the `read` sequence to the `write` sequence must be done whenever it is found that a subterm must be built. Therefore these jump tests must be carried out *before* entering the code sequence corresponding to a subterm.

To understand the instructions that perform this, it is important to remember that the Prolog term's distinction between a variable and a structure is no longer appropriate for OSF terms. Here, a term is made of uniform structures, all built of nodes (the cells in HEAP) and edges (the tables in FHEAP). So, while Prolog code for the program must build a subterm on the heap if the corresponding HEAP cell is a variable, in the case OSF terms, the program must enter `write` mode only if the feature leading to that subterm is undefined. This is indeed the only time where a new structure must be created.

Now, testing the existence of a feature is as expensive as accessing it. Therefore, there is no point in using a single abstract instruction to perform the test: it is more effective to embed such test in all the instructions that access a feature.

Under these considerations, the instructions used by an OSF program term in `read` mode are the following:

- `intersect_sort` X_i, s
- `test_feature` $\mathsf{X}_i, \ell, \mathsf{X}_j, \mathit{Level}, \mathit{Label}$
- `unify_feature` $\mathsf{X}_i, \ell, \mathsf{X}_j$

The `intersect_sort` X_i, s instruction attempts to refine the sort of the HEAP cell indicated by X_i by intersecting it with s.

The `test_feature` $\mathsf{X}_i, \ell, \mathsf{X}_j, \mathit{Level}, \mathit{Label}$ instruction checks whether the HEAP cell indicated by X_i has feature ℓ. If it does, then X_j is set to point to its value. Otherwise, it causes a jump to the `write` sequence at the appropriate *Label* and with the appropriate *Level* to build the corresponding subterm (*i.e.*, always a `set_sort` X_j, s). Therefore, when the write sequence terminates, X_j will point to the root of this newly built subterm.

The `unify_feature` $\mathsf{X}_i, \ell, \mathsf{X}_j$ instruction also tests for the existence of feature ℓ in the feature table of X_i. However, whereas `test_feature` initializes X_j with a jump to the `write` sequence and back, `unify_feature` assumes that X_j already points to a structure on the heap. Hence, this instruction is used rather than `test_feature` if and only if X_j is guaranteed to be initialized (*i.e.*, if it has already occurred in a preceding instruction in this sequence). If feature ℓ is not part of those of X_i, then it is added with the cell pointed to by X_j as value. Otherwise, if ℓ does exist for X_i, it calls a full OSF unification procedure on X_i and X_j.

As for an example, refer to Figure 9. It shows the code generated using the method we just described for the OSF term (1) when compiled as a program. There is a subtle remark to be made here regarding the use of `set` instructions in the `write` sequence of a program code. As we explained, the task that is carried out by that sequence is exactly the same term-building work as the instructions of a query term---hence the use of the same instructions. In a query, however, it is safe for `set_feature` $\mathsf{X}_i, \ell, \mathsf{X}_j$ to assume that the heap cell at address X_i is not bound (since necessarily just created by a `set_sort`). This is no longer the case when `set_feature` $\mathsf{X}_i, \ell, \mathsf{X}_j$ is used in the code of a program (since execution may come to it with a jump from a `read` instruction). Therefore, the effect of `set_feature` given by Figure 6 is not quite correct when used in the code for program terms. It is simple to correct it, as shown in Figure 10, using the operation *deref* (defined in Figure 11) to follow dereference pointers created by binding HEAP cells to other HEAP cells.

The effect of `read` sequence instructions is summed up in Figure 12. The `unify_feature` instruction may need to perform a full-fledged OSF term unification. This procedure is given in Figure 13. It uses a *push down list* (PDL) as in the WAM. The difference is that sort intersection is used instead of comparison of cell tags and equality of functors, and unification of subterms depends on gathering together the relevant features present in both terms. The former is done thanks to the $\wedge$ operation assumed defined on the (representation of) sorts, and the latter thanks to the *carry_features* operation given in Figure 14.

4 Optimizations

The code generation scheme we just sketched is rather naïve in that it produces code that can be immediately optimized in several ways. Indeed it is possible to identify systematically some instructions in the generated code that either will never be executed or that performs unnecessary work. Here we will identify superfluous instructions and explain how to recognize them *after* the generation, and also how to improve the compilation scheme to simply avoid generating them.

4.1 Uneffective code

Let us first remark that it is possible to generate `intersect_sort` $\mathsf{X}_1, \top$ instructions, that have no effect whatsoever. Indeed, any sort intersected with $\top$ stays the same. There is no point then in executing such instructions. It is very easy to get rid of them after generation, but

```
          intersect_sort X1, person          %  X1 : person
          test_feature X1, name, X2, 1, W1   %  X1.name = X2
          intersect_sort X2, id              %  X2 : id
          test_feature X2, first, X3, 2, W2  %  X2.first = X3
          intersect_sort X3, string          %  X3 : string
R1    :   test_feature X2, last, X4, 2, W3   %  X2.last = X4
          intersect_sort X4, string          %  X4 : string
R2    :   test_feature X1, spouse, X5, 1, W4 %  X1.spouse = X5
          intersect_sort X5, person          %  X5 : person
          test_feature X5, name, X6, 2, W5   %  X5.name = X6
          intersect_sort X6, id              %  X6 : id
          unify_feature X6, last, X4         %  X6.last = X4
R3    :   unify_feature X5, spouse, X1       %  X5.spouse = X1
R4    :   jump W6                            %  (skip write code)
      :   push_cell X1                       %  X1 : person
          set_sort X1, person                %  X1 : person
W1    :   push_cell X2                       %  X2 : id
          set_feature X1, name, X2           %  X1.name = X2
          set_sort X2, id                    %  X2 : id
W2    :   push_cell X3                       %  X3 : string
          set_feature X2, first, X3          %  X2.first = X3
          set_sort X3, string                %  X3 : string
          write_test 2, R1                   %
W3    :   push_cell X4                       %  X4 : string
          set_feature X2, last, X4           %  X2.last = X4
          set_sort X4, string                %  X4 : string
          write_test 2, R2                   %
          write_test 1, R2                   %
W4    :   push_cell X5                       %  X5 : person
          set_feature X1, spouse, X5         %  X1.spouse = X5
          set_sort X5, person                %  X5 : person
W5    :   push_cell X6                       %  X6 : id
          set_feature X5, name, X6           %  X5.name = X6
          set_sort X6, id                    %  X6 : id
W5bis :   set_feature X6, last, X4           %  X6.last = X4
          write_test 3, R3                   %
          write_test 2, R3                   %
W5ter :   set_feature X5, spouse, X1         %  X5.spouse = X1
          write_test 2, R4                   %
          write_test 1, R4                   %
          write_test 0, R4                   %
W6    :                                      %
```

Figure 9 Code for program OSF term (1)

```
set_feature Xi, ℓ, Xj ≡  add_feature(HEAP[deref(Xi)].FTAB, ℓ, Xj)
```

Figure 10 Modified `set_feature` instruction

```
function deref(a : address) : address;
  begin
    b ← HEAP[a].CREF;
    if a = b
      then return a
      else return deref(b)
  end deref;
```

Figure 11 The *deref* operation

```
intersect_sort Xi, s                ≡  addr ← deref(Xi);
                                       ns ← s ∧ HEAP[Xi].SORT;
                                       if ns = ⊥
                                         then fail ← true
                                         else HEAP[Xi].SORT ← ns

test_feature Xi, ℓ, Xj, Level, Label ≡  addr ← deref(Xi);
                                       ⟨ found , value ⟩ ← get_feature(HEAP[addr].FTAB, ℓ);
                                       if found
                                         then Xj ← value
                                         else
                                           begin
                                             D ← Level;
                                             goto Label
                                           end

unify_feature Xi, ℓ, Xj             ≡  addr ← deref(Xi);
                                       ⟨ found , value ⟩ ← get_feature(HEAP[addr].FTAB, ℓ);
                                       if found
                                         then osf_unify(value, Xj)
                                         else add_feature(HEAP[addr].FTAB, ℓ, Xj)
```

Figure 12 Instructions for OSF term program `read` sequence

```
procedure osf_unify(a1 : address, a2 : address);
  begin
    push(a1, PDL);
    push(a2, PDL);
    fail ← false;
    while ¬empty(PDL) ∨ fail do
      begin
        d1 ← deref(pop(PDL));
        d2 ← deref(pop(PDL));
        if d1 ≠ d2
          then
            begin
              ns ← HEAP[d1].SORT ∧ HEAP[d2].SORT;
              if ns = ⊥
                then fail ← true
                else
                  begin
                    bind_refine(d1, d2, ns);
                    if deref(d1) = d2
                      then carry_features(d1, d2)
                      else carry_features(d2, d1)
                  end
            end
      end
  end osf_unify;
```

Figure 13 The OSF term unification procedure

```
procedure carry_features(d1, d2 : address);
  begin
    for ⟨ ℓ , v1 ⟩ in HEAP[d1].FTAB do
      begin
        ⟨ found , v2 ⟩ ← get_feature(HEAP[d2].FTAB, ℓ);
        if found
          then
            begin
              push(v1, PDL);
              push(v2, PDL)
            end
          else add_feature(HEAP[d2].FTAB, ℓ, v1)
      end
  end carry_features;
```

Figure 14 The *carry_features* operation

```
procedure bind_refine(d1, d2 : address, s : sort);
  begin
    HEAP[d1].CREF ← d2;
    HEAP[d1].SORT ← s
  end bind_refine;
```

Figure 15 The *bind_refine* operation

it is even easier to avoid generating them, so this is the solution we will choose.

4.2 Unreachable code

The instruction sequence that builds a new term assumes that it has to do so from scratch. But, when executing code corresponding to a program term, one knows that at least the root of the term is already on the heap and will never need to be built. Therefore, the `push_cell` and `set_sort` instructions of the write sequence relative to X_1 will never be executed. They can hence be safely eliminated. We simply erase the two `push_cell` X_1 and `set_sort` X_1, s from the write sequence, or, more properly, avoid generating such instructions by skipping $X_1 : s$ tokens and not generating the `push_cell` X_1 for $X_1.l = X_i$ tokens.

4.3 Unreachable labels and redundant write tests

When in the read sequence we generate a `unify_feature` X_j, l, X_i instruction, we are sure that we will never need to build the subterm relative to X_j alone. Indeed, during execution, either we reach this unify instruction, and in that case we do not jump to the `write` sequence, or we jump to the `write` sequence before reaching this `unify` instruction, in correspondence with an external subterm X_h at lower depth containing X_i, and in that case we will not have to jump back to `read` mode after generating X_i, but after generating X_h. In either case,

- we never enter `write` mode at the label for X_i, so such label is unused and redundant;
- we never exit from the `write` sequence at the depth of X_i, but at a lower depth, so the write test generated for X_i will never succeed and is useless.

Eliminating useless labels and tests is a little harder than for the previous optimization. One needs to check that there is no test feature involving that label, and that in the corresponding read sequence there is no test feature involving the depth specified in the test. It is much easier to avoid producing that code, by recognizing that X_i in the token $X_j.l = X_i$ has already been seen, so no label nor test must be generated for the term rooted at X_i.

4.4 Compressing chains of write tests

The easiest and most effective optimization is the possibility to compress chains of write tests into one single write test. When generating the write code for a deeply nested term, like for example $f(g(h(l(a))))$, the write tests that we must generate at the end of each subterm will actually lie side by side, all accumulated in a chain, as shown in Figure 16.

This is because when we finish building a, we also finish building $l(a)$, and $h(l(a))$ and all the more external terms where a appears as a rightmost subterm. Here, the sequence of tests can be compressed in just one test on the smallest depth d: this sequence of tests tells us to jump to R_i is the depth register is greater than or equal to $d + 4$, or $d + 3$, ... or d, i.e. we must jump if it is greater or equal to d.

```
write_test d+4, R_i   % end of subterm rooted at a
write_test d+3, R_i   % end of subterm rooted at l
write_test d+2, R_i   % end of subterm rooted at h
write_test d+1, R_i   % end of subterm rooted at g
write_test d, R_i     % end of subterm rooted at f
```

Figure 16 Optimized for program term $f(g(h(l(a))))$

This condition is most easily recognized after code generation, as a chain is clearly identifiable locally, but also in phase of code generation one can just pass on a flag identifying the leftmost subterm, to inhibit generation of write tests for it.

Figure 17 shows the result of applying all the above optimizations to the code of Figure 9.

This concludes our description of our basic compilation scheme for OSF term unification. This scheme can be improved much further to exploit several particular situations as shown in [8].

```
        intersect_sort X1, person          %  X1 : person
        test_feature X1, name, X2, 1, W1   %  X1.name = X2
        intersect_sort X2, id              %  X2 : id
        test_feature X2, first, X3, 2, W2  %  X2.first = X3
        intersect_sort X3, string          %  X3 : string
R1  :   test_feature X2, last, X4, 2, W3   %  X2.last = X4
        intersect_sort X4, string          %  X4 : string
R2  :   test_feature X1, spouse, X5, 1, W4 %  X1.spouse = X5
        intersect_sort X5, person          %  X5 : person
        test_feature X5, name, X6, 2, W5   %  X5.name = X6
        intersect_sort X6, id              %  X6 : id
        unify_feature X6, last, X4         %  X6.last = X4
R3  :   unify_feature X5, spouse, X1       %  X5.spouse = X1
R4  :   jump W6                            %  (skip write code)
    :   push_cell X1                       %  Unreachable code
        set_sort X1, person                %  Unreachable code
W1  :   push_cell X2                       %  X2 : id
        set_feature X1, name, X2           %  X1.name = X2
        set_sort X2, id                    %  X2 : id
W2  :   push_cell X3                       %  X3 : string
        set_feature X2, first, X3          %  X2.first = X3
        set_sort X3, string                %  X3 : string
        write_test 2, R1                   %
W3  :   push_cell X4                       %  X4 : string
        set_feature X2, last, X4           %  X2.last = X4
        set_sort X4, string                %  X4 : string
        write_test 2, R2                   %  Redundant test: nested subterms
        write_test 1, R2                   %
W4  :   push_cell X5                       %  X5 : person
        set_feature X1, spouse, X5         %  X1.spouse = X5
        set_sort X5, person                %  X5 : person
W5  :   push_cell X6                       %  X6 : id
        set_feature X5, name, X6           %  X5.name = X6
        set_sort X6, id                    %  X6 : id
        set_feature X6, last, X4           %  X6.last = X4
        write_test 3, R3                   %  Redundant test: nested and unreachable subterm
        write_test 2, R3                   %
        set_feature X5, spouse, X1         %  X5.spouse = X1
        write_test 2, R4                   %  Redundant test: nested and unreachable subterm
        write_test 1, R4                   %  Redundant test: nested subterms
        write_test 0, R4                   %  Redundant test: end of write code
W6  :                                      %
```

Figure 17 Optimized for program OSF term (1)

References

1. Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, Cambridge, MA (1991).

2. Hassan Aït-Kaci. An introduction to LIFE---programming with logic, inheritance, functions, and equations. In Dale Miller, editor, *Proceedings of the International Symposium on Logic Programming*, pages 52--68, Cambridge, MA (October 1993). MIT Press.

3. Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. PRL Research Report 11, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (June 1991).

4. Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3-4):195--234 (July-August 1993).

5. Mohamed Amraoui. *Une Expérience de Compilation de Prolog II sur MALI*. Thèse de doctorat, Université de Rennes I, France (January 1988).

6. André Mariën and Bart Demoen. A new scheme for unification in WAM. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 257--271, Cambridge, MA (1991). MIT Press.

7. Micha Meier. Compilation of compound terms in Prolog. In Saumya Debray and Manuel Hermenegildo, editors, *Logic Programming, Proceedings of the 1990 North American Conference*, pages 63--79, Cambridge, MA (1990). MIT Press.

8. Richard Meyer. Compiling life. PRL Technical Note 8, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison (December 1993).

9. David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA (October 1983).

PRL Technical Notes

The following documents may be ordered by regular mail from:

Librarian -- Technical Notes
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92563 Rueil-Malmaison Cedex
France.

It is also possible to obtain them by electronic mail. For more information, send a message whose subject line is `help` to `doc-server@prl.dec.com` or, from within Digital, to `decprl::doc-server`.

Technical Note 1: *Wild-LIFE, a User Manual*. Hassan Aït-Kaci and Richard Meyer. (being revised).

Technical Note 2: *Wild-LIFE, an Implementation Manual*. Richard Meyer. (being revised).

Technical Note 3: *Characterising Perle0*. Alan Skea. October 1990.

Technical Note 4: *Perle1DC: a C++ Library for the Simulation and Generation of DECPeRLe-1 Designs*. Hervé Touati. February 1994.

Technical Note 5: *TiGeR Version 1.0 User Guide*. Olivier Coudert, Jean-Christophe Madre, and Hervé Touati. January 1994.

Technical Note 6: *Tgr Version 1.0 Reference Manual*. Olivier Coudert, Jean-Christophe Madre, and Herve Touati. August 1993.

Technical Note 7: *Compiling Order-Sorted Feature Term Unification*. Hassan Aït-Kaci and Roberto Di Cosmo. December 1993.

Technical Note 8: *Compiling LIFE*. Richard Meyer. December 1993.

7

Compiling Order-Sorted Feature Term Unification

Hassan Aït-Kaci and Roberto Di Cosmo

digital

PARIS RESEARCH LABORATORY