NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS
SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

WAM extensions for implementing higher order logic languages

Alexandros V. Tasos

Supervisors: **Panagiotis Rondogiannis,** Professor
Angelos Charalambidis, Postdoctoral researcher - NCSR Demokritos

ATHENS
SEPTEMBER 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Επεκτάσεις της WAM για υλοποίηση λογικών γλωσσών υψηλής τάξης

Αλέξανδρος Β. Τάσος

Επιβλέποντες: **Παναγιώτης Ροντογιάννης,** Καθηγητής
Άγγελος Χαραλαμπίδης, Μεταδιδ. ερευν. - ΕΚΕΦΕ Δημόκριτος

ΑΘΗΝΑ
ΣΕΠΤΕΜΒΡΙΟΣ 2016

BSc THESIS

WAM extensions for implementing higher order logic languages

Alexandros V. Tasos
S.N.: 1115201100085

SUPERVISORS: **Panagiotis Rondogiannis**, Professor
Angelos Charalambidis, Postdoctoral researcher - NCSR Demokritos

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Επεκτάσεις της WAM για υλοποίηση λογικών γλωσσών υψηλής τάξης

Αλέξανδρος Β. Τάσος

Α.Μ.: 1115201100085

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Παναγιώτης Ροντογιάννης**, Καθηγητής
Άγγελος Χαραλαμπίδης, Μεταδιδ. ερευν. - ΕΚΕΦΕ Δημόκριτος

ABSTRACT

In their publication, Charalambidis et al. propose a Higher-order version of Prolog with Extensional Semantics (HOPES). One of the suggestions in the paper for further research is to create a modification for WAM capable of running HOPES. Such an implementation is described in this thesis.

SUBJECT AREA: Logic programming
KEYWORDS: Higher order logic programming, Intermediate representation, Compilers, WAM, Extensionality

ΠΕΡΙΛΗΨΗ

Στη δημοσίευσή τους, οι Χαραλαμπίδης κ.ά. παρουσιάζουν μία εκδοχή της Prolog η οποία βασίζεται σε λογική υψηλότερης τάξης με Extensional Semantics (HOPES). Μία από τις προτάσεις της δημοσίευσης για περαιτέρω έρευνα είναι η δημιουργία μίας τροποποιημένης εκδοχής της WAM, ικανής να εκτελέσει τη HOPES. Μία τέτοια υλοποίηση περιγράφεται στην εργασία αυτή.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Λογικός προγραμματισμός
ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Λογικός προγραμματισμός ανώτερης τάξης, Ενδιάμεση αναπαράσταση, Μεταγλωττιστές, WAM, Επεκτασιμότητα

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Objective

The purpose of this thesis is to create an extension of the Warren Abstract Machine [14] (an abstract machine consisting of a memory layout and an instruction set tailored to executing Prolog programs) so as to improve the performance of extensional higher-order logic languages. More specifically, the work presented here is intended to provide an extended version of WAM for the programming language HOPES, developed by Charalambidis et al. [7].

The version of the WAM presented here does not cover the entirety of the HOPES language; there exist a few edge cases (which we'll describe later on) which the extended WAM is not able to execute and produce results that abide by the SLD-resolution algorithm presented for HOPES in [7].

1.2 Motivation

Although Prolog is the most well-known programming language with regards to the logic programming paradigm, an obvious limitation of it is the fact that it is first order. To remedy this, there have been attempts at incorporating higher-order features into logic programming, mainly λProlog [11], HiLog [3] and a framework developed by Wadge [13] called ”the definitional subset of higher-order Horn logic”. More recently, Charalambidis et al introduced the H programming language [7] and Koukoutos [9] introduced $polyHOPES$, an extensional higher-order superset of Prolog.

Higher-order logic programming is divided into intensional and extensional logic programming. In intentional higher-order logic programming, two predicates are considered equal if they have the same name, whereas in extensional higher-order logic programming two predicates are considered equal if they succeed for the same set of instances. The former category already demonstrates a couple of implementations [12] [6] (albeit partial), while the latter category currently only demonstrates an implementation by Charalambidis in Haskell [2]. In [7], a few optimizations to the implementation are proposed as part of a future work, one of which is to devise and implement a WAM-based implementation.

In 1983, David Warren [14] designed an abstract machine for the execution of Prolog, which consisted of a memory layout and an instruction set. This design is known as the Warren Abstract Machine and has become the *de facto* standard target for Prolog compilers and interpreters. Apart from the fact that it is a lower-level instruction set, additional optimizations have been proposed and implemented on it [5].

Given the popularity of the WAM and the research performed into it to optimize Prolog programs, it is appealing to construct one for higher-order logic programming. However, since the WAM is designed around the premise that Prolog programs will be translated into it, it lacks any concepts related to higher-order logic languages. As such, we need to incorporate concepts such as partial appication and higher-order variables into it in an efficient manner, while adhering to the design philosophies of the WAM as close as possible; this thesis demonstrates that the above is actually feasible and in an efficient manner.

1.3 Outline of this thesis

The rest of this thesis is organized as follows: In chapter 2, we introduce the WAM, its semantics and how a Prolog program is transformed to a WAM equivalent. In chapter 3, we present HOPES, its traits, as well as what programs it cannot support. In chapter 4, we present our extension to the WAM, its implementation details and examples of compilation. In chapter 5, we discuss our implementation against other implementations of the WAM (or any other instruction set intended for logic languages). Finally, in chapter 6, we present our conclusions and provide suggestion for future work.

2. HOPES

In this chapter, we introduce HOPES (Higher-order Prolog with Extensional Semantics), an extensional (that is, instead of considering two predicates equal if they have the same name, it considers two predicates equal if they succeed for the same for the same instances) higher-order language. We, then analyze the issues of using HOPES as-is to extend classical Prolog and, then, introduce $polyHOPES$, which is a strict superset of classical Prolog with extensional semantics.

2.1 An overview of HOPES

HOPES (Higher-order Prolog with Extensional Semantics) is an extensional developed by Charalambidis et al. [7]. The language features support for higher-order predicates, i.e. predicates that are allowed to receive other predicates as parameters. For instance, considering the following predicate (in a Prolog-esque syntax):

```
closure(R, X, Y) :- R(X, Y).
closure(R, X, Y) :- R(X, Z), closure(R, Z, Y).
```

And given the following fact database:

```
friends(jake, sarah).
friends(lina, jake).

friends(john, nathan).
friends(lester, nathan).
```

Then by specifying the goal `?- closure(friends, jake, X)`, we can find all transitively mutual friends of Jake:

```
X = lina
X = sarah
```

We are also allowed to apply currying to predicates. For instance, consider the predicates:

```
filter([], P, []).
filter([X|Xs], P, [X|Xs]) :- P(X), filter(Xs, P, Xs).
filter([X|Xs], P, Xs) :- filter(Xs, P, Xs).

unify(X, X).
```

Then, we can compute a list of all elements that do not unify with constants a, b, respectively, by giving the goals:

```
?- filter([a, e, d, a, c, b, e], unify(a), X).
X = [a, a]
?- filter([a, e, d, a, c, b, e], unify(b), X).
X = [b]
```

Since HOPES supports higher order variables, given the following clauses:

```
p(Q) :- Q(0), Q(s(0)).
```

The answer we receive, assuming the goal is `?- p(R)`, will be

```
R = {0, s(0)}
```

In [7] Charalambidis et al. do not provide a formal definition of HOPES. What they do, however, is provide a formal definition of a higher-order extensional language called H, then provide a means of tansforming HOPES programs to H equivalent ones.

Not all programs in classical Prolog are translatable to HOPES. For instance, given a binary relation P, we cannot construct a higher-order predicate $commutative(P)$ which succeeds if and only if P is a commutative relation. The culprit is the fact that it is impossible to express a universal quantification in H, and that negation in expressions is not supported. Thus, HOPES programs are only a strict superset of positive first order logic programming.

2.2 The higher-order language H

The H language defined in [7] is based on a type system that supports two base types: The booleans domain, which is denoted by o and the individuals domain (domain of data objects), which is denoted by ι. Composite types are partitioned into functional (assigned to function symbols and deonted by σ), predicate (assigned to predicate symbols and denoted by ρ) and argument (assigned to predicate parameters types and denoted by π). They are defined, respectively, as:

$$\sigma := \iota | (\iota \to \sigma)$$
$$\rho := \iota | \pi$$
$$\pi := o | (\rho \to \pi)$$

We will also give an abridged version of the syntax of H. Assuming that:

- X are variables (of type π or of type ι)
- c are constants (of type π or of type ι)
- f are functional symbols of every functional type $\sigma \neq \iota$

Then the following are the positive expressions of H:

1. Every predicate variable and predicate constant (with a type of π).
2. Every individual variable and individual constant (with a type of ι).
3. The constants `true` and `false` (with a type of o).
4. Given positive expressions $E_1, \ldots, E_n$ of type ι, $fE_1 \ldots E_n$ (with a type of ι).
5. Given positive expressions E_1 of type $\rho \to \pi$, E_2 of type ρ, $E_1 E_2$ (with a type of π).
6. Given argument variabel V of type ρ and positive expression E of type $\rho \to \pi$, $\lambda V.E$ (with a type of $\rho \to \pi$).
7. Given positive expressions E_1, E_2 of type π, $E_1 \wedge_p iE_2, E_1 \vee_\pi E_2$ (with a type of π).
8. Given positive expressions E_1, E_2 of type ι, $E_1 \approx E_2$ (with a type of o).
9. If E an expression of type $o$and V argument variable of type ρ, $\exists_p V\ E$ (of type o).

The clausal expressions of H are as follows:

- If p a predicate constant of type π and E a closed positive expression of type π, then $p \leftarrow_\pi E$ is a clausal expression of H, also called a program clause.
- If E a closed positive expression of type π, then $false \leftarrow_o E$ is a clausal expression of H, also called a goal clause.

A convenient property of H is that it every HOPES program can be transformed into an equivalent program into H with a trivially derived set of transformations. For example, given:

```
closure(R, X, Y) :- R(X, Y).
closure(R, X, Y) :- R(X, Z), closure(R, Z, Y).
```

We can convert this HOPES program into H (and the resulting program will look lke the following):

$$closure \leftarrow_\pi \lambda R.\lambda X.\lambda Y.(RXY)$$
$$closure \leftarrow_\pi \lambda R.\lambda X.\lambda Y.\exists z\ ((RXZ) \wedge (RXY))$$

An example of an ill-defined program in HOPES is the following:

```
foo(H) :- H = a, H(a).
```

The above program will be converted into H as follows:

$$foo \leftarrow_{\pi} \lambda H.((H \simeq a) \wedge (Ha))$$

Given rule 6, we deduce from $(H \simeq a)$ that H is of type ι. In addition, given rule 3, we deduce that in the expression (Ha), the type of the experssion should be π and the type of H should be $\rho \to \pi$. We have reached a contradiction, hence the above program is not a legal H program.

2.3 Limitations of H (and HOPES)

When it comes to extending the classical Prolog, although H satisfies the properties of completeness and soundness, there exist a few issues that make it impossible to integrate H and, subsequently, HOPES into a classical Prolog implementation.

The first of these issues is name aliasing. In classical Prolog definition of predicates with different arities is allowed. There is no confusion during the runtime of which predicate should be invoked, since there is no partial application. However, in a higher-order setting, partial applications may introduce aliases. For example:

```
p.
p(0).
q(X) :- X(p).
```

There is an ambiguity as to what the parameter p represents (the predicate p/0, the predicate p/1 or the structure p/0). HOPES works this ambiguity around by disallowing aliased names. This strategy, however, is incompatible with programs that are written in classical Prolog.

Another issue is that we cannot tell whether or not the parameter p is an partial application, that is whether or not the argument p is actually the predicate $p/1$ or a partially applied $p/2$, $p/3$, etc. HOPES offers a simple partial application mechanism, which ignores arity. Thus, creating a higher-order language that allows classic Prolog porgrams without any modifications should have this mechanism redesigned.

A third issue is the fact that H offers no parametric polymorphism. That is, consider the `closure` predicate defined in the previous section. Then, the type inference mechanism of H would infer that `closure`'s type is $(\iota \to \iota \to o) \to \iota \to \iota \to o$. However, this type is not the most general possible type (we would like to he able to have `closure` handle relations on any argument type.

Due to these, we will introduce $polyHOPES$ [9], introduced by Koukoutos, which explicitly deals with the issues above.

2.4 polyHOPES

$polyHOPES$ is an extensional higher-order language that is also a strict superset of Prolog. Due to the fact that the subset of Prolog described in [1] does not introduce any concept of operators or, for instance, clauses with no predicate name (such as `:- op(500, yfx, '-')`, we can assume that there exists a lowering pass that transforms regular Prolog programs to programs described by the syntax in [1].

Thus, the grammar of ”lowered” $polyHOPES$ will not contain operators or anonymous predicates (as these can be emulated), hence it will be as follows:

- The syntax of expressions will be:

$$\begin{aligned} E &::= V|c|num|pred\ c[/m]|E(E1,...,En)||list|(E) \\ list &::= []|[E1,...,En[|(V|list)]] \end{aligned}$$

- While the syntax of sentences and programs will be:

$$\begin{aligned} Sentence &::= Clause|Goal|Dir \\ Program &::= Sentence^{+} \\ Clause &::= Head[ImplBody]. \\ Head &::= AppHead|c/n \\ AppHead &::= c|AppHead(E1,...,En) \\ Impl &::=: -| < - \\ Body &::= E \end{aligned}$$

The three additional features, compared to the Prolog subset defined in [1] are the following:

- The fact that a partial application can be constructed out of an existing variable, as well as called (assuming the type is correct).
- The `pred` keyword, which defines a partial application of a predicate (so as to avoid ambiguity between a partial application between a predicate p/m and defining a functional term of the form $p(E1, \ldots, Em)$.
- The fact that the arguments in predicates can be grouped together (for instance `foo(X)(Y, Z) :- ...`). This is actually something we do not need to take into consideration; this syntax only ensures that the first partial application must consist of exactly k_1 parameters (where k_1 is the number of parameters in the syntactically first set of parentheses), the second partial application must consist of exactly k_2 parameters (k_2 is defined in the same manner as for k_1) and so on. This syntax does not alter the fact that the predicate `foo` has an arity of 3.

We also need to assume that each higher-order variable will be a:

- First-order variable
- Higher-order variable of an arbitrary order, as long as it is located on the head of a clause. Thus all higher order variables which will be requested to be generated will be located either on the clause body (and not on the clause head) or on the goal.

The rationale for this limitation will be discussed in the next chapter, where we introduce our modified WAM.

We will see in the next chapter how to transform this subset of $polyHOPES$ to the WAM and what modifications to the WAM we need to perform in order to actieve this.

3. A WAM PRIMER

The Warren Abstract Machine (WAM) is an abstract machine designed by Warren in 1983 [14] with the intent of creating an instruction set and a memory layout model for the execution of Prolog programs. The instruction set of the WAM consists of 49 instructions, as well as 5 different memory segments, which shall be explained later on in detail.

The purpose of this chapter is to provide a breve introduction to the WAM. The concepts and explanations given are derived from [1] and presented in such a way so that someone unfamiliar with the WAM can be brought up to speed. For those who wish to delve further into the WAM itself should consider referring to [1] and [14].

The WAM is a register-based ISA, that is, unlike stack-based architectures, whose instructions manipulate a stack as well its elements when executed (such as the Java bytecode specification [10]), operate on a finite or an arbitrarily large amount of registers. In the case of the WAM it is presumed that the abstract machine can make use of an arbitrarily large amount of them. All registers are also callee-saved, that is any clause can clobber an unspecified amount of registers (thus the calling clause has to store the register values that need to be preserved during a function call to a stack).

The naming convention used in [1] to refer to registers is to use the labels `X1, X2, ..., Xn`. When passing arguments between clauses, the first k registers are used to store the arguments. As such, the first k registers are also aliased to the labels `A1, ..., Ak`.

The abstract machine is also expected to be able to reserve a finite number of slots on the stack in order to deal with preserving register values etc. If, during the compilation of a clause, it is deemed that m slots need to be allocated on the stack, then the labels `Y1, ..., Ym` shall refer to their corresponding slots. When dealing with reading from and writing to a stack slot for each instruction, the abstract machine shall perform the appropriate memory access on the stack under the hood (hence, when it comes to the WAM ISA, stack variables and registers are treated in the vast majority of cases in the same way).

3.1 Creating terms

In order to perform unification in the WAM, we first need to find a suitable representation for terms in memory (that is variables and functors with any arity possible). Our ideal memory representation and allocation scheme would be one that fulfills all of the following criteria:

- It allocates only the absolute minimum of memory every time.
- It allocates objects in memory as close as possible address wise to each other (so that the issue of memory fragmentation is minimized and cache locality turns out to be as much of a free lunch as possible).

- If possible, we would like to get away with not using a GC at all and just get away with simple methods of performing memory management.

The memory layout presented in [1] fulfills all of the above.

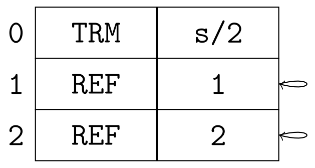
The heap is implemented as a growable vector of memory cells. Two state registers are also specified, `H` and `S`, where the former indicates the end of the heap (hence if the heap was represented as a growable vector, the state register `H` would correspond to the vector size and not the capacity, while the `S` status register is essiential in unification, as we will see below.

Memory cells are also typed; each memory cell can have one of the following types:

- Term name - arity tuple, such as $f/2$ (`TRM`)
- Reference to the beginning of a structure: Term name followed by its arguments (`STR`).
- Refererence to another variable (`REF`).
- Reference to a list term (`LIS`). The list term is, essentially, a term with an arity of 1 optimized to consume less memory space than any other term with the same arity.
- Reference to a constant (term with 0 arity) (`CON`).

Due to the way the instructions are implemented in [1], references on the heap will point from higher addresses to lower addresses (except for the case of unbounded variables, which we will be discussed a bit later on).

Representing the term `s(X, Y)` on the heap, for instance, is shown in figure 3.1.



0	TRM	s/2
1	REF	1
2	REF	2

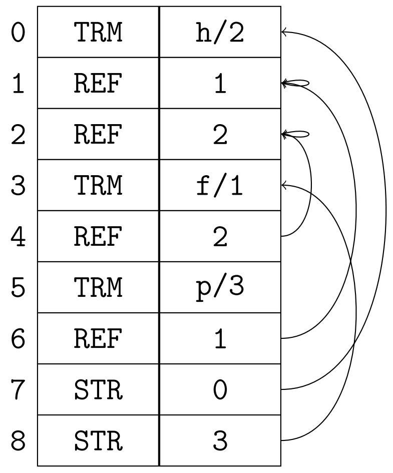
Figure 3.1: In-memory representation of `s(X, Y)`

The reason there exists a self loop in the aforementioned figure is because a variable in WAM can be either:

- Unbounded, i.e. not yet unified with anything (in which case it will point to itself).
- Bounded, i.e. unified with another variable or a structure (in which case it will point to the relevant cell).

Variable unification in the WAM is represented as a disjoint-set data structure [4] [1]. That is, given two variables allocated as two distinct cells on the heap, they will be considered as unified if they belong to the same disjoint set (i.e. we can find a heap cell h such that subsequent dereferences of the two cells will yield the address of h).

An example that clarifies the distinction between unbounded and bounded variables for the term `p(Z, h(Z, W), f(W))` is shown in figure 3.2:



0	TRM	h/2
1	REF	1
2	REF	2
3	TRM	f/1
4	REF	2
5	TRM	p/3
6	REF	1
7	STR	0
8	STR	3

Figure 3.2: In-memory representation of `p(Z, h(Z, W), f(W))`

Z and W are represented by cells 1 and 2. Should either of them become bounded during the program execution, all other (direct/indirect) respective references to them will also become bounded. In addition, heap cells 6 and 4 bind to heap cells 1 and 2, respectively, which are, as we just mentioned, the locations of the corresponding unbound variables Z and W.

To create structures on the heap, we use the following 5 instructions:

1. `put_structure f/n, Xi`: Create a new term on the heap
2. `put_list Xi`: Create a new list term on the heap
3. `set_variable Xi`: k-th term parameter is unbounded and `Xi` points to it.
4. `set_value Xi`: k-th term parameter points to where `Xi` currently points at.
5. `set_constant c`: k-th term parameter points to the constant `c`.
6. `set_void n`: Starting from the k-th term parameter, the next n term parameters are unbounded (deals with the underscore parameter).

The first instruction (`put_structure`) simply creates a new functor on the heap by allocating $n+1$ memory cells, setting the type of the first cell to be of type `STR`, then making the `S` status register point to the second cell of the heap (`put_list` works in the same number, except for the fact that no STR cell is created). Afterwards, all set instructions perform their corresponding operation to the k-th term, which the status register S points to, then increment the S status register (except for the set_void instruction, which will increase the value of S by a variable amount).

As an example, creating the term f([X, a], _, X, g(Y, X)), would result in the following code to be emitted:

```
put_list X1           % X1 = [_|_]
set_constant a        % X1 = [a|_]
set_constant []       % X1 = [a]
put_list X2           % X2 = [_|_]
set_variable X3       % X2 = [X3|_]
set_value X1          % X2 = [X3|X1], X1 = [a]
put_structure g/2, X4 % X4 = g(_, _)
set_variable X5       % X4 = g(X5, _)
set_value X3          % X4 = g(X5, X3)
put_structure f/4, X6 % X6 = f(_, _, _, _)
set_value X2          % X6 = f(X2, _, _, _)
set_void 1
set_value X3          % X6 = f(X2, _, X3, _)
set_value X4          % X6 = f(X2, _, X3, X4)
```

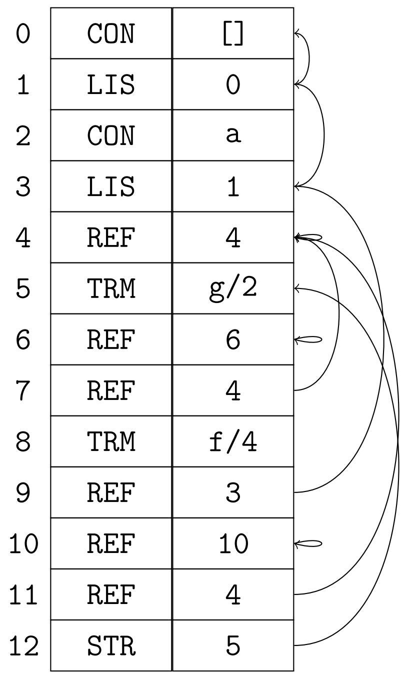
When executed (the instructions are executed in a sequential order), the heap should math the layout of figure 3.3.

3.2 Unification

Unification of variables requires adding the concept of read and write mode: If a term unification is performed against an unbounded variable, then the term corresponding to the unified variable has to be reconstructed on the heap. Determining which mode we should enter is performed in the get_structure and get_list instructions, by checking if Xi is bounded or not and, then, enter read or write mode accodringly.

To perform unification, the WAM ISA introduces the following instructions:

1. get_structure f/n, Xi
2. get_list Xi
3. unify_variable Xi
4. unify_value Xi
5. unify_constant c



0	CON	[]
1	LIS	0
2	CON	a
3	LIS	1
4	REF	4
5	TRM	g/2
6	REF	6
7	REF	4
8	TRM	f/4
9	REF	3
10	REF	10
11	REF	4
12	STR	5

Figure 3.3: In-memory representation of `f([X, a], _, X, g(Y, X))`

6. `unify_void n`

Since these instructions have semantics that correspond to the semantics of the instructions mentioned above, we shall not delve into them.

For instance, given the predicate `foo(f(X), g(X, a)).` assuming that the argument parameters are `X1` and `X2`, the following code would be emitted:

```
foo:
    get_structure f/1, X1 % X1 = f(_, _)
    unify_variable X3     % X1 = f(X3, _)
    unify_variable X4     % X1 = f(X3, X4)

    get_structure g/2, X2 % X2 = g(_, _)
    unify_value X3        % X2 = g(X3, _)
M1: unify_constant a      % X2 = g(X3, a)
```

There also exist two additional unification intsructions, called `get_variable` and `get_value`. Both of these instructions take a source and a destination variable as arguments. The first simply copies the contents of the source to the destination. The second one performs recursive unification (which can also be an occurs-check unification). To perform a recursive unification, the WAM ISA reserves a memory segment called the PDL, which is, essentially, just a scratch buffer used by the depth-first search unification algorithm. The purpose of this instruction is to handle cases where performing an arbitrary amount of heap allocations in one instruction is necessary, such as when unifying the terms $f(a(X,b),Y), f(Z,h(Z,a))$, where it turns out that $Z = a(X,b), Y = h(Z,a)$ and X is a (possibly) unbound variable.

3.3 Predicate invocation

Just like in procedural languages, when a predicate is invoked in the WAM, a stack frame is created, where it stores its own registers to be saved during inter-predicate calls, as well as status registers so that it can return control to the predicate that called it.

The stack frame of a predicate in the WAM has the layout described in figure 3.4. There are two registers we need to consider here: The frame pointer(referred to as the environment register (E) in the WAM), which points to the beginning of the stack frame and the instruction pointer, which points to the current instruction being executed (refered to as the program counter in the WAM).

It is not necessary for a predicate invocation to allocate a stack; a leaf predicate (that is, a predicate that calls no other predicates can simply perform its execution and return.

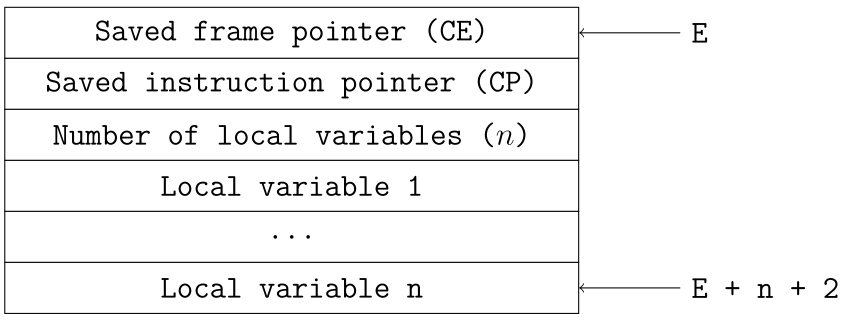


Figure 3.4: Stack frame layout

The end of the stack frame need not be saved; it can be inferred from the frame pointer E and the number of local variables that it is $SP = E + n + 2$.

The following instructions deal with calling and parameter passing:

- Calling and returning:
  - `call p/n`: Calls the predicate `p/n`.
  - `execute p/n`: Assuming the predicate `p/n` is the last one in the current clause, performs a tail call optimization by passing the arguments to the corresponding registers and then trimming the stack so that `p/n` takes control.
  - `allocate n`: Allocates a stack frame with n local variables. First instruction that must be executed from a predicate as soon as it's called (if it is necessary to allocate a stack frame).

  - `deallocate`: Cleans up the current stack frame (if the predicate did alloate one) and returns to the parent.

- Parameter passing:
  - `put_variable Vn, Xi`: Binds `Vn, Ai` to a new (unbounded) heap cell (akin to its one argument counterpart).
  - `put_value Vn, Xi`: Assigns the contents of `Vn` to `Ai`.
  - You can also use `put_structure, put_list`, as well as `put_constant` (which has the same behaviour as its `set` counterpart).
  - `get_variable Vn, Xi`: Assigns the contents of `Ai` to `Vn`.
  - `get_value Vn, Xi`: Performs a deep unification of `Vn, Ai`. A special segment of memory (called the PDL is reserved for deep unification). Notice that deep unification can also be performed by using, for instance, an occurs-check.
  - The instructions `get_structure, get_list`, and `get_constant` can also be used as well.

For instance, the rule `p(X, Y) :- q(X, Z), r(Z, Y).` would be translated into the following:

```
p/2:
    allocate 2
    get_variable Y1, A1
    get_variable Y2, A2
    put_value Y1, A1
    put_variable Y3, A2
    call q/2
    put_value Y3, A1
    put_value Y2, A2
    call r/2, 0
    deallocate
```

A tail-call optimized version of the above would be the following:

```
p/2:
    allocate 2
    get_variable Y1, A1
    get_variable Y2, A2
    put_value Y1, A1
    put_variable Y3, A2
    call q/2
    put_value Y3, A1
    put_value Y2, A2
    deallocate
    execute r/2
```

3.4 Backtracking

To implement backtracking in WAM, we need to introduce the concept of a choice point. A choice point is a memory segment that contains all the relevant information so that should the current goal be unattainable, the control flow will use the choice point data to perform a reversion. When allocated, the layout of a choice point frame adheres to the one described in figure 3.5. A keen observer will notice the trail pointer in the frame layout; we will ignore this saved register for now.

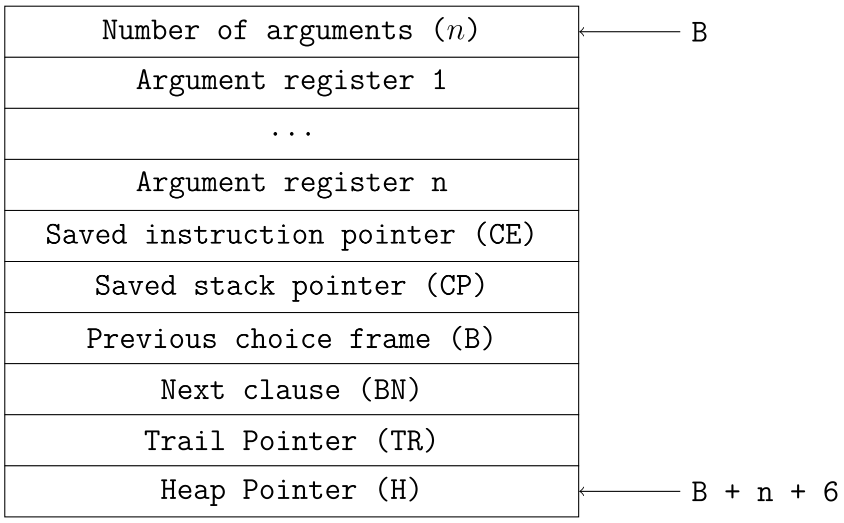


Figure 3.5: Choice point frame layout

Thus, whenever we enter a predicate with multiple choices for the first time, we create a choice point by saving the relevant information in it, then we perform the typical stack allocation boilerplate.

While this concept might look similar to stack-based exception handling, there is a significant catch: Even after returning from the predicate, we should be able to unwind the program execution to the point after the choice point has been created.

For instance, consider the following Prolog snippet:

```
a :- b(X), c(X).
b(X) :- e(X).
e(X) :- f(X).
e(X) :- g(X).
c(1).
f(2).
g(1).
```

A choice point is created when `e` is invoked (figure 3.6), but if we discard the choice point because `f(2)` succeeds (figure 3.7), we'll abort with failure instead of returning with the correct answer, which is `X = 2`.

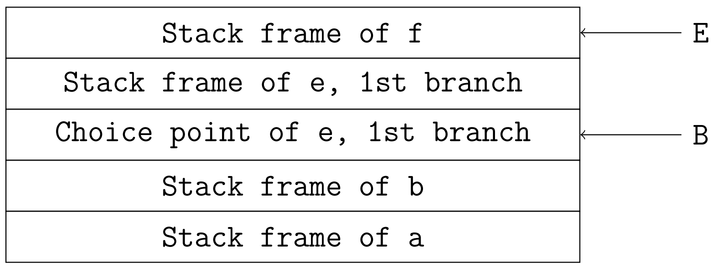


Figure 3.6: Creation of choice point for e

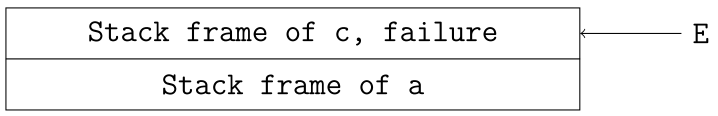


Figure 3.7: Deallocation of choice point of e yields no answer

To remedy this, WAM adds the following additional rules regarding choice points:

- When allocating a stack frame, allocate it after the latest choice point.
- When deallocating a stack frame, don't deallocate its corresponding choice point. Instead, just deallocate your stack frame and leave the choice point on its own.
- Upon failure, the choice point will resurrect all stack frames that were deallocated and drop all stack frames allocated after it, because they correspond to an execution that must be rewound.

After applying these rules, when a choice point is created when `e` is invoked (figure 3.8), but we keep the stack frame of predicate `b` allocated even when it has finished executing (figure 3.9). Thus, even if `b` is deallocated and then `c` is called (figure 3.10), we'll be able to backtrack successfully when `f` is called (which will fail for `f(1)`), we will be able to yield the correct answer by backtracking to the second branch of predicate `e` (figure 3.11 , and that answer is `X = 2`.

The WAM has 3 choice point instructions:

- `try_me_else L`: Allocate and initialize a new choice point such that in case of a backtrack, the instruction at label L will be executed.
- `retry_me_else L`: Reset all information corresponding to the current choice point and set the instruction upon backtrack to be at the label L.
- `trust_me`: Reset all information corresponding to the current choice point and deallocate it.

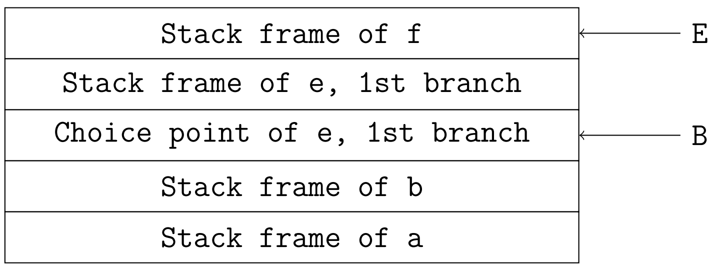


Figure 3.8: Creation of choice point for e

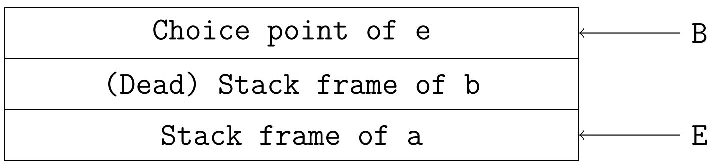


Figure 3.9: Creation of choice point for e, after applying choice point rules

For example, for the following program:

```
p(X, a).
p(b, X).
p(X, Y) :- p(X, a), p(b, Y).
```

Is translated into the following:

```
p/2: try_me_else L1
     allocate 0
     get_constant a, A2
     deallocate
L1:  retry_me_else L2
     allocate 0
     get_constant b, A1
     deallocate
L1:  trust_me
     allocate 1
     get_variable Y1, A2
     put_constant a, A2
     call p/2
     put_constant b, A1
     put_value Y2, A2
     call p/2
     deallocate
```

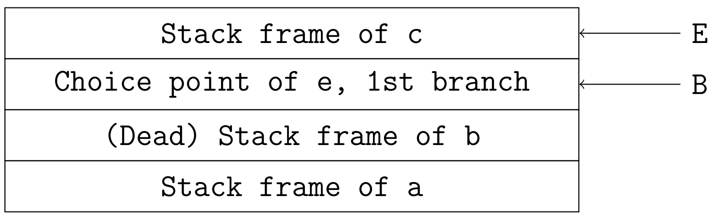


Figure 3.10: Choice point of e is still allocated

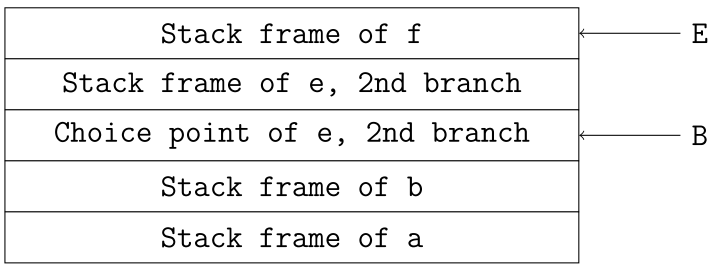


Figure 3.11: Aborting properly resurrects stack frame of b

There is still an additional issue we have to deal with: When unwinding to a choice point, we also have to reset all variables that were modified by the instructions executed between the choice point creation and the instruction that forced the unwinding to occur. This is defined as the trail in the WAM, which stores all changes in variable values. The trail is, essentially, a memory segment where, whenever a variable is changed, the old value is pushed to the top of it, while when it is necessary to backtrack, all values from the top of the trail to the choice point trail pointer (TR) are stored. We can do better, however: According to [], instead of storing all values, we only need to store all variables that were initially unbound when modified, thus significantly reducing the size of the trail.

The trail defines two functions, called $trail$ and $unwindTrail$, which append a stack or heap address to the trail and unwind the trail up to a certain point, respectively. These functions are defined in figure 3.12.

```
Function trail($addr$, $higherOrder$):
    if $addr$ < HB or (H < $a$ and $a$ < B) then
        TRAIL[TR] = $addr$;
        TR = TR + 1;
    end
Function unwindTrail($addr1$, $addr2$):
    foreach $i$ = $addr1$ to $addr2 - 1$ do
        STORE[$val$] = (REF, $val$);
    end
return
```

Figure 3.12: Trail utility operations

3.5 Additional constructs

This is not a full-featured presentation of the WAM. There are still features not presented here, such as cutting (implementation of the cut in classical Prolog), as well as indexing (which is, essentially, a method of minimizing the number of choice points we are supposed to create by peeking at an argument register’s tag and value. A reader willing to learn about the implementations of cut and indexing is free to refer to [1].

4. EXTENDING THE WAM TO EXECUTE POLYHOPES PROGRAMS

As it stands right now, WAM is not a fit candidate for translating *polyHOPES* programs into it. There are two reasons for this:

- We are currently lacking any concept of callable variables (such as function pointers, as defined by the C standard [8]). The `call` and `execute` instructions only allow term names as invocation targets.
- Even if we did have a concept of callable variables, we also require the ability to represent the result of a partial application. For instance, consider the following *polyHOPES* program:

  ```
  foo(P)(A, B, C) :- ap(P(A))(B), ap(P(A))(C).
  ap(Q)(X) :- Q(X).
  bar(s, t).

  ?- foo(pred bar)(s, t, u).
  ```

  If we had to transform this program into a WAM-esque one, it would be mandatory to represent the fact that when the predicate *foo* is being executed, two subsequent partial applications are performed on the predicate *P*, with corresponding arguments *A* and *B*.
- We are also lacking a means of representing the set corresponding to a higher order variable. For instance, in a hypothetical scenario where we had support for higher order variables, while executing the following program:

  ```
  map(P)([], []).
  map(P)([X|Xs], [Y|Ys]) :- P(X, Y), map(P)(Xs, Ys).

  ?- map(X)([2, 3, 5, 7, 11], [1, 2, 3, 4, 5]).
  ```

  The set S corresponding to the higher-order variable X during each resursive invocation of *map/3* would be $\{(2,1)\}$, then $\{(2,1),(3,2)\}$, then $\{(5,3)\}$, etc.
- When backtracking, it is necessary to also revert the current set corresponding to a higher-order variable, if necessary. For instance, in the following program:

  ```
  foo(X)(A, B) :- X(A), fail.
  foo(X)(A, B) :- X(B).

  ?- X(s), foo(X)(t, u).
  ```

When the predicate *foo* is executed, the higher-order variable *X* is first set to $\{s, t\}$, then the *fail* predicate is invoked, which will cause a backtracking (since the *fail* predicate is obviously non-existent) hence *X* will be reverted to $\{s\}$ and, finally, when the second branch is executed, *X* will be assigned to $\{s, u\}$.

In order to extend the WAM so that it can also run programs written for HOPES, we have to incorporate partial application, as well as the introduction of higher order variables and treating them as callables. We also need to ensure that classic Prolog programs still execute as intended when these extensions are added.

4.1 Introducing partial application support

An approach of introducing partial application support in WAM is by introducing two new heap cell types, called *APP* and *APPSTR* (derived from ”partial application” and ”partial application structure”, respectively).

Just like the *REF* and *STR* types, a *APP* cell is parameterized by an address value, which points to the heap where the partial application is stored. An *APPSTR* cell, on the other hand, is parameterized by an integer n, which is the number of parameters that have been passed to the partial application.

A partial application is represented on the heap as follows: Let n be the number of arguments to a partial application. Then a sequence of $n + 2$ contiguously-allocated cells represent the partial application allocation as follows:

- The first heap cell parameter has a tag of *APPSTR*, along with the value n. If $n = 0$, then the partial application is, essentially, a no-op partial application.
- The second heap cell parameter can have a tag of type *APPSTR* (to denote a partial application applied to another partial application), *TRM*, (along with the corresponding predicate name and arity, such as *pred/3*), *REF* or *HOV* (higher-order variable), which we shall delve into later on.
- For $3 \leq k \leq n + 2$, the k-th cell corresponds to the $n - 2$-th parameter in the partial application.

For instance, consider the following *polyHOPES* program:

```
ap(X)(Y) :- X(Y).
q(X, Y).

?- ap(q(a))(b).
```

When passing the first parameter to the predicate *ap*, the heap layout, as well as the value of the register *X1* will look like the one depicted on figure 4.1.

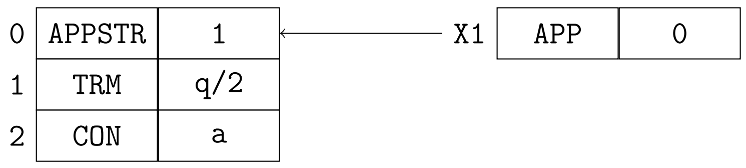


Figure 4.1: Heap representation of partial application `q(a)`

We will also demonstrate an example of chained partial application. Given the following *polyHOPES* program:

```
ap(P)(X) :- P(X).
ap2(P)(X, Y) :- ap(P(X))(Y).
q(A, B, C).

?- ap2(q(a))(b, c).
```

When passing the first parameter to the predicate *ap*, the heap layout, as well as the value of the register *X1* will look like the one depicted on figure 4.2:

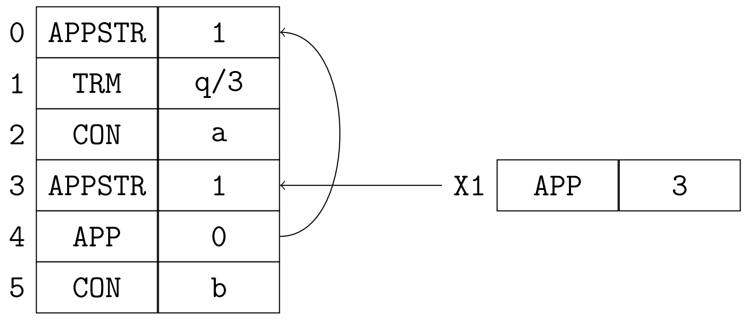


Figure 4.2: Heap representation of chained partial application `(q(a))(b)`

An astute observation is that it may be necessary to also retain the number m of parameters in the final predicate (where $n \leq m$), so that an error is raised if the partial application is called with a superfluous or an insufficient number of arguments. However, this is actually unnecessary: If more than n arguments were passed to an n-ary predicate, higher-order variable or an existing partial application of other, the *polyHOPES* type system would have caught it for us in the first place. Additionally, if an n-ary predicate, higher-order variable or existing partial application is called with strictly less than n arguments, then the type system will also reject it.

We have introduced all of the appropriate heap concepts for partial application (barring higher-order variables). We also need to introduce commands to invoke partial applications. We, thus, define the following commands:

- `put_application m Yi, f/n`: Constructs a partial application of predicate `f/n` with an arity of m on the heap and stores its address into variable `Yi`, along with a tag of *APP*.

Just like when allocating structure types on the heap, putting the partial application arguments is performed with the usual `set` instructions.

To construct the partial application into the heap, the algorithm shown on 4.3 is performed:

```
HEAP[H] = (APPSTR, m);
HEAP[H + 1] = (TRM, f/n);
S = H + 2;
Yi = (APP, H);
H = H + m + 2;
P = P + $instrSize$(P);
```

Figure 4.3: Partial application of predicate

- `put_application m Yi, Yj`: Constructs a partial application of the value pointed to by the variable Yj with an arity of m on the heap and stores its address into variable `Yi`, along with a tag of *APP*.

  There is a culprit we have to mitigate, though: If Yj is an unbound stack variable, we have to bind it to a newly allocated variable on the heap, otherwise, there is a hazard of introducing a reference from the heap to the stack, which is forbidden by the WAM. As we will see later on, this hazard will be introduced when we delve into higher-order variables, thus it is mandatory that we deal with it at this point.

  Just like when allocating structure types on the heap, putting the partial application arguments is performed with the usual `set` instructions.

  To construct the partial application into the heap, the algorithm shown on 4.4 is executed.

- `call_variable Xi, N`: Similarly to `call`, it invokes the variable `Xi`, with N stack variables remaining on the caller's frame. The variable Xi must be of tag of either `APPSTR`, `REF` or `HOV`; if it's not, an error is issued (although this case will never happen, since it will be caught by the type checker.

  The `call_variable` command is, also, required to extract the arguments from the partial application chain and apply them in a right-to-left order (arguments of first partial application in a chain correspond to the last arguments in the invocation, etc).

  We will assume that the partial application is applied to a predicate (and describe later on the case for a higher-order variable). The algorithm for performing the invocation is as follows in 4.5 (where $ExecuteCall$ can be assumed to be the function that handles the `call f/n`.

- `execute_variable Xi`: Similar to `call_variable`, except that it replaces the stack frame of the caller with the callee's one.

```
$addr$ = $deref$(E + j + 1);
if $addr$ < $E$ then
    HEAP[H] = (REF, H);
    H = H + 1;
    $bind$($addr$, H);
    $addr$ = H;
end
HEAP[H] = (APPSTR, m);
HEAP[H + 1] = $addr$;
S = H + 2;
H = H + m + 2;
Yi = (APP, H);
P = P + $instrSize$(P);
```

Figure 4.4: Partial application of register or stack variable

For example, consider the following program:

```
p(a, b, c).
ap(P)(A, B) :- ap(P(A))(B).
ap(P)(A) :- P(A).

?- ap2(pred p(a))(b, c).
```

A WAM translation that uses the concepts we just defined is the following:

```
p/3:  get_constant a, X1
      get_constant b, X2
      get_constant c, X3
ap/3: put_application 1 X1, X1
      set_variable X2
      put_variable X3, X2
      execute ap/2
ap/2: put_application 1 X1, X1
      set_variable X2
appl: execute_variable X1
main: put_application 2 X1, p
      set_constant a
      put_constant b, X2
      put_constant c, X3
      execute ap/3
```

When the `appl` label is reached, the layout of the heap and registers will look like the representation on figure 4.6.

```
$args$ = [];
$addr$ = HEAP[Xi];
while true do
    $addr$ = $deref(addr)$;
    switch val do
        case (REF, _) To be discussed later on
        case (HOV, _) To be discussed later on
        case (APP, m)
            $addr$ = addr + 1;
            $args$ = [HEAP[$addr$ + 2], ..., HEAP[$addr$ + m + 2]] + $args$;
            continue;
        end
        case (TRM, f/n)
            foreach i in 1 .. len(args) do
                Xi = $args[i]$;
                $ExecuteCall$(f/n, N);
            end
        end
    endsw
end
```

Figure 4.5: Procedure for calling a variable

4.2 Integrating higher order variables

An issue that has not been discussed yet is the fact that we have not settled on what should happen in a top-down implementation of *polyHOPES* when we encounter a call to an unbounded or higher-order variable. Luckily for us, this issue can be answered by referring to the definition of SLD-resolution of the language H, as demonstrated in [7]. From the SLD-resolution of H, we can determine that:

- If an unbound variable is called, then we need to create a new higher order variable to the heap and make the former point to the latter. The arity n of that variable is equal to the number of arguments passed to it.
- If a higher-order variable of arity n is called, then a list of the parameters that were called have to be stored as a tuple alongside the variable, so as to be displayed as a set when the solution is found and presented to the user.
- It must be possible to remove argument tuples from the higher-order variable should it be deemed necessary due to backtracking in a last-to-first manner(stack-esque).

Consider, for example, the HOPES program:

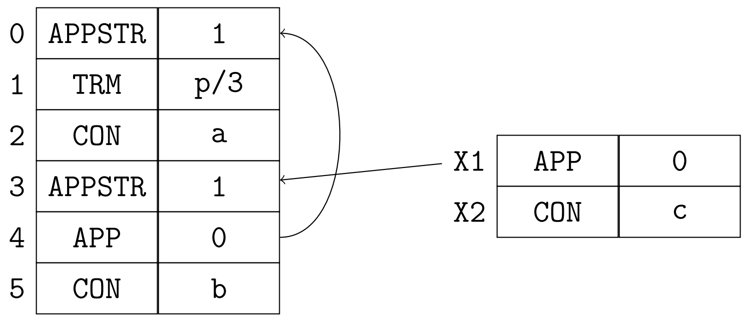


Figure 4.6: Heap representation of a more complex partial application

```
closure(R, X, Y) :- R(X, Y).
closure(R, X, Y) :- R(X, Z), closure(R, Z, Y).

?- closure(Q, a, b).
```

Then, by applying the SLD-resolution algorithm, we can get an arbitrary number of answers for the program, such as $\{(a, b)\}$, $\{(a, Z_1), (Z_1, b)$ etc. A hypothetical top-down implementation that executed the above program would first yield the answer `Q = {(a, b)}`, then backtrack from the first branch to yield the answer `Q = {(a, Z1), (Z1, b)}` and so on (hence a successful implementation of higher-order variables should not only allow the appending of tuples, but also the removal of them).

For a representation to be suitable for dealing with higher-order variables, it should, preferably, comply to the following:

- It should be simple to implement from an implementor's point of view.
- Adding and removing of items should be done in a stack-esque manner.
- It should be easy to integrate to the WAM memory layout (i.e. it should not require any additional memory segments to be defined for the WAM or, if so, those new memory segments should be minimized).
- It should minimize the amount of memory required.

Given these criteria, we come to the conclusion that a linked-list based stack implementation is the best answer. Someone could also implement a more sophisticated data structure, such as a linked hash set (hash map that retains the insertion order of elements), so that duplicate tuples (tuples whose heap cell contents are equal for each n-th cell) will be removed. Should they decide to do so, however, they will start running into issues: Since a hash set is usually backed by a resizable contiguous array [4], performing a resize on the hash map will result in heap fragmentation. We could introduce more sophisticated logic to the WAM memory allocation scheme, but, unlike our decision, it will complicate the allocation and freeing of cells from the heap. We may need to introduce a garbage collector in the process, whereas simply using a linked linked-list backed stack only requires a stack-esque allocation scheme.

Thus, we shall introduce two new heap cell types to implement higher-order types: `HOV` and `HOVATY` (standing for ”higher-order variable” and ”higher-order variable arity” respectively). A `HOV` cell is parameterized by an address value, while a `HOVATY` type is parameterized by an integer that denotes the arity of the higher-order variable. An astute observer will remark that cells of type `HOVATY` are unnecessary, since we can always know the eventual number of parameters from a partial application and the type system of *polyHOPES* will reject a program if the invocation arity differs for the same higher-order variable. However, when it comes to displaying the answer to the user, we will not be able to tell the number of arguments. Thus, although superfluous during normal execution, `HOVATY` cells are important for displaying answers back to the user.

A higher-order variable is represented on the heap as follows: Let n be the variable's arity. Then, a sequence of 2 contiguously-allocated cells represent the higher-order variable as follows:

- The first cell has a tag of `HOVATY` and a value of n.
- The second cell has a type of `REF`, whose address is greater than or equal to this cell's address on the heap. In case of equality, the higher-order variable is deemed to be an empty relation, otherwise it is deemed to me a non-empty relation.

A higher-order variable relation tuple is represented on the heap as a contiguously-allocated segment of $n + 1$ heap cells (assuming that the higher-order variable has an arity of n). The contents of the cells are as follows:

- The first cell has a type of `REF`, which references the first cell of the previous tuple stored in the higher-order variable relation. If the relation is unitary, then the reference points to the second cell (reference to last element in the relation) of the higher-order variable representation.
- The rest n types represent the argument tuple.

A heap cell, register or stack variable with a cell type of `HOV` simply refers to a higher-order variable.

For example, if `P` is a higher-order variable with arity 2 which is stored in register `X3`, then assuming that $\{(a, b), (c, d), (e, f)\}$ (and they have been added in the order they are displayed on this thesis), then the in-memory representation should match that of figure 4.7.

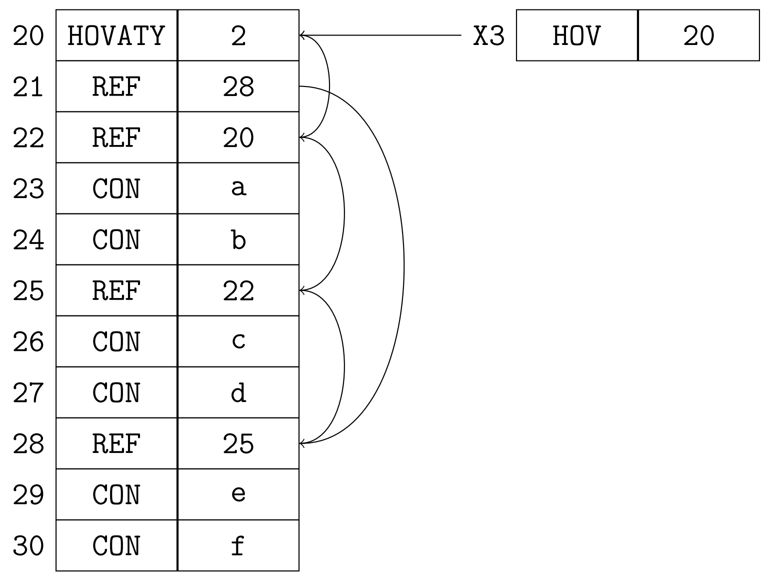


Figure 4.7: Heap representation of higher-order variable relation

What remains to be done is to incorporate the stack operations into the relevant parts of the WAM implementation. There are two cases we need to consider:

- When executing a `call_variable` instruction, we have to possibly initialize an unbound variable on the heap to one binding to a higher-order variable on the heap, then perform a push operation onto the stack.

  We will not delve into pushing into an already existing stack; this can be performed trivially [4]. What we need to delve into is that the variable we bind a newly-created higher-order variable to has been already allocated on the heap and that the correct number of arguments have been passed.

  Thankfully, for us, this becomes a non-issue thanks to the way `put_application` was implemented. When we wrote the implementation for the aforementioned instruction, we took special care to bind the argument to an unbound heap variable (if it was an unbound stack argument).

  In addition, the only way of invoking a higher-order variable is to perform a partial application first (through the `put_partial`) set of instructions, then call the partial application through `call_variable`, thus all arguments will be located on the heap (and be traversable as a singly linked list of partial applications).

  The (now complete) implementation of `call_variable` (minus the stack manipulation boilerplate) is shown in figure 4.8.

- When backtracking from a call, we have to undo all stack pushes we performed from the point we entered the stack frame up to the point the backtracking was initiated.

  We can solve this issue by looking at the WAM trail: As a reminder, the trail is simply a separate memory segment where whenever a binding is performed to an unbound variable, this is recorded on the trail so that it can be reverted in case of backtracking.

We, thus, modify the trail so that it can accept both unbound variable addresses and higher-order variables on the heap (with each case being represented by a different tag).

The modifications to the trail procedures, as well as the utility functions we need to introduce are shown on figure 4.9.

Since mutation of the tuples of a higher order variable relation is now simply a matter of pushing and popping into a stack, we now modify `call_variable` to also support higher-order variables, as shown in figure 4.8.

```
$args$ = [];
$addr$ = HEAP[Xi];
while true do
    $addr$ = $deref(addr)$;
    switch val do
        case (REF, ptr)
            /* ptr is an unbound variable */
            STORE[$ptr$] = (HOV, H);
            $bind(ptr$, H);
            HEAP[H] = (HOVATY, $len(args)$);
            HEAP[H + 1] = (REF, H + 1);
            H = H + 2;
            $appendTuple(ptr, args)$; break;
        end
        case (HOV, ptr)
            $appendTuple(ptr, args)$;
            break;
        end
        case (APP, m)
            $addr$ = $addr$ + 1;
            $args$ = [HEAP[$addr$ + 2], ..., HEAP[$addr$ + m + 2]] + $args$;
            continue;
        end
        case (TRM, f/n)
            foreach $i$ in 1 .. len(args) do
                Xi = $args[i]$;
            end
            $ExecuteCall$(f/m, N);
            break;
        end
    endsw
end
```

Figure 4.8: Procedure for calling a variable, revisited

A keen observer will point out that the algorithm presented in 4.8 does not cover all cases regarding the invocation of a higher-order variable. For instance, given:

```
nat(0).
nat(s(X)) :- nat(X).
?- P(nat).
```

A correct implementation should print all natural numbers, starting from zero.

This case is the reason we made the following assumptions for our programs (which ensures that algorithm 4.8 will be correct) in chapter 3, that is all higher order variables can be:

- First-order variables
- Higher-order variables of an arbitrary order, as long as they are located on the head of a clause. Thus all higher order variables which will be requested to be generated will be located either on the clause body (and not on the clause head) or on the goal.

```
Function trail($addr$, $higherOrder$):
    if $addr$ < HB or (H < $a$ and $a$ < B) then
        TRAIL[TR] = ($addr$, $higherOrder$);
        TR = TR + 1;
    end
Function unwindTrail($addr1$, $addr2$):
    foreach $i$ = $addr1$ to $addr2 - 1$ do
        ($val$, $higherOrder$) = TR[i];
        if $higherOrder$ then
            $removeHeadTuple$(addr);
        else
            STORE[$val$] = (REF, $val$);
        end
    end
return
Function appendTuple($addr$, $args$):
    $headPtr$ = STORE[$addr$] + 1;
    HEAP[H] = (REF, HEAP[$headPtr$]);
    foreach $i$ = 1 to $len(args)$ do
        HEAP[H + $i$] = $args[i]$;
    end
    HEAP[$headPtr$] = (REF, H);
    H = H $+n+1$;
return
Function removeHeadTuple($addr$):
    $headPtr$ = $deref$($addr$) + 1;
    (_, $prev$) = HEAP[$headPtr$]; HEAP[$headPtr$] = (REF, $prev$);
return
```

Figure 4.9: Utility operations for higher-order variable manipulation

An example of execution using higher-order variables will now be shown. Given the following program:

```
foo(P)(A, B, C, D) :- P(A, B), fail.
foo(P)(A, B, C, D) :- P(C, D).
?- foo(P)(a, b, c, d).
```

The WAM instruction outputted will be the following:

```
foo/5: try_me_else L2
       put_application 2 X1, Y1
       set_variable X2
       set_variable X3
M0:    call_variable X1, 0
M1:    execute fail/0
L2:    trust_me
       put_application 2 X1, X1
       set_variable X4
       set_variable X5
M2:    execute_variable X1
main:  put_variable X1, X1
       put_constant X2, a
       put_constant X3, b
       put_constant X4, c
       put_constant X5, d
       execute foo/5
```

Then:

- Before the label `M0` is executed, the heap representation will match that of figure 4.10.
- Before the label `M1` is executed, the heap representation will match that of figure 4.11, the higher-order variable will consist of 1 tuple. The variable will also exist on the trail twice: Once as an unbound variable and once as an already existing higher-order variable.
- After the label `M1` is executed, a backtracking will occur. The backtracking will cause the higher-order variable to be an empty one, then, because the variable that pointed to it was also an unbound variable before the stack frame was executed, it will also become unbound, hence before `L1` is executed, the heap will be empty.
- Just like in the case of label `M0`, before the label `M2` is executed, the heap representation will match that of figure 4.12.

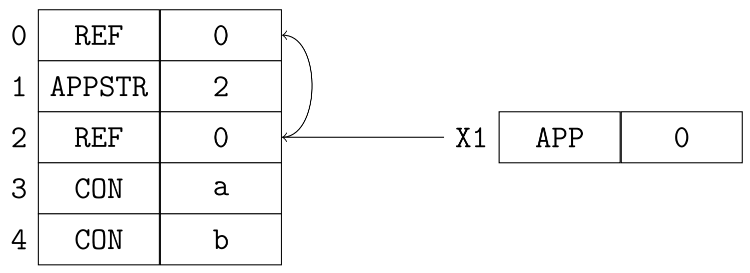


Figure 4.10: Heap representation of higher-order backtracking, state 1

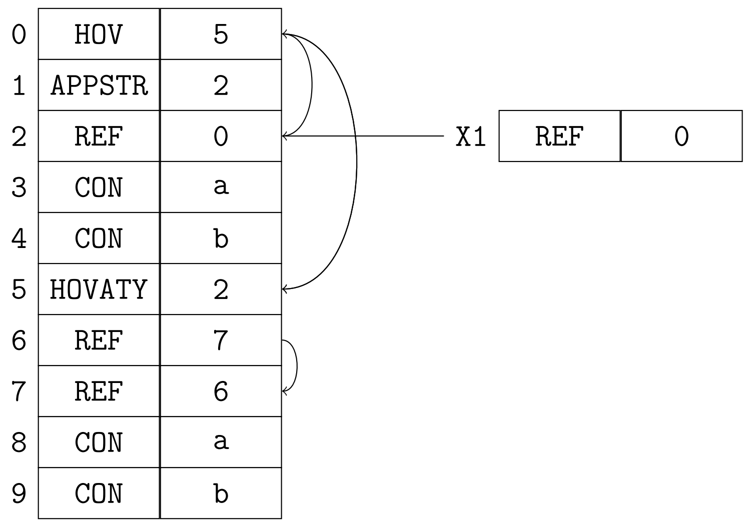


Figure 4.11: Heap representation of higher-order backtracking, state 2

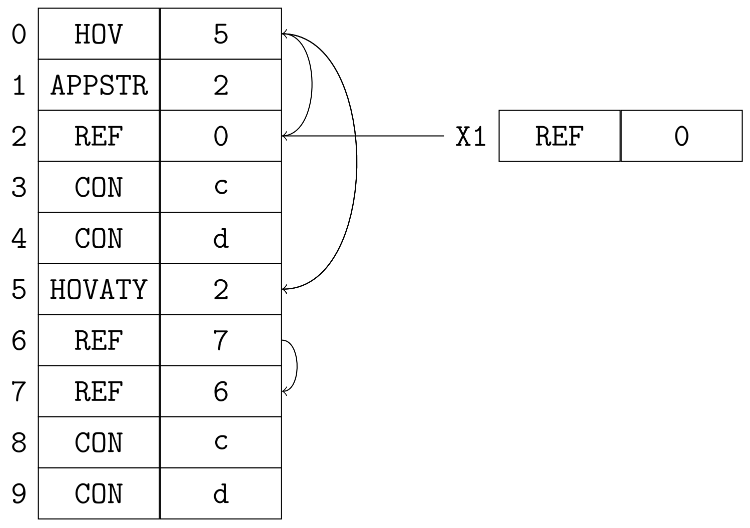


Figure 4.12: Heap representation of higher-order backtracking, state 3

We also observe a convenient property after incorporating the trail to track higher-order variable relations: In the classical WAM, addresses on the heap point to heap elements of a lower address, thus deallocating part of the heap by decrementing the heap pointer is a non-issue. This is, on first sight, not possible in our case, because it is possible for references to point to heap cells of a higher address. However, thanks to the addition of higher-order variable cleanup by the trail, this becomes a non-issue, hence there is no necessity in incorporating a more complex allocation scheme.

4.3 Conclusion

We now have seen that it is possible to extend the WAM to allow execution of `polyHOPES` programs, without introducing a significant number of changes to the WAM itself. A major question arising from this is whether or not existing Prolog programs translated into the classical WAM will continue to run unmodified. The answer is affirmative; if a `polyHOPES` program does not make use of partial application and/or higher-order variables, none of the instructions defined in this chapter will be executed, thus no partial application or higher order variable constructs will be present on the heap. Thus, the modification of the WAM shown in this chapter is a strict subset of the classical WAM.

5. RELATED WORK

In this chapter, we will describe the work performed on Teyjus [12] (a λProlog interpreter) and XSB (which partly implements HiLog) [6].

5.1 Teyjus

Just like the case in this thesis, the abstract machine implemented in Teyjus is based on the WAM. What differs, compared to our own implementation, is that an extended WAM that is going to be used by Teyjus must also fulfill the following requirements:

- The language contains primitives that can alter the name space and the definitions of procedures in the course of execution. This means, in particular, that unification has to pay attention to changing signatures and that the solution to each (sub)goal has to be relativized to the relevant program context.
- In contrast to other languages, lambda terms are used in Lambda Prolog as data structures. A representation must therefore be provided for these terms that permits their structures to be examined and compared in addition to supporting reduction operations efficiently.
- Higher-order unification is used in an intrinsic way in the language. This operation is conceptually more complex than the unification operation of Prolog and a practical way of supporting it must be found. In doing this, it may sometimes be necessary to delay the solution of unification problems. For this reason, a mechanism must be devised for representing unification problems explicitly.
- In addition to having a role in determining program correctness, types could be relevant to the dynamic behavior of programs. A scheme must therefore be designed for reducing the runtime impact of types and this must be augmented by a good mechanism for carrying types along into computatations when this cannot be avoided.
- Programming in the language is done relative to modules. In realizing this feature, it is necessary to support certain operations for composing different modules. Moreover, if modularity is to be genuine, a mechanism must be devised for realizing separate compilation.

Work carried out by Nadathur et al [12] in the design of a virtual machine that included devices for treating all these aspects well. The solution to the problem of changing signatures was based on an scheme for tagging constants and variables and using these tags in unification.

To realize changing program contexts, a fast method was designed for adding and removing code that is also capable of dealing with backtracking. Code that needs to be added may sometimes contain global variables and this possibility was dealt with by an adaptation to logic programming of the idea of a closure. To facilitate a sensible representation of lambda terms, an explicit substitution calculus called the suspension calculus was designed and deployed in the low-level steps for manipulating lambda term that are contained in the abstract machine. This abstract machine handled full higher-order unification, an operation that is characterized by its branching behaviour. Techniques were developed for treating such branching and also for compiling unification steps and for prioritizing deterministic parts of the unification computation. In treating types, ideas were introduced for utilizing information available at compile time about their structure to substantially reduce the effort expended at runtime in creating and analyzing types.

Finally, towards supporting modular programming, a method was designed for realizing separate compilation with one of the module interaction mechanisms known as module importation. This mechanism also required the addition and removal of blocks of code. Techniques for dealing with changing program contexts in the core language could be used to implement this aspect. However, these methods had to be embellished with new mechanisms for avoiding redundancy in the added code, something that could only be determined at runtime.

5.2 HiLog

Unlike Teyjus, an implementation of HiLog incorporates no modifications to the WAM, but, instead, the HiLog source code is transpiled into Prolog at first [15]. As such, the XSB implementation of HiLog [6] needs no modifications to the WAM so that it can run.

The rationale for not modifying the WAM specification, according to [15] is as follows:

- All low-level Prolog optimisations and compilation techniques developed throughout the years would be immediately applicable.
- Prolog programs would not incur the extra cost of the WAM modifications.
- HiLog programs could run on any Prolog implementation.

According to the paper, the WAM should be an adequate abstract machine for the execution of any logic language with first-order semantics. Thus, [15] focuses on providing:

- A complete solution to the problem of HiLog implementation, which stays within the WAM framework.
- A formal proof that HiLog programs that do not use any higher-order features execute at the same speed as Prolog programs, when compiled with the proposed scheme.
- A completely automated call specialisation algorithm that uses global static information, but does not require any user supplied annotations, information about the

queries, or approximation of the dynamic behaviour of HiLog programs using abstract interpretation.

6. CONCLUSIONS AND FUTURE WORK

We have, thus, presented a modification to the WAM, which is capable of implementing partial application and higher order variables, thus translating HOPES to WAM can be done with only a couple of somewhat trivial modifications.

The modifications we made to the WAM are also exteremely memory and execution time efficient: No significant modifications to the memory allocation schemes of the WAM were necessary and, except for the case of performing the invocation of a partial application with n final variables (which is obviously an $\Omega(n)$ operation), all other instructions have a constant running time.

The main limitation of our current implementation is the fact that calling a higher-order variable with another higher order variable or predicate as an argument is currently not implemented. We speculate that implementing the case for first-order variables is a task that is neither trivial nor complex. On the other hand, we speculate that adding support for second (and higher) order variables may require significant changes to the WAM layout.

The original paper for HOPES [7] suggests looking for approaches regarding adding *negation as failure* to the language. An interesting approach for doing so is by adding negation-call and negation-execute instructions (the latter one being useful for tail call optimization). The negation-call would proceed as expected for ”normal” calls, whereas for higher order variables an approach that makes use of three cyclic intrusive (`prev` and `next` nodes are located in the list elements themselves) linked lists, one of which is used to store the affirmative clauses, one the negative and all clauses would be in the final list, which contains all the elements and is used by the trail to perform unwinding.

Another suggestion for improving the implementation presented above is to add closure support, instead of an ad-hoc partial application. A closure is, essentially, a partial application with the exception that instead of being a partial application of a variable, it is a partial application to a predicate (either labelled or anonymous). In addition, instead of shuffling registers around to construct the argument list, we can add another register referring to the captured parameters in the closure. This also enables us to easily add support for anonymous predicates, with only having to modify the syntax. Closure support would also make it easier to add typechecking for predicates (anonymous, higher-ordered or even neither of them). The only minor downside is to invent additional syntax to refer to a predicate with arity n instead of a structure with the same name and with arity n. We can also invent a syntax to express the creation of an ”empty” higher-order variable with an arity of n (which is not possible with the current syntax).

ABBREVIATIONS - ACRONYMS

WAM	Warren Abstract Machine
ISA	Instruction Set Architecture
HOPES	Higher Order Prolog with Extensional Semantics

REFERENCES

[1] Hassan Ait-Kaci. Warren's abstract machine-a tutorial reconstruction. 1999.

[2] Aggelos Charalambidis. Hopes haskell interpreter. `http://code.haskell.org/hopes/`, 2012. [Online; accessed 10-November-2016].

[3] Weidong Chen, Michael Kifer, and David S Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, 1993.

[4] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[5] Vítor Santos Costa. Optimising bytecode emulation for prolog. In *International Conference on Principles and Practice of Declarative Programming*, pages 261–277. Springer, 1999.

[6] Luis Fernando P. de Castro. Xsb. `http://xsb.sourceforge.net/shadow_site/manual1/node36.html`, 2012. [Online; accessed 10-November-2016].

[7] Angelos Charalambidis Konstantinos Handjopoulos, Panos Rondogiannis, and William W Wadge. Extensional higher-order logic programming.

[8] ISO ISO. Iec 9899: 2011 information technology—programming languages—c. *International Organization for Standardization, Geneva, Switzerland*, 2011.

[9] Emmanouil Koukoutos. A higher-order extension of prolog with polymorphic type inference.

[10] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification: Java SE 8 Edition*. Pearson Education, 2014.

[11] Dale A Miller and Gopalan Nadathur. Higher-order logic programming. In *International Conference on Logic Programming*, pages 448–462. Springer, 1986.

[12] Gopalan Nadathur and Dustin J Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λprolog. In *International Conference on Automated Deduction*, pages 287–291. Springer, 1999.

[13] William W Wadge. Higher-order horn logic programming. In *ISLP*, pages 289–303, 1991.

[14] David HD Warren. An abstract prolog instruction set. *Tech. Note 309*, 1983.

[15] Konstantinos Sagonas David S Warren. Efficient execution of hilog in wam-based prolog implementations. In *Logic Programming: Proceedings of the Twelfth International Conference on Logic Programming*, volume 12, page 349. MIT Press, 1995.