An Experiment in Abstract Machine Design

stephan diehl

FB 14-Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, Germany (email: diehl@cs.uni-sb.de)

SUMMARY

In this article we present Typed Feature Structures as an extension of Prolog, and show how to come up with a compilation scheme and an abstract machine using a design methodology based on partial evaluation. First we define the transformations used by our partial evaluator. Then we present the design methodology which we will use later. Next, we clarify the notion of Typed Feature Structures that underlies our work, and formally define the unification of such structures. Based on this definition, we develop a unification procedure with explicit heap representation. By partially evaluating this procedure with respect to some example programs, we show how to come up with the machine instructions and translation schemes. Finally, we briefly address coreferences, cyclic structures and the unification of types.

key words: logic programming; typed feature structures; abstract machines; WAM; compiler design; partial evaluation

INTRODUCTION

The more programming languages are freed from the ‘Von-Neumann style’, the more important are abstract machines. Abstract machines provide an intermediate target language for compilation. First, the compiler generates code for the abstract machine, then this code can be further compiled into real machine code or it can be interpreted. By dividing compilation into two stages, abstract machines increase the portability and maintainability of compilers. For compiling Prolog, and as a basis for compilers of other logic programming languages, the Warren Abstract Machine (WAM) is widely used. Peter Kursawe shows how some of the WAM instructions for unification, originally invented by Warren, can be reinvented by using partial evaluation. Later, Nilsson derived some of the control instructions of the WAM in a similar way. To our knowledge, we are the first to use this methodology to ‘invent’ WAM-like instructions, not ‘reinvent’ WAM instructions. Moreover, the methodology both guides the design and relates source language constructs to machine instructions. Although the methodology has only been applied in the context of logic programming, we are convinced that it can be applied to other languages, too. For example, it is not difficult to design some of the P-machine instructions by applying the methodology to an interpreter for arithmetic expressions.

In the following sections we apply the methodology to come up with machine instructions for Typed Feature Structures. These are an extension of first-order terms in Prolog. Basically, we allow structures and predicates to have typed and labelled...
arguments similar to records in Pascal. Via types, a certain kind of inheritance is added to the language. Typed Feature Structures have been used as knowledge representation formalisms in AI and computational linguistics for a number of years.

**PARTIAL EVALUATION**

In logic programming there are several program transformation techniques which can be used in a partial evaluator. Some of these are unfolding and folding, forward unification, backward unification, deletion of unification, computation of standard predicates, disjunction elimination and many other source-to-source transformations.  

Next, we define those transformations used in our partial evaluator.

*Unfolding:* a literal in the intermediate code is replaced by the corresponding instance of the body of one or more of its defining clauses.

Let \( L \) be a literal and \( L_i : \neg G_i \) all variants of clauses for \( L \). Then replace \( L \) by the disjunction \( (G_1, \ldots, G_k) \), where \( G_i = (t_{01} = t_{i1}, \ldots, t_{0n} = t_{in}, G_i) \) and \( L = p(t_{01}, \ldots, t_{0n}) \) and \( L_i = p(t_{i1}, \ldots, t_{in}) \).

*Folding:* if a part of the intermediate code matches a defining clause of a predicate, that part is replaced by the corresponding instance of the predicate. Note that this transformation should only be applied if the predicate has only one defining clause.

Let \( C = L_1 : \neg G_1, \ldots, G_k \) be a clause and \( L_2 : \neg G_{i1}, \ldots, G_{in} \) a variant of another clause, such that \( \{G_{i1}, \ldots, G_{in}\} \subseteq \{G_1, \ldots, G_k\} \). Then delete all occurrences of \( G_j \) in \( C \) and add the goal \( L_2 \) to the body of the clause \( C \).

*Evaluation of a predicate:* if the evaluation of a predicate succeeds, it is replaced by \( true \) and the corresponding bindings take place; otherwise it is replaced by \( fail \).

**Syntactic Transformations:**

\[
(\text{pl}, \ldots, \text{true}, \ldots, \text{pn}) \rightarrow (\text{pl}, \ldots, \text{pn}) \\
(\text{pl}, \ldots, \text{fail}, \ldots, \text{pn}) \rightarrow \text{fail} \\
(\text{pl}; \ldots; \text{true}; \ldots; \text{pn}) \rightarrow \text{true} \\
(\text{pl}; \ldots; \text{fail}; \ldots; \text{pn}) \rightarrow (\text{pl}, \ldots, \text{pn}) \\
(\text{true} \rightarrow \text{alt1}; \text{alt2}) \rightarrow \text{alt1} \\
(\text{fail} \rightarrow \text{alt1}; \text{alt2}) \rightarrow \text{alt2} \\
(\text{true} \rightarrow \text{alt1}) \rightarrow \text{alt1} \\
(\text{false} \rightarrow \text{alt1}) \rightarrow \text{true}
\]

Our partial evaluator is based on the work of Lakhotia and Sterling. The heart of the partial evaluator consists of the following definitions:

\[
\text{peval(Head,BodyResidue)} : - \\
\quad \text{should_unfold(Head),!}, \text{clause(Head,Body), peval(Body,BodyResidue)}. \\
\text{peval(Goal,true)} : - \text{evaluable(Goal),call(Goal)}. \\
\text{peval(Goal,fail)} : - \text{evaluable(Goal),!}.
\]

The predicate \texttt{should_unfold} provides us with the ability to control unfolding, i.e., the user of the partial evaluator can define this predicate depending on the interpreter that is going to be partially evaluated. In this way, meta-knowledge about the interpreter can be provided. The predicate \texttt{evaluable} decides whether or not a literal should be evaluated.
Kursawe derived the unification instructions of the WAM by partially evaluating a unification procedure written in Prolog. Nilsson used the same approach to derive some of the control instructions of the WAM. He also expounds the view that partial evaluation and other program transformations are part of a methodology to design abstract machines for logic programming.

Kursawe introduced special unification predicates, and used these to make unification explicit in Prolog clauses. Then he partially evaluated these clauses with respect to the definition of the unification predicates. Next he replaced recurring patterns in the resulting partially evaluated clauses by ‘machine instructions’. The following example shows this process for the simple case of the clause append([],L,L):

\[
\begin{align*}
\text{append}([],L,L). & \quad \text{↓ make unification explicit} \\
\text{append}(A1,A2,A3) & : - \text{unipp}([],A1),\text{unipp}(L,A2),\text{unipp}(L,A3). & \quad \text{↓ partial evaluation} \\
\text{append}(A1,A2,A3) & : - (\text{var}(A1), A1 = []; [] == A1), \text{unipp}(A2,A3). & \quad \text{↓ define ‘machine instructions’} \\
\text{append}(A1,A2,A3) & : - \text{unipp\_constant}([],A1),\text{unipp}(A2,A3).
\end{align*}
\]

Note that the result of this method is twofold: (1) compilation of a program into machine code; and (2) design of an abstract machine by defining its machine instructions.

In this paper, we use the following similar methodology, which consists of the following steps.*

1. Define an interpreter.
2. Change the interpreter by making the heap representation and the unifications explicit.
3. Partially evaluate the interpreter with respect to some example inputs.
4. Look for patterns in the intermediate code and define corresponding machine instructions.
5. Fold the result of the partial evaluation using the machine instructions.
6. If the resulting code contains predicates or Prolog constructs which are not machine instructions, or the machine instructions are not deterministic or are unsuitable for conventional computer architectures, then repeat the design loop (steps 2 to 5).

**TYPED FEATURE STRUCTURES**

In computational linguistics, Typed Feature Structures and extensions thereof are used to write grammars for natural language processing. In AI, Typed Feature Structures are at the heart of frame-like knowledge representation formalisms. First we give the syntax of Typed Feature Structures, which we will use throughout this paper. Basically, a Typed Feature Structure consists of its type and a list of attribute/value pairs (AVPAIRS). The types are organized in a type hierarchy, which

---

* Here the interpreter is a procedure for unifying Typed Feature Structures.
has to be defined by the user. Typed Feature Structures as we present them here are similar to the $\psi$-terms in LOGIN.\textsuperscript{10}

\[
\text{TFS} \rightarrow \text{TYPE:}\left[\text{AVPAIRS}\right]\text{TYPE:}\left[\right]\text{SYMBOL = TFS}\text{TERM}
\]

\[
\text{AVPAIRS} \rightarrow \text{SYMBOL TFS}\text{SYMBOL TFS, AVPAIRS}
\]

The following are two unit clauses in Prolog and Typed Feature Structures:

\[
\text{person:}\left[\text{name arthur, office address:}\left[\text{street park_ave, no 14, city boston}\right]\right].
\]
\[
\text{homeworker:}\left[\text{office X=address:}\left[\right], \text{home X}\right].
\]

In the Typed Feature Structure of type \text{homeworker} above, the coreference \text{X} denotes that the attributes \text{office} and \text{home} share the same value \text{address:}\left[\right]. Here \left[\right] does not denote the empty list, but a typed feature structure without attribute/value pairs, i.e., \text{address:}\left[\right] unifies with every Typed Feature Structure of type \text{address}. Coreferences are closely related to Prolog’s logic variables. Actually, we will treat them as syntactic sugar. In Prolog the notion of types and inheritance can be implemented using chain rules: \text{animal(X) :- mammal(X). mammal(X) :- dog(X).}

These rules state that every dog is an animal, thus all rules for animals can be applied for dogs. This is a very inefficient way in which to express subtype relations. If types and inheritance hierarchies are explicit, then we can use typed unification instead of standard Prolog unification. In the following, we assume that there is such an explicit hierarchy (directed acyclic graph), and that we can compute the greatest lower bound $\text{glb}(t_1,t_2)$ and test inheritance $t_1 \subseteq t_2$ of two types $t_1$ and $t_2$ in that hierarchy. In Prolog each predicate has a fixed arity and arguments are positional. The following example shows how the use of labels and types differs from the Prolog notation:

\[
\text{in Prolog:} \quad \text{person(name(john,X),Y).}
\]
\[
\text{using labels and types:} \quad \text{person:}\left[\text{name name:}\left[\text{first john, last X}\right], \text{address Y}\right].
\]
\[
\text{person:}\left[\text{address Y, name name:}\left[\text{last X, first john}\right]\right].
\]

The main difference is that in Typed Feature Structures, neither the order in which arguments occur nor the number of arguments is fixed (cf. keyword parameter in LISP). Typed Feature Structures have been essential parts of recently proposed linguistic formalisms.\textsuperscript{11} Some of these are Functional Unification Grammar (FUG), Parse and Translate II (PATR-II), Lexical Functional Grammar (LFG) and Head-driven Phrase Structure Grammar (HPSG). In HPSG rules, principles and lexicon entries are represented as Typed Feature Structures. The following is an example of an HPSG-like feature structure:
Using our notation, this could be written as

\[
\text{reflexive-verb} : \{\text{syn syntax} : [\text{loc local} : [\text{head vhead} : []]], \text{sem semantic} : [\text{cont basic-circumstance} : [\text{agent } X, \text{patient } X]]]\.
\]

**Unification of acyclic Typed Feature Structures**

Before we formally define the unification of Typed Feature Structures, we give an example, which covers many of the interesting cases:

\[
X = \text{person} : [\text{name } X, \text{office address} : [\text{street main street}, \text{age } 35]], \\
Y = \text{person} : [\text{office home address} : [\text{number } 4], \text{name john}].
\]

Assuming that \text{home address} is a subtype of \text{address}, the unification of \text{$X$} and \text{$Y$} yields:

\[
\text{person} : [\text{name john}, \\
\text{office home address} : [\text{street main street, number } 4], \text{age } 35]
\]

Readers interested in the methodology and not so much in the details of the unification of Typed Feature Structures might skip the rest of this section.

For the following definitions we translate coreferences into path equations. Further-
more, we will use a special symbol $\top$ (top), which unifies with every other value. The notation $2^A$ denotes the powerset of $A$, i.e. the set of all subsets of the set $A$. We will write $\hat{e}$ to denote the symmetric, transitive closure of the relation $e$. Furthermore, the domain of a function is $\text{dom}(f) = \{a : f(a) \text{ is defined}\}$ and the restriction of a function $f$ to a set $A$ is defined as $f|_A = \{(a,f(a)) : a \in A \text{ and } f(a) \text{ is defined}\}$.

Every Typed Feature Structure can be regarded as a triple comprising its type, a partial mapping from attributes to values, and a set of path equations.

**Definition**

\[
\begin{align*}
\mathit{TF} & = \mathit{TYPES} \times (\mathit{ATTRIBUTES} \rightarrow \mathit{VALUES}) \times \mathit{EQ} \\
\mathit{VALUES} & = \mathit{TF} \cup \mathit{ATOMS} \cup \{\top\} \\
\mathit{EQ} & = 2^{\mathit{PATH} \times \mathit{PATH}} \\
\mathit{PATH} & = \mathbb{N} \rightarrow \mathit{ATTRIBUTES}
\end{align*}
\]

The value of a path $p$ in a given Typed Feature Structure $(t,f,e)$ is denoted by $\psi(p,(t,f,e),0)$, which is formally defined as:

\[
\psi(p,f(p(n)),n+1) \quad \text{if } p(n) \text{ is defined} \\
\psi(p,v,n) = \begin{cases} v & \text{if } p(n) \text{ is undefined} \\ \top & \text{if } v = \top \end{cases}
\]

We call a Typed Feature Structure well-formed if it does not violate any of its path equations, i.e., a Typed Feature Structure $(t,f,e)$ is well-formed iff, for every $(p,q) \in \hat{e}$, we have $\psi(p,(t,f,e),0) \equiv \psi(q,(t,f,e),0)$ and for every $a \in \mathit{ATTRIBUTES}$ the value $f(a)$ is well-formed.

Now we have to define the congruence ($\equiv$) of two Typed Feature Structures

\[
\begin{align*}
a \equiv a & \quad \text{if } a \in \mathit{ATOMS} \\
\top \equiv v & \quad \text{if } v \in \mathit{VALUES} \\
v \equiv \top & \quad \text{if } v \in \mathit{VALUES} \\
(t_1,f_1,e_1) \equiv (t_2,f_2,e_2) & \quad \text{if } t_1 \sqsubseteq t_2 \text{ or } t_2 \sqsubseteq t_1 \\
 & \quad \text{and } \forall a \in \text{dom}(f_1) \cup \text{dom}(f_2): f_1(a) \equiv f_2(a)
\end{align*}
\]

The unification of two Typed Feature Structures basically consists of the greatest lower bound of their types, the unification of their attribute/value lists and the union of their path equations. In addition, the resulting Typed Feature Structure must be well-formed:

\[
\begin{align*}
a \sqcup a & = a & \text{if } \text{glb}(t_1,t_2) \neq \bot, \\
(t_1,f_1,e_1) \sqcup (t_2,f_2,e_2) & = (\text{glb}(t_1,t_2),f_1 \sqcup f_2, e_1 \sqcup e_2) & \text{if } f_1 \sqcup f_2 \text{ is defined} \\
 & & \text{and } (\text{glb}(t_1,t_2),f_1 \sqcup f_2, e_1 \sqcup e_2) \text{ is well-formed}
\end{align*}
\]
\[ f_1 \sqcup f_2 = f_1|_{\text{dom}(f_1) - \text{dom}(f_2)} \cup f_2|_{\text{dom}(f_2) - \text{dom}(f_1)} \]
\[ \cup \{(a,f_1(a) \sqcup f_2(a)) : a \in \text{dom}(f_1) \cap \text{dom}(f_2)\} \]

Note that \( a \sqcup b \) is not defined for two atoms \( a, b \) and \( a \neq b \). In this case, we say the unification fails. Furthermore, if \( f_1(a) \sqcup f_2(a) \) is undefined for some attribute \( a \in \text{dom}(f_1) \cap \text{dom}(f_2) \), then \( f_1 \sqcup f_2 \) is undefined, i.e., the unification fails. Last but not least, if \( \text{glb}(t_1,t_2) = \bot \), then \( (t_1,f_1,e_1) \sqcup (t_2,f_2,e_2) \) is not defined, and as before the unification fails.

**THE METHODOLOGY AT WORK**

The above definition of the unification of two Typed Feature Structures was the basis for a unification procedure based on Prolog structures and lists. Next we made unifications and the heap representation explicit using the following predicates:

- `heap(Addr, Tag, Value)` reads the Tag and Value at Addr
- `heap_entry(Addr, Tag, Value)` writes the Tag and Value on top of the heap and returns that Addr
- `heap_change(Addr, Tag, Value)` writes the Tag and Value at Addr

Some of the properties of the representation shown in Figure 2 are:

(i) Nodes can point to newer ones, i.e., after unifying two nodes, one node points to the other, which represents the result of the unification.

(ii) The heap cells subsequent to the node always contain the first element of the attribute value list.

(iii) Four heap cells above the node, the type entry is stored.

(iv) Type entries can point to newer ones, i.e., when two nodes with incomparable types are unified, the type entry points to a new type entry for the unified type.

(v) Attribute value list elements can point to the next attribute value list element.

(vi) The value of an attribute value list is always a variable (REF), which can point to the actual value.

(vii) The end of the attribute value list or the empty attribute value list always allocates two heap cells for the attribute and the value, which might be stored there later. This is necessary to merge the attribute value lists of two structures.

When executing a Prolog program, arguments of a goal are unified with the arguments of a head of a program clause. If this unification succeeds, the literals of the body of the clause become new goals. Now the idea is that goals are already stored on the heap, whereas the program clauses are represented as Prolog terms. After unification of the goal on the heap and the Prolog term in the program clause, the result is stored on the heap and can be unified with another argument.

Thus, in our unification procedure we have to make the way in which the feature structure is stored on the heap explicit. Furthermore unifications, and thus the use of logical variables, have to be made explicit and replaced by assignments or simple unifications. Last but not least, we have to strive for determinism, because the WAM-like instructions we are going to derive should be deterministic and suitable for conventional computer architectures. As a result, we actually use a sublanguage of Prolog, which can be given a different interpretation as an imperative language.
Figure 2. Heap representation of Typed Feature Structures

with pattern matching. In this setting the predicate \(=/2\) is read as assignment and pattern matching, \(==/2\) as an equality test and \((X \rightarrow Y; Z)\) as conditional (no backtracking). After many iterations of the design loop, we obtained the final interpreter. The excerpts of the source code (Figure 3) show the low level of abstraction which is used to write the interpreter. We use the prefix `uniph_` for predicates, which unify Prolog terms with terms stored on the heap.

**Meta-knowledge**

To guide partial evaluation we had to provide some meta-knowledge. First, all predicates with the prefix `uniph_` should be unfolded if their first argument is
uniph_tfs(TFS,Addr,Mode)
   :- (TFS=node(T,AVL)
      -> (/* if MODE is write, then create a new NODE with
            value BUILD_TYPE and create an empty AVL */
         (Mode=write,
          heap_entry(Addr,'NODE', 'BUILD_TYPE'),
          heap_entry(AVLSAddr,'AVL',AVLSAddr),
          heap_entry(_, 'AV',_), heap_entry(_, 'REF', _),
          PatchAddr=Addr)
      ; (Mode=read, deref(Addr, DAddr),
         /* else if MODE is read and value at ADDR is a variable
            then create a new NODE with value BUILD_TYPE
            and create an empty AVL */
         (heap(DAddr,'REF', DAddr),
          heap_entry(NextAddr,'NODE', 'BUILD_TYPE'),
          PatchAddr=NextAddr,
          heap_entry(AVLSAddr,'AVL',AVLSAddr),
          heap_entry(_, 'AV',_), heap_entry(_, 'REF', _),
          heap_change(DAddr, 'REF', NextAddr))
      )*/
   ; /* else if MODE is read and value Addr is a NODE, then unify
       the attribute-value list and the type of TFS with that
       of the structure stored at Addr */
   ; (heap(DAddr,'NODE', DAddr),
      TypeAddr=DAddr+4, PatchAddr=DAddr, AVLSAddr=DAddr+1)),
   /* finally unify the type and the attribute value
      list with those stored on the heap. */
   uniph_type(T,TypeAddr,PatchAddr,Mode),
   uniph_avls(AVL,AVLSAddr,Mode))
   ; /* last but not least, if TFS is a standard term,
       then use Kursawe’s uniph() definitions */
   (TFS=node(C,V) -> fail ; uniph(TFS,Addr))).

uniph_avls(AVL,Addr,Mode)
   :- (/* if MODE is write and AVL is the empty list, then create an
        empty list on the heap. (actually the empty list has already
        been created by uniph_tfs, so nothing has to be done. */
       Mode=write, AVL=[])
   ; /* if MODE is read and AVL is the empty list,
       then test, whether there is an av list on the heap */
       (Mode=write, AVL=[], deref(Addr, DAddr), heap(DAddr,'AVL', NAddr))
   ; /* if MODE is write and AVL is a non-empty list, then build the
       attribute value list AVL on the heap, else if MODE is read,
       then compare the AVL lists and make according bindings. */
   (AVL=[AV|R], (Mode=write -> AvlAddr=Addr),
    uniph_nonempty_avl(AVL, AvlAddr, Mode)).

Figure 3. Final version of interpreter
bound. Furthermore, the predicate $==$ can be evaluated if its arguments are bound. The predicate $=\backslash 2$ can be evaluated if its first argument is bound, and the predicates \texttt{var/1} and \texttt{nonvar/1} have to be evaluated whatever their argument might be.

Intermediate code produced by partial evaluation

In the following sections, we only consider \textit{read} mode. For details on \textit{write} mode see elsewhere.\textsuperscript{3} The intermediate code, as we present it here, is difficult to read because of the heavy use of parentheses. These indicate the tree structure of the code, which is important for the later folding phase.

Partial evaluation of the intermediate code produces the code in \textit{Figure 4}.

\begin{verbatim}
((deref(\$,0),_293),
 (heap(\_293,REF,\_293),
 (heap_entry(\_275,NODE,BUILD_TYPE),
  (_304=_275, (heap_entry(_323,AVL,_323),
   (heap_entry(_255,AVL,_257),
    (heap_entry(_248,REF,_250),
     heap_change(_293,REF,_275)))))));
(heap(_\_293,NODE,\_293),
  (_231=\_293+4, (304=\_293, _323=\_293+1))))),
(heap(_\_304,NODE,BUILD_TYPE)
  ->(heap_entry(_231,TYPE,_231),
    (heap_entry(_1906,TYPENAME,append),
     heap_change(_304,NODE,_304))))
);(deref_type(_231,\_1897),
 ((heap(_1897+1,TYPENAME,\_1888), _1888=append)
  ;(heap(_1897+1,TYPENAME,\_1888),
   (heap_entry(_1865,TYPE,\_1865),
    (glb(append,\_1888,\_1868),
     (heap_entry(_1849,TYPENAME,\_1858),
      heap_change(_1897,TYPE,\_1865))))),
 (_3986=_3119,
  ((deref(_3119,\_4803),
   ...

\end{verbatim}

\textit{Figure 4. Intermediate code}

The intermediate code is not very readable, but the parentheses make the tree structure of the intermediate code explicit. This tree structure is important for the pattern matching, which takes place when we try to fold the intermediate code.

After partially evaluating some example programs with respect to the unification procedure \texttt{uniph\_tfs} and flattening the resulting code, we found recurring patterns. These have been the basis for the definition of WAM-like instructions for Typed Feature Structures. All variables which occur in a pattern but are not local to that pattern have to be passed in registers in the resulting machine instruction. In addition, the instructions are defined in such a way that all disjunctions and implications in the intermediate code are within a single instruction. In other words, after folding...
we want the resulting code to be a conjunction of machine instructions. To achieve this goal we had to find out which parts of the interpreter caused problems for defining such instructions. Then we had to solve these problems and change the interpreter accordingly. Based on this experience, the following restrictions should be satisfied for an interpreter to be suitable for our methodology:

(i) The clauses should be tail-recursive. If we allow general recursion, in the resulting abstract machine the content of variables would have to be stored on a stack, in order to be available after returning from recursive calls.
(ii) The number of arguments passed to uniph calls has to be as small as possible, because these arguments have to be passed in registers in the resulting abstract machine.

Looking at the intermediate code produced by partial evaluation of some examples, we defined the instructions shown in Figure 5.

**Code produced by partial evaluation and folding**

After folding the above code (Figure 4) by using our machine instructions (Figure 5), i.e., replacing patterns, which match the definition of a machine instruction

```prolog
get_type(T1,Addr, PatchAddr)
  :- ( heap(PatchAddr, 'NODE', 'BUILD_TYPE'),
           heap_entry(Addr, 'TYPE', Addr),
           heap_entry(_, 'TPNAME', T1),
           heap_change(PatchAddr, 'NODE', PatchAddr),
          ( deref_type(Addr, DAddr),
            ( heap(DAddr+1, 'TPNAME', T2), T2=T1,
              ; heap(DAddr+1, 'TPNAME', T2),
              heap_entry(NewAddr, 'TYPE', NewAddr),
              glb(T1, T2, T3), heap_entry(_, 'TPNAME', T3),
              heap_change(DAddr, 'TYPE', NewAddr))).

get_node(Addr, AvlAddr, TypeAddr, PatchAddr)
  :- deref(Addr, DAddr),
     (( heap(DAddr, 'REF', DAddr),
       heap_entry(NextAddr, 'NODE', 'BUILD_TYPE'),
       PatchAddr=NextAddr, heap_entry(AvlAddr, 'AVL', AvlAddr),
       heap_entry(_, 'AV', _), heap_entry(_, 'REF', _),
       heap_change(DAddr, 'REF', NextAddr))
    ; ( heap(DAddr, 'NODE', DAddr), TypeAddr=DAddr+1,
       PatchAddr=DAddr, AvlAddr=DAddr+1)).

get_empty_avl(Addr) :- deref(Addr, DAddr), heap(DAddr, 'AVL', NAddr).

get_attribute(Attr, AvlAddr, VAddr, Addr)
  :- AvlAddr=Addr, deref(Addr, DAddr), find_attribute(Attr, DAddr, VAddr).
```

*Figure 5. Definitions of abstract machine instructions*
by that instruction, we obtained the ‘compiled’ program in Figure 6. Next we optimized the instructions slightly. We combined \texttt{get\_node} and \texttt{get\_type} to \texttt{get\_typed\_node} and saved one register. Furthermore, in \texttt{get\_attribute}, the purpose of the last two arguments can be combined. Finally we got the following functionality:

(a) \texttt{get\_typed\_node t X1 X2}
    If \( X1 \) points to an unbound variable, this instruction creates a new node with an empty attribute value list and saves its address in the unbound variable pointed to by \( X1 \) and the address of the empty attribute value list in \( X2 \). If \( X1 \) is bound to a Typed Feature Structure, then \( X2 \) gets bound to its attribute value list and its type is unified with \( t \).

(b) \texttt{get\_attribute a X1 X2}
    \( X2 \) points to an attribute value list. This instruction looks for the attribute \( a \) in that attribute value list and binds \( X1 \) to the value of the attribute value pair found. If there is no such attribute in the attribute value list, a new attribute value pair is created at the end of the attribute value list, and \( X1 \) points to its value (an unbound variable). The binding of \( X2 \) is changed only if the first element in the attribute value list has the attribute \( a \). In this case, \( X2 \) gets bound to the next element in the list.

Translation schemes

A closer look at the WAM code generated above, and similar examples, reveals the translation schemes in Table I, where \( <t> \) stands for a type name, \( <avl> \) for an attribute value list, \( <a> \) for an attribute and \( <v> \) for a value.

Coreferences, cycles and types

In our implementation coreferences are treated as syntactic sugar, e.g.

\[
\text{person:} [\text{name bill, home X=address: [], office X}].
\]

can be rewritten as

\[
\text{person:} [\text{name bill, home X, office X}] :- \text{unify(X, address: [])}.
\]
Our version of the WAM can also handle cyclic bindings. First, the modifications for Typed Feature Structures work as well for cyclic ones. In addition, we had to change the way in which Prolog structures are represented in the WAM. Now instead of one cell, two cells are allocated for the head of each structure. In Table II, we show the heap after the structures at address 45 and 56 have been unified.

The STR cell is handled like a REF cell. Finally, the computation of greatest lower bounds of types in the inheritance hierarchy is based on a bit encoding of the inheritance relation.12

CONCLUSIONS

Using a methodology for the design of abstract machines, we invented WAM-like instructions for Typed Feature Structures. The methodology both guides the design and relates source language constructs to machine instructions. Finally, we optimized the new machine instructions slightly, and came up with compilation schemes for Typed Feature Structures.

Partial evaluation has been done fully automatically using the partial evaluator described in the author’s thesis.5 The abstract machine has been implemented in C without any problems, since the definitions in Figure 5 can be translated into C using macros for heap_entry, deref, etc. Integrating the compilation schemes of Table I into an existing Prolog compiler13 was not difficult. This is not surprising, because when we wrote the unification algorithm uniph_tfs we tried to adopt the WAM principles.14

We feel that our use of the design methodology is very promising, and that it is

---

**Table I. Translation schemes in READ mode**

<table>
<thead>
<tr>
<th>TFS construct</th>
<th>Abstract machine code</th>
</tr>
</thead>
</table>
| ⟨t⟩: []       | get_typed_node (t) X1 X2  
               | get_empty_av1 X2       |
| ⟨t⟩: [(av1)]  | get_typed_node (t) X1 X2  
               | ⟨GET instructions for (av1) and the av1 stored in X2⟩ |
| ⟨a⟩ ⟨v⟩, ⟨av1⟩ | get_attribute (a) X3 X1  
               | ⟨GET instructions for (v) and the value stored in X3⟩ |
               | ⟨GET instructions for (av1) and the av1 stored in X1⟩ |

---

**Table II. Modified WAM representation of Prolog structures**

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>STR</td>
<td>56</td>
</tr>
<tr>
<td>46</td>
<td>STRNAME</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>STR</td>
<td>56</td>
</tr>
<tr>
<td>57</td>
<td>STRNAME</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
a very interesting question as to whether it scales up. Can we use the methodology to derive instructions for extensions of Typed Feature Structures (e.g., disjunctions, functional uncertainty, ...), or can we even use the methodology to derive new control instructions for a totally different search algorithm which would replace the Prolog depth-first search? Looking at current knowledge representation formalisms in AI (e.g., KL-ONE, LOOM, CLASSIC, TDL) and recently developed hybrid programming languages (e.g., LIFE, OZ), we find that logic becomes more and more important. We expect that the use of a methodology to design abstract machines and code generation schemes will make it much easier and less error-prone to write compilers for such AI formalisms and programming languages. Although we have presented the methodology in the context of logic programming, we are convinced that it is applicable to other language paradigms as well. Usually, abstract machines are designed in an ad hoc manner, often based on experience with other abstract machines. In contrast, the methodology provides some guidance. Basically, what we suggest is: (i) write an interpreter, (ii) partially evaluate it with respect to example programs, (iii) find recurring patterns, and (iv) generalize the patterns as machine instructions.

acknowledgement

I wish to thank Ulf Nilsson for his comments on a draft of this paper, Stanley Selkow for many discussions, and the Fulbright Commission for making my stay at WPI possible.

REFERENCES

11. S. Shieber, An Introduction to Unification-Based Approaches to Grammar, Center for the Study of Language and Information, Stanford University, 1986.