Implementation strategies for Scheme-based Prolog systems

Michael Levin Daniel P. Friedman R. Kent Dybvig

April 19, 1998

Abstract

Schemelog is a logic programming language similar to Prolog with three important distinctions. Schemelog supports a simple module system, provides access to the run-time system through arbitrary Scheme expressions, and defines a syntax abstraction facility. This paper describes a compiler and a run-time architecture suitable for implementing Schemelog or any other Prolog-like language in Scheme.

1 Introduction

There are many ways to implement a new language. Writing an interpreter or a set of macros that expand into host language expressions usually results in a slow system. On the other hand, it is quite time-consuming to develop a native code compiler. A compromise between these two approaches is to write a compiler to another high level language. For a Prolog-like language implementation, Scheme is a natural choice as the target language. It manages the type tags and heap automatically. Scheme procedures can represent predicates since all Scheme implementations handle tail calls properly. We do not need to implement the WAM's stack explicitly, but can use Scheme local variables to achieve most the desired effects. Finally, we can use first-class continuations to realize the backtracking mechanism.

Schemelog is a logic programming language implemented on top of Scheme. We implemented its original version as a collection of simple macros. A lot of work that could be done at compile time was deferred until run time, and the performance of the system was unacceptably slow. To improve it, we studied

the Warren Abstract Machine and the ways to compile Prolog programs into it (see [AK91]). As we became more comfortable with the WAM, we realized that it could be altered a little to fit the Scheme syntax and its computational model better. The artifact that we obtained by implementing the WAM in Scheme and taking advantage of various Scheme features is what we call the Scheme Abstract Machine. Our goal is to translate Schemelog programs into Scheme in such a way that while all the important WAM techniques are preserved, as many of them as possible are handled by Scheme and are not represented explicitly in the SAM.

In addition to well known compilation techniques such as clause indexing ([AK91]) and unification factoring ([DRR$^+$]), Schemelog employs a compile-time unification algorithm that performs several local optimizations by analyzing explicit unification statements in a predicate. This algorithm and the presentation of the Scheme Abstract Machine are the contributions of this paper.

The outline of this paper is as follows. First, we describe the run-time data structures used by Schemelog and the SAM. We then present the Schemelog syntax. We elide the discussion of the semantics of Schemelog, but, basically, it diligently mimics the semantics of Prolog. As a next step, we define the output language of the Schemelog compiler, the SAM target language. Finally, we focus on the compiler and its optimizations.

2 The Schemelog abstract machine

Having committed to Scheme as the basis of our Schemelog implementation, we now discuss the mapping between Schemelog programs and their Scheme counterparts. First, let us discuss how we can represent Prolog terms in Scheme. (Since Schemelog is in many ways similar to Prolog, we will use Prolog and Schemelog syntax notation interchangeably.) Prolog has the following data types: atoms, numbers, pairs, and structures. In Scheme, we can conveniently represent them by symbols, numbers, pairs, and vectors whose first element is the symbolic name of the structure. Additionally, we have to represent logic variables by some kind of assignable boxes. For example, we can use vectors whose first element is some special tag disjoint from structure names. Our Scheme implementation ([Cad94]) supports boxes whose type is different from other data types. Boxes are a natural choice for logic variable representation. Figure 1 shows the correspondence between Prolog

Prolog term	Scheme construction	Scheme representation
`1`	`1`	`1`
`a`	`'a`	`a`
`X`	`(box 'unbound)`	`#&unbound`
`[1|2]`	`(cons 1 2)`	`(1 . 2)`
`f(1,2)`	`(vector 'f/2 1 2)`	`#(f/2 1 2)`

Figure 1: Term representation

and Scheme terms.

More interesting representational issues are how to handle predicates and what data structures are needed for backtracking implementation. As we mentioned in the previous section, a predicate can be conveniently represented by a procedure that is executed whenever the predicate is queried. The only effect that a procedure invocation has is the new heap image where some of the unbound variables become bound as the result of the unification statements in the body of the procedure. We have to choose between different kinds of procedures, however. The first idea is to represent predicates by Scheme top level procedures. To maintain backtracking information, we also need to store the value of the current choice point and the list of variables that have been bound since the last choice point in global variables. That setup indeed corresponds to Prolog systems where every predicate is top level and visible everywhere. Figure 2 demonstrates this method.

This representation, however, does not address the modularity and privacy issues. Moreover, top level Scheme procedures cannot be inlined and their invocations cannot be converted to direct jumps by a Scheme compiler.

An alternative to global procedures is `letrec` bound procedures (see Figure 3). In this representation, a database of rules and facts, a logic unit, would correspond to a `letrec` expression whose procedures implement predicates and whose body implements a query of a predicate. Now, the current choice point and trail stack do not have to be stored in global variables but can be held in lexical variables visible inside the `letrec` expression only. Thus, logic units are independent of each other and can be queried simultaneously. Also, a Scheme compiler can do a better job optimizing `letrec`

Prolog program	Scheme representation

`predicate1(X) :-`
`  predicate2(X).`
 	
`predicate2(X) :-`
`  predicate1(X).`
	`(define predicate1`
	`  (lambda (x) (predicate2 x)))`
	
	`(define predicate2`
	`  (lambda (x) (predicate1 x)))`

`predicate3() :-`
`  predicate1(X),`
`  predicate2(X).`
	`(define predicate3`
	`  (lambda ()`
	`  (let ([x (box 'unbound)])`
	`  (predicate1 x)`
	`  (predicate2 x))))`

Figure 2: Predicates as top-level procedures

procedures and calls to them. A big problem with this representation is that it is not extensible. Whenever we want to add a predicate to a logic unit or query the unit with a different goal, we have to construct and recompile a new `letrec` expression from scratch.

In object-oriented languages, class definitions contain method declarations just like `letrec` expressions contain procedure declarations. Additionally, classes are extensible via subclassing and, consequently, provide a natural way to implement logic units. To realize this, we use a set of Scheme macros defining a single inheritance object-oriented language. That enables us to denote a logic unit by a class whose methods correspond to predicates. Representing some predicates by private methods, we can limit the scope of the predicate just to the logic unit where the predicate is defined. It cannot be overridden when the unit is extended or queried from outside of the unit. Therefore, this design provides for defining logic units and specifying what predicates are exported from the unit. Figure 4 illustrates our representation.

The luxury of being able to extend classes with new predicates comes at a price. Calls to public methods incur the overhead of virtual method calls. Private methods, however, are implemented by `letrec` procedures and can be translated to direct jumps. In summary, a collection of predicates, a unit, is represented by a class. The predicates that are exported to the outside world are represented by public methods, whereas the remaining predicates are represented by `letrec` bound procedures. Logic units do not communicate with each other. A predicate in one logic unit cannot query a

Prolog program	Scheme representation

`predicate1(X) :-`
`  predicate2(X).`
 	
`predicate2(X) :-`
`  predicate1(X).`
	`(letrec ([predicate1`
	`  (lambda (x) (predicate2 x))]`
	`  [predicate2`
	`  (lambda (x) (predicate1 x))])`
	`  (predicate1 5))`

`predicate3() :-`
`  predicate1(X),`
`  predicate2(X).`
	`(letrec ([predicate1`
	`  (lambda (x) (predicate2 x))]`
	`  [predicate2`
	`  (lambda (x) (predicate1 x))]`
	`  [predicate3`
	`  (lambda ()`
	`  (let ([x (box 'unbound)])`
	`  (predicate1 x)`
	`  (predicate2 x)))])`
	`  (predicate1 6))`

Figure 3: Predicates as letrec procedures

Prolog program	Scheme representation

`predicate1(X) :-`
`  predicate2(X).`
 	
`predicate2(X) :-`
`  predicate1(X).`
	`(define unit1`
	`  (class <initial-unit> () ()`
	`  (predicate1`
	`  (method (x)`
	`  (predicate2 x)))`
	`  (predicate2`
	`  (method (x)`
	`  (predicate1 x)))))`

`predicate3() :-`
`  predicate1(X),`
`  predicate2(X).`
	`(define unit2`
	`  (class unit1 () ()`
	`  (predicate3`
	`  (method ()`
	`  (let ((x (box 'unbound)))`
	`  (predicate1 x)`
	`  (predicate2 x))))))`

Figure 4: Predicates as methods

Prolog program	Scheme representation
`? predicate3().`	`(call (new unit2) predicate3)`

Figure 5: Query representation

predicate of an unrelated logic unit.

If a class is a logic unit, what is an object of that class? Schemelog creates an object whenever a query to a predicate in the unit is issued. Each object maintains two instance variables: the current choice point and the trail stack. In this architecture, we can pose several queries to the same logic unit and let them run simultaneously independently of each other. Figure 5 gives an example of querying a logic unit.

The target language of the Schemelog compiler is a subset of Scheme extended with the following operations and features.

- `bind: Box * Term -> Void`
- `unify/2: Term * Term -> Void`
- `orelse`
- A small object-oriented system implemented on top of Scheme

Procedure `bind` takes an uninitialized box and a term and assigns the box to the term. Then, it saves the box on the trail stack to be unbound when backtracking occurs. Procedure `unify/2` takes two terms and unifies them. Form `orelse` takes several statements and executes them nondeterministically. These forms and some other helper forms that have not been mentioned here are implemented as global macros or as public methods defined in the initial logic unit provided by the system. The choice point and trail stack instance variables are also defined in the initial unit and, therefore, are visible inside its descendants.

The `orelse` form is implemented as a macro that uses `call/cc` and stores the current continuation in the current choice point instance variable `choice-point`. When backtracking happens, Schemelog calls the helper procedure `unwind-trail` that goes through the list stored in the instance variable `trail-stack` and unbinds the variables stored there. Figure 6 shows the expansion of the macro for the case of `orelse` with two statements.

```
 (orelse command1 command2)

==>

 (let ([old-trail-stack trail-stack]
       [old-choice-point choice-point])
   (if (not (call/cc
              (lambda (k) (set! choice-point k) command1)))
       (begin (set! choice-point old-choice-point)
              (unwind-trail trail-stack old-trail-stack)
              (set! trail-stack old-trail-stack)
              command2)))
```

Figure 6: Expansion of `orelse`

3 Schemelog syntax

A Schemelog program is a collection of logic units. A unit is a collection of predicate definitions. A predicate is defined by its name, the list of formal parameters and a goal statement that constitutes the body of the predicate. A goal can be a disjunction or a conjunction of other goals, a `branch` expression, a call to a predicate, an explicit unification statement, a call to a predicate in the parent logic unit, or an invocation of an arbitrary Scheme expression. Figure 7 presents Schemelog syntax. This syntax is somewhat different, lower level, than that of Prolog where a predicate is defined by a collection of clauses and the head of each clause can contain arbitrary terms. Schemelog receives programs written in Prolog notation and converts them to the described format introducing explicit unification statements and disjunctive goals. Figure 8 shows an example of this conversion for the `member/2` predicate.

4 Schemelog compiler

Schemelog programs are very similar to Scheme programs. In fact, the only difference is that in Schemelog, local variables are declared implicitly whenever they are referenced for the first time while in Scheme every variable must be declared explicitly. To compile Schemelog programs, it is sufficient to gen-

```
P ::=                                   ;;; Programs
      U ...

U ::=                                   ;;; Units
      initial-unit
      (extend U (Pred-name ...)  D ...)

D ::=                                   ;;; Predicate definitions
      (Pred-name (Var ...)  G)

G ::=                                   ;;; Goal statements
      (Pred-name T ...)
      (super Pred-name T ...)
      (begin G G ...)
      (orelse G G ...)
      (branch Var (T G) ...)
      (= T T)
      (scheme (Var ...)  S-exp)

T ::=                                   ;;; Terms
      C
      Var
      (cons T T)
      (Struct-name T ...)

C ::=                                   ;;; Constants
      Number
      (quote Symbol)
```

Figure 7: Schemelog syntax

Prolog notation

```
(define-predicate member/2
  [(X (cons X _))]
  [(X (cons _ Y)) (member/2 X Y)])
```

Schemelog notation

```
(define member/2
  (predicate (a1 a2)
    (orelse
      (begin
        (unify/2 a1 X)
        (unify/2 a2 (cons X _)))
      (begin
        (unify/2 a1 X)
        (unify/2 a2 (cons _ Y))
        (member/2 X Y)))))
```

Figure 8: From Prolog to Schemelog notation

erate `let` declarations for all the local variables. See Figure 9 for an example of this transformation on the `member/2` predicate. A compiler that performs only this transformation would overlook optimization opportunities. In the example in Figure 9 for instance, variable `X` is always aliased to `a1` and need not be created. The unification statements in both branches of the disjunction are partially instantiated and should be compiled into specialized code. Also, the two unification statements try to unify the same variable `a2` with terms of the same shape causing the program to perform redundant checks on backtracking.

The Schemelog compiler has the following objectives.

- **Specialize partially instantiated unifications**. For partially instantiated unification statements, generate specialized unification code instead of a call to the generic `unify/2` procedure.
- **Eliminate unnecessary variables**. Perform copy propagation by avoiding creation of variables that are known to be bound to constants or aliased to other variables.
- **Reduce the number of redundant unifications**. Reduce the number of unifications by extracting common unification statements from a disjunction.
- **Recognize determinism**. Finally, eliminate creation of some choice points by converting nondeterministic `orelse` statements into equivalent deterministic `branch` statements.

Our compiler achieves these objectives in four major passes.

- Compile-time unification
- Unification factoring
- Clause indexing
- Code generation

Prolog program

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
```

Schemelog program

```
(define member/2
  (predicate (a1 a2)
    (orelse
      (begin
        (unify/2 a1 X)
        (unify/2 a2 (cons X _)))
      (begin
        (unify/2 a1 X)
        (unify/2 a2 (cons _ Y))
        (member/2 X Y)))))
```

Translation

```
(define member/2
  (method (a1 a2)
    (orelse
      (let ((X (box 'undefined)) (_ (box 'undefined)))
        (unify/2 a1 X)
        (unify/2 a2 (cons X _)))
      (let ((X (box 'undefined))
            (_ (box 'undefined))
            (Y (box 'undefined)))
        (unify/2 a1 X)
        (unify/2 a2 (cons _ Y))
        (member/2 X Y)))))
```

Figure 9: Schemelog syntax

4.1 Compile-time unification

The compile-time unification pass takes a predicate and executes explicit unifications in its body. According to the result of the unifications, the compiler outputs a residual program. This pass of the compiler accomplishes several local optimizations. Let us illustrate them by examples.

- **Copy propagation**. In the following example, the local variable `Y` is aliased to `X` and the compiler replaces all occurrences of `Y` by `X`

```
test1(X) :- Y=X,test2(Y).

==>

test1(X) :- test2(X).
```

- **Unification strength reduction**. Here, two pairs are unified, and the compiler can save a little bit of work by unifying the components of the pairs instead.

```
test(V,X,Y,Z) :- [V|X] = [Y|Z].

==>

test(V,X,Y,Z) :- V=Y, X=Z.
```

- **Dead code elimination**. Dead code is a piece of code that will never be reached due to a unification failure. In the following example, the compiler determines that the `X=2` unification statement will always fail and replaces it and the rest of the clause with the explicit `fail` statement.

```
test1(X) :- X=f(1),test2(X),X=2,test3(X).

==>

test1(X) :- X=f(1),test2(X),fail.
```

- **Box elimination**. Sometimes, local variables are introduced just to be bound to a term immediately. The compiler does not need to generate a box for these variables, but can represent them by lexical Scheme variables. In the following program, `Y` is a variable of this kind. The compiler does not perform copy propagation when a variable is bound to a compound term since it might lead to code explosion.

```
test1(X) :- Y=f(X),test2(Y)

==>

test1(X) :- let_bind(Y,f(X)),test2(Y).
```

Next, we consider how the compile-time unification pass achieves the optimizations discussed above. First, we introduce the notions of shared variables and the compile-time store.

A variable occurrence is *shared* if its binding cannot be inferred at compile-time. The compiler generates unification code for a variable based on whether the variable is shared or not. The four examples below motivate the importance of shared variable analysis.

- **Example 1**. Here a predicate checks whether its input argument is a pair of 1 and 2. This operation can potentially fail, and therefore, we must generate code for it and cannot optimize it any further. Variable `X` in this example is shared since we cannot assume that it is bound to a term of any particular shape. In summary, we have to generate code for any explicit unification statement whose left hand side is a shared variable.

```
test1(X) :- X=[1|2].
```

- **Example 2**. In this example, the predicate binds a pair of 1 and 2 to a local variable. Since variable `Y` does not escape outside of the predicate (i.e., it is not shared), This operation will always succeed and will have no effect on the execution of the program. Thus, we can avoid generating code for this unification.

```
test1(X) :- Y=[1|2].

==>

test1(X).
```

- **Example 3**. This example shows that the same variable can have shared and non-shared occurrences in a predicate. Here, the local variable `Y` is not shared until it gets passed as the parameter to predicate `test2`. The predicate `test2` can bind `Y` to an arbitrary term; thus, `Y` becomes shared and we have to generate code that unifies it with the pair.

```
test1(X) :- test2(Y),Y=[1|2].
```

- **Example 4**. This example is similar to the previous one. A local variable again becomes shared at some point in the predicate. The reason is different here. The local variable `Y` is bound to a subterm of the shared variable `X`. We cannot assume anything about that subterm and, thus, have to generate the explicit unification statement.

```
test1(X) :- X=f(Y),Y=[1|2].
```

As our compile-time unification algorithm traverses the predicate, it executes the explicit unifications with respect to the symbolic *store*. The store

provides a mapping between variables and terms and represents the information that the compiler has inferred about the variables in the range of the store. In the store, a variable can be:

- unbound
- bound to a constant or another variable. These cases enable us to perform copy propagation as in the example below.

```
test1(X) :- X=1,test2(X),test3(X).

==>

test1(X) :- X=1,test2(1),test3(1).
```

- bound to a compound term. In this example, the shared variable `X` is bound to the term `f(1)` in the symbolic store.

```
test1(X) :- X=f(1),test2(X),test3(X).

==>

test1(X) :- X=f(1),test2(X),test3(X).
```

The only result of the unifications performed by the compiler is the updated version of the store. We do not generate any code as we encounter explicit unifications. As soon as we reach a call or the end of the predicate, we have to produce code from the information collected while performing the explicit unifications. The compiler generates unification statements only for the variables that are referenced in the call and the shared variables. The following example demonstrates this point.

```
test1(X) :- X=f(1),Y=g(2),Z=h(3),test2(Y),test(Z).

==>

test1(X) :-
  X=f(1),
  let_bind(Y,g(2)),
  test2(Y),
  let_bind(Z,h(3)),
  test(Z).
```

Here, the unification statement for `X` is generated because `X` is shared and the unification can potentially fail preventing the first call from being executed. The local variable `Y` is bound since it is referenced in the first call. The local variable `Z`, however, has no bearing on the first call and the compiler can postpone generating its unification code until it is used in the second call.

In summary, the compile-time unification algorithms takes a predicate and traverses the sequence of goals that define its body. As it traverses the body, the algorithm maintains the list of shared variables and the symbolic store. Initially, the list of shared variables contains the formal parameters of the predicate, and the store maps every variable that is referenced in the body to the symbol `unbound`. The algorithm performs explicit unification statements by updating the variable bindings in the store. When it encounters a call or a `scheme` expression, the algorithm outputs unification statements for the variables referenced in the call and the shared variables. Then, it outputs the call itself dereferencing the variables with respect to the store. To process each disjunct of an `orelse` expression, the algorithm calls itself recursively extending the list of shared variables with the variables that are referenced both inside the disjunct and outside of the `orelse` goal. For more details, refer to Figure 10 that presents pseudo-code of the algorithm.

Let us examine two more programs presented in Figure 11 that are improved by the compile-time unification pass. In the first example, the compiler preserves the first unification goal since `X` is a shared variable. Variable `Z` is not shared, and it is not referenced in the first call; therefore, our compiler can postpone generating its unification code. The only purpose of the third unification goal is to associate `W` with `1` and `Y` with `A`, and, again, there

```
procedure CTU(goals, shared, store) {

  goal = first(goals);

  case goal

    unify/2(t1,t2) =>

      success,store := CTU_unify(t1,t2,store);
      if success CTU(rest(goals), shared, store);
      else EMIT_fail();

    p(t1,t2) =>

      shared := GENERATE_code(shared,store,goal);
      CTU(rest(goals), shared, store);

    orelse(goal1,goal2) =>

      shared := shared + common_vars(goal1,goal2,rest(goals));
      EMIT_orelse_start();
      CTU(goal1, shared, store);
      CTU(goal2, shared, store);
      EMIT_orelse_stop();
      CTU(rest(goals), shared, store);

}
```

Figure 10: Compile-time unification

Before	After

`temp(X) :-`
`  X=[A|1],`
`  Z=[Y|W],`
`  temp(X),`
`  Z=X,`
`  temp1(Y,W).`
	`temp(X) :-`
	`  X=[A|1],`
	`  temp(X),`
	`  temp1(A,1).`

`temp(X) :-`
`  A=f(X),`
`  temp1(X),`
`  temp2(A),`
`  A=g(X),`
`  temp3(A).`
	`temp(X) :-`
	`  temp1(X),`
	`  let_bind(A,f(X)),`
	`  temp2(A),`
	`  fail.`

Figure 11: Compile-time unification examples

Before	After	Should be

`temp(X) :-`
`  (Y=f(X);Y=g(X)),`
`  h(Z,X)=h(Y,1).`
	`temp(X) :-`
	`  (Y=f(X);Y=g(X)),`
	`  X=1.`
		`temp(X) :-`
		`  (succeed;succeed),`
		`  X=1.`

Figure 12: Compile-time unification counter example

is no need to generate any code for this unification statement. Finally, in the second call, the algorithm replaces `W` by `1` and `Y` by `A`.

In the second example, the binding of `A` is not needed until the second call `temp2(A)`, and, since `A` is not shared, the unification statement can be replaced by a `let` binding. The second unification goal cannot succeed; so, we replace the dead code by a call to `fail`.

Sometimes, our algorithm produces sub-optimal programs. For example, consider the predicate in Figure 12. As the compiler recursively traverses the branches of the disjunction, it judges `Y` to be shared since it occurs both inside and outside of the disjunction. Of course, then, it has to generate the unification statements in both disjuncts. Later, the compiler infers that `Y` is not needed outside of the disjunction. Thus its original judgment that it is a shared variable was overly conservative.

4.2 Unification factoring

The purpose of unification factoring is to hoist common unification statements occurring in multiple arms of a disjunction outside of the disjunction (see [DRR+]). Consider the following predicate.

```
test1(X) :- X=f(1),test2(X).
test1(X) :- X=f(2),test3(X).

==>

test1(X) :- X=f(Y),helper(X,Y).

helper(X,Y) :- Y=1,test2(X).
helper(X,Y) :- Y=2,test3(X).
```

It is defined by two clauses, and, in the head of each clause, the argument variable `X` is unified with a term whose outer constructor is `f`. If the unification statements are not factored outside of the disjunction, the program might check whether `X` unifies with the term `f(1)`, succeed at that, and fail while executing the goal `test2(X)`. At this point, the program will backtrack, and perform tests to unify `X` with `f(2)`. But while evaluating the first clause, the program has already determined whether `X` unifies with an `f` structure, and it would be wasted effort to perform the same checks again while evaluating the second clause. The transformed program introduces a helper predicate and avoids the extra checks.

It is important that only the unifications in the head of a disjunct be considered for factoring out. The next example does not present opportunities for unification factoring because the call `test3(X)` has to take place prior to the explicit unification `X=f(2)`.

```
test1(X) :- X=f(1),test2(X).
test1(X) :- test3(X),X=f(2).
```

The Schemelog compiler generalizes the above example to a case with any number of disjuncts. In an `orelse` statement, the compiler finds the longest sequence of disjuncts containing a common unification command. It factors out the unification and performs the same algorithm on the resulting

expression until there is no more unifications to be factored out.

4.3 Clause indexing

Clause indexing ([AK91]) turns nondeterministic disjunctions into deterministic `branch` statements whenever possible. In the following example, if `X` is bound to a `(f 1)` term and the corresponding unification in the first clause succeeds but the call `(test2 X)` fails, there is no reason to backtrack to the second clause of the disjunction since `X` is known to be bound to an `f` structure and will not unify with `(g 2)`.

```
(orelse
  (begin (unify/2 x f (1)) (test2 x))
  (begin (unify/2 x g (2)) (test3 x)))
==>

(branch
  x
  ((f y) (begin (unify/2 y 1) (test2 x)))
  ((g y) (begin (unify/2 y 2) (test3 x))))
```

A `branch` goal checks whether its variable is bound to a term. If that is the case, it matches the term against patterns specified in the alternatives and proceeds to execute the goals in the selected alternative. If one of the goals fails, the whole `branch` goal fails. If the variable of a `branch` goal is unbound, Schemelog nondeterministically selects an alternative, binds the variable to the pattern specified by the alternative and executes the rest of the goals.

The advantage of this optimization is that choice points are not created when the unification variable is bound to a term. In this case the program will execute deterministically akin to a functional program. If the variable is unbound at run time, nondeterminism cannot be eliminated and the execution trace will be the same for both the original `orelse` goal and the equivalent `branch` goal.

Again, only head unifications can be subject to indexing. Our Schemelog compiler finds the longest sequence of disjuncts in an `orelse` goal such that each disjunct has a unification statement. The left hand side of each unifica-

tion statement should be the same variable, and the right hand sides should be terms of distinct shapes. Clause indexing is performed until there is no more indexing left to be done.

The compile-time unification pass aides the unification factoring and clause indexing passes by removing superfluous code and turning every explicit unification statement into a form with a variable on the left hand side. Also, unification factoring may open more possibilities for clause indexing. On the other hand, clause indexing does not enable any more unification factoring. Furthermore, neither clause indexing nor unification factoring introduce opportunities for compile-time unification. Therefore, these passes must be performed in exactly the order they are introduced in this paper.

4.4 Code generation

Code generation in Schemelog is quite straight forward. It accomplishes two purposes: making variable declarations explicit with `let` bindings and compiling partially instantiated unification statements into two-stream unification code. (For an explanation of the two-stream unification approach, refer to [Roy94].) For an illustration of two-stream unification code, let us consider the following predicate.

```
test([1|[2|3]]).
```

Predicate `test` checks whether its argument is a triple of 1, 2 and 3. The table in Figure 13 compares code produced by the Schemelog compiler with pseudo-code similar to what the original WAM translation prescribes. Code in the left column of the table exemplifies one-stream unification. If the input variable `a1` is unbound, execution enters the write mode and allocates a fresh variable `t1`. Then it comes back into the stream and checks whether `t1` is unbound to determine whether to branch into the write or read mode. For this particular execution trace, this test is redundant since the program has just created `t1`, and it is obviously unbound. The program on the right branches into the write stream as soon as it determines that `a1` is unbound and never comes back. It performs no checks as it builds the whole term on the heap. To avoid code explosion, Schemelog creates a constructor procedure for each subterm of the outer term. A Scheme compiler need not create closures for these procedures and can convert calls to them into goto statements.

WAM translation	SAM translation

a1 := dereference(a1);
if a1 is unbound
{
  t1 := fresh-variable();
  a1 := make-pair(1,t1);
}
else if a1 is a pair
{
  unify(first(a1),1);
  t1 := second(a1);
}
else fail();
 	
t1 := dereference(t1);
if t1 is unbound
{
  t1 := make-pair(2,3);
}
else if t1 is a pair
{
  unify(first(t1),2);
  unify(second(t1),3);
}
else fail();
	(letrec ([cns2
	  (lambda () (cons 1 (cns1)))]
	  [cns1 (lambda () (cons 2 3))])
	  (let ([a1 (deref a1)])
	  (cond
	  [(var? a1) (bind a1 (cns2))]
	  [(pair? a1)
	  (let ([t1 (cdr a1)])
	  (unify/2 (car a1) 1)
	  (let ([t1 (deref t1)])
	  (cond
	  [(var? t1)
	  (bind t1 (cns1))]
	  [(pair? t1)
	  (unify/2 (car t1) 2)
	  (unify/2 (cdr t1) 3)]
	  [else (fail)]))]
	  [else (fail)])))

Figure 13: Two-stream unification code

5 Conclusion and related work

This paper has introduced the Scheme Abstract Machine, an abstract machine that is geared toward compilation of logic languages into Scheme. We use the SAM to implement Schemelog, a Scheme based logic language. Schemelog features a simple module facility, syntactic abstraction mechanism, and provides access to Scheme and the run-time data structures through the `scheme` form.

In the presence of a macro system, it is important for a compiler to handle cases of trivial optimizations. While a human programmer rarely would write a program that is subject to these optimizations, the expanded code often contains easily optimizable fragments. To deal with that, we propose compile-time unification technique which optimizes explicit unifications. In its nature, it is similar to copy propagation and useless code elimination optimizations found in a Scheme compiler. A combination of compile-time unification, unification factoring, and clause indexing provides an efficient yet simple compiler for Schemelog.

There have been other attempts to implement Prolog in Scheme. Dorai Sitaram describes a language that is an "embedding of Prolog-style language in Scheme" [Sit97]. There are several important differences between Schemelog and Schelog. Schelog defines simple macros that transform the source code into Scheme without performing any compiler optimizations. As a result Schelog is slower than Schemelog. Although, Schelog has lexical predicates, it does not have module facilities; thus, only one query can be executed at a time, and there is no inheritance. On the other hand, Schelog is coupled with Scheme more tightly, and Scheme expressions and Prolog goals are indistinguishable and can interleave freely. Schelog supports any data type that can be encoded in Scheme as long as the pair of a constructor and a destructor is provided for each data type. For example, his unification algorithm handles string terms; while ours does not.

Another Scheme implementation of a Prolog-like language in Scheme is Prolog 1.2 [DKR94] It is implemented as an interpreter, and, therefore, is slower than both Schemelog and Schelog. However, Prolog 1.2 supports delayed goals and interval arithmetic.

Other systems attempted to compile Prolog to C [CD95]. We discussed the advantages of compilation to Scheme as opposed to C in the introduction. They are closer execution model, similar data types, garbage collection, and interactive compilation.

When developing Schemelog, we benefited from the extensive research conducted in the field of Prolog implementation. A description of the Warren Abstract Machine and various compiler optimizations including clause indexing can be found in [AK91]. Unification factoring is described in [DRR+]. Other modifications of the Warren Abstract Machine have been proposed in the literature. In particular, the SAM (as well as some compilation techniques) have been inspired by [Roy90] which describes the Berkeley Abstract Machine (BAM) and an efficient Prolog compiler. The idea of two-stream unification is discussed in [Roy94].

References

[AK91] H. Ait-Kaci. *Warren's Abstract Machine, A Tuturial Reconstraction.* Logic Programming. The MIT Press, 1991.

[Cad94] Cadence Research Systems, Bloomington, Indiana. *Chez Scheme System Manual, Rev. 2.4*, July 1994.

[CD95] Philippe Codognet and Daniel Diaz. WAMCC: Compiling Prolog to C. In Leon Sterling, editor, *Proc. The Twelfth International Conference on Logic Programming*, pages 317–333. The MIT Press, 1995.

[DKR94] Alan Dewar, Vinit Kaushik, and Sue Rempel. Prolog 1.2. `ftp://ftp.cpsc.ucalgary.ca:/pub/projects/prolog1.2/prolog12.tar.Z`, 1994.

[DRR+] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. Technical report, Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY.

[Roy90] Peter Lodewijk Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, UC Berkley, 1990.

[Roy94] Peter Lodewijk Van Roy. Issues in implementing logic languages. `http://www.info.ucl.ac.be/people/PVR/impltalk.html`, May 1994.

[Sit97] Dorai Sitaram. Programming in schelog. `ftp://titan.cs.rice.edu/public/dorai/schelog3e.tar.gz`, 1997.