7° Convegno
sulla Programmazione Logica

Tremezzo, Lago di Como
17-19 Giugno 1992

Contexts as First-Class Objects: an implementation based on the SICStus Prolog system

Enrico Denti*, Antonio Natali*, Andrea Omicini **

**DEIS, Università di Bologna*
Viale Risorgimento, 2
40136 Bologna - Italy
email: natali@deis33.cineca.it

***DS Logics*
Viale Silvani,1
40122 Bologna - Italy
email: andrea@deis33.cineca.it

Why structured logic programming?

- need for structuring mechanisms

  applications in the large, AI, ...

- various proposals

  blocks, modules, points of view, objects, ...

Why contextual logic programming? [1]

- uniform static and dynamic composition to build software components
- support and integration of different structuring policies [2]
- declarative style
- neat semantics
- efficiency(?): implementation founded on simple basic mechanisms, with a possible, natural extension of the standard Prolog machine

[1] L. MONTEIRO, A. PORTO, *Contextual Logic Programming*, in G. Levi and M. Martelli (eds.), Proceedings 6th International Conference and Symposium on Logic Programming, The MIT Press, Cambridge (USA), 1989.

[2] A. BROGI, E. LAMMA, P. MELLO, *A General Framework for Structuring Logic Programs*, C.N.R. Technical Report "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", N. 4/1, May 1990.

An example

Given the following units:

```
:- unit(carOwner).
car(antonio,alfa33).
car(evelina,volvoPolar).
car(paola,fiat500).


:- unit(hasA).
hasAcar(X) :- car(X,_).
hasApc(X) :- pc(X,_).
.....
```

calling

```
:- [hasA,carOwner] <- hasAcar(X).
```

one should expect to have the following answers:

```
X = antonio;
X = evelina;
X = paola;
```

Context as a structured software component

(programmer level)

- context as a composition of units
- units: partition of the logic theory space
- unit: • unalterable
  - collection of clauses
  - unique name

Context as a binding environment

(implementation level)

- different binding policies [3]

  (a) conservative policy
  (b) evolutive policy

  C = [Un,...,Ui,...,U1], p/n called in Ui

  (a) late binding: *(evolutive policy)*

  p/n solved in [Un,...,Ui,...,U1]
  (current context, global —, lazy —)

  (b) eager binding: *(conservative policy)*

  p/n solved in [Ui,...,U1]
  (current bindcontext, partial —, eager—)

Syntactic sugar

- binding operators

contextCall(Context,Goal)	`Context <- Goal`
lazyCall(Goal)	`# Goal`
eagerCall(Goal)	`: Goal` *(default, no op)*

- context extension operators

`Unit >>> Goal` *(current context extension & context call)*
`Unit >:> Goal` *(current bindcontext extension & context call)*

[3] E. LAMMA, P. MELLO, A. NATALI, *The design of an abstract machine for efficient implementation of contexts in Logic Programming*, Proceedings of the Sixth International Conference of Logic Programming, Lisboa, Portugal, MIT Press, June 1989.

Context as an abstract data type

- constructors
  - `createContext(Unit,OldCtx,NewCtx)`
  - `emptyCtx`

- selectors
  - `topUnit(Ctx,Unit)`
  - `firstSubcontext(Ctx,Sub1ctx)`

- predicates

  (many)

Implementation guidelines

- fulfilment of the above described abstractions & constructs
- efficiency
- industrial-level programming environment
- time is money

⇓

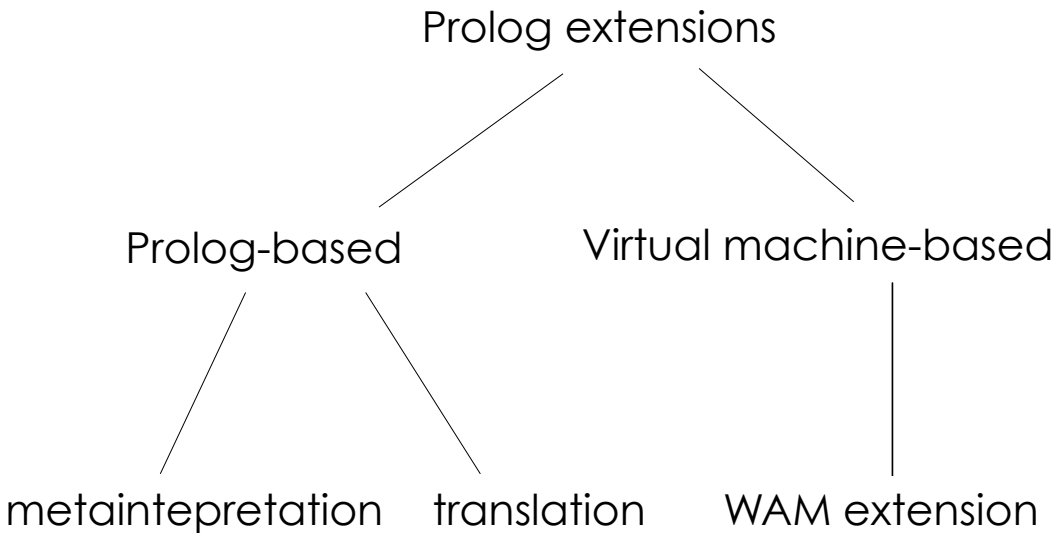
cannot start implementation from scratch, but from an existing advanced and efficient Prolog system:

⇓

SICStus Prolog [4]

[4] SWEDISH INSTITUTE OF COMPUTER SCIENCE, *SICStus Prolog User's Manual*, Kista (Sweden),1991.

Implementation in a compiled environment

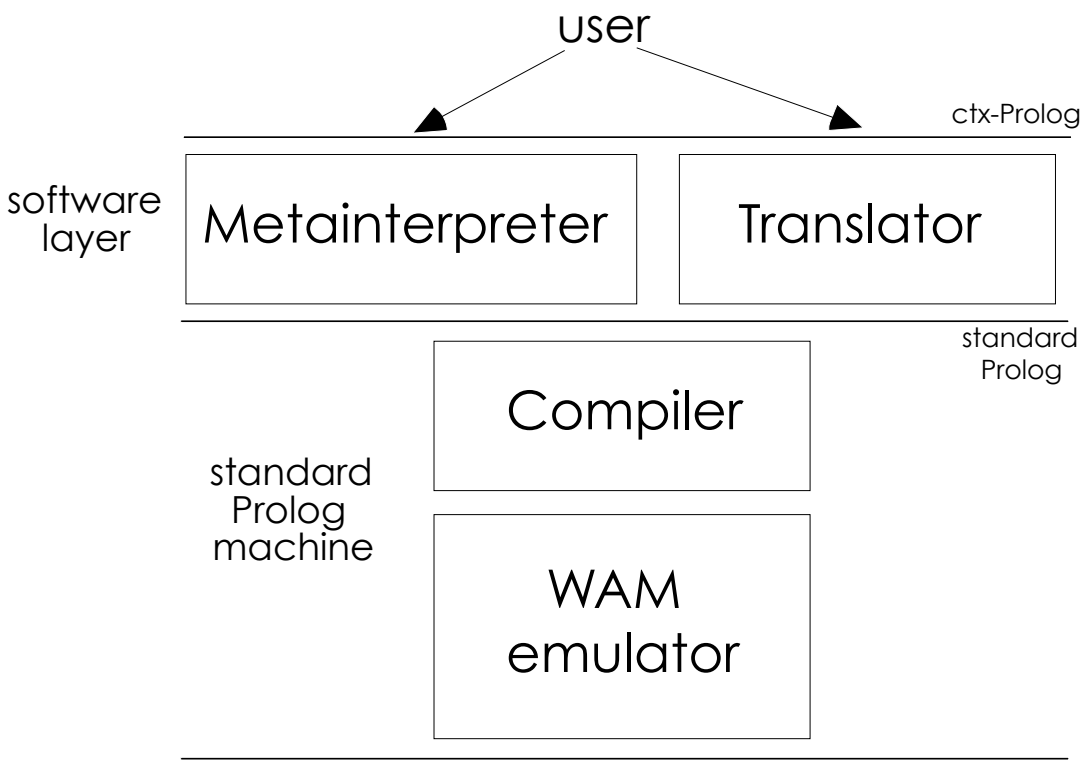


Implementation (1)

1) Prolog-based approach

   - metainterpretation
   - translation

   easy implementation, no great efficiency



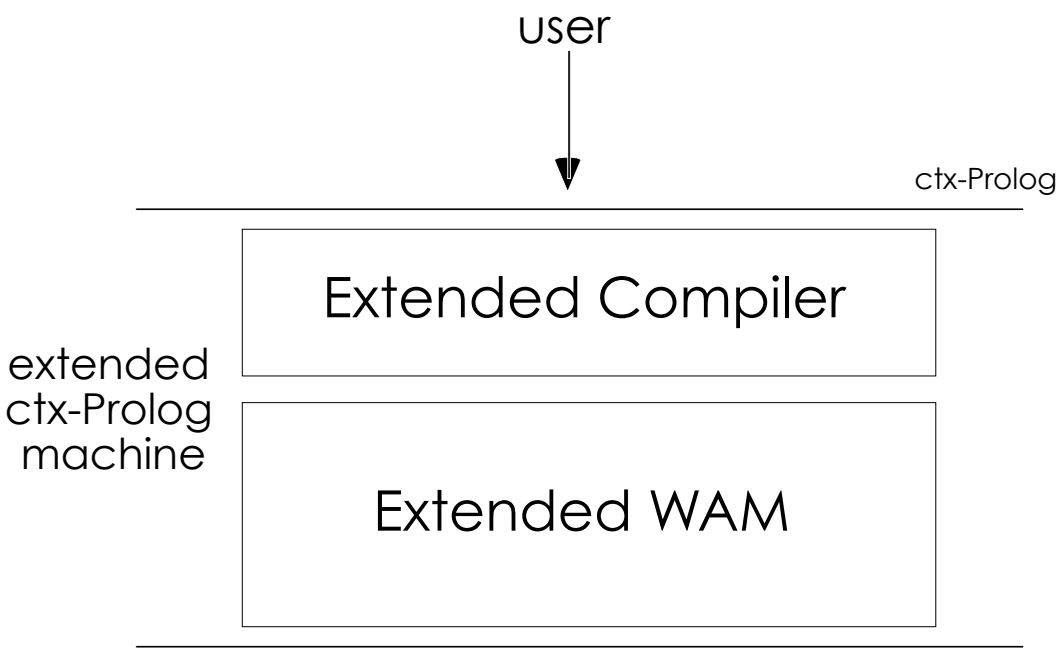
Implementation (2)

2) Virtual machine-based approach

- WAM-extension
  (S-WAM)

efficient, but

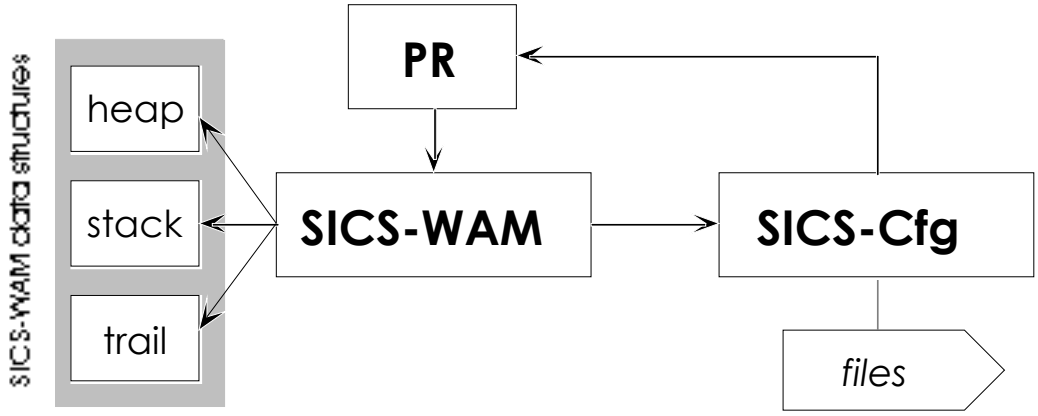
- difficult to implement
- computational overhead



SICStus Prolog

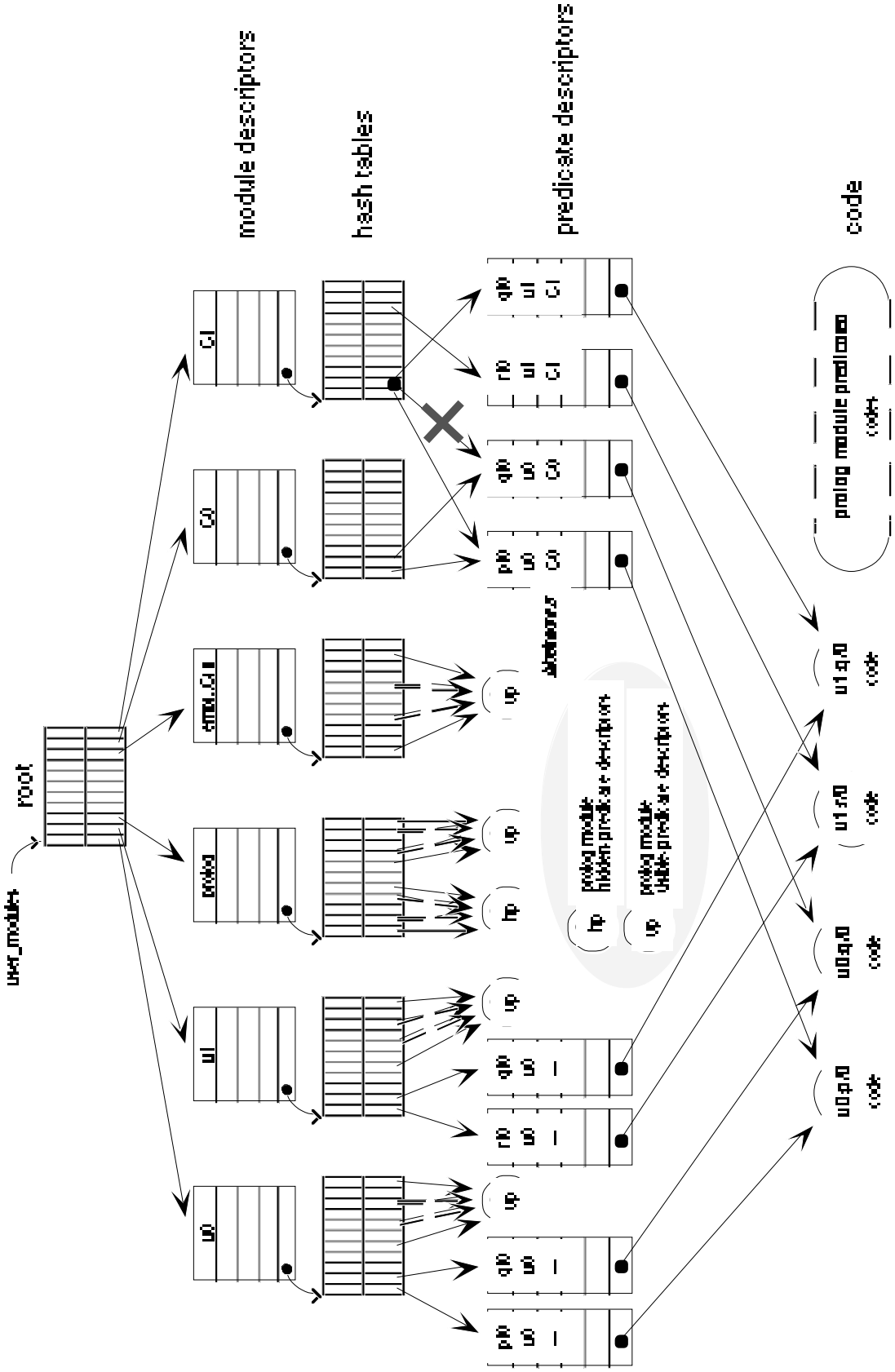
- refinement of the standard Prolog machine:

  it deserves to be described by a more complex scheme



PR *program representation part*
SICS-Cfg *configurator*
SICS-WAM *SICStus version of the WAM*

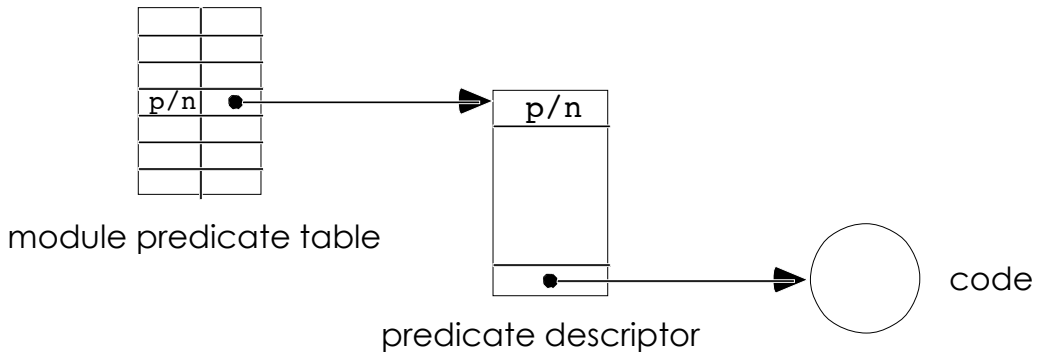
Code tree data-structure - a snapshot



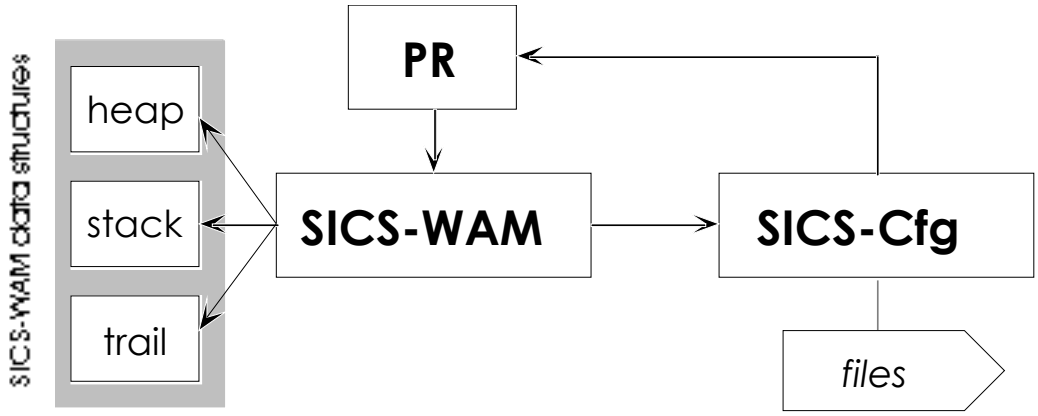
Indirect addressing mechanism

- support for a more efficient indexing scheme
- support for modularity
- incremental loading of clauses
- coexistence of both compiled and interpreted predicate-code.

An example of SICStus indirectness



Program-representation-based approach



What do contexts need?

1) global naming mechanism, partition of clause space

   (module names and separate predicate spaces)

   -> units as SICStus modules

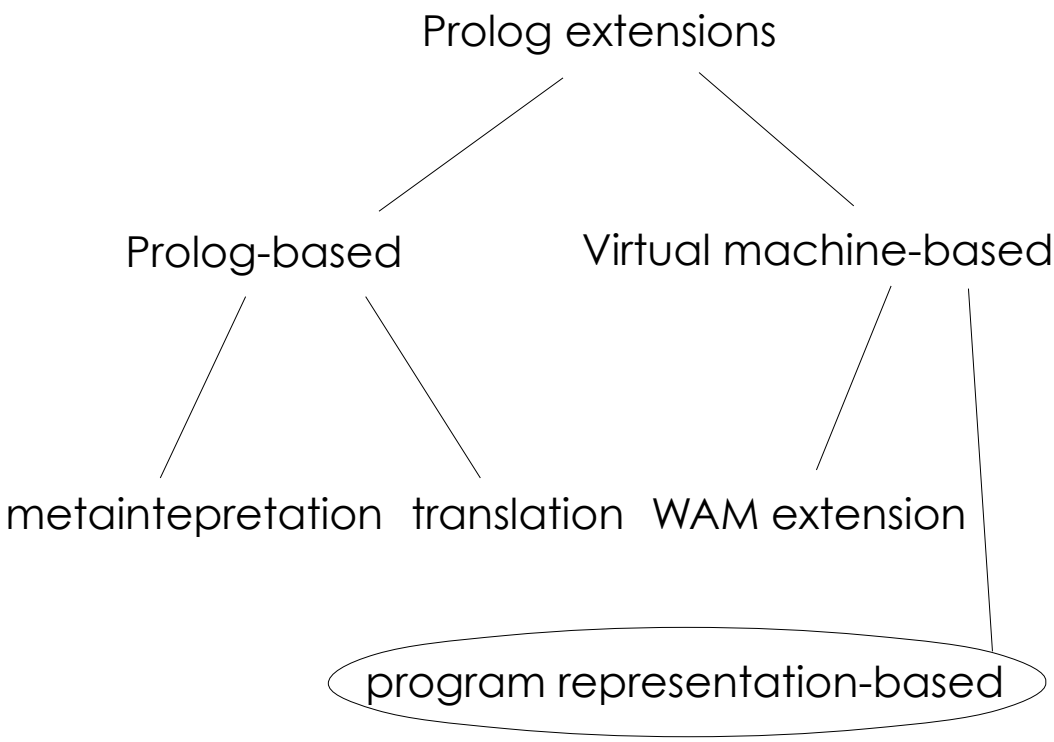
2) indirect addressing mechanism

   (module addressing scheme for context binding mechanism)

   -> **C**ontexts as **S**ICStus **M**odules

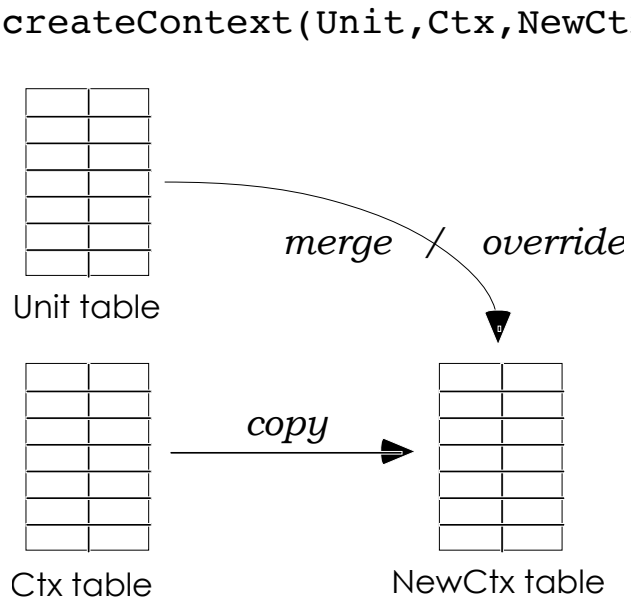
CSM

Implementation in SICStus Prolog



Context implementation

- context creation
  - a new table for every new context
  - memoisation —> “smart” creation



- context use
  - *local binding*: “pure” SICStus
  - *CSM binding*: indirectness via context tables

CSM approach results

1) easy to implement

   - context calls as SICStus module calls
   - context creation as SICStus module creation

2) trasparence w.r.t. SICStus Prolog

   - no computational overhead
   - no semantic distorsions

3) efficient implementation

   - based on an intrinsically efficient support
   - CSM binding policies *(late binding included)* rely on SICStus own binding mechanism

4) contexts as first-class objects

   - contexts can be referred to by logic variables
   - smart context creation through context memoisation in a Knowledge Base
   - natural definition of structured software components allowing easy (incremental) specialization