Issues in Implementing Constraint Logic Languages

Peter Van Roy
DEC Paris Research Laboratory

École de Printemps
Châtillon/Seine
May 1994

Overview

- High-level issues
  - The simplification principle
  - How to implement a new logic language
  - The compiler intermediate form
  - Example of LIFE compilation
  - Using types when compiling constraints
- Prolog implementation issues
  - The Prolog language and the WAM
  - Evolution of Prolog performance
  - The WAM as a constraint language
  - How to compile unification
  - Using types when compiling unification
- Some general implementation techniques
  - Backtracking
  - Timestamping
  - Suspensions and the CLP(FD) implementation
- Conclusions and further work
- Bibliography

The simplification principle

- The heart of any efficient implementation of a constraint language is the simplification principle:

> Simplify each occurrence of a constraint as much as possible

- This is done at compile-time and run-time (statically and dynamically) and by looking at various-sized pieces of program (locally and globally).

- Examples of simplification:

Static:	"get_variable" in the WAM is a move
Dynamic:	Clause indexing
Local:	WAM unification is done per functor
Global:	The two-stream unification algorithm

- This principle has continued to hold from the early days of Prolog implementation (e.g., the DEC-10 compiler) to the present. It holds for the WAM, for native code systems, and for systems that do global analysis.

- All experimental evidence so far shows no contradiction with the hypothesis that the principle continues to hold until performance has reached that of imperative programming.

The simplification principle in the WAM

Simplify unification

- Consider the unification `X=s(a,T).`
- Since the second argument `s(a,T)` is known, the general algorithm can be specialized for this case, resulting in:

```
get_structure A1, s/2
unify_constant a
unify_value A2
```

Simplify backtracking

- Consider the predicate p with definition:

```
p(a).
p(b).
```

- If the first argument is an atom, then picking the right clause can be done with hashing instead of backtracking:

```
   switch_on_term V,C,fail,fail

C: switch_on_constant 2, T  % Hash lookup

A: get_constant a
   proceed

B: get_constant b
   proceed

T: entry a, A
   entry b, B

V: % Try both clauses with backtracking
```

How to implement a new logic language

- The *quickest* way to implement a new logic language is to write an interpreter in Prolog (e.g., Concurrent Prolog, Still-Life, Timed Gentzen, ...).

- There are several ways to get an *efficient* implementation:

    1. Extend the WAM. This technique has been the most popular (e.g., CLP(R), CLP(FD), ...). It is not the right way for languages very different from Prolog (e.g., λ-Prolog).

    2. Compile the language into an existing implementation of Prolog (e.g., QD-Janus). Modern implementations of Prolog are fast and support advanced control constructs such as coroutining (e.g., SICStus) and first-class suspensions (e.g., ECLiPSe).

    3. If the above is difficult, then try to compile the language into a system that provides useful primitives. For example, λ-Prolog has been implemented with MALI, a memory management library tailored for logic programming.

- To get the *fastest possible* implementation, it is necessary to compile directly to a lower-level language (e.g., C or assembly).

Implementation trade-offs

Approach	Portability	Execution speed	Compilation speed	Foreign language interface	Dynamicity (assert/retract)	
Interpreter in Prolog	yes	slow	fast	poor	yes	(1)
Interpreter in C or C++	yes	slow	fast	poor	yes	
Emulator in C or C++	yes	average	fast	poor	yes	
Generate Prolog	yes	average	depends *	depends *	yes	(2)
Generate C or C++	yes	fast †	slow	best	no	(3)
Assembly	no	fast	fast	average	no	
Machine code (binary)	no	fast	fast §	average	yes	

* Depends on the underlying Prolog system

† Using RISC-macro technique (see next slide)

§ Compilation becomes more complex with modern (superscalar) RISC

- Three good trade-offs:
  - (1) Smallest development time to get a system that works
  - (2) Smallest development time to get a system that is fast enough to be useful
  - (3) Best speed while maintaining excellent portability and interoperability

C as a portable assembler

- Goals: speed, portability, and interoperability.

- Generating assembly is efficient but nonportable.
  Generating emulated byte code is portable but inefficient.
  Naive use of C as a target language is inefficient.

- A solution (R. Meyer):
  The RISC-macro technique using GNU C.

  1. Generate RISC code, where the RISC instructions are C macros. The macros are chosen so that the C compiler translates them into actual RISC instructions.

  2. Do not use the C call stack. Compile short branches as gotos and long branches as functions (using a simple interpretive loop). Control overhead < 5% and C procedure sizes are bounded.

  3. Assign registers to global variables (a feature of GNU C).

  4. Let the C compiler handle register allocation, peepholing, architecture-dependent issues and calls to C routines.

- This technique is being used in the LIFE compiler development at DEC PRL. It is fast (3 times faster than the first C code generation and within 30% of pure native) and portable (same code runs on MIPS, Alpha, and i486).

The compiler intermediate form for constraint language implementation

- The intermediate form is the compiler's internal representation of the program.

- A good intermediate form should be able to express correctness independently of termination and efficiency.

- Ideally, an intermediate form has 3 orthogonal components:

    1. Primitive constraints: the basic data manipulation instructions, which are executable and non-directional. They may include structural constraints (e.g., unification of Herbrand terms) and arithmetic constraints (e.g., equalities and inequalities).

    2. Control flow: all modifications of control flow. This includes calls, jumps, closures and continuations.

    3. Type attributes: standard data types as well as modedness and aliasing information.

- This allows expressing purely declarative execution as well as efficient operational execution.

- Compilation proceeds in two steps:

    1. Add control and type information to the constraints. This can be done through programmer annotation, compiler transformations and global analysis.

    2. Translate the annotated constraints to the target language.

Examples of intermediate forms

- We present two examples of intermediate forms that have been designed for efficient execution of Prolog.

The Warren Abstract Machine (WAM) instruction set

- The WAM instruction set can be divided into a constraint part (get, put, unify instructions) and a control part (call, switch, choice points). The instructions have execution order and type information wired in. Mapping from Prolog is straightforward and many important optimizations are designed in.

- Research issue: The constraints are too tightly bound with execution order and types to allow for significant further optimization. For example, the "unify" instructions must be executed in a given order to unify a term's arguments.

The Aquarius Kernel Prolog language

- Aquarius Kernel Prolog is a simplified representation of Prolog with all syntactic sugar removed. The three components are orthogonal (e.g., type information is stored separately from the program).

- Research issue: The only constraints and control that are represented are Prolog's. Other constraints (e.g., arithmetic) and other control (e.g., coroutining) are not represented.

Example of LIFE compilation (1)

- LIFE is a constraint language implementing unification and matching on ψ-terms.
- A ψ-term is a generalization of a Prolog term. It is to a Prolog term as a dynamic record is to a static array. That is, it has named fields and fields may be added at will. There is no notion of a *ground* term.
- In this example, we show how the RISC-macro technique is used to compile ψ-term unification in LIFE.
- A ψ-term can be considered as a conjunction of three primitive constraints: a sort constraint $(X : s)$, a feature constraint $(X.f = Y)$, and an equality constraint $(X = Y)$.
- The ψ-term $X : person(age \Rightarrow 25)$ is equivalent to $X : person \wedge (X.age = Y) \wedge (Y : 25)$.
- The representation of ψ-terms, if used as Prolog terms, has at most a single word of memory overhead.

Example of LIFE compilation (2)

- Consider a LIFE program consisting of the single fact p(X:f(X)). This program is translated into Kernel LIFE and then into FLAM code. The FLAM translation uses the two-stream unification algorithm:

```
extern(p)
   allocate
      pred_args([v(1)])
      begin_unify
         sort_and_features(v(1),f,[1],[v(2)],0,intern(6))
         if_new_feat(v(2),intern(7),1)
         unify(v(2),v(1))
      intern(8)
         jump(intern(9))
      intern(6)
      intern(7)
         write_feature(v(1),v(2))
         write_test(intern(8),1)
      intern(9)
      end_unify
   deallocate
   return
```

- The instruction sort_and_features(Reg,Func,FeatList,RegList,Level,Label) corresponds to the constraint $v = f(f_1 \Rightarrow v_1, \ldots, f_n \Rightarrow v_n)$, with:

Reg	V
Func	f
FeatList	$[f_1, \ldots, f_n]$
RegList	$[V_1, \ldots, V_n]$
Level	(two-stream unification)
Label	(branch to write stream)

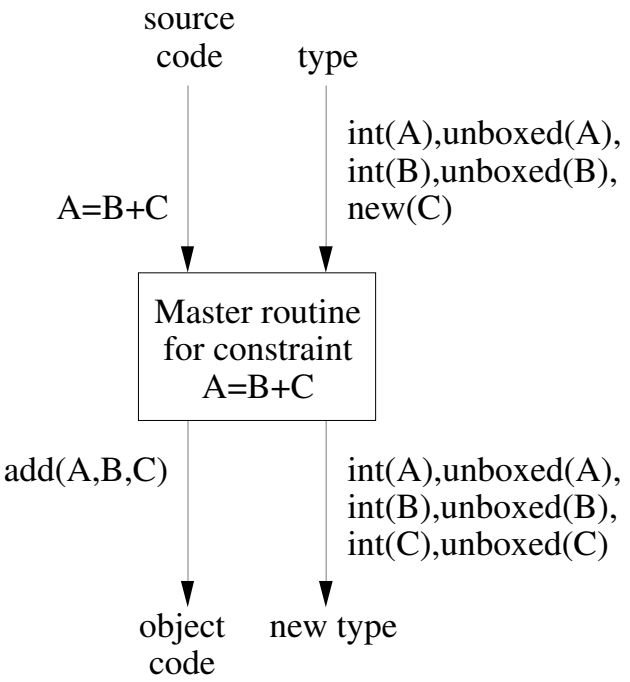
- The instructions unify(Reg1,Reg2) and write_feature(Reg1,Reg2) correspond to the constraint $v = w$, with Reg1 and Reg2 corresponding to v and w.
- The FLAM code is translated into C using the RISC-macro technique.

Example of LIFE compilation (3)

- The C code resulting from the compilation of p(X:f(X)) is a sequence of macros that strongly resembles a RISC assembly program. It is 101 lines long and begins as follows:

```
extern(p)
   start_block(block(local_vars => [t(0),t(1),l(2),t(2)],number => 0))
   comment(p = allocate)
   blt(frame,choice,intern(10))
   sub(choice,frame_block(0),t(0))
   b(intern(11))
intern(10)
   sub(frame,frame_block(0),t(0))
intern(11)
   sw(frame,frame_previous,t(0))
   sw(cont,frame_cont,t(0))
   mv(t(0),frame)
   lv(t(0),0)
   sw(t(0),frame_size,frame)
   comment(p = begin_unify)
   comment(p = sort_and_features(v(1),f,[1],[v(2)],0,intern(6)))
   b(intern(13))
intern(12)
   mv(t(0),r(0))
intern(13)
   lref(t(0),r(0))
   bref(t(0),intern(12))
   beq(t(0),static_add(sort_f,feat_tag),intern(14))
   beq(t(0),topAtom,intern(18))
   beq(t(0),static_add(topAtom,feat_tag),intern(15))
   beq(t(0),sort_f,intern(14))
   jmp_fail
intern(14)
   b(intern(17))
   jmp_fail
intern(15)
   lv(t(1),static_add(sort_f,feat_tag))
   sw(t(1),0,heap)
   set_ref(heap,t(1))
   add(heap,word,heap)
   c_call(CopyTable(r(0)))
   bgt(r(0),last_heap,intern(16))
...
```

Using types when compiling constraints



- The source code is annotated with type information.
- A single *master routine* per constraint generates specialized object code and updates the type.
- The master routine can be used as the abstract operation in global analysis. The analyzer is easily integrated in the compiler.
- This technique is planned for the LIFE compiler. It generalizes the technique of entry specialization used in the Aquarius compiler. Using an accumulator preprocessor the source code is kept compact.

Sample code for the master routine

```
master_add(X=Y+Z, Tin, Code, Tout) :-
    if (Tin=>(int(Y),int(Z)) then
        % Generate add instruction:
        Code = [add(T,Y,Z),unify(T,X)]
        % Update type:
        Tout = update(int(X),Tin)
    else
        % Handle other special cases
        ...
    else
        % Default (most general) case:
        Code = [general_add(X,Y,Z)]
        Tout = update((int(X),int(Y),
                       int(Z)),Tin))
    endif.
```

- Special cases can be added according to need. Leaving them out affects speed and does not affect not correctness.

The Prolog language and the WAM

- Prolog is a constraint logic language that solves equality constraints over finite trees or rational trees. Constraint solving is done by unification. Control flow is sequential augmented with chronological backtracking.

- Prolog has been used as a base to build constraint systems using other domains, both practically and theoretically. For example, CLP(R) handles linear equalities and inequalities over floats, and is built on top of a standard Prolog engine.

- Prolog systems are reaching the performance of imperative programming. Hence it is important to understand what makes Prolog run fast.

- The main breakthrough in Prolog implementation was the development of the Warren Abstract Machine (WAM) in 1983. The WAM defines a high-level instruction set and execution model for Prolog. Implementation work since then has built on this foundation.

Evolution of Prolog performance

System	Machine	Clock	Benchmark					
		(MHz)	N	Q	D	S	R	HM
‡ DEC-10 Prolog	DEC-10		1	1	1	1	1	1
XSB 1.3	SPARC	25	7	4	2	4	3	3
Quintus 2.0	Sun	20	11	4	3	4	3	4
‡ MProlog 2.3	386	33	13	6	5	5	2	5
ECLiPSe 3.3.7	SPARC	25	11	6	4	6	3	5
NU-Prolog 1.5.38	SPARC	25	22	7	5	7	2	5
SICStus 2.1	DEC	25	37	16	10	10	5	10
Quintus 2.5	SPARC	25	33	16	9	13	8	12
‡ BIM 3.1 beta	SPARC	25	34	21	8	16	8	13
‡ SICStus 2.1	SPARC	25	39	26	15	20	8	17
‡ § † Aquarius	SPARC	25	120	140	28	25	12	29
‡ § IBM Prolog	IBM		120	59	74	69	33	60
‡ § Aquarius 1.0	DEC	25	180	210	63	44	46	71
‡ § † Parma	MIPS	25	330	350	130	170	59	140

- This table compares popular software implementations of Prolog.
- Annotations: ‡ (native code system), § (system with global analysis), † (research system).
- Benchmarks with times on DEC-10 Prolog (year 1977, in ms): N (naive reverse, 53.7), Q (quicksort, 75.0), D (deriv, 10.1), S (serialise, 40.2), R (query, 185.0), HM (harmonic mean).
- Machines: DEC (DECstation 5000/200), IBM (IBM System 370 ES/9000 Model 9021), MIPS (MIPS R3230), SPARC (SPARCstation 1+), Sun (Sun 3/60 MC68020), 386 (IBM PC clone).

The WAM as a constraint language

- The data manipulation instructions of the WAM instruction set are executable constraints, specialized with execution order and type information.
- The WAM provides two primitive constraints, a functor constraint and an equality constraint.

Constraint

Instruction sequences that implement the constraint

Constraint	Decomposed constraints	Two possible instruction sequences
X=f(X1,...,Xn) (n≥0, Xi variable or constant)	X=f/n X.1=X1 ... X.n=Xn	n unify instructions put_... unify_... ... unify_... get_... unify_... ... unify_...

Constraint	Eight possible instructions	
X=Y	put_variable ...	unify_variable ...
	put_value ...	unify_value ...
	get_variable ...	put_unsafe_value ...
	get_value ...	unify_local_value ...

- The instructions are specialized for variables occurring for the first time (denoted “variable”) and others (denoted “value”).
- The “unify” instructions may only occur in the positions shown.

The WAM instruction set

Constraint	Specialized constraint (with execution order and type)	
	Loading argument registers (just before a call)	
$X = Y$	**put_variable** V_n, R_i	Create a new variable, put in V_n and R_i.
$X = Y$	**put_value** V_n, R_i	Move V_n to R_i.
$X = C$	**put_constant** C, R_i	Move the constant C to R_i.
$X = nil$	**put_nil** R_i	Move the constant *nil* to R_i.
$X = F/N$	**put_structure** F/N, R_i	Create the functor F/N, put in R_i.
$X = ./2$	**put_list** R_i	Create a list pointer, put in R_i.
	Unifying with registers (head unification)	
$X = Y$	**get_variable** V_n, R_i	Move R_i to V_n.
$X = Y$	**get_value** V_n, R_i	Unify V_n with R_i.
$X = C$	**get_constant** C, R_i	Unify the constant C with R_i.
$X = nil$	**get_nil** R_i	Unify the constant *nil* with R_i.
$X = F/N$	**get_structure** F/N, R_i	Unify the functor F/N with R_i.
$X = ./2$	**get_list** R_i	Unify a list pointer with R_i.
	Unifying with structure arguments (head unification)	
$X.i = Y$	**unify_variable** V_n	Move next structure argument to V_n.
$X.i = Y$	**unify_value** V_n	Unify V_n with next structure argument.
$X.i = C$	**unify_constant** C	Unify the constant C with next structure argument.
$X.i = nil$	**unify_nil**	Unify the constant *nil* with next structure argument.
$\bigwedge_{1 \le j \le N} X.(i+j-1) = Y_i$	**unify_void** N	Skip next N structure arguments.
	Managing unsafe variables	
$X = Y$	**put_unsafe_value** V_n, R_i	Move V_n to R_i and globalize.
$X.i = Y$	**unify_local_value** V_n	Unify V_n with next structure argument and globalize.
	Procedural control	
	call P, N	Call predicate P, trim environment size to N.
	execute P	Jump to predicate P.
	proceed	Return.
	allocate	Create an environment.
	deallocate	Remove an environment.
	Selecting a clause (conditional branching)	
	switch_on_term V, C, L, S	Four-way jump on type of A_1.
	switch_on_constant N, T	Hashed jump (size N table at T) on constant in A_1.
	switch_on_structure N, T	Hashed jump (size N table at T) on structure in A_1.
	Backtracking (choice point management)	
	try_me_else L	Create choice point to L, then fall through.
	retry_me_else L	Change retry address to L, then fall through.
	trust_me_else *fail*	Remove top-most choice point, then fall through.
	try L	Create choice point, then jump to L.
	retry L	Change retry address, then jump to L.
	trust L	Remove top-most choice point, then jump to L.

How to compile unification

In the WAM (1983)

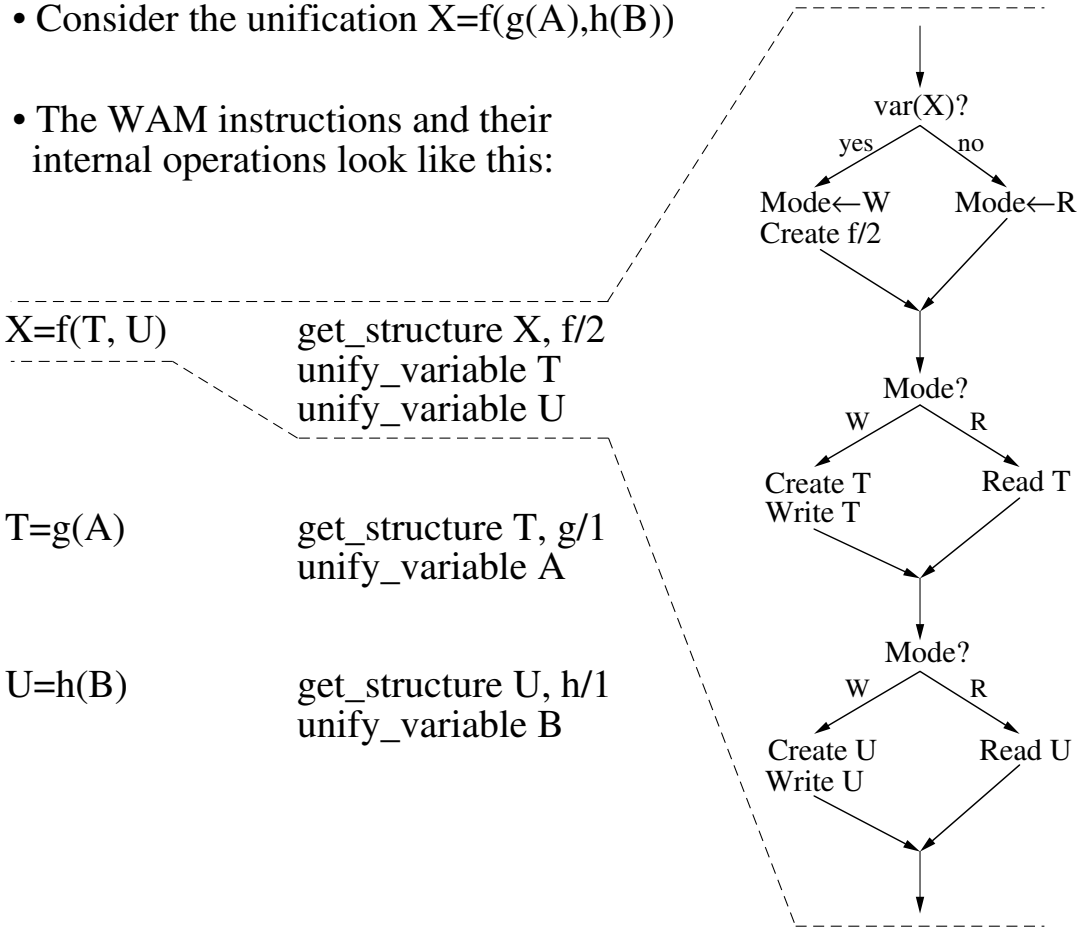
- Single instruction stream
- A mode flag distinguishes between Read and Write mode
- Breadth-first traversal of terms: X=f(g(A),h(B)) is compiled as X=f(T,U), T=g(A), U=h(B)
- Problems:
  - Write mode is not propagated to substructures, resulting in superfluous variable creations and bindings
  - Every instruction sets or tests the mode flag
  - Superfluous work on failure, e.g. X=f(g(a),_,_) always unifies the last two arguments

The two-stream algorithm (1989)

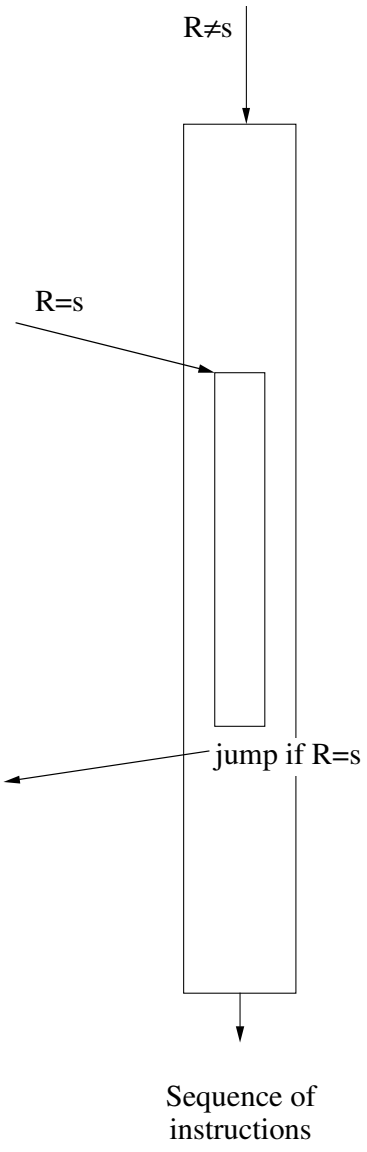
- Two instruction streams: for Read and for Write mode
- Depth-first traversal of terms
- Key idea: a mechanism to jump between the Read and Write mode streams as needed, with very low overhead
- Advantages:
  - No superfluous operations
  - Downward propagation of Write mode (to substructures)
  - Linear code size
  - Efficient expansion to native code

How to compile unification: The WAM

- Consider the unification X=f(g(A),h(B))
- The WAM instructions and their internal operations look like this:



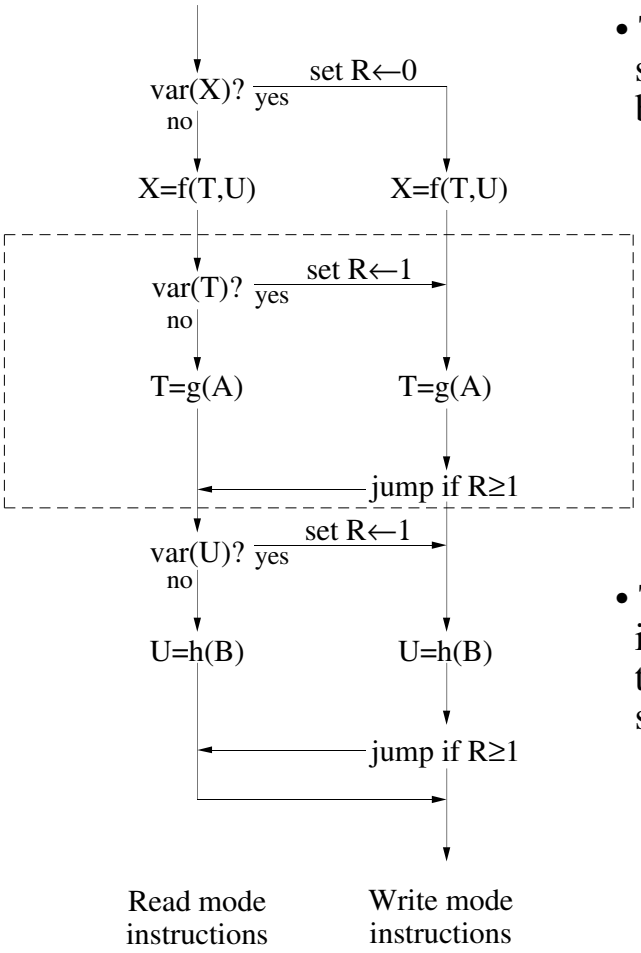
How to compile unification: The two-stream idea



- Given a sequence of instructions.
- It is possible to efficiently execute any contiguous subsequence. Give each subsequence a unique identifier *s*. Then a single comparison per subsequence and a single register R are all that is needed.
- Arrange the unification instructions in a depth-first traversal of the term. Then two properties are true:
  - Each subterm is a contiguous sequence of instructions.
  - Nested subterms are also nested sequences of instructions. (This permits further reduction of overhead.)
- Idea due to Mohamed Amraoui, André Mariën and Bart Demoen, Kent Boortz, and Micha Meier.

How to compile unification: The two-stream algorithm

- Consider the unification X=f(g(A),h(B))
- The two-stream compilation looks like this:



- There are two instruction sequences, with jumps between them

Instructions that implement the unification of T with the subterm g(A)

- The constraint T=g(A) is compiled differently in the Read and Write mode sequences

Using types when compiling unification

- This table describes how unification is compiled in Aquarius.
- unify(X,Y) generates code for the unification X=Y using type T.
- "T $\Rightarrow$ var(X)" tests whether type T implies var(X).

Name	Condition	Actions
unify(X, Y)	var(X), var(Y)	var_var(X, Y)
	var(X), nonvar(Y)	var_nvar(X, Y)
	nonvar(X), var(Y)	var_nvar(Y, X)
	nonvar(X), nonvar(Y)	nvar_nvar(X, Y)
nvar_nvar(X, Y)		$\forall$ args X_i, Y_i: unify(X_i, Y_i)
var_nvar(X, Y)	T $\Rightarrow$ new(X)	new_old(X, Y)
	T $\Rightarrow$ ground(X)	old_old(X, Y)
	otherwise	old_old(X, Y) (depth limited)
var_var(X, Y)	T $\Rightarrow$ (old(X), old(Y))	oldv_oldv(X, Y)
	T $\Rightarrow$ (old(X), new(Y))	Generate store instruction
	T $\Rightarrow$ (new(X), old(Y))	Generate store instruction
	T $\Rightarrow$ (new(X), new(Y))	new_new(X, Y)
new_new(X, Y)		Generate store and move instructions
new_old(X, Y)	compound(Y)	W_seq(X, Y)
	atomic(Y)	Generate store instruction
	var(Y)	var_var(X, Y)
old_old(X, Y)	compound(Y), (T$\Rightarrow$nonvar(X))	Test Y type, then old_old_R(X, Y)
	compound(Y)	Generate switch, R & W branches
	atomic(Y), (T $\Rightarrow$ nonvar(X))	old_old_R(X, Y)
	atomic(Y)	Generate unify_atomic instruction
	nonvar(Y), (T $\Rightarrow$ var(X))	old_old_W(X, Y)
	var(Y)	var_var(X, Y)
oldv_oldv(X, Y)	A = atomic_value(T, X)	unify(Y, A)
	A = atomic_value(T, Y)	unify(X, A)
	T $\Rightarrow$ (atomic(X), atomic(Y))	Generate comparison instruction
	T $\Rightarrow$ (var(X), nonvar(Y))	Generate store instruction
	T $\Rightarrow$ (nonvar(X), var(Y))	Generate store instruction
	otherwise	Generate unify instruction
old_old_W(X, Y)	compound(Y)	W_seq(X, Y)
	atomic(Y)	Generate store instruction
old_old_R(X, Y)	compound(Y)	$\forall$ args X_i, Y_i: old_old(X_i, Y_i)
	atomic(Y)	Generate comparison instruction
W_seq(X, Y)		Generate W mode sequence

Timestamping (1)

- To *trail* a binding means to store enough information on a stack (the *trail stack*) so that backtracking may restore the unbound state.

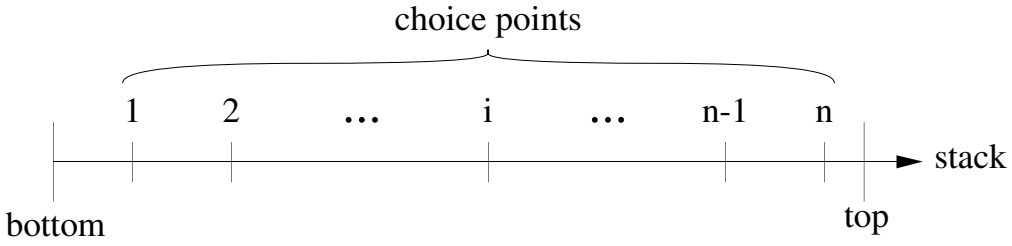
- WAM trail condition: Trail a variable binding if the address of the variable is less than the address of the top-most choice point.

  - This works well for the WAM since an unbound variable can only be bound *once* on forward execution.
  - Other constraints may be modified ("bound") more than once on forward execution (e.g., refinement of finite domains). The WAM condition results in too much trailing for them (c.f., CHIP).

- Improved trail condition: Trail a constraint modification if the previous modification was done before the creation time of the top-most choice point.

- Implementing this trail condition requires that each choice point and each constraint contain a timestamp marking its creation or modification time.

Timestamping (2)



- At any execution point, number the choice points on the stack from 1 to n.
- Let ts_i be the timestamp corresponding to choice point i and top_i be the stack top at the creation time of choice point i.
- The following invariants are maintained:

$$i<j \Rightarrow top_i \leq top_j$$
$$i<j \Rightarrow ts_i < ts_j$$

- Trail condition when modifying constraint c:

if $ts_n > ts_c$ then trail

- Maintaining consistent values of the timestamps:

1. Keep a global timestamp counter gts
2. Store gts in choice points and constraints at their creation
3. Increment gts at choice point creation
4. Never decrement gts

The CLP(FD) language

- CLP(FD) is a constraint logic language that incorporates a finite-domain solver built using the "glass-box" approach.
- "Glass-box": provide primitive operations to allow efficient implementation of constraint solvers in the user language.
- CLP(FD) provides the single constraint **X in R** where X is a finite-domain variable and R is a range. Various ranges are provided, e.g., L..H, min(Y), max(Y), dom(Y), val(Y).
- The finite-domain constraint "X=Y+C" can be implemented as follows:

  Partial lookahead scheme:

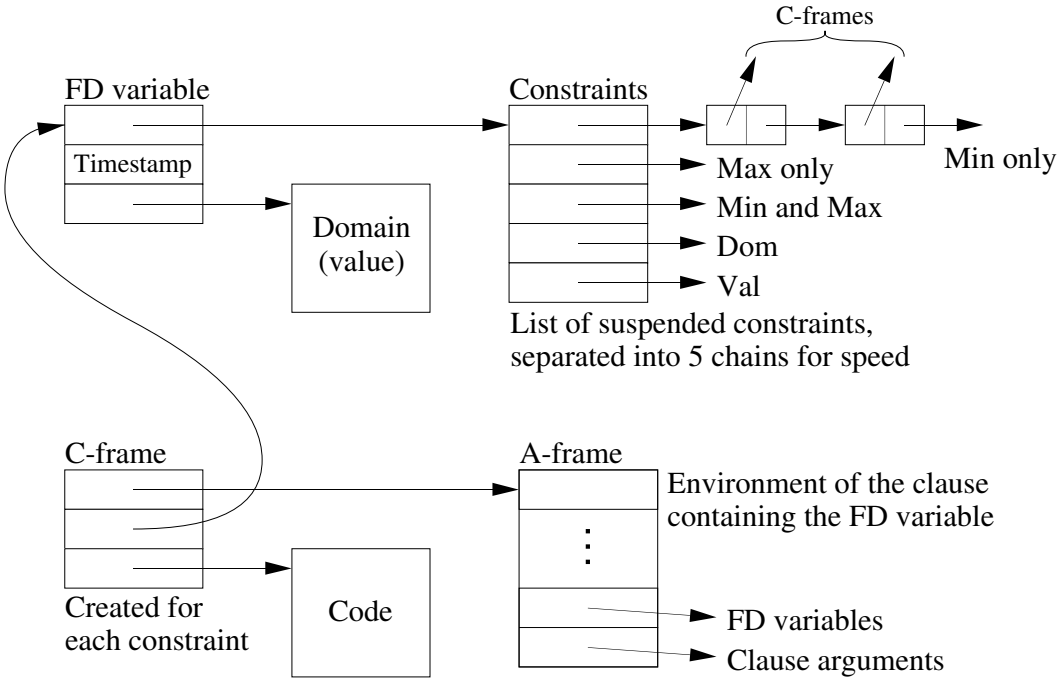
```
'X=Y+C'(X,Y,C) :-
    X in min(Y)+C..max(Y)+C,
    Y in min(X)-C..max(X)-C.
```

  Full lookahead scheme:

```
'X=Y+C'(X,Y,C) :-
    X in dom(Y)+C,
    Y in dom(X)-C.
```

The CLP(FD) implementation

- CLP(FD) is implemented by translation to C. The base engine is a WAM, and the WAM instructions are C macros.
- A new class of FD variables is added, with a new tag.
- Unification of FD variables and standard variables is defined in the obvious way.
- Data structures and abstract instructions are added to support the range constraints.
- Several non-trivial constraint optimizations are implemented. The system is significantly faster than CHIP.
- The following data structures support the constraint solver:



Conclusions and Further Work

- The field of constraint logic language implementation is relatively new. Much remains to be done.

- Prolog is a constraint language over a simple domain. Prolog implementation technology has progressed much in the last decade, and implementation of other constraint domains can profit from this work.

- Some fruitful areas for further work:

    - Development of constraint compilers.

    - Design of a common intermediate form to be shared between researchers to avoid duplication of work.

    - Extension of the "glass-box" approach to other constraint domains.

    - Better understanding of "cooperation" between constraint solvers.

    - Development of global analysis for constraint systems.

Partial bibliography

Daniel Diaz and Philippe Codognet. A Minimal Extension of the WAM for clp(FD). In *10th ICLP*, pages 774–790, Budapest, Hungary, MIT Press, June 1993.

Joxan Jaffar and Michael J. Maher. Constraint Logic Programming: A Survey. In *JLP*, Tenth Anniversary Issue, 1994.

Niels Jørgensen, Kim Marriott and Spiro Michaylov. Some Global Compile-Time Optimizations for CLP(R). In *8th ILPS*, pages 420–434, MIT Press, October 1991.

Peter Van Roy. 1983–1993: The Wonder Years of Sequential Prolog Implementation. In *JLP*, Tenth Anniversary Issue, 1994.