INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig
Maximilians-
Universität
München

SatAM – An Abstract Machine for Model Generation with PUHR Tableaux

Alexander von Drach **David Sacher**

Fortgeschrittenenpraktikum

Beginn der Arbeit:	September 1997
Abgabe der Arbeit:	September 1998
Aufgabensteller:	Prof. Dr. François Bry
Betreuer:	Dipl.-Inf. Tim Geisler
	Dipl.-Inf. Sven Panne

Contents

1 Introduction

The model-generation paradigm for automated deduction [MB88] is to demonstrate the *satisfiability* of first-order formulas by constructing satisfying Herbrand interpretations (i. e., models) instead of demonstrating their *unsatisfiability* by constructing a refutation (e. g., a resolution proof). Model generation was originally motivated by database applications (a detailed description of such an application appears in [BEST98]). Other successful applications of model generation include model-based diagnosis [BFFN97], planning [EG98], and even the solution of previously open problems in finite algebra [FSB93].

Among the various approaches for building models of first-order formulas, we concentrate in this paper on the *positive unit hyper-resolution (PUHR) tableau method* [BY96]. It combines positive hyper-resolution with unit clauses and a beta or splitting tableau rule.

The PUHR tableau method is actually a formalization of the theorem prover Satchmo [MB88], an early attempt to integrate resolution into the tableau framework for automated deduction in order to yield efficient tableau methods. The Prolog programs given in [MB88] are remarkably short and simple, yet efficient, because they re-use rather than re-implement Prolog's features like backtracking and matching.

Two techniques enhancing the efficiency of PUHR tableau method were introduced with *Compiling Satchmo* [SG96]: *incremental evaluation*, a technique known from deductive databases as Δ-*iteration*, and *compilation*. Compiling Satchmo compiles a set of first-order clauses into a Prolog program, which computes its minimal models more efficiently.

A problem with the approach of using Prolog as a target language for compilation is that the language is still too high-level: Compiling the Prolog rules to native code by a Prolog compiler takes prohibitively long when the initial clause set is large, and using a Prolog system based on an abstract machine (e. g. WAM [War83, Aït91]) adds another interpretation layer. We therefore developed a target language and corresponding machine at a level which lies more in the "middle" between clause sets and native code. This *abstract machine* directly supports incremental evaluation, backtracking, matching, and an appropriate efficient data structure for the representation of the models under construction. We briefly call the Satchmo abstract machine *SatAM*.

In the following, we first review the PUHR tableau method, incremental evaluation, and the compilation done by Compiling Satchmo. We then describe the systematic development of a prototype of an abstract machine in Prolog, following the approach developed by Kursawe [Kur86] and Nilsson [Nil92, Nil93] to derive an abstract machine for Prolog. We further present the necessary modifications to the result of this development process in order to obtain an abstract machine that can be implemented in a suitable low-level imperative language. Finally we describe the compiler *SatAMC*, which compiles range-restricted clauses to bytecode suitable for SatAM.

This paper is based on joint work with Tim Geisler and Sven Panne [vGPS98] and describes its major contributions in more detail.

2 The PUHR Tableau Method

The PUHR tableau method [BY96] is a calculus for range-restricted clauses. We call the set of negative literals of a clause its *body*, and the set of positive literals its *head*. A clause is range-restricted if each of its variables occurs in the body. Many practical problem domains can be naturally formalized with range-restricted clauses [EG98]. If not, clauses can always be transformed into a range-restricted form [MB88, BY96].

PUHR tableaux are trees in which every node is labeled with a set $\mathcal{U}$ of ground atoms (**u**nits) and a (multi-)set $\mathcal{P}$ of **p**ositive ground clauses. PUHR tableaux for a given clause set are built starting from the tree consisting of a node labeled with the empty set of units and the set of positive ground clauses from the input clauses, by repeatedly applying the following two tableau rules to a branch with leaf node $(\mathcal{U}, \mathcal{P})$.

The *PUHR rule* resolves the body literals of an input clause simultaneously with units from $\mathcal{U}$ and extends the branch with a new leaf $(\mathcal{U}, \mathcal{P}')$, where $\mathcal{P}'$ is the result from adding the (hyper-)resolvent to $\mathcal{P}$.

The *splitting rule* extends the branch either with n new leaf nodes $(\mathcal{U} \cup \{p_i\}, \mathcal{P} \backslash \{p_1 \vee \cdots \vee p_n\})$, if none of the p_i is in $\mathcal{U}$, or otherwise with one new leaf node $(\mathcal{U}, \mathcal{P} \backslash \{p_1 \vee \cdots \vee p_n\})$. As a borderline case, if the empty clause is selected from $\mathcal{P}$ (i.e., $n = 0$), the branch is marked as *closed*.

If none of these tableau rules is applicable to the leaf node of a branch, then its set $\mathcal{U}$ represents a Herbrand model of the input clause set. If all branches are closed, then the input clause set is unsatisfiable.

The PUHR tableau method has properties such as refutation completeness and minimal-model completeness which are desirable for model generation [BY96], provided the two tableau rules are applied in a fair way. Note that this *fair control* can be realized by computing all possible hyper-resolvents in one step and by implementing $\mathcal{P}$ as a queue.

Incremental evaluation can be used to obtain significantly more efficient implementations of the PUHR tableau method [SG96]. To avoid repeated application of the PUHR rule with the same clauses and to accelerate its applicability test we apply this rule *incrementally*: If a unit is added to $\mathcal{U}$ in a splitting step, then exactly those PUHR steps that use this unit have to be applied as the next steps.

Most implementations of the PUHR tableau method *interpret* a set of clauses. *Compilation techniques* can be used to avoid this level of interpretation, because the PUHR tableau method with incremental evaluation is well suited for compilation [SG97].

The PUHR rule and the splitting rule can be mostly specialized for each input clause. The set of input clauses is fixed. Furthermore, a hyper-resolvent generated by applications of the PUHR rule to an input clause is always a (ground) instance of this clause's head. As a consequence, it even suffices to store head identifiers and substitutions in $\mathcal{P}$.

In order to speed up the discovery of potentially resolvable clauses, the incremental application of the PUHR rule can be specialized for each body literal of a clause and for each predicate symbol.

This compilation approach was applied to Prolog implementations of the PUHR tableau method, resulting in Compiling Satchmo. It compiles a set of clauses into a Prolog program, which efficiently computes the set of minimal models of the given clauses [SG96, GPS97].

3 Systematic Development of an Abstract Machine

The purpose of an abstract machine is to bridge the conceptual gap between language and hardware. The design of an abstract machine is a difficult process, which requires a lot of ingenuity. From the existing work towards a methodology for the design of abstract machines, we followed the approach of Kursawe [Kur86] and Nilsson [Nil92, Nil93], who systematically reconstructed from the instruction set of the WAM those parts that deal with unification and control, respectively.

Their approach starts from interpreters for logic languages implemented in Prolog with stepwise enhanced expressivity. Simultaneously to these extensions to the implemented logic language, Prolog prototypes are developed by partial deduction and other program transformation techniques. Each prototype implements the abstract machine instructions and hand-compiled abstract machine code, represented as Prolog rules.

In order to count as a *machine*, the abstract machine instructions have to be implemented in Prolog in such a way that at the machine level they can be interpreted deterministically and neither recursion (apart from tail-recursion, i. e., looping) nor backtracking is used. Unification has to be reduced to simple parameter passing [Nil92].

We follow this methodology, with the extension that global data structures in the interpreter (e. g., the two sets $\mathcal{U}$ and $\mathcal{P}$ in our case) remain global in the abstract machine. Those data structures give rise to abstract machine operations to manipulate them.

We develop the abstract machine for the PUHR tableau method in several steps. In each step, we enhance the expressivity of the logic source language, and simultaneously extend the design of the abstract machine, i. e., introducing or modifying machine instructions and global data structures. During this process, we maintain two invariants: The machine should provide incremental evaluation and a fair control.

In the following sections we describe the stepwise development of the abstract machine. Instead of giving the Prolog prototypes mentioned above, the invented machine instructions are presented as state-transition rules like Peyton Jones and Lester do for describing the abstract machine for a lazy, purely functional language [PL92]. They define a state transition system as a notation for describing the behavior of a sequential machine. At any time, the machine is in some state, beginning with a specified initial state. If the machine state matches one of the state-transition rules, the rule fires and specifies a new state for the machine. When no state transition rule matches, execution halts. If more than one rule matches, one of them is arbitrarily chosen to fire; in this case the machine is non-deterministic. However, the machine presented in this paper is deterministic.

3.1 Definite Propositional Clauses

As a starting point, we choose to develop a machine for definite propositional clauses, i. e., propositional clauses with exactly one head atom. This language already requires the PUHR rule with incremental evaluation, but requires neither matching, the application of substitutions, branching in the tableau, nor backtracking.

Machine and Machine Instructions A definite propositional clause of the form $\alpha_1 \wedge \cdots \wedge \alpha_n \rightarrow \beta$ with a non-empty body has $n \geq 1$ *entry points* (also denoted α_k) for incremental evaluation: each of the atoms α_k could have been added to $\mathcal{U}$ by the preceding application of the splitting rule. The following operations are performed by the *body code sequence* for entry point α_k: Lookup all α_i (with $i \neq k$) in $\mathcal{U}$; if all lookups are successful, add to $\mathcal{P}$ the address of the *head code sequence* corresponding to β (PUHR rule) and return to the caller; if any lookup fails, return to the caller immediately.

Head code sequences test if their corresponding β is already in $\mathcal{U}$; if not, β is inserted into $\mathcal{U}$ and all body code sequences with entry point β are called as a subroutine. After this, the next address is fetched from $\mathcal{P}$ and jumped to.

The initial code sequence of the machine adds the addresses of all head code sequences for clauses with an empty body (i. e., the facts) to $\mathcal{P}$.

The machine handles two global data structures: the set of atoms $\mathcal{U}$, which are just symbols in the propositional case, and a queue of head code addresses – the *continuation queue* implementing the set of atoms $\mathcal{P}$. The queue ensures fairness.

The initial state[1] is

$$\boxed{0 \quad c \quad \emptyset \quad \emptyset \quad 0}$$

There are three instructions dealing with the set of atoms $\mathcal{U}$. The instruction `insert` inserts an atom into $\mathcal{U}$:

[1]In this step a state is a quintupel

$$\boxed{p \quad c \quad \mathcal{U} \quad \mathcal{P} \quad r}$$

where p is the program counter, c the code (an array of instructions and their arguments), $\mathcal{U}$ and $\mathcal{P}$ the introduced set and queue resp. and r a register used to store a return address.

$c[p : \texttt{instr}\ a_1, \ldots, a_n]$ means that $c[p] = \texttt{instr}$, `instr` has the arity n and $a_1, \ldots, a_n$ are the arguments stored at $c[p+1], \ldots, c[p+n]$. p has to be incremented by $n+1$ to reach the next machine instruction.

All future extensions of the abstract machine, i. e., all new global data structures, will be initialized empty (or with 0, respectively). Therefore, we will not give the new initial state in each step.

$$\boxed{\begin{array}{lllll} & p & c[p: \texttt{insert}\ \beta] & \mathcal{U} & \mathcal{P}\ \ r \\ \Rightarrow & p+2 & c & \mathcal{U}\cup\{\beta\} & \mathcal{P}\ \ r \end{array}}$$

The lookup instruction for a body atom `proved` returns to the caller of the body code sequence, if the atom is not in $\mathcal{U}$. Otherwise it does nothing:

$$\boxed{\begin{array}{llllll} & p & c[p: \texttt{proved}\ \alpha] & \mathcal{U} & \mathcal{P} & r \\ \Rightarrow & r & c & \mathcal{U} & \mathcal{P} & r \\ \text{if} & \alpha\notin\mathcal{U} & & & & \end{array}}$$

$$\boxed{\begin{array}{llllll} & p & c[p: \texttt{proved}\ \alpha] & \mathcal{U} & \mathcal{P} & r \\ \Rightarrow & p+2 & c & \mathcal{U} & \mathcal{P} & r \\ \text{if} & \alpha\in\mathcal{U} & & & & \end{array}}$$

In contrast, the lookup instruction `test` for a head atom does nothing if the atom is not in $\mathcal{U}$. Otherwise it fetches the next address from the continuation queue and jumps to it:

$$\boxed{\begin{array}{llllll} & p & c[p: \texttt{test}\ \beta] & \mathcal{U} & \mathcal{P} & r \\ \Rightarrow & p+2 & c & \mathcal{U} & \mathcal{P} & r \\ \text{if} & \beta\notin\mathcal{U} & & & & \end{array}}$$

$$\boxed{\begin{array}{llllll} & p & c[p: \texttt{test}\ \beta] & \mathcal{U} & q:\mathcal{P} & r \\ \Rightarrow & q & c & \mathcal{U} & \mathcal{P} & r \\ \text{if} & \beta\in\mathcal{U} & & & & \end{array}}$$

There is an instruction `call` to enqueue a code address of a head code sequence into $\mathcal{P}$ and an instruction `proceed` to set the program counter to the next element of $\mathcal{P}$: if the queue is not empty, it dequeues a code address and jumps to it.

$$\boxed{\begin{array}{llllll} & p & c[p: \texttt{call}\ q] & \mathcal{U} & \mathcal{P} & r \\ \Rightarrow & p+2 & c & \mathcal{U} & \mathcal{P}:q & r \end{array}}$$

$$\boxed{\begin{array}{llllll} & p & c[p: \texttt{proceed}] & \mathcal{U} & q:\mathcal{P} & r \\ \Rightarrow & q & c & \mathcal{U} & \mathcal{P} & r \end{array}}$$

If `proceed` is called and $\mathcal{P}$ is empty, there is no matching state transition. In this case the machine terminates and a model is found. The positive unit clauses in $\mathcal{U}$ represent the generated model.

Two further instructions are required for calling and returning from body code sequences: `call_subroutine` and `return_subroutine`. A single register suffices to store the return address, because there are no subroutine calls in body code sequences, i. e., subroutine calls are not nested:

$$\boxed{\begin{array}{llllll} & p & c[p: \texttt{call\_subroutine}\ q] & \mathcal{U} & \mathcal{P} & r \\ \Rightarrow & q & c & \mathcal{U} & \mathcal{P} & p+2 \end{array}}$$

$$\boxed{\begin{array}{llllll} & p & c[p: \texttt{return\_subroutine}] & \mathcal{U} & \mathcal{P} & r \\ \Rightarrow & r & c & \mathcal{U} & \mathcal{P} & r \end{array}}$$

Definite propositional clauses are compiled to a sequence of these machine instructions.

Example In each step we want to give a small example. Consider the following clauses:

$$\top \to a \tag{1}$$

$$a \to b \tag{2}$$

$$a \wedge b \to c \tag{3}$$

They are compiled to the following sequence of machine instructions:

0:	`call`	3	– *initial call*
2:	`proceed`		
3:	`test`	a	– *head code sequence for (1)*
5:	`insert`	a	
7:	`call_subroutine`	12	
9:	`call_subroutine`	22	
11:	`proceed`		
12:	`call`	15	– *single entry point for (2)*
14:	`return_subroutine`		
15:	`test`	b	– *head code sequence for (2)*
17:	`insert`	b	
19:	`call_subroutine`	27	
21:	`proceed`		
22:	`proved`	b	– *1st entry point for (3)*
24:	`call`	32	
26:	`return_subroutine`		
27:	`proved`	a	– *2nd entry point for (3)*
29:	`call`	32	
31:	`return_subroutine`		
32:	`test`	c	– *head code sequence for (3)*
34:	`insert`	c	
36:	`proceed`		

Remark SatAM and SatAMC combine the instructions `test` and `insert` to a single instruction `test_and_insert`. Nevertheless, the former instructions are still necessary in the next steps. For the sake of brevity, we do not give a state transition for the combined instruction.

3.2 Propositional Clauses

In the previous step every clause had exactly one head atom. We now extend the logic language to full propositional clauses, i. e., the head of a clause can now be a (possibly empty) disjunction. With this extension, the splitting rule introduces branching into the tableau, with the possibility of closed branches. This can be implemented using backtracking. We postpone the introduction of variables – the alternative choice for the current development step – because backtracking is easier to manage without variables.

Machine and Machine Instructions In the abstract machine backtracking gives rise to a new global data structure – the *splitting choice point stack* $\mathcal{S}$. The instruction `try_me_else` is introduced which pushes a new choice point onto $\mathcal{S}$, saving the states of $\mathcal{U}$ and the continuation queue $\mathcal{P}$ and an address for an alternative which is passed to the machine instruction.

$$\begin{array}{|lllllll|}\hline & p & c[p: \texttt{try\_me\_else}\ q] & \mathcal{U} & \mathcal{P} & \mathcal{S} & r \\ \Rightarrow & p+2 & c & \mathcal{U} & \mathcal{P} & (\mathcal{U},\mathcal{P},q):\mathcal{S} & r \\ \hline\end{array}$$

The head code sequences have to be adapted to non-definite clauses. First, for each head atom a lookup instruction is performed. Then, if non of the head atoms is in $\mathcal{U}$, a choice point is generated for each but the rightmost head atom. For each head atom β_i, subroutine calls to all body code sequences with entry point β_i have to be performed after the possible generation of the corresponding choice point and the insertion of β_i into $\mathcal{U}$.

Instead of compiling empty clause heads into code which is called by the `call` instruction and therefore executed later (if $\mathcal{P}$ is not empty), this occurrence of `call` is replaced by a new instruction `fail`. If possible, this instruction pops a choice point, resets $\mathcal{U}$ and the continuation queue $\mathcal{P}$ and jumps to the saved alternative address. Otherwise the machine terminates with the result that no model is found.

$$\begin{array}{|lllllll|}\hline & p & c[p: \texttt{fail}] & \mathcal{U} & \mathcal{P} & (\mathcal{U}',\mathcal{P}',q):\mathcal{S} & r \\ \Rightarrow & q & c & \mathcal{U}' & \mathcal{P}' & \mathcal{S} & r \\ \hline\end{array}$$

The semantics of `proceed` changes a bit: after a model is found and presented to the user, the user is asked whether another model should be computed. For this computation further backtracking is initiated. [2]

[2]Note that the generated choice points are not recycled. The WAM improves this by the use of the instruction `retry_me_else`. This optimization is provided neither in the

Example For example the clauses

$$\top \to a \vee b \vee c \tag{4}$$
$$a \to \bot \tag{5}$$

are compiled to:

0:	`call`	3
2:	`proceed`	
3:	`test`	a
5:	`test`	b
7:	`test`	c
9:	`try_me_else`	16
11:	`insert`	a
13:	`call_subroutine`	24
15:	`proceed`	
16:	`try_me_else`	21
18:	`insert`	b
20:	`proceed`	
21:	`insert`	c
23:	`proceed`	
24:	`fail`	

3.3 Ground Clauses

We now introduce compound ground terms into the logic language. Note that this logic language has the same expressivity as propositional clauses. This step prepares the introduction of variables.

Composite ground terms have to be linearized because the parameters of a machine instruction should have a fixed size. Therefore, every machine instruction having a (possibly compound) atom as argument is replaced by a sequence of instructions with arguments of a fixed size.

For example, instead of `insert` $p(a, f(g(b)))$ the sequence

`insert`	p	2
`insert`	a	0
`insert`	f	1
`insert`	g	1
`insert`	b	0

is used.

prototype, nor in the concrete implementation of the abstract machine. Of course, this would be a reasonable improvement.

As the Prolog prototype completely reconstructs the term again and provides the insertion and the lookups with compound terms, we can treat such a sequence of machine instructions as a single instruction with compound argument in the following. This makes the description of the further steps shorter and much more transparent, because we can use a higher abstraction level.[3]

3.4 Clauses with Non-Ground Bodies and Ground Heads

We now extend the logic language by introducing variables in clause bodies. We postpone the introduction of variables into the clause heads, because the specialization of the incremental application of the PUHR rule is more complicated with non-ground terms than with ground terms in the clause heads.

Machine and Machine Instructions We introduce a new data structure $\mathcal{E}$ – the *environment* – which represents the current bindings of variables to terms[4].

If a body code sequence with non-ground entry point α_i is called, the variables in α_i have to be bound to terms by the caller (according to the caller's head atom just inserted into $\mathcal{U}$). This parameter passing is implemented by letting the caller allocate the environment with the instruction `alloc_env`[5] and subsequently bind the appropriate variables in the environment with the instructions `env_entry_for` and `write_env`.

[3] Otherwise we would need to introduce two new global data structures: a counter (to manage the reconstruction) and a store to save the (partially) reconstructed term. This would give raise to numerous state-transition rules, because in each rule it has to be checked, if the term is completely reconstructed or not.

The instructions used by the concrete abstract machine will be presented later.

[4] The environment is a (ground) substitution in the further text.

id is the empty substitution. If σ, σ' are substitutions, v a variable, V a set of variables and t a term, we use the following notations:

1. $\sigma[v/t]$ replaces the binding of v in σ with t
2. $\sigma \circ \sigma'$ is the composition of σ and σ'
3. $v\sigma := \sigma(v)$
4. $dom(\sigma)$ is the set of all variables v with $v\sigma \neq v$
5. $ran(\sigma) := \bigcup_{v \in dom(\sigma)} vars(v\sigma)$
6. $\sigma|_V(x) := \begin{cases} x\sigma & \text{if } x \in V \\ x & \text{otherwise} \end{cases}$
7. $\sigma \setminus_V (x) := \begin{cases} x & \text{if } x \in V \\ x\sigma & \text{otherwise} \end{cases}$

   That is, "reset" the bindings of the variables V in σ.

[5] Representing the environment as a substitution in this context, `alloc_env` just resets

$$\begin{array}{|rlllllllll|}\hline & p & c[p: \texttt{alloc\_env}] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & \mathcal{E} & r \\ \Rightarrow & p+2 & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & id & r \\ \hline\end{array}$$

$$\begin{array}{|rlllllllll|}\hline & p & c[p: \texttt{env\_entry\_for}\ w] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & \mathcal{E} & r \\ \Rightarrow & p+2 & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & w & \mathcal{E} & r \\ \hline\end{array}$$

$$\begin{array}{|rlllllllll|}\hline & p & c[p: \texttt{write\_env}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & \mathcal{E} & r \\ \Rightarrow & p+2 & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & \mathcal{E}[v/t] & r \\ \hline\end{array}$$

The incremental evaluation technique requires to perform all PUHR steps that use the unit added to $\mathcal{U}$ most recently. Therefore, another backtracking mechanism is needed to find a compatible substitution in the hyper-resolution step. We introduce a second stack – the *hyper-resolution stack* $\mathcal{HS}$. An entry of this stack consists of a code address to proceed if another variable binding has to be tried, a list of substitutions and an index denoting the substitution to be used in the next backtracking step.

Every code sequence for each body atom containing unbound variables is preceded by the instruction `create_hs_entry` for creating a hyper-resolution stack entry with an empty list of substitutions.

$$\begin{array}{|rlllllllll|}\hline & p & c[p: \texttt{create\_hs\_entry}\ q] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & \mathcal{E} & r \\ \Rightarrow & p+2 & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & (0,[],q):\mathcal{HS} & v & \mathcal{E} & r \\ \hline\end{array}$$

Note that due to the range restriction of the input clauses, it can be determined at compilation time for each variable whether it is already bound.

Additional matching instructions are needed in body code sequences, which match and bind unbound variables or match an already bound variable, respectively. A matching instruction for a body atom α fails, if there is no substitution σ such that $\alpha\mathcal{E}\sigma \in \mathcal{U}$. In this case an alternative variable binding has to be tried. To accommodate this, the top entry of the hyper-resolution stack is considered: If there is another substitution represented in this entry, the bindings of the environment are updated to that substitution and the control flow continues at the proceed address of this stack entry; otherwise, the entry is popped and the previous stack entry is considered; if there is none, control returns to the caller.

Treating the machine instructions as if they could have compound arguments we have to give the state transition function just for the two unary

---

the current environment. In the concrete realisation `alloc_env` allocates space of a particular size, of course. Therefore, in the concrete realization the instruction will be supplied with a parameter specifying this size.

instructions prove_bound and prove_ubound (prove_bound replaces the former instruction proved). In the concrete realization there are some more instructions to introduce.[6]

$$\boxed{\begin{array}{lllllllllll} & p & c[p: \texttt{prove\_bound}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \emptyset & v & \mathcal{E} & r \\ \Rightarrow & r & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \emptyset & v & \mathcal{E} & r \\ \multicolumn{10}{l}{\text{if } t\mathcal{E} \notin \mathcal{U}} \end{array}}$$

$$\boxed{\begin{array}{lllllllll} & p & c[p: \texttt{prove\_bound}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & (m,\bar{\sigma},q):\mathcal{HS} & v & \mathcal{E} & r \\ \Rightarrow & q & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & (m+1,\bar{\sigma},q):\mathcal{HS} & v & \sigma_m \circ \mathcal{E}' & r \\ \multicolumn{10}{l}{\text{if } t\mathcal{E} \notin \mathcal{U} \wedge \bar{\sigma} = [\sigma_0,\ldots,\sigma_n] \wedge 0 < m \leq n} \\ \multicolumn{10}{l}{\text{where } \mathcal{E}' = \mathcal{E}\backslash_{dom(\sigma_m)}} \end{array}}$$

$$\boxed{\begin{array}{llllllllll} & p & c[p: \texttt{prove\_bound}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & (m,[\sigma_0,\ldots,\sigma_n],q):\mathcal{HS} & v & \mathcal{E} & r \\ \Rightarrow & p & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & \mathcal{E} & r \\ \multicolumn{10}{l}{\text{if } t\mathcal{E} \notin \mathcal{U} \wedge m > n} \end{array}}$$

$$\boxed{\begin{array}{llllllllll} & p & c[p: \texttt{prove\_bound}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & \mathcal{E} & r \\ \Rightarrow & p+2 & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & \mathcal{E} & r \\ \multicolumn{10}{l}{\text{if } t\mathcal{E} \in \mathcal{U}} \end{array}}$$

$$\boxed{\begin{array}{llllllllll} & p & c[p: \texttt{prove\_ubound}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \emptyset & v & \mathcal{E} & r \\ \Rightarrow & r & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \emptyset & v & \mathcal{E} & r \end{array}}$$

$$\boxed{\begin{array}{llllllllll} & p & c[p: \texttt{prove\_ubound}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & (0,[],q):\mathcal{HS} & v & \mathcal{E} & r \\ \Rightarrow & q & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & (1,\bar{\sigma},q):\mathcal{HS} & v & \sigma_0 \circ \mathcal{E} & r \\ \multicolumn{10}{l}{\text{if } \bar{\sigma} = [\sigma_0,\ldots,\sigma_n] \text{ is the list of all substitutions, such that } t\mathcal{E}\sigma_i \in \mathcal{U} \wedge n \geq 0} \end{array}}$$

$$\boxed{\begin{array}{llllllllll} & p & c[p: \texttt{prove\_ubound}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & (0,[],q):\mathcal{HS} & v & \mathcal{E} & r \\ \Rightarrow & p & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & \mathcal{E} & r \\ \multicolumn{10}{l}{\text{if } \neg\exists\sigma : t\mathcal{E}\sigma \in \mathcal{U}} \end{array}}$$

[6] For example, instead of the unary instruction prove_ubound we would need the binary instructions: prove_and_bind for predicate and function symbols, prove_and_bind_envvar and prove_and_bind_bvar for variables already bound and prove_and_bind_uvar for still unbound variables.

$$\begin{array}{|llllllllll|}
\hline
 & p & c[p: \texttt{prove_ubound}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & (m, \bar{\sigma}, q) : \mathcal{HS} & v & \mathcal{E} & r \\
\Rightarrow & q & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & (m+1, \bar{\sigma}, q) : \mathcal{HS} & v & \sigma_m \circ \mathcal{E}' & r \\
\multicolumn{10}{|l|}{\text{if } \bar{\sigma} = [\sigma_0, \ldots, \sigma_n] \wedge 0 < m \leq n} \\
\multicolumn{10}{|l|}{\text{where } \mathcal{E}' = \mathcal{E} \backslash_{dom(\sigma_m)}} \\
\hline
\end{array}$$

$$\begin{array}{|llllllllll|}
\hline
 & p & c[p: \texttt{prove_ubound}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & (m, [\sigma_0, \ldots, \sigma_n], q) : \mathcal{HS} & v & \mathcal{E} & r \\
\Rightarrow & p & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & \mathcal{E} & r \\
\multicolumn{10}{|l|}{\text{if } m > n} \\
\hline
\end{array}$$

Note that $\mathcal{E}$ and σ_i are ground substitutions. Note also that q in the $\mathcal{HS}$-entry always equals $p + 2$. This does not hold anymore, if the instruction `prove_ubound` is replaced by a sequence of instructions with fixed argument size. Then q points to the first instruction after this sequence.

At this stage there is another machine instruction to introduce, which becomes obsolete in the next step: After having found a suitable substitution $\mathcal{E}$ such that for all body atoms $\alpha_i\mathcal{E} \in \mathcal{U}$ holds and having called the corresponding head code sequence, the hyper-resolution stack $\mathcal{HS}$ has to be emptied and a return to the caller is performed (see `return_subroutine`, which is from now on needed in the case of ground bodies only).

$$\begin{array}{|lllllllll|}
\hline
 & p & c[p: \texttt{reset_hs}] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v\ \mathcal{E} & r \\
\Rightarrow & r & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \emptyset & v\ \mathcal{E} & r \\
\hline
\end{array}$$

As the heads are ground until now $\mathcal{E}$ has not to be saved in the choice points generated by `try_me_else`.

Example Finally, let us have a look at a small example again:

$$\top \rightarrow p(f(a, b)) \tag{6}$$

$$\top \rightarrow q(a) \tag{7}$$

$$p(f(X, Y)) \wedge q(X) \rightarrow r(a) \tag{8}$$

These clauses are compiled to:

0:	`call`	5	– *initial calls*
2:	`call`	22	
4:	`proceed`		
5:	`test`	p(f(a, b))	– *head code sequence for (6)*
7:	`insert`	p(f(a, b))	
9:	`alloc_env`	2	

```
11:  env_entry_for     X
13:  write_env         a
15:  env_entry_for     Y
17:  write_env         b
19:  call_subroutine   35
21:  proceed
22:  test              q(a)          – head code sequence for (7)
24:  insert            q(a)
26:  alloc_env         2
28:  env_entry_for     X
30:  write_env         a
32:  call_subroutine   40
34:  proceed
35:  prove_bound       q(X)          – 1st entry point for (8)
37:  call              47
39:  reset_hs
40:  create_hs_entry   44            – 2nd entry point for (8)
42:  prove_ubound      p(f(X, Y))
44:  call              47
46:  reset_hs
47:  test              r(a)          – head code for (8)
49:  insert            r(a)
51:  proceed
```

3.5 Non-Ground, Range-restricted Clauses

We finally extend the logic language by introducing variables in the clause heads. This results in the logic language the PUHR tableau method can handle: range-restricted first-order clauses.

Machine and Machine Instructions The PUHR rule ground instantiates non-ground clause heads. In order to represent an instance of a clause head, the continuation queue entries additionally carry an environment, binding the variables of the corresponding head atom.

The body code sequences are extended, such that in case of a successful match for all body atoms an environment binding for the head variables is constructed and enqueued together with the address of the head code sequence. For this reason the instruction `push_param` is introduced. An entry of the continuation queue is a pair now, consisting of a substitution and an code address.[7]

[7] In the following state-transition rules the environment $\mathcal{E}$ is replaced by a pair of environments, because in a special case – as we will see later – the old environment has to be stored. The first element of the pair is the actual environment always.

$$\begin{array}{|lllllllll|}\hline & p & c[p: \texttt{push\_param}\ h] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \Rightarrow & p+2 & c & \mathcal{U} & \mathcal{P} : (\mathcal{E}|_{vars(h)}, 0) & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \hline\end{array}$$

The semantics of `proceed` changes, if $\mathcal{P}$ is not empty:

$$\begin{array}{|llllllll|}\hline & p & c[p: \texttt{proceed}] & \mathcal{U} & (\mathcal{E}', q) : \mathcal{P} & \mathcal{S} \ \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \Rightarrow & q & c & \mathcal{U} & \mathcal{P} & \mathcal{S} \ \mathcal{HS} & v & (\mathcal{E}',\mathcal{E}_o) \quad r \\ \hline\end{array}$$

The changed semantics of the instruction `call` and other instructions manipulating $\mathcal{P}$ is obvious and therefore not given here (note that `call` manipulates the last queue entry in $\mathcal{P}$ now).

Having found a possible substitution (which is given by $\mathcal{E}$) such that for all body atoms $\alpha_i\mathcal{E} \in \mathcal{U}$ holds and having called the head code sequence, hyper-resolution backtracking is now enforced by the instruction `try_next_binding` in order to find all possible bindings for the head variables.

$$\begin{array}{|lllllllll|}\hline & p & c[p: \texttt{try\_next\_binding}] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \emptyset & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \Rightarrow & r & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \emptyset & v & (\mathcal{E}_o,\mathcal{E}_o) \quad r \\ \hline\end{array}$$

$$\begin{array}{|lllllllll|}\hline & p & c[p: \texttt{try\_next\_binding}] & \mathcal{U} & \mathcal{P} & \mathcal{S} & (m, \bar{\sigma}, q) : \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \Rightarrow & q & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & (m', \bar{\sigma}, q) : \mathcal{HS} & v & (\mathcal{E}',\mathcal{E}_o) \quad r \\ \multicolumn{9}{|l|}{\text{if } \bar{\sigma} = [\sigma_0, \ldots, \sigma_n] \wedge 0 < m \leq n} \\ \multicolumn{9}{|l|}{\text{where } \mathcal{E}' = \sigma_m \circ \mathcal{E} \setminus_{dom(\sigma_0)} \wedge m' = m+1} \\ \hline\end{array}$$

$$\begin{array}{|lllllllll|}\hline & p & c[p: \texttt{try\_next\_binding}] & \mathcal{U} & \mathcal{P} & \mathcal{S} & (m, \bar{\sigma}, q) : \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \Rightarrow & p & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \multicolumn{9}{|l|}{\text{if } \bar{\sigma} = [\sigma_0, \ldots, \sigma_n] \wedge m > n} \\ \hline\end{array}$$

The instruction `reset_hs` is not necessary anymore, because `try_next_binding` subsequently pops all $\mathcal{HS}$-entries.

In the previous development step it was known at compilation time whether a body literal of a clause is resolvable with another head atom. Therefore, it was possible to perform the subroutine calls unconditionally. Now, subroutine calls to an entry point may have to be preceded by a code sequence which tests the bindings of the head variables for compatibility. For example, it is not known at compilation time, if the head atom $p(X,Y)$ is resolvable with the body atom $p(Z,Z)$. The bindings of X and Y have to be checked for equality at runtime.

Let β be a head atom and α a possibly resolvable body atom, i.e., α and β are unifiable. We assume that α and β have no common variables. In fact, as α and β could belong to the same clause, this is not necessarily true. Nevertheless, for the sake of transparency this assumption is made here.

σ_β and σ_α are two substitutions with the following properties:

1. $\sigma = mgu(\alpha, \beta)$, i.e., σ is the idempotent most general unifier of α and β.
2. $\sigma_\alpha = \sigma|_{vars(\alpha)} \wedge \sigma_\beta = \sigma|_{vars(\beta)}$

Each test sequence starts with the instruction `test_me_and_then` which saves the address to proceed, if the following tests fail. Note that `test_me_and_then` can use the same register as `call_subroutine` without the possibility of conflicts.

$$\boxed{\begin{array}{llllllll} p & c[p:\ \texttt{test\_me\_and\_then}\ q] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \Rightarrow\ p+2 & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad q \end{array}}$$

Then, after allocating a new environment with `alloc_env`, for each variable v in $dom(\sigma_\beta)$ a sequence of instructions is created, starting with `match_to` v.

The semantics of `alloc_env` changes: The previous environment is saved.

$$\boxed{\begin{array}{llllllll} p & c[p:\ \texttt{alloc\_env}] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \Rightarrow\ p+2 & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (id,\mathcal{E}) \quad r \end{array}}$$

$$\boxed{\begin{array}{llllllll} p & c[p:\ \texttt{match\_to}\ w] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \Rightarrow\ p+2 & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & w & (\mathcal{E},\mathcal{E}_o) \quad r \end{array}}$$

Each `match_to` is followed by the instruction `match`, which has the term $t = v\sigma_\beta$ as parameter. Note that t can contain variables from α and β also, and $v\mathcal{E}_o$ is ground.

$$\boxed{\begin{array}{llllllll} p & c[p:\ \texttt{match}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \Rightarrow\ r & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \multicolumn{8}{l}{\text{if } v\mathcal{E}_o \text{ and } t\mathcal{E}_o\mathcal{E} \text{ are not matchable}} \end{array}}$$

$$\boxed{\begin{array}{llllllll} p & c[p:\ \texttt{match}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) \quad r \\ \Rightarrow\ p+2 & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\sigma\circ\mathcal{E},\mathcal{E}_o) \quad r \\ \multicolumn{8}{l}{\text{if } \sigma \text{ is a substitution, such that } v\mathcal{E}_o = t\mathcal{E}_o\mathcal{E}\sigma} \end{array}}$$

Now all tests are completed. The variables in $dom(\sigma_\alpha)$ are not bound in $\mathcal{E}$ by now[8]. For this purpose the instruction `write_env` is used. The semantics of `write_env` changes, because σ_α is not necessarily a ground substitution:

$$\boxed{\begin{array}{llllllllll} & p & c[p: \texttt{write\_env}\ t] & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E},\mathcal{E}_o) & r \\ \Rightarrow & p+2 & c & \mathcal{U} & \mathcal{P} & \mathcal{S} & \mathcal{HS} & v & (\mathcal{E}[v/t\mathcal{E}_o\mathcal{E}],\mathcal{E}_o) & r \end{array}}$$

Now the code for the corresponding body can be called by the use of `call_subroutine`.

All this, i.e. saving the address to proceed if the tests fail, the tests, and writing to the environment, can be specialized for several cases. For example, if σ_β is empty, no tests and therefore no saving of the proceeding address are necessary and the environment can be filled up immediately.

Finally, note that the actual environment $\mathcal{E}$ now has to be saved in the choice points. The semantics of `try_me_else` changes in the straightforward way. The instructions `test` and `insert` now apply $\mathcal{E}$ on the passed atom before inserting it or executing the lookup.

Example The clauses

$$\top \rightarrow p(f(a), g(a)) \tag{9}$$

$$\top \rightarrow q(a, b) \tag{10}$$

$$p(f(X), Y) \wedge q(X, B) \rightarrow p(g(Y), X) \tag{11}$$

$$p(g(V), f(W)) \rightarrow r(V) \tag{12}$$

are compiled to

0:	`call`	5	– *initial calls*
2:	`call`	22	
4:	`proceed`		
5:	`test`	p(f(a), g(a))	– *head code sequence for (9)*
7:	`insert`	p(f(a), g(a))	
9:	`alloc_env`	3	
11:	`env_entry_for`	X	
13:	`write_env`	a	
15:	`env_entry_for`	Y	
17:	`write_env`	g(a)	
19:	`call_subroutine`	39	
21:	`proceed`		
22:	`test`	g(a, b)	– *head code sequence for (10)*

[8] A variable $v \in dom(\sigma_\alpha)$ could have been bound in the previous steps only if $v \in ran(\sigma_\beta)$. This is impossible because σ is idempotent.

```
24:  insert             g(a, b)
26:  alloc_env          3
28:  env_entry_for      X
30:  write_env          a
32:  env_entry_for      B
34:  write_env          b
36:  call_subroutine    48
38:  proceed
39:  create_hs_entry    43          – 1st entry point for (11)
41:  prove_ubound       q(X, B)
43:  push_param         p(g(Y), X)
45:  call               57
47:  try_next_binding
48:  create_hs_entry    52          – 2nd entry point for (11)
50:  prove_ubound       p(f(X), Y)
52:  push_param         p(g(Y), X)
54:  call               57
56:  try_next_binding
57:  test               p(g(Y), X)  – head code sequence for (11)
59:  insert             p(g(Y), X)
61:  test_me_and_then   75
63:  alloc_env          2
65:  match_to           X
67:  match              f(W)
69:  env_entry_for      V
71:  write_env_entry    Y
73:  call_subroutine    76
75:  proceed
76:  push_param         r(V)        – single entry point for (12)
78:  call               61
80:  return_subroutine
61:  test               r(V)        – head code sequence for (12)
63:  insert             r(V)
65:  proceed
```

4 The Implementation – An Overview

Designing an abstract machine is only half of the business: In order to compare its performance with existing implementations of the PUHR calculus and to validate our design, we developed a bytecode [Kin97] compiler/interpreter system. The compiler was implemented in Haskell [PH97], re-using an existing front-end from Functional Satchmo [GPS97]. The output of the compilation phase is a compact bytecode, which is then executed by an interpreter written in C++. Although this part could have been done in Haskell, too, we decided not to do so because of the danger of leaving too many things implicit (e. g., garbage collection).

Our system consists of this two parts:

- The compiler SatAMC and
- the bytecode interpreter SatAM.

An diagrammatical overview over the system is given in Figure 1.

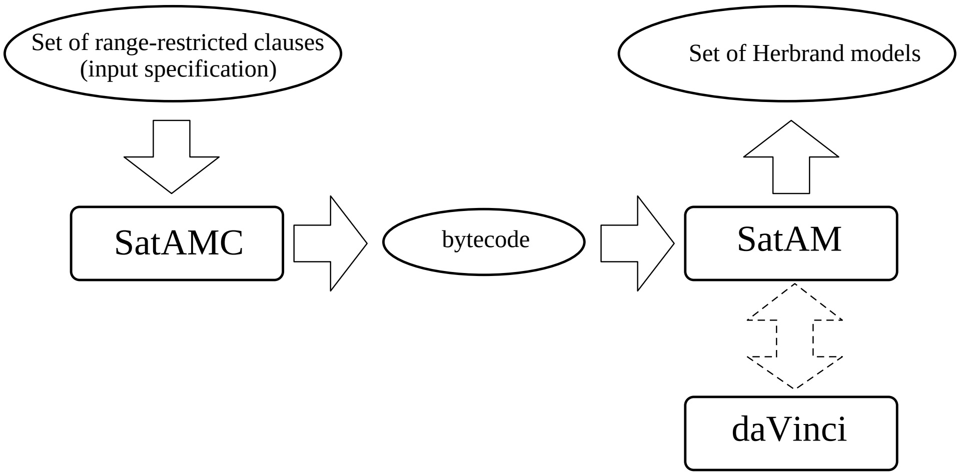


Figure 1: Diagrammatical overview of the SatAMC–SatAM system.

The input for the compiler SatAMC, the *input specification*, is a set of formulas in first-order logic. We re-used Functional Satchmo's [GPS97] front end for parsing the input specification and converting it into a set of range-restricted clauses. SatAMC compiles these clauses into bytecode as specified in section 6.

The interpreter SatAM reads this bytecode and executes it. The execution results in a computation of the models of the input specification. This computation is described in section 5.

Optionally, it is possible to visualize the current machine state and to control the machine flow with the graph visualization system *da Vinci* [FW94].

5 Concrete Realization of the Abstract Machine

5.1 Data Structures and Machine Registers

The machine data structures and registers of the concrete realization of the abstract machine are shown in Table 6.

The data structures and registers that were developed during the stepwise design were translated into the C++ implementation as directly as possible, with the differences described in the following.

As the machine is implemented in C++, all data structures are realized as objects. The methods they support are also presented in the following.

Note that in the following we use some simple data types like `stack_address`, `symbol_identifier`, or `variable_identifier` without further explanation. In the realization, these types are modeled as 32 bit integers, here they are only used for documentation purposes.

Data Structures	
`Interpretation`	The set $\mathcal{U}$
`Stack`	Combination of the choice-point stack and the hyper-resolution stack
`ContQ`	Stores the addresses of the head code sequences to be executed
`ParamQ`	Stores parameters for head calls (`ContQ` and `ParamQ` together form $\mathcal{P}$)
`Heap`	Stores term representations
`Code`	Stores the bytecode
`SymbolTable`	Maps the symbol identifiers (which are numbers in the code) to their names (Strings)
Registers	
`PC`	Program counter
`ReturnAddr`	Return or continuation address (for `call_subroutine` or `test_me_and_then`, resp.)
`Env`	Current environment (stack address)
`ActHSE`	The current hyper-resolution stack entry (stack adress)
`RootHSE`	The first hyper-resolution stack entry of the current body code sequence (stack adress)
`LastChP`	The latest choice point (stack adress)
`HeapPtr`	Pointer to a term on the Heap (required for processing linearized terms)

Table 6: Data structures and machine registers of the concrete realization of the abstract machine.

5.1.1 `SymbolTable`

It is not possible to identify the symbols in the code by their names used in the input specification, because all arguments to machine instructions should have a fixed size. So we introduced a symbol identifier[9], which is used everywhere in the machine. Symbols with the same name but a different arity get different symbol identifiers. Further on it is no longer necessary to store the arity of a symbol directly in the bytecode. Instead, the compiler generates a mapping from the symbol names and arities to their symbol identifier. This mapping is stored in the bytecode file, in the machine it is represented by `SymbolTable`. Every time the arity or name of a symbol is required, the appropriate method of `SymbolTable` is called.

Methods

`GetSymbolName(symbol : symbol_identifier) : string`

> Returns the name of the symbol with symbol identifier `symbol`. Used for output purposes only.

`GetSymbolArity(symbol : symbol_identifier) : integer`

> Returns the arity of the symbol with symbol identifier `symbol`. The arity is an important information, for it is required to work with the linearized terms correctly.

5.1.2 `Heap`

This data structure was not specified by the design, but for the implementation it is essential for storing the linearized terms. Instead of implementing a heap, we simply use the C++-heap (i. e., allocating with `new` and deallocating with `delete`). The data structure `Heap` stores only two types of nodes:[10] RefCountHeapEntries (consisting of a reference counter `Counter` and a pointer to a SymbolHeapEntry `Term`), which are required for garbage collection, and SymbolHeapEntries (consisting of a symbol identifier `Symbol` and a continuation pointer `Next`), which are used for modelling the linearized terms in the environment, in the substitutions in $\mathcal{HS}$ entries and for the machine instructions `compare`, `match`, etc. While for most dynamically allocated memory the lifetime is statically known, this is not the case for those term representations, as they may be referenced by choice points or $\mathcal{HS}$ entries. To avoid copying these terms, we introduced the RefCountHeapEntries, which are a kind of prefix to the linearized

[9]Simply an unique 32 bit number.

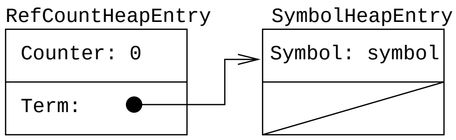
[10]While the C++-heap stores most of the data occurring in the machine (e. g., the stack, the queue nodes, the discrimination tree of the interpretation, and the temporary data the interpretation requires for computing substitutions).

terms. These RefCountHeapEntries implement reference-counting garbage collection [Col60]. This is sufficient due to the impossibility of cyclic terms. The reference counter has to be incremented when a reference to the RefCountHeapEntry is copied (e. g. when executing `push_param`), and it has to be decremented when a reference is deleted (e. g. removing an environment) or overwritten (e. g. when executing `try_next_binding`). To deallocate a RefCountHeapEntry in the implementation, it is sufficient to call the heap method `Free_Termrepresentation`. This will decrement the counter and eventually deallocate the heap entry.

Methods There are several allocation functions. After presenting the signature of each allocation function, we depict the heap entry allocated by the respective function.

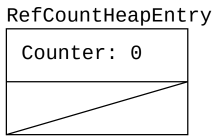
`Allocate_Termrepresentation(symbol : symbol_identifier) : RefCountHeapEntry`

Returns a pointer to the following structure:



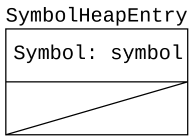
`Allocate_Termrepresentation() : RefCountHeapEntry`

Returns a pointer to the following structure:



`Allocate_Symbol(symbol : symbol_identifier) : SymbolHeapEntry`

Returns a pointer to the following structure:



`Copy_term(term : SymbolHeapEntry) : SymbolHeapEntry`

Returns a heap pointer to a new Heap entry which is a (deep) copy of `term`.

`Free_Termrepresentation(addr : RefCountHeapEntry)`

> Decrements the reference counter and deallocates the memory referenced by **addr**, if the counter is 0. Will also deallocate the term referenced by **addr.Term**[11], if not *nil*.

`Free_Symbol(addr : SymbolHeapEntry)`

> Deallocates the memory referenced by **addr**. Will also deallocate the term referenced by **addr.Next**[12], if not *nil*.

5.1.3 `Interpretation`

In the stepwise design, $\mathcal{U}$ is simply a set of atoms. This set is replaced by a kind of *discrimination tree* [McC88], resulting in two enhancements. Memory usage is reduced by sharing some common prefixes of the terms, but more importantly, matching is more efficient: The machine instructions representing a linearized term implement a depth-first traversal of the tree, aborting an unsuccessful match as soon as possible. Note that the terms have to be linearized to ensure that the parameters of each machine instruction have a fixed size (as mentioned in section 3.3 and section 5.2.1). Note that (in opposite to the binary machine instructions mentioned in section 3.3) it is not required to pass the symbol arity to the machine instructions, because this information is stored in **SymbolTable**. As specified in the design, it is required to save $\mathcal{U}$ in the choice points for backtracking purposes. In order to save stack space (and improve performance), a choice point will not store the whole **Interpretation** (or an appropriate representation), but only an state identifier.

In the following we will only describe the interface of the **Interpretation** and will not take a look at implementation details[13]. In fact, a discrimination tree is not the only possible solution, any data structure that fulfills the interface specification will be appropiate (e. g., [Gra96] presents various methods for the efficient representation of sets of terms). Due to the term representation in linearized form and the processing of a term using a sequence of machine instructions (as seen in section 3.3), the **Interpretation** has to be able to remember a current lookup/insertion position. Further on, it has to know when the linearization of a term is finished in order to reset the actual position pointer. Note that this is quite simple because the arities of the symbols are known.

[11] By calling `Free_Symbol`.

[12] Also by calling `Free_Symbol` – so `Free_Symbol` is recursive.

[13] Some tasks of the interpretation, especially the computing of substitutions for an $\mathcal{HS}$ entry, belong to the most complex parts of the abstract machine. A more detailed description would be very technical without any use for understanding the machine implementation.

Methods

`Save_state() : interpretation_state_identifier;`

Saves the current state of the interpretation for backtracking. It returns a state identifier that will be passed to `Reset_state`.

`Reset_state(oldState : interpretation_state_identifier);`

Resets the `Interpretation` to the state specified by `oldState`.

`Contains(symbol : symbol_identifier) : bool;`

Returns true if `symbol` can be found at the actual position.

`Contains(addr : heap_address) : bool;`

Return true if the term `addr` is referring to can be found at the actual position.

```
TestContains(symbol : symbol_identifier) : bool;
TestContains(addr : heap_address) : bool;
```

Same as `Contains`, but true is returned only if the linearization is finished.

```
Add(symbol : symbol_identifier) : bool;
Add(addr : heap_address) : bool;
```

Adds the symbol (the term referred to) at the current position, returns false if the symbol (one of the symbols) is already present at the current position.

```
Update_actl_HSE(symbol : symbol_identifier) : bool;
Update_actl_HSE(addr : heap_address) : bool;
Update_actl_HSE(var : variable_identifier, bound : bool) : bool;
```

There functions have to compute all substitutions that match a term stored in the `Interpretation` with the term defined by the parameters[14]. If the computation is successful (i. e., there are suitable substitutions), the actual $\mathcal{HS}$ entry is updated in the way shown in Figure 4, and true is returned, otherwise false is returned.

`Show();`

Prints the current state of `Interpretation` in readable form (i. e., complex terms with symbol names) on the screen.

[14] In other words, those substitutions σ, such that $t\sigma \in \mathcal{U}$, where t is the term defined by the parameters (which is identical to the body atom the actual $\mathcal{HS}$ entry was generated for).

5.1.4 `Stack`

The choice-point stack and the hyper-resolution stack (which were introduced in the systematic development of the abstract machine in section 3) can be merged into a single stack, because the hyper-resolution stack, which is only used in body code sequences, is always completely emptied before a previous splitting choice point (which was generated in a head code sequence) is accessed (which also happens always while executing a head code sequence). Further on, it is possible to store the environment $\mathcal{E}$ on this stack, too. It is accessed via the machine register `Env`, which simply contains an index into `Stack`. The memory layout of the stack frames is given in figures 2 to 5.[15]

Remarks

$\vdots$	$\vdots$
$frame\_ptr + 9 + (n-1)$	binding for variable n
$\vdots$	$\vdots$
$frame\_ptr + 9$	binding for variable 1
$frame\_ptr + 8$	number of variables in environment $(= n)$
$frame\_ptr + 7$	`Env`
$frame\_ptr + 6$	tail of `ParamQ`
$frame\_ptr + 5$	head of `ParamQ`
$frame\_ptr + 4$	tail of `ContQ`
$frame\_ptr + 3$	head of `ContQ`
$frame\_ptr + 2$	`Interpretation` state identifier
$frame\_ptr + 1$	previous choice point
$frame\_ptr$	address of alternative
$\vdots$	$\vdots$

Figure 2: Memory layout of a choice point.

Choice Point Figure 2 shows that instead of saving $\mathcal{P}$ and $\mathcal{U}$ completely, we store as less data as possible in a choice point. For the queues (i. e., $\mathcal{P}$), only the current head and tail addresses are stored (see section 5.1.5 for details), for the interpretation (i. e., $\mathcal{U}$), a simple state identifier is sufficient. It is a task of `Interpretation` to reset the saved state using this state identifier as the only external information. This leads to a drastic reduction

[15]Note that the variable numbers shown in these figures (which begin with 1) should not be confused with the variable indices used in the representation (which start with 0 – a tribute to the implementation language; see pseudo code in section 5.2.2 for details).

of the stack size, as the current environment is the only data structure to be stored completely in a choice point. The size of a frame in the choice-point stack is linear to the number of variables stored in the actual environment, which is identical to the number of variables in the head of the actual clause. Note that there is no space allocated in the environment for variables that exist only in the body of the actual clause.

⋮	⋮
$frame\_ptr + 4$	next $\mathcal{HS}$ entry (initially $frame\_ptr + 5$)
$frame\_ptr + 3$	substitution pointer (initially nil)
$frame\_ptr + 2$	number of variables in $\mathcal{HS}$ entry
$frame\_ptr + 1$	previous $\mathcal{HS}$ entry
$frame\_ptr$	continuation address
⋮	⋮

Figure 3: Memory layout of a $\mathcal{HS}$ entry after `create_hs_entry`.

$\mathcal{HS}$ Entry After a $\mathcal{HS}$ entry is generated by `create_hs_entry`, it has the form shown in Figure 3. After the following `prove_and_bind` sequence is executed successfully[16], the found substitutions are written to the $\mathcal{HS}$ entry in the form shown in Figure 4. Note that it is required to store additional information about the environment indices of the variables bound in the found substitutions (e. g., if the body $p(X, Y, Z) \wedge q(Y)$ is called after the insertion of $q(a)$, it is required to find all possible substitutions for X and Z, i. e., the variables with the indices 0 and 2). The size of a $\mathcal{HS}$ entry depends on two parameters. First, it depends on the number of variables to be bound, i. e., the size of a single substitution[17]. Second, the size of a $\mathcal{HS}$ entry depends on the number of possible substitutions[18].

Environment In addition to the stored bindings for the variables (which are initialized with *nil* when `alloc_env` is called), the environment has to store the number of variables it contains, the (stack) address of the previous environment (in some cases, e. g., for matching, it is required to access the previous environment), and a flag that specifies whether this environment can be removed from the stack. This flag is always true except when the environment is referenced by a choice point.

[16] Which means that suitable substitutions exist.

[17] This is the number of unbound variables in the atom the $\mathcal{HS}$ entry is generated for (as mentioned in section 3.4, a $\mathcal{HS}$ entry is generated for each atom containing unbound variables.), which is equal or less than the number of variables in the actual clause.

[18] Whis is the number of atoms already stored in `Interpretation` that can match with the atom the $\mathcal{HS}$ entry was generated for.

$\vdots$	$\vdots$
$frame\_ptr + 5 + mn + (n-1)$	mth substitution – binding for variable n
$\vdots$	$\vdots$
$frame\_ptr + 5 + mn$	mth substitution – binding for variable 1
$\vdots$	$\vdots$
$frame\_ptr + 5 + 2n + (n-1)$	2nd substitution – binding for variable n
$\vdots$	$\vdots$
$frame\_ptr + 5 + 2n$	2nd substitution – binding for variable 1
$frame\_ptr + 5 + n + (n-1)$	1st substitution – binding for variable n
$\vdots$	$\vdots$
$frame\_ptr + 5 + n$	1st substitution – binding for variable 1
$frame\_ptr + 5 + (n-1)$	environment index of variable n
$\vdots$	$\vdots$
$frame\_ptr + 5$	environment index of variable 1
$frame\_ptr + 4$	next $\mathcal{HS}$ entry (now $frame\_ptr + 5 + (m+1)n$)
$frame\_ptr + 3$	substitution pointer (now $frame\_ptr + 5 + n$)
$frame\_ptr + 2$	number of variables in $\mathcal{HS}$ entry ($= n$)
$frame\_ptr + 1$	previous $\mathcal{HS}$ entry
$frame\_ptr$	continuation address
$\vdots$	$\vdots$

Figure 4: Memory layout of a $\mathcal{HS}$ entry after `prove_and_bind`.

$\vdots$	$\vdots$
$frame\_ptr + (n-1)$	binding for variable n
$\vdots$	$\vdots$
$frame\_ptr + 1$	binding for variable 2
$frame\_ptr$	binding for variable 1
$frame\_ptr - 1$	number of variables in environment ($= n$)
$frame\_ptr - 2$	previous environment
$frame\_ptr - 3$	flag if environment can be deleted
$\vdots$	$\vdots$

Figure 5: Memory layout of an environment

Methods The symbol `various` is in the following not a special type. Instead, it means that the parameter or return value can have different types.[19]

`Push(new_entry : various);`

Pushes `new_entry` on the stack.

`Pop() : various;`

Returns the topmost entry of the stack and removes this entry.

`Top() : stack_address`

Returns the top of the stack. Top is the index of the first free stack entry, this means it is the index of the next entry to be pushed.

`operator [](idx : stack_address) : various`

Returns the stack entry at the index `idx`.

`Set_top(newTop : stack_address)`

Sets the top of stack to `newTop`. This is useful if a lot of data should be popped at once.

5.1.5 `ContQ` and `ParamQ`

While in the design phase there was only one queue storing instances of clause heads (in the form of the start address of the corresponding head code sequences and later as pairs of the start address and an environment), it is required to split it into two synchronous queues: `ContQ`, which stores the addresses of the head code sequences, and `ParamQ`, which stores pointers to the bindings of the variables (in a defined order). This splitting is required for correct deallocation: in some cases, e. g., if the current branch is closed with `fail`, there can be many entries left in $\mathcal{P}$ which are not longer used. These entries must be deallocated correctly. While this requires nothing to do for `ContQ`, `ParamQ` has to free the stored bindings if they are not referenced from elsewhere (see section 5.1.2 for details).

To enable backtracking and to prevent that too much data is stored in a choice point, another kind of reference counting [Col60] is used. Each queue node has a reference counter to state by how many choice points it is referenced. When saving the state of a queue in a choice point, the reference counter of all nodes between head and tail are incremented. when popping a queue entry, the entry is only removed from the queue if it is not referenced in a choice points (i. e., if its reference counter is 0). So it is sufficient to store only two pointer for a queue inside a choice point (four pointers for both queues) instead of storing the whole content.

[19] In the concrete implementation in C++ this requires a lot of type-casting.

Methods

```
EnQuque(entry : code_address);
EnQuque(entry : heap_address);
```

Adds a new entry at the tail of the queue.

```
DeQueue() : code_address;
Dequeue() : heap_address;
```

Returns the head of the queue and eventually removes it from the queue, if its reference counter is 0.

```
LastEntryDeleted() : bool
```

Returns true, if the last call of `DeQueue` removed the entry from the queue, else it returns false.

```
IsEmpty() : bool
```

Returns true if there is no data stored in the queue.

```
GetHead() : queue_address;
GetTail() : queue_address;
```

Returns a pointer to the head or tail, respectively, of the queue. This is required for saving state information in choice points. `GetTail` increments the reference counter of all nodes from the heap to the tail of the queue.

```
Reset_state(head, tail : queue_address)
```

Resets the queue to the state where `head` is the new head and `tail` is the new tail. The reference counters of the queue entries between head and tail are decremented.

5.2 The Machine Instruction Set

5.2.1 From Design to Concrete Realization

Table 7 shows the differences between the instruction set developed in the design phase and that used in the concrete realization. Most instructions are identical, the difference result from the different modelings of the terms: In the design code a complex term can be argument of a machine instruction, in the realization the terms have to be split into their components, i. e., symbols and variables. The different components have to be progressed in a different way, so all machine instructions operating on terms exist in two or more versions: one for symbols, and one (ore more, if the concerned variables can have more than one state, like bound or unbound) for variables.

A special case is `match`. the instructions `match_ ... _begin`, which are executed at the beginning of a match-sequence, are necessary because of the structure of the linearized terms: as described in section 5.1.2, the linearized term always begins with a RefCountHeapEntry – this reflects in the necessity of `match_ ... _begin`.

Some instructions were not introduced by the design:

- `pop_param`, which dequeues the binding for a variable of the head atoms from `ParamQ`, was not required because in the design the whole environment was dequeued in a single step together with the code start address in `proceed`. This is no longer possible in the more explicit sight of the realization: Although it would have been possible to have the number of variable bindings to be popped as a second parameter of `proceed`, there is no way to hand over the indices of the variables to be bound without violating the constraint that all machine instructions should have a fixed size.

- `compare` is a special case of `match`, where all related terms are either ground, or all variables are already bound. This is an optimization, which is not really essential for soundness and completeness of the machine (`match` would be sufficient – with a bit more overhead).

5.2.2 Implementation of the Machine Instructions

In this section, the machine will be described with the help of pseudo-code.[20] Note that this is a simplified presentation; technical details like type-casts, especially when referencing stack entries, have been omitted for better readability.

Main Loop

```
Main()
BEGIN
  // all data structures are initialized empty
  PC := 0;
  HeapPtr := nil;
  ReturnAddr := invalid;
  Env := invalid;
  ActHSE := invalid;
  RootHSE := invalid;
  read bytecode from file;
  // main loop
```

[20]The symbol `invalid` denotes an invalid stack or code address just like `nil` denotes an invalid heap address.

Design	**Realization**	**Parameter**
`call`	`call`	*code address*
`call_subroutine`	`call_subroutine`	*code address*
`return_subroutine`	`return_subroutine`	
`test_me_and_then`	`test_me_and_then`	*code address*
`proceed`	`proceed`	
`try_me_else`	`try_me_else`	*code address*
`create_hs_entry`	`create_hs_entry`	*number*, *code address*
`try_next_binding`	`try_next_binding`	
`fail`	`fail`	
`alloc_env`	`alloc_env`	*number*
`env_entry_for`	`env_entry_for`	*variable identifier*
`write_env`	`write_env_begin`	*symbol identifier*
	`write_env_old_var_begin`	*variable identifier*
	`write_env_new_var_begin`	*variable identifier*
	`write_env`	*symbol identifier*
	`write_env_old_var`	*variable identifier*
	`write_env_new_var`	*variable identifier*
not introduced	`pop_param`	*variable identifier*
`insert`	`insert`	*symbol identifier*
	`insert_var`	*variable identifier*
`test`	`test`	*symbol identifier*
	`test_var`	*variable identifier*
`test_and_insert`	`test_and_insert`	*symbol identifier*
	`test_and_insert_var`	*variable identifier*
`proved, prove_bound`	`proved`	*symbol identifier*
	`proved_bvar`	*variable identifier*
`prove_ubound`	`prove_and_bind`	*symbol identifier*
	`prove_and_bind_envvar`	*variable identifier*
	`prove_and_bind_bvar`	*variable identifier*
	`prove_and_bind_uvar`	*variable identifier*
not introduced	`compare_to`	*variable identifier*
not introduced	`compare`	*symbol identifier*
	`compare_var`	*variable identifier*
`match_to`	`match_to`	*variable identifier*
`match`	`match`	*symbol identifier*
	`Match_bvar_oldenv`	*variable identifier*
	`Match_bvar_env`	*variable identifier*
	`Match_uvar_env`	*variable identifier*
`push_param`	`push_param`	*variable identifier*

Table 7: Differences between the machine instruction set developed in the design phase and that usage in the realization.

```
  WHILE true DO
    run instruction handler for Code[PC];
  ENDWHILE
END
```

Helper Functions

- `Go_on` dequeues the start address of the next head code sequence from `ContQ`, if possible. Else `Interpretation` contains a model of the input specification. This model is printed and the user is asked if more models should be computed. If yes, backtracking is enforced. Else the machine is stopped.

```
Go_on()
BEGIN
  Remove_env();
  IF NOT ContQ.IsEmpty() THEN
    PC := ContQ.DeQueue();
  ELSE
    Interpretation.Show();
    IF user wants more models THEN
      Backtrack();
    ELSE
      Halt();
    ENDIF
  ENDIF
END
```

- `Backtrack` restores `Interpretation`, `ContQ`, `ParamQ` and the environment according to the data found in the last choice point. If there is no such choice point, the machine is stopped.

```
Backtrack()
  binding : RefCountHeapEntry
  i       : integer
  actlChP : stack_address
BEGIN
  IF LastChP = invalid THEN
    Halt();
  ELSE
    PC := Stack[LastChP];
    Interpretation.Reset_state(Stack[LastChP + 2]);
    ContQ.Reset_state(Stack[LastChP + 3], Stack[LastChP + 4]);
    ParamQ.Reset_state(Stack[LastChP + 5], Stack[LastChP + 6]);
    IF Stack[LastChP + 7] != invalid THEN
      FOR i := 0 TO Stack[LastChP + 8] - 1 DO
        binding := Stack[Stack[LastChP + 7] + i];
```

```
        IF binding != nil THEN
          Free_Termrepresentation(binding);
        ENDIF
        Stack[Stack[LastChP + 7] + i] := Stack[LastChP + 9 + i];
      ENDFOR
      Stack[Stack[LastChP + 7] - 3] := true;
    ENDIF
    actlChP := LastChP;
    LastChP := Stack[LastChP + 1];
    Stack.Set_top(actlChP);
  ENDIF
END
```

- `Halt` stops the machine.

```
Halt()
BEGIN
  stop the machine;
END
```

- `Return` sets the `PC` to the value of `ReturnAddr`. It has a bool as parameter, specifying whether he environment should be removed or not.

```
Return(remove_env : bool)
BEGIN
  IF remove_env THEN
    Remove_Env();
  ENDIF
  PC := ReturnAddr;
  ReturnAddr := invalid;
END
```

- `Remove_Env` pops the environment from the stack, deallocating all bindings in the environment.

```
Remove_Env()
  binding : RefCountHeapEntry;
  i       : integer;
BEGIN
  IF Env != invalid AND Stack[Env - 3] = true THEN
    FOR i := 1 TO Stack[Env - 1] DO
      binding := Stack.Pop();
      IF binding != nil THEN
        Free_Termrepresentation(binding);
      ENDIF
    ENDFOR
```

```
    Stack.Pop();
    Env := Stack.Pop();
    Stack.Pop();
  ENDIF
END
```

- `Next_HSE_binding` updates the environment according to the next substitution stored in the current $\mathcal{HS}$ entry. If there is no substitution or there is no substitution left, `ActHSE` is set to the previous $\mathcal{HS}$ entry and the next substitution of that $\mathcal{HS}$ entry is used (by calling `Next_HSE_binding` recursively). If there is no $\mathcal{HS}$ entry (which happens only in recursive calls), this means all possible substitutions have been used; so the $\mathcal{HS}$ entries can be popped from stack and control is returned to the caller

```
Next_HSE_binding()
  binding : RefCountHeapEntry
  i       : integer
BEGIN
  IF ActHSE = invalid THEN
    Remove_HSEs();
    Return(true);
  ELSE
    IF Stack[ActHSE + 3] = invalid THEN
      ActHSE := Stack[ActHSE + 1];
      Next_HSE_binding();
    ELSE
      IF Stack[ActHSE + 3] + Stack[ActHSE + 2] < Stack[ActHSE + 4] THEN
        Stack[ActHSE + 3] := Stack[ActHSE + 3] + Stack[ActHSE + 2];
        FOR i := 0 TO Stack[ActHSE + 2] - 1 DO
          binding := Stack[Env + Stack[ActHSE + 5 + i]];
          IF binding != nil THEN
            Free_Termrepresentation(binding);
          ENDIF
          binding := Stack[Stack[ActHSE + 3] + i];
          binding.Counter := binding.Counter + 1;
          Stack[Env + Stack[ActHSE + 5 + i]] := binding;
        ENDFOR
        PC := Stack[ActHSE];
      ELSE
        Stack[ActHSE + 3] := ActHSE + 5 + Stack[ActHSE + 2];
        FOR i := 0 TO Stack[ActHSE + 2] - 1 DO
          binding := Stack[Env + Stack[ActHSE + 5 + i]];
          IF binding != nil THEN
            Free_Termrepresentation(binding);
          ENDIF
          Stack[Env + Stack[ActHSE + 5 + i]] := nil;
        ENDFOR
```

```
        ActHSE = Stack[ActHSE + 1];
        Next_HSE_binding();
      ENDIF
    ENDIF
  ENDIF
END
```

- Remove_HSEs removes all $\mathcal{HS}$ entries from the stack, the bindings in the substitutions are deallocated. As this is done while returning from a subroutine call, there are only $\mathcal{HS}$ entries above RootHSE on the stack. The exact number is unknown (but limited by the number of atoms in the body of the clause). So the $\mathcal{HS}$ entries are removed successive, beginning from RootHSE. To free all variable bindings, it is required to iterate over all substitutions (their number is also unknown but limited by the size of $\mathcal{U}$)[21]. Because of the layout of a $\mathcal{HS}$ entry and because all substitutions inside a single $\mathcal{HS}$ entry have the same size, it is possible to start with the first and continue as long as the stack index of the next substitutions is less than the index of the next $\mathcal{HS}$ entry.

```
Remove_HSEs()
  currentHSE, nextHSE, currentSubst : stack_address
  binding                           : RefCountHeapEntry
  i                                 : integer
BEGIN
  currentHSE := RootHSE;
  WHILE currentHSE < Stack.Top() DO
    nextHSE := Stack[currentHSE + 4];
    currentSubst := currentHSE + 5 + Stack[currentHSE + 2];
    IF currentSubst != invalid THEN
      WHILE currentSubst < nextHSE DO
        FOR i := 0 TO Stack[currentHSE + 2] - 1 DO
          binding := Stack[currentSubst + i];
          Free_Termrepresentation(binding);
        ENDFOR
        currentSubst := currentSubst + Stack[currentHSE + 2];
      ENDWHILE
    ENDIF
  ENDWHILE
  Stack.Set_top(RootHSE);
END
```

- Check_Var_Binding tests if the term referenced by addr is a prefix ot the term referenced by HeapPtr. If this is the case, returns true, otherwise false is returned.

[21] The size of $\mathcal{U}$ is arbitrarily large, e. g., if there are recursive clauses in the input specification.

```
Check_Var_Binding(addr : SymbolHeapEntry) : bool
BEGIN
  WHILE addr != nil AND HeapPtr != nil
        AND addr.Symbol = HeapPtr.Symbol DO
    addr := addr.Next;
    HeapPtr := HeapPtr.Next;
  ENDWHILE
  IF addr = nil THEN
    return true;
  ELSE
    return false;
```

Instruction Handlers In the C++ implementation of the machine, there is a single `switch` statement dispatching the instruction. Therefore, we do not name the instruction handlers in the following.

- `call`

```
BEGIN
  ContQ.EnQueue(Code[PC + 1]);
  PC := PC + 2;
END
```

- `call_subroutine`

```
BEGIN
  ReturnAddr := PC + 2;
  PC := Code[PC + 1];
END
```

- `return_subroutine`

```
BEGIN
  Return(true);
END
```

- `test_me_and_then`

```
BEGIN
  ReturnAddr := Code[PC + 1];
  PC := PC + 2;
END
```

- `proceed`

```
BEGIN
  Go_on();
END
```

- `try_me_else`

```
  oldChP : stack_address
  i      : integer
BEGIN
  oldChP := LastChP;
  LastChP := Stack.Top();
  Stack.Push(Code[PC + 1]);
  Stack.Push(oldChP);
  Stack.Push(Interpretation.Save_state());
  Stack.Push(ContQ.GetHead());
  Stack.Push(ContQ.GetTail());
  Stack.Push(ParamQ.GetHead());
  Stack.Push(ParamQ.GetTail());
  Stack.Push(Env);
  IF Env = invalid
    Stack.Push(0);
  ELSE
    Stack.Push(Stack[Env - 1]);
    FOR i := 0 TO Stack[Env - 1] - 1 DO
      Stack.Push(Stack[Env + i]);
      (Stack[Env + i]).Counter := (Stack[Env + i]).Counter + 1;
    ENDFOR
    Stack[Env - 3] := false;
  ENDIF
  PC := PC + 2;
END
```

- `create_hs_entry`

```
  old_top_of_stack : stack_address
BEGIN
  old_top_of_Stack := Stack.Top();
  Stack.Push(Code[PC + 2]);
  Stack.Push(ActHSE);
  Stack.Push(Code[PC + 1]);
  Stack.Push(invalid);
  Stack.Push(old_top_of_Stack + 5);
  ActHSE := old_top_of_Stack;
  PC := PC + 3;
END
```

- `try_next_binding`

```
BEGIN
  Next_HSE_binding();
END
```

- `fail`

```
BEGIN
  Remove_HSEs();
  IF ReturnAddr != invalid THEN
    Return(true);
  ENDIF
  Remove_env();
  Backtrack();
END
```

- `alloc_env`

```
  i : integer
BEGIN
  Stack.Push(true);
  Stack.Push(Env);
  Stack.Push(Code[PC + 1]);
  Env := Stack.Top();
  FOR i := 0 TO Code[PC + 1] - 1 DO
    Stack.Push(nil);
  ENDFOR
  ActHSE := invalid;
  RootHSE := Stack.Top();
  PC := PC + 2;
END
```

- `env_entry_for`

```
BEGIN
  HeapPtr := Allocate_Termrepresentation();
  Stack[Env + Code[PC + 1]] := HeapPtr;
  HeapPtr.Counter := HeapPtr.Counter + 1;
  PC := PC + 2;
END
```

- `write_env_begin`

```
BEGIN
  HeapPtr.Term := Allocate_Symbol(Code[PC + 1]);
  HeapPtr := HeapPtr.Term;
  PC := PC + 2;
END
```

- `write_env_old_var_begin`

```
BEGIN
  HeapPtr.Term := Copy_term((Stack[Stack[Env - 2] + Code[PC + 1]]).Term);
  HeapPtr := HeapPtr.Term;
  PC := PC + 2;
END
```

- write_env_new_var_begin

```
BEGIN
  HeapPtr.Term := Copy_term((Env + Code[PC + 1]]).Term);
  HeapPtr := HeapPtr.Term;
  PC := PC + 2;
END
```

- write_env

```
BEGIN
  HeapPtr.Next := Allocate_Symbol(Code[PC + 1]);
  HeapPtr := HeapPtr.Next;
  PC := PC + 2;
END
```

- write_env_old_var

```
BEGIN
  HeapPtr.Next := Copy_term((Stack[Stack[Env - 2] + Code[PC + 1]]).Term);
  HeapPtr := HeapPtr.Next;
  PC := PC + 2;
END
```

- write_env_new_var

```
BEGIN
  HeapPtr.Next := Copy_term((Stack[Env + Code[PC + 1]]).Term);
  HeapPtr := HeapPtr.Next;
  PC := PC + 2;
END
```

- pop_param

```
  addr : RefCountHeapEntry
BEGIN
  addr := ParamQ.DeQueue();
  Stack[Env + Code[PC + 1]] := addr;
  IF NOT ParamQ.LastEntryDeleted() THEN
    addr.Counter := addr.Counter + 1;
  ENDIF
  PC := PC + 2;
END
```

- insert

```
BEGIN
  Interpretation.Add(Code[PC + 1]);
  PC := PC + 2;
END
```

- `insert_var`

```
BEGIN
  Interpretation.Add(Stack[Env + Code[PC + 1]]);
  PC := PC + 2;
END
```

- `test_and_insert`

```
BEGIN
  IF Interpretation.Add(Code[PC + 1]) THEN
    PC := PC + 2;
  ELSE
    Go_on();
  ENDIF
END
```

- `test_and_insert_var`

```
BEGIN
  IF Interpretation.Add(Stack[Env + Code[PC + 1]]) THEN
    PC := PC + 2;
  ELSE
    Go_on();
  ENDIF
END
```

- `test`

```
BEGIN
  IF Interpretation.TestContains(Code[PC + 1]) THEN
    Go_on();
  ELSE
    PC := PC + 2;
  ENDIF
END
```

- `test_var`

```
BEGIN
  IF Interpretation.TestContains(Stack[Env + Code[PC + 1]]) THEN
    Go_on();
  ELSE
    PC := PC + 2;
  ENDIF
END
```

- proved

```
BEGIN
  IF Interpretation.Contains(Code[PC + 1]) THEN
    PC := PC + 2;
  ELSE
    Next_HSE_binding();
  ENDIF
END
```

- proved_bvar

```
BEGIN
  IF Interpretation.Contains(Stack[Env + Code[PC + 1]]) THEN
    PC := PC + 2;
  ELSE
    Next_HSE_binding();
  ENDIF
END
```

- prove_and_bind

```
BEGIN
  IF Interpretation.Update_actl_HSE(Code[PC + 1]) THEN
    PC := PC + 2;
  ELSE
    Next_HSE_binding();
  ENDIF
END
```

- prove_and_bind_envvar

```
BEGIN
  IF Interpretation.Update_actl_HSE(Stack[Env + Code[PC + 1]]) THEN
    PC := PC + 2;
  ELSE
    Next_HSE_binding();
  ENDIF
END
```

- prove_and_bind_bvar

```
BEGIN
  IF Interpretation.Update_actl_HSE(Code[PC + 1], true) THEN
    PC := PC + 2;
  ELSE
    Next_HSE_binding();
  ENDIF
END
```

- `prove_and_bind_uvar`

```
BEGIN
  IF Interpretation.Update_actl_HSE(Code[PC + 1], false) THEN
    PC := PC + 2;
  ELSE
    Next_HSE_binding();
  ENDIF
END
```

- `compare_to`

```
BEGIN
  HeapPtr := (Stack[Env + Code[PC + 1]]).Term;
  PC := PC + 2;
END
```

- `compare`

```
BEGIN
  IF HeapPtr.Symbol = Code[PC + 1] THEN
    HeapPtr := HeapPtr.Next;
    PC := PC + 2;
  ELSE
    Return(false);
  ENDIF
END
```

- `compare_var`

```
BEGIN
  IF Check_Var_Binding((Stack[Env + Code[PC + 1]]).Term) THEN
    PC := PC + 2;
  ELSE
    Return(false);
END
```

- `match_to`

```
BEGIN
  HeapPtr := (Stack[Stack[Env - 2] + Code[PC + 1]]).Term;
  PC := PC + 2;
END
```

- `match`

```
BEGIN
  IF HeapPtr.Symbol = Code[PC + 1] THEN
    HeapPtr := HeapPtr.Next;
```

```
    PC := PC + 2;
  ELSE
    Return(true);
  ENDIF
END
```

- `match_bvar_oldenv`

```
BEGIN
  IF Check_Var_Binding((Stack[Stack[Env - 2] + Code[PC + 1]]).Term) THEN
    PC := PC + 2;
  ELSE
    Return(true);
  ENDIF
END
```

- `match_bvar_env`

```
BEGIN
  IF Check_Var_Binding((Stack[Env + Code[PC + 1]]).Term) THEN
    PC := PC + 2;
  ELSE
    Return(true);
  ENDIF
END
```

- `match_uvar_env`

```
  addr    : heap_address;
  counter : integer
BEGIN
  addr := Allocate_Termrepresentation(HeapPtr.Symbol);
  Stack[Env + Code[PC + 1]] := addr;
  addr.Counter := addr.Counter + 1;
  addr := addr.Term;
  HeapPtr := HeapPtr.Next;
  counter := SymbolTable.GetSymbolArity(addr.Symbol);
  WHILE counter > 0 DO
    addr.Next := Allocate_Symbol(HeapPtr.Symbol);
    counter := counter - 1 + SymbolTable.GetSymbolArity(HeapPtr.Symbol);
    addr := addr.Next;
    HeapPtr := HeapPtr.Next;
  ENDWHILE
  PC := PC + 2;
END
```

- push_param

```
  addr : RefCountHeapEntry
BEGIN
  addr := Stack[Env + Code[PC + 1]];
  ParamQ.EnQueue(addr);
  addr.Counter := addr.Counter + 1;
  PC := PC + 2;
END
```

5.3 Visualization of the Abstract Machine

For demonstration purposes we connected the implementation of the abstract machine with the graph visualization system *daVinci* [FW94]. This graphical user interface to the abstract machine enables the user to step through the abstract machine code and to view all data structures of the machine in a graph representation. An example can be seen in Figure 6. Additionally, it was extremely useful for debugging the implementation of

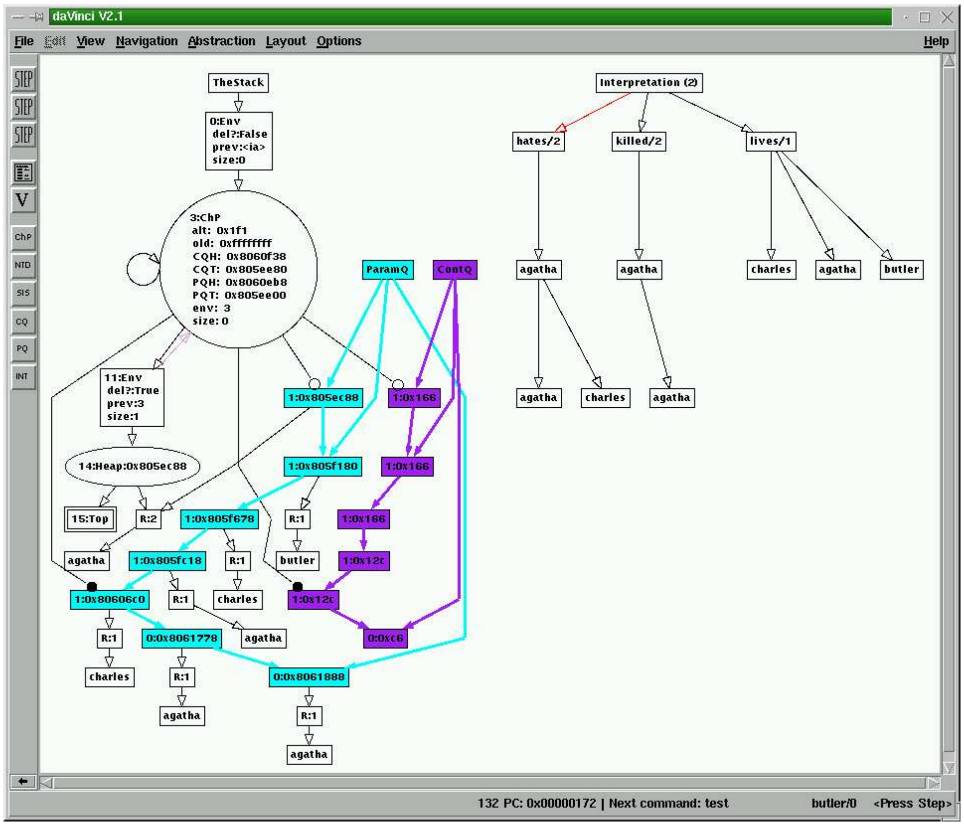


Figure 6: Graphical user interface to the abstract machine using *daVinci*.

the machine. Furthermore, we expect to recognize possible inefficiencies and design flaws of the machine and machine code.

6 The Compiler SatAMC

This section describes the compiler SatAMC, which translates range-restricted clauses to bytecode appropriate for the abstract machine. First and mainly, the compilation schemes are presented. Then an idea of the concrete implementation is given.

6.1 The Compilation Schemes

6.1.1 Notation

The compilation schemes are a set of functions. Every function returns a flat sequence of machine instructions. The function names are written in $\mathcal{CALLIGRAPHIC}$ font, instructions as above – in `typewriter`. A sequence of instructions is written in square brackets. Let $r \circ s$ be the sequence which results from appending the sequence s to the sequence r. If an argument is a syntactical part of the clauses to translate, it is surrounded by pointed brackets.

If "ϕ :" precedes a code sequence, ϕ is a label denoting the address of the first instruction in this sequence. *codeAddr* dereferences such a label, i.e., $codeAddr(\phi)$ is the code address of the instruction denoted by ϕ.

$\mathcal{F}a_i \circ \cdots \circ \mathcal{F}a_j$ means [], if $i > j$ and $\mathcal{F}a_i \circ (\mathcal{F}a_{i+1} \circ \cdots \circ \mathcal{F}a_j)$ otherwise. Similarly, $[\texttt{instr}\ a_i, \ldots, \texttt{instr}\ a_j]$ means [], if $i > j$, otherwise $[\texttt{instr}\ a_i] \circ [\texttt{instr}\ a_{i+1}, \ldots, \texttt{instr}\ a_j]$.

The function *vars* is extended from atoms and terms to clauses and its parts. *vars* returns a list of variables in order of their appearance in the argument. If set theoretic operations like $\cup$ are applied to such lists, the lists are treated as sets. In this case the ordering is ignored, of course. *length* calculates the length of a list.

6.1.2 Description of the Schemes

In Figure 7 we give a call graph of the 21 compilation schemes given in section 6.1.2 in order to simplify the orientation.

$\mathcal{TRANSLATION}$ This scheme gets a list of the given clauses as argument. Let $Addr_1, \ldots, Addr_m$ be the code addresses of the translations of the facts appearing in the clauses. This scheme is the main one, i.e., to translate clauses to bytecode, they are passed to $\mathcal{TRANSLATION}$.

$$\mathcal{TRANSLATION}\ \langle Clause_1, \ldots, Clause_n \rangle =$$

$$[\texttt{call}\ Addr_1,\ \ldots,\ \texttt{call}\ Addr_m,\ \texttt{proceed}] \circ$$
$$\mathcal{CLAUSE}\ \langle Clause_1 \rangle \circ \ldots \circ \mathcal{CLAUSE}\ \langle Clause_n \rangle$$

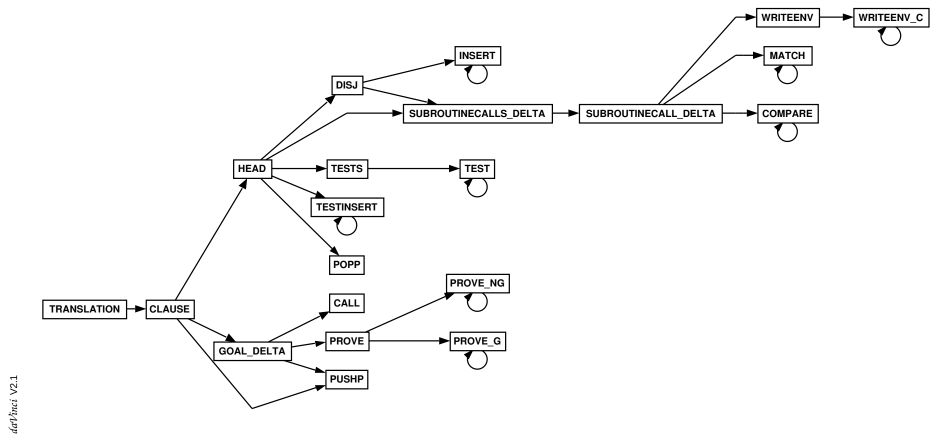


Figure 7: Call graph for the compilation schemes given in section 6.1.2. The nodes are labeled with the function names. An edge from a to b means that a can call b.

$\mathcal{CLAUSE}$ This scheme gets a clause as argument.

$\mathcal{CLAUSE}\ \langle \alpha_1, \ldots, \alpha_n \rightarrow \beta_1; \ldots; \beta_m \rangle =$

- If $n \leq 1 \wedge m = 0$:
  [`fail`]

- If $n = 0 \wedge m \neq 0$:
  $\mathcal{HEAD}\ \langle \beta_1; \beta_2; \ldots; \beta_m \rangle$

- If $n = 1 \wedge m \neq 0$:
  $\mathcal{PUSHP}\ \ vars([\beta_1; \ldots; \beta_m])\ \circ$
  [`call` $codeAddr(\phi)$] $\circ$
  [`return_subroutine`] $\circ$
  ϕ: $\mathcal{HEAD}\ \langle \beta_1; \beta_2; \ldots; \beta_m \rangle$

- Else ($n > 1$):
  $\mathcal{GOAL}_\Delta\ \langle \alpha_1, \ldots, \alpha_n \rightarrow \beta_1; \ldots; \beta_m \rangle\ 1\ \ \phi\ \circ$
  $\mathcal{GOAL}_\Delta\ \langle \alpha_1, \ldots, \alpha_n \rightarrow \beta_1; \ldots; \beta_m \rangle\ 2\ \ \phi\ \circ$
  $\vdots$
  $\mathcal{GOAL}_\Delta\ \langle \alpha_1, \ldots, \alpha_n \rightarrow \beta_1; \ldots; \beta_m \rangle\ n\ \ \phi\ \circ$
  ϕ: $\mathcal{HEAD}\ \langle \beta_1; \beta_2; \ldots; \beta_m \rangle$

$\mathcal{HEAD}$ This scheme gets the head of a clause as argument.

$\mathcal{HEAD}\ \langle \beta_1; \ldots; \beta_m \rangle =$

- If $m = 0$
  []
- If $m = 1$:
  [`alloc_env` $\mathit{length}(\mathit{vars}(\beta_1))$] $\circ$
  $\mathcal{POPP}$ $\mathit{vars}(\beta_1)$ $\circ$
  $\mathcal{TESTINSERT}$ $\langle\beta_1\rangle$ $\circ$
  $\mathcal{SUBROUTINECALLS}_\Delta$ $\langle\beta_1\rangle$
- Else ($m > 1$):
  [`alloc_env` $\mathit{length}(\mathit{vars}([\beta_1;\ldots;\beta_m]))$] $\circ$
  $\mathcal{POPP}$ $\mathit{vars}([\beta_1;\ldots;\beta_m])$ $\circ$
  $\mathcal{TESTS}$ $\langle\beta_1;\ldots;\beta_m\rangle$ $\circ$
  $\mathcal{DISJ}$ $\langle\beta_1;\ldots;\beta_m\rangle$

$\mathcal{DISJ}$ This scheme gets the head of a clause as argument.

$\mathcal{DISJ}$ $\langle\beta_1;\ldots;\beta_n\rangle$ =

[`try_me_else` $\mathit{codeAddr}(\phi_1)$] $\circ$
$\mathcal{INSERT}$ $\langle\beta_1\rangle$ $\circ$
$\mathcal{SUBROUTINECALLS}_\Delta$ $\langle\beta_1\rangle$ $\circ$
ϕ_1 : [`try_me_else` $\mathit{codeAddr}(\phi_2)$] $\circ$
$\mathcal{INSERT}$ $\langle\beta_2\rangle$ $\circ$
$\mathcal{SUBROUTINECALLS}_\Delta$ $\langle\beta_2\rangle$ $\circ$
$\vdots$
ϕ_{n-1} : $\mathcal{INSERT}$ $\langle\beta_n\rangle$ $\circ$
$\mathcal{SUBROUTINECALLS}_\Delta$ $\langle\beta_n\rangle$

$\mathcal{INSERT}$ This scheme gets a head atom or a subterm of a head atom as argument.

$\mathcal{INSERT}$ $\langle\beta\rangle$ =

- If $\beta = f(t_1,\ldots,t_n)$:
  [`insert` f_n] $\circ$
  $\mathcal{INSERT}$ $\langle t_1\rangle$ $\circ \ldots \circ$ $\mathcal{INSERT}$ $\langle t_n\rangle$
- Else:
  [`insert_var` β]

$\mathcal{TESTINSERT}$ This scheme gets a head atom or a subterm of a head atom as argument.

$\mathcal{TESTINSERT}$ $\langle\beta\rangle$ =

- If $\beta = f(t_1, \ldots, t_n)$:
  $[\texttt{test\_and\_insert}\ f_n] \circ$
  $\mathcal{TESTINSERT}\ \langle t_1 \rangle\ \circ \ldots \circ \mathcal{TESTINSERT}\ \langle t_n \rangle$

- Else:
  $[\texttt{test\_and\_insert\_var}\ \beta]$

$\mathcal{TESTS}$ This scheme gets the head of a clause as argument.

$\mathcal{TESTS}\ \langle \beta_1; \ldots; \beta_m \rangle =$

$\mathcal{TEST}\ \langle \beta_1 \rangle\ \circ \ldots \circ \mathcal{TEST}\ \langle \beta_n \rangle$

$\mathcal{TEST}$ This scheme gets a head atom or a subterm of a head atom as argument.

$\mathcal{TEST}\ \langle \beta \rangle =$

- If $\beta = f(t_1, \ldots, t_n)$:
  $[\texttt{test}\ f_n] \circ$
  $\mathcal{TEST}\ \langle t_1 \rangle\ \circ \ldots \circ \mathcal{TEST}\ \langle t_n \rangle$

- Else:
  $[\texttt{test\_var}\ \beta]$

$\mathcal{SUBROUTINECALLS}_\Delta$ This scheme gets a head atom β as argument. Let $g_1, \ldots, g_n$ be a list of quintuples $(addr, n, m, \sigma_\beta, \sigma_\alpha)$ where

- *addr* is the code address of an entry point α of a possibly resolvable clause ψ, i.e., α is a body atom of ψ such that α and β are unifiable.
- n is the number of variables in α.
- m is the number of variables in ψ.
- σ_α and σ_β are substitutions with the properties described in section 3.5. Additionally we require $\forall v \in \beta : v\sigma_\beta \notin vars(\alpha)$. This is not necessary, but makes the generation of more efficient code possible.

$\mathcal{SUBROUTINECALLS}_\Delta\ \langle \beta \rangle =$

$\mathcal{SUBROUTINECALL}_\Delta\ g_1\ \ vars(\beta)\ \ \phi_1\ \circ$
ϕ_1: $\mathcal{SUBROUTINECALL}_\Delta\ g_2\ \ vars(\beta)\ \ \phi_2\ \circ$
$\vdots$
ϕ_{n-1}: $\mathcal{SUBROUTINECALL}_\Delta\ g_n\ \ vars(\beta)\ \ \phi_n\ \circ$
ϕ_n: $[\texttt{proceed}]$

$\mathcal{SUBROUTINECALL}_\Delta$ This scheme gets a quintuple $(addr, n, m, \sigma_\beta, \sigma_\alpha)$ as described above, a set of variables V and a label ϕ as arguments. V is the set of variables of the head atom β, which is possibly resolvable with an body atom α of a clause ψ. $\mathcal{SUBROUTINECALL}_\Delta$ generates the code for the runtime testing of resolvability (described in section 3.5). If this tests fail at runtime, the proceeding point is at the address labeled by ϕ.

Let $\sigma_\beta = [(v_1, t_1), \ldots, (v_k, t_k)]$ and $\sigma_\alpha = [(v_{\Delta 1}, t_{\Delta 1}), \ldots, (v_{\Delta l}, t_{\Delta l})]$.

$\mathcal{SUBROUTINECALL}_\Delta\ [addr, n, m, \sigma_\beta, \sigma_\alpha]\ V\ \ \phi =$

- If $k = 0$ ($\Rightarrow l = n$):
  [`alloc_env` m] $\circ$
  [`env_entry_for` $v_{\Delta 1}$] $\circ$
  $\mathcal{WRITEENV}\ \langle t_{\Delta 1}\rangle\ V\ \circ$
  $\vdots$
  [`env_entry_for` $v_{\Delta l}$] $\circ$
  $\mathcal{WRITEENV}\ \langle t_{\Delta l}\rangle\ V\ \circ$
  [`call_subroutine` $addr$]

- If $k > 0 \wedge ran(\sigma_\beta) \subset V$ ($\Rightarrow l = n$):
  [`test_me_and_then` $codeAddr(\phi)$] $\circ$
  [`compare_to` v_1] $\circ$
  $\mathcal{COMPARE}\ \langle t_1\rangle\ \circ$
  $\vdots$
  [`compare_to` v_k] $\circ$
  $\mathcal{COMPARE}\ \langle t_k\rangle\ \circ$
  [`alloc_env` m] $\circ$
  [`env_entry_for` $v_{\Delta 1}$] $\circ$
  $\mathcal{WRITEENV}\ \langle t_{\Delta 1}\rangle\ V\ \circ$
  $\vdots$
  [`env_entry_for` $v_{\Delta l}$] $\circ$
  $\mathcal{WRITEENV}\ \langle t_{\Delta l}\rangle\ V\ \circ$
  [`call_subroutine` $addr$]

- Else:
  [`test_me_and_then` $codeAddr(\phi)$] $\circ$
  [`alloc_env` m] $\circ$
  [`match_to` v_1] $\circ$
  $\mathcal{MATCH}\ \langle t_1\rangle\ V\ \emptyset\ \circ$
  [`match_to` v_2] $\circ$
  $\mathcal{MATCH}\ \langle t_2\rangle\ V\ \ vars(t_1)\ \circ$
  $\vdots$
  [`match_to` v_k] $\circ$

$\mathcal{MATCH}\ \langle t_k \rangle\ V\ \bigcup_{i=1}^{k-1} vars(t_i)\ \circ$
[`env_entry_for` $v_{\Delta 1}$] $\circ$
$\mathcal{WRITEENV}\ \langle t_{\Delta 1} \rangle\ V\ \circ$
$\vdots$
[`env_entry_for` $v_{\Delta l}$] $\circ$
$\mathcal{WRITEENV}\ \langle t_{\Delta l} \rangle\ V\ \circ$
[`call_subroutine` $addr$]

$\mathcal{COMPARE}$ This scheme gets a subterm of a head atom as argument.

$\mathcal{COMPARE}\ \langle \beta \rangle =$

- If $\beta = f(t_1, \ldots, t_n)$:
  [`compare` f_n] $\circ$
  $\mathcal{COMPARE}\ \langle t_1 \rangle\ \circ \ldots \circ \mathcal{COMPARE}\ \langle t_n \rangle$
- Else:
  [`compare_var` β]

$\mathcal{MATCH}$ This scheme gets a term as first argument, which may contain variables of a head atom β and a body atom α. The second argument is the set of all variables of β (note that these variables are *all* bound at runtime, when the generated code is reached) and the third is the set of those variables of α which will be already bound at runtime when the generated code is reached. This is a subset of $vars(\alpha)$.

$\mathcal{MATCH}\ \langle T \rangle\ V_\beta\ V_\alpha =$

- If $T = f(t_1, \ldots, t_n)$:
  [`match` f_n] $\circ$
  $\mathcal{MATCH}\ \langle t_1 \rangle\ V_\beta\ V_\alpha\ \circ$
  $\mathcal{MATCH}\ \langle t_2 \rangle\ V_\beta\ vars(t_1) \cup V_\alpha\ \circ$
  $\vdots$
  $\mathcal{MATCH}\ \langle t_n \rangle\ V_\beta\ \bigcup_{i=1}^{n-1} vars(t_i) \cup V_\alpha$
- Else:
  - $T \in V_\beta$:
    [`match_bvar_oldenv` T]
  - $T \in V_\alpha$:
    [`match_bvar_env` T]
  - Else ($T \notin V_\beta \wedge T \notin V_\alpha$):
    [`match_uvar_env` T]

$\mathcal{WRITEENV}$ This scheme gets a term as first argument, which also may contain variables of a head atom β and a body atom α. The second argument is the set of the variables of β.

$\mathcal{WRITEENV}\ \langle T\rangle\ V_\beta =$

- If $T = f(t_1, \ldots, t_n)$:
  [`write_env_begin` f_n] $\circ$
  $\mathcal{WRITEENV}_\mathcal{C}\ \langle t_1\rangle\ V_\beta\ \circ \ldots \circ \mathcal{WRITEENV}_\mathcal{C}\ \langle t_n\rangle\ V_\beta$
- If $T \in V_\beta$:
  [`write_env_old_var_begin` T]
- Else:
  [`write_env_new_var_begin` T]

$\mathcal{WRITEENV}_\mathcal{C}$ This scheme gets a term as first argument, which also may contain variables of a head atom β and a body atom α. The second argument is the set of the variables of β.

$\mathcal{WRITEENV}_\mathcal{C}\ \langle T\rangle\ V_\beta =$

- If $T = f(t_1, \ldots, t_n)$:
  [`write_env` f_n] $\circ$
  $\mathcal{WRITEENV}_\mathcal{C}\ \langle t_1\rangle\ V_\beta\ \circ \ldots \circ \mathcal{WRITEENV}_\mathcal{C}\ \langle t_n\rangle\ V_\beta$
- If $T \in V_\beta$:
  [`write_env_old_var` T]
- Else:
  [`write_env_new_var` T]

$\mathcal{PUSHP}$ This scheme gets a list of variables as argument.

$\mathcal{PUSHP}\ [v_1, \ldots, v_n] =$

[`push_param` v_1, ..., `push_param` v_n]

$\mathcal{POPP}$ This scheme gets a list of variables as argument.

$\mathcal{POPP}\ [v_1, \ldots, v_n] =$

[`pop_param` v_1, ..., `pop_param` v_n]

$\mathcal{GOAL}_\Delta$ This scheme gets a clause, the index of the body atom for which an entry point should be generated, and a label ϕ as arguments. ϕ denotes the address of the corresponding head code sequence.

$\mathcal{GOAL}_\Delta\ \langle \alpha_1, \ldots, \alpha_n \rightarrow \beta_1; \ldots; \beta_m\rangle\ i\ \ \phi =$

$$\begin{array}{l}
\mathcal{PROVE}\ \ \langle\alpha_1\rangle\ \ vars(\alpha_i)\ \ \phi_1\ \ \circ \\
\phi_1 : \mathcal{PROVE}\ \ \langle\alpha_2\rangle\ \ vars(\alpha_1) \cup vars(\alpha_i)\ \ \phi_2\ \ \circ \\
\vdots \\
\phi_{i-2} : \mathcal{PROVE}\ \ \langle\alpha_{i-1}\rangle\ \ (\bigcup_{j=1}^{i-2} vars(\alpha_j)) \cup vars(\alpha_i)\ \ \phi_{i-1}\ \ \circ \\
\phi_{i-1} : \mathcal{PROVE}\ \ \langle\alpha_{i+1}\rangle\ \ \bigcup_{j=1}^{i} vars(\alpha_j)\ \ \phi_i\ \ \circ \\
\vdots \\
\phi_{n-2} : \mathcal{PROVE}\ \ \langle\alpha_n\rangle\ \ \bigcup_{j=1}^{n-1} vars(\alpha_j)\ \ \phi_{n-1}\ \ \circ \\
\phi_{n-1} : \mathcal{PUSHP}\ \ vars([\beta_1;\ldots;\beta_m])\ \ \circ \\
\mathcal{CALL}\ \ \phi\ \ m
\end{array}$$

$\mathcal{CALL}$ This scheme gets a label denoting the address of a head code sequence and the number of atoms in the corresponding head as arguments.

$\mathcal{CALL}\ \ \phi\ \ m =$

- If $m = 0$:
  [`fail`]

- Else ($m > 0$):
  [`call` $codeAddr(\phi)$] $\circ$
  [`try_next_binding`]

$\mathcal{PROVE}$ This scheme gets a body atom, a set of variables V, and a label ϕ as arguments. V is the set of variables which will be already bound in the environment at runtime when the generated code is reached. ϕ is the label of the address which should be inserted in the possibly generated hyperresolution stack entry.

$\mathcal{PROVE}\ \ \langle\alpha\rangle\ \ V\ \ \phi =$

- If $vars(\alpha) \subset V$:
  $\mathcal{PROVE}_{\mathcal{G}}\ \ \langle\alpha\rangle$

- Else:
  [`create_hs_entry` $|\ vars(\alpha) \setminus V\ |$ $codeAddr(\phi)$] $\circ$
  $\mathcal{PROVE}_{\mathcal{NG}}\ \ \langle\alpha\rangle\ \ V\ \ \emptyset$

$\mathcal{PROVE}_{\mathcal{G}}$ This scheme gets a body atom or one of its subterms as argument.

$\mathcal{PROVE}_{\mathcal{G}}\ \ \langle\alpha\rangle =$

- If $\alpha = f(t_1, \ldots, t_n)$:
  [`proved` f_n] $\circ$
  $\mathcal{PROVE}_{\mathcal{G}}\ \langle t_1 \rangle\ \circ \ldots \circ \mathcal{PROVE}_{\mathcal{G}}\ \langle t_n \rangle$

- Else:
  [`proved_bvar` α]

$\mathcal{PROVE}_{\mathcal{NG}}$ This scheme gets a body atom (or one of its subterms) and two sets of variables as arguments. In *Env* are the variables which at runtime will be already bound in the environment, when the generated code is reached. In V are the variables which already appeared in the current atom and therefore are bound in a temporary data structure in the machine.

$\mathcal{PROVE}_{\mathcal{NG}}\ \langle \alpha \rangle\ Env\ V =$

- If $\alpha = f(t_1, \ldots, t_n)$:
  [`prove_and_bind` f_n] $\circ$
  $\mathcal{PROVE}_{\mathcal{NG}}\ \langle t_1 \rangle\ Env\ V\ \circ$
  $\mathcal{PROVE}_{\mathcal{NG}}\ \langle t_2 \rangle\ Env\ V \cup vars(t_1)\ \circ$
  $\vdots$
  $\mathcal{PROVE}_{\mathcal{NG}}\ \langle t_n \rangle\ Env\ V \cup \bigcup_{i=1}^{n-1} vars(t_i)$

- If $\alpha \in Env$:
  [`prove_and_bind_envvar` α]

- If $\alpha \in V$:
  [`prove_and_bind_bvar` α]

- If $\alpha \notin Env \wedge \alpha \notin V$:
  [`prove_and_bind_uvar` α]

6.2 The Implementation of the Compiler

The compiler SatAMC is implemented in the lazy, purely functional language Haskell, mainly due to the fact that scanner and parser could be reused from an existing implementation – Functional Satchmo [GPS97]. Additionally, we already had good experiences implementing a compiler for a functional language in Haskell [GP97].

To make a peephole optimizer possible, SatAMC generates non flat code in the first stage. A fix-up of the code references cannot be done (or prepared) in the first pass, because the peephole optimizer could eliminate code. This would result in *non-local* changes of code references which is contrary to the general idea of *peephole* optimization. After the optimization step the code is flattened and the references are dissolved passing another two times over (parts of) the code.

7 Preliminary Benchmark Results

We performed some preliminary benchmark experiments with the comcrete realization of the compiler and the abstract machine. The details on these and some other experiments are given in [vDGPS98].

7.1 Bytecode Size

Specialization and compilation approaches can suffer from an explosion of the size of the specialized/compiled code. In particular, the specialization of the incremental application of the PUHR rule might lead to a nonlinear increase in code size.

We compared the size of the generated bytecode with the size of the input problem for all problems in clause normal form of the TPTP library [SSY94]. This problem library contains theorem proving problems with a large variety of features like clause size or term nesting depth.

We observed (again, the details are given in [vDGPS98]) that the bytecode size grows non-linearily, but sub-quadratically with the problem size. We further observed that the size of the bytecode grows faster than the size of the compiled code generated by Compiling Satchmo and subsequently compiled to WAM code with a Prolog compiler. We suppose that this is due to the larger amount of compilation and specialization done by our approach.

Note that the compilation times depend on the size of the resulting code as well as on the size and some syntactic features of the input clauses. We do not report the compilation times here because the implementation of our compiler is not optimized for efficiency yet.

7.2 Model-Search Time

The model-search times for implementations of PUHR tableaux depend heavily in the choice of the tableau rule to apply next, because this influences the structure of the search space. Preliminary experiments using some artificial examples showing a unique search space with our system and with Compiling Satchmo show a speedup of about an order of magnitude for our system compared with Compiling Satchmo.

8 Related Work

We are not aware of any abstract machine directly intended and used for model generation. We consider Prolog and KL1, which are used by Compiling Satchmo [SG96] and MGTP [Ins95], respectively, as too high-level for being counted as "abstract machines".

Abstract machines are successfully used to implement efficient refutational theorem provers. Most prominently, SETHEO's implementation is based on an abstract machine [LSBB92].

The most important differences between our abstract machine and the WAM [War83, Aït91] are on one hand the (complex) global data structures for the set of positive unit clauses and the set of disjunctions to be split, and on the other hand the absence of full unification. Note that the complex data structures are required for the forward reasoning performed by our machine opposed to a simple stack in the WAM which is sufficient for backward reasoning, i. e., SLD resolution.

The SLG-WAM is another abstract machine for a Prolog-like language which uses a discrimination tree data structure [RRS$^+$95].

9 Conclusions and Further Work

We designed an abstract machine for model generation with the PUHR tableau method, which ensures fairness and incrementality. During the design process, the expressivity of the logic language was enhanced stepwise and Prolog prototypes of the machine were developed in parallel. Finally, we implemented a bytecode compiler/interpreter system in order to validate the design and to be able to compare the performance of the system with other implementations of the PUHR tableau method.

During the whole process, we used the methodology proposed by Kursawe and Nilsson as a guideline. The Prolog prototypes were extremely useful for the design of the instruction set of the abstract machine and for the development of the compilation schemes. Nevertheless, we did not strictly follow the proposed methodology. Sometimes the creative steps in the design of the abstract machine are very difficult. Our past experiences from implementations of the PUHR tableau method suggested creative steps that are difficult to justify by just following the proposed methodology.

References

[Aït91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1991.

[BEST98] François Bry, Norbert Eisinger, Heribert Schütz, and Sunna Torge. SIC: Satisfiability checking for integrity constraints. In Piero Fraternali, Ulrich Geske, Carolina Ruiz, and Dietmar Seipel, editors, *Proc. 6th International Workshop on Deductive Databases and Logic Programming, DDLP '98*, GMD Report 22, Manchester, UK, June 1998.

[BFFN97] Peter Baumgartner, Peter Fröhlich, Ulrich Furbach, and Wolfgang Nejdl. Tableaux for diagnosis applications. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 97*, number 1071 in Springer LNCS, pages 76–90, Pont-a-Mousson, France, May 1997. Springer-Verlag.

[BFK+98] Peter Baumgartner, Ulrich Furbach, Michael Kohlhase, William McCune, Wolfgang Reif, Mark Stickel, and Tomás Uribe, editors. *Problem-solving Methodologies with Automated Deduction, Workshop at CADE-15*, Lindau, Germany, July 1998.

[BY96] François Bry and Adnan Yahya. Minimal model generation with positive unit hyper-resolution tableaux. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Proc. TABLEAUX '96*, number 1071 in Springer LNCS, pages 143–159, Terrasini, Palermo, Italy, May 1996. Springer-Verlag.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 12(3):655–657, 1960.

[EG98] Norbert Eisinger and Tim Geisler. Problem solving with model-generation approaches based on PUHR tableaux. Technical Report PMS-FB-1998-8, Institut für Informatik, LMU München, 1998. In [BFK+98].

[FSB93] Masayuki Fujita, John Slaney, and Frank Bennett. Automatic generation of some results in finite algebra. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence, IJCAI 93*, pages 52–57. Morgan Kaufmann, 1993.

[FW94] Michael Fröhlich and Mattias Werner. The graph visualization system daVinci – A user interface for applications. Technical Report 5/94, Institut für Informatik, Universiät Bremen, September 1994.

[GP97] Tim Geisler and Sven Panne. Übersetzerbau-Praktikum. Praktikum im Hauptstudium Informatik, Sommersemester 1997, Institut für Informatik, LMU München, 1997. Slides available from http://www.pms.informatik.uni-muenchen.de/lehre/praktikum/uebersetzerbau/97ss/.

[GPS97] Tim Geisler, Sven Panne, and Heribert Schütz. Satchmo: The compiling and functional variants. *Journal of Automated Reasoning*, 18(2):227–236, 1997.

[Gra96] Peter Graf. *Term Indexing*. Number 1053 in Springer LNAI. Springer-Verlag, 1996.

[Ins95] Institute for New Generation Computer Technology. *Model Generation Theorem Prover: MGTP*, 1995. http://www.icot.or.jp/ICOT/IFS/IFS-abst/082.html.

[Kin97] Andreas Kind. Bytecode-Interpretierung. *Informatik-Spektrum*, 20(2), April 1997. In German.

[Kur86] Peter Kursawe. How to invent a Prolog machine. In Ehud Y. Shapiro, editor, *Third International Conference on Logic Programming*, number 225 in Springer LNCS. Springer-Verlag, July 1986.

[LSBB92] Reinhold Letz, Johann Schumann, S. Bayerl, and Wolfgang Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[MB88] Rainer Manthey and François Bry. SATCHMO: A theorem prover implemented in Prolog. In E. L. Lusk and R. A. Overbeek, editors, *9th Int. Conf. on Automated Deduction (CADE)*, number 310 in Springer LNCS, pages 415–434, Argonne, IL, USA, 1988. Springer-Verlag.

[McC88] William McCune. An indexing method for finding more general formulas. *Association for Automated Reasoning Newletter*, 9(1):7–8, January 1988.

[Nil92] Ulf Nilsson. *Abstract Interpretation & abstract machines: contributions to a methodology for the implementation of logic programs*. PhD thesis, Linköping University, Sweden, 1992.

[Nil93] Ulf Nilsson. Towards a methodology for the design of abstract machines for logic programming languages. *Journal of Logic Programming*, 16(1–2):163–189, 1993.

[PH97] John Peterson and Kevin Hammond. Haskell 1.4: A non-strict, purely functional language. http://www.haskell.org/onlinereport/, April 1997.

[PL92] Seymon L. Peyton Jones and David R. Lester. *Implementing Functional Languages – A Tutorial.* Prentice Hall, 1992.

[RRS+95] I. V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S. Warren. Efficient tabling mechanisms for logic programs. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming*, pages 697–711. MIT Press, 1995.

[SG96] Heribert Schütz and Tim Geisler. Efficient model generation through compilation. In Michael McRobbie and John Slaney, editors, *13th Int. Conf. on Automated Deduction (CADE)*, number 1104 in Springer LNAI, pages 433–447. Springer-Verlag, 1996.

[SG97] Heribert Schütz and Tim Geisler. Efficient model generation through compilation. Technical Report PMS-FB-1997-21, Institut für Informatik, LMU München, 1997. Accepted for publication. Extended version of [SG96].

[SSY94] Geoff Sutcliffe, Christian Suttner, and Theodor Yemenis. The TPTP problem library. In Alan Bundy, editor, *Automated Deduction — CADE-12*, number 814 in Springer LNAI, pages 252–266, Nancy, France, 1994. Springer-Verlag.

[vDGPS98] Alexander von Drach, Tim Geisler, Sven Panne, and David Sacher. SatAM – An abstract machine for model generation with PUHR tableaux. Technical Report PMS-FB-1998-14, Institut für Informatik, LMU München, October 1998.

[vGPS98] Alexander von Drach, Tim Geisler, Sven Panne, and David Sacher. An abstract machine for model generation with PUHR tableaux (extended abstract). In Uwe Egly, editor, *13. Workshop Logische Programmierung, WLP '98*, Technische Universität Wien, Wien, Austria, October 1998.

[War83] D.H.D. Warren. An abstract PROLOG instruction set. Technical Report 309, SRI, 1983.