Modular Multiset Rewriting

Iliano Cervesato
iliano@cmu.edu

Edmund S.L. Lam
sllam@qatar.cmu.edu

Carnegie Mellon University

Supported by QNRF grants NPRP 09-667-1-100 and NPRP 4-341-1-059

LPAR-20, November 2015

Outline

Rule-based Languages

Modify a global state by concurrently rewriting disjoint portions

- Forward logic programming
  - Datalog (databases, networking, robotics, security, spreadsheets)
- Rewriting-based languages
  - MSR, Petri nets (concurrency)
  - CHR (general purpose)
  - Maude (specification and reasoning)
  - CoMingle (distributed mobile programming)
- Process algebras
  - π-calculus (specification and reasoning)
  - Strand spaces (protocol verification)
- Others
  - jQuery (web programming)
  - Actor model
  - Agent paradigm, ...

An Increasingly Popular Paradigm

- Appeal
  - Declarative
  - Native support for concurrency
  - Concise specifications
  - Logic-based
  - Potential for effective reasoning
- But limited support for programming-in-the-large
  - Team-based development
  - Hierarchical components with clear interfaces
  - Swappable implementations
  - Code reuse
  - Separate compilation

An Increasingly Popular Paradigm

- Appeal
  - Declarative
  - Native support for concurrency
  - Concise specifications
  - Logic-based
  - Potential for effective reasoning
- But limited support for programming-in-the-large
  - Team-based development
  - Hierarchical components with clear interfaces
  - Swappable implementations
  - Code reuse
  - Separate compilation

Modularity tames complexity

The Challenges of Writing Large Rule-based Programs

- Flat name space
  - No protection against reusing a name
  - No shadowing
  - No local definitions or private names
- Proactive semantics
  - Rule fires once applicable
  - No explicit “calls”
- Pittfalls of concurrency
  - Race conditions
  - Deadlocks, livelocks, etc

Writing correct code of even moderate size is difficult

Modularity in Logic Languages

- Mainly for backward-logic programming languages
  - Exceptions: Maude, MSR2
- Largely ad-hoc
  - Module layer on top of a logic language
  - Provided in commercial languages (e.g., SWI and SICStus Prolog)
- A few logic-based proposals
  - Use of embedded implication in λProlog [2]

$$p(x) \leftarrow (M \rightarrow q(x))$$

    Clauses in M available to prove body atom $q(x)$
  - Use of second-order quantification [1]

$$\forall sorted.\, \exists le.\, sorted\ [x] \wedge (sorted\ [x, y|z] \rightarrow le(x, y) \wedge sorted\ [y|z])$$

Modularity in Logic Languages

- Mainly for backward-logic programming languages
  - Exceptions: Maude, MSR2
- Largely ad-hoc
  - Module layer on top of a logic language
  - Provided in commercial languages (e.g., SWI and SICStus Prolog)
- A few logic-based proposals
  - Use of embedded implication in λProlog [2]

$$p(x) \leftarrow (M \rightarrow q(x))$$

    Clauses in M available to prove body atom $q(x)$
  - Use of second-order quantification [1]

$$\forall sorted.\, \exists le.\, sorted\ [x] \wedge (sorted\ [x, y|z] \rightarrow le(x, y) \wedge sorted\ [y|z])$$

Outline

$\mathcal{L}^1$, a Prototypical Rule-based Language

$$
\begin{array}{rlll}
\textit{First-order terms} & t & ::= & x \mid \cdots \\
\textit{Atoms} & A & ::= & p\ t \\
\textit{LHS} & L & ::= & \cdot \mid A, L \\
\textit{Rules} & R & ::= & L \multimap P \mid \forall x.\, R \\
\textit{Programs} & P & ::= & \cdot \mid P, P \mid A \mid R \mid\ !R \mid \exists x.\, P
\end{array}
$$

Rules have the form

$$\forall \vec{x}.\, L \multimap P$$

Safety requirement:

- free variables in P occur in enclosing LHS

Examples

- Adding two numbers

$$\begin{array}{rcl} !\forall x.\,\forall y.\ \mathsf{add}(\mathsf{s}(x), y) & \multimap & \mathsf{add}(x, \mathsf{s}(y)) \\ \forall m.\,\forall n.\quad \cdots & \multimap & \cdots, \mathsf{add}(n, m) \\ \forall r.\ \mathsf{add}(\mathsf{z}, r), \cdots & \multimap & \cdots \end{array}$$

- Adding two numbers — nested rules

$$\forall m.\,\forall n.\ \cdots \quad \multimap \quad \left[\begin{array}{rcl} !\forall x.\,\forall y.\ \mathsf{add}(\mathsf{s}(x), y) & \multimap & \mathsf{add}(x, \mathsf{s}(y)), \\ \cdots, \mathsf{add}(n, m), & & \\ \forall r.\ \mathsf{add}(\mathsf{z}, r), \cdots & \multimap & \cdots \end{array}\right]$$

Examples

- Queue as a linked list

$$\cdot \quad \multimap \quad \exists d. \left[\begin{array}{l} \mathsf{head}(d), \\ \mathsf{tail}(d) \end{array}\right]$$

$$!\forall e.\, \forall d. \quad \left[\begin{array}{l} \mathsf{enq}(e), \\ \mathsf{head}(d) \end{array}\right] \quad \multimap \quad \exists d'. \left[\begin{array}{l} \mathsf{data}(e, d', d), \\ \mathsf{head}(d') \end{array}\right]$$

$$!\forall e.\, \forall d.\, \forall d'. \quad \left[\begin{array}{r} \mathsf{deq\_req}, \\ \mathsf{tail}(d'), \\ \mathsf{data}(e, d, d') \end{array}\right] \quad \multimap \quad \left[\begin{array}{l} \mathsf{deq}(e), \\ \mathsf{tail}(d) \end{array}\right]$$

Compositional Semantics

$$\textit{Signatures} \quad \Sigma \quad ::= \quad \cdot \mid \Sigma, f \mid \Sigma, p$$

State: $\Sigma.\,\Pi$ with Π *ground* program

$$\boxed{\Sigma.\,\Pi \mapsto \Sigma'.\,\Pi'}$$

$$\begin{array}{rcll}
\Sigma.\,(\Pi, L, L \multimap P) & \mapsto & \Sigma.\,(\Pi, P) & \\
\Sigma.\,(\Pi, \forall x.\,R) & \mapsto & \Sigma.\,(\Pi, [t/x]R) & \text{if } \cdot \vdash_\Sigma t : \iota \\
\Sigma.\,(\Pi, !R) & \mapsto & \Sigma.\,(\Pi, !R, R) & \\
\Sigma.\,(\Pi, \exists x.\,P) & \mapsto & (\Sigma, x).\,(\Pi, P) &
\end{array}$$

Atomic Semantics

$\Sigma.\,\Pi$ with no top-level $\exists$ is *stable*

$$\boxed{\begin{array}{l}\Sigma.\,\Pi \Mapsto \Sigma'.\,\Pi' \\ \Sigma.\,\Pi \Mapsto \Sigma'.\,\Pi'\end{array}}$$

$$\begin{array}{rcl}\Sigma.\,(\Pi, \exists x.\,P) & \Mapsto & (\Sigma, x).\,(\Pi, P) \\ \Sigma.\,\Pi, L\theta, \forall(L \multimap P) & \Mapsto & \Sigma.\,(\Pi, P\theta) \\ \Sigma.\,\Pi, L\theta, !\forall(L \multimap P) & \Mapsto & \Sigma.\,(\Pi, !\forall(L \multimap P), P\theta)\end{array}$$

Theorem (Soundness)

1. *If* $\Sigma.\,\Pi \Mapsto \Sigma'.\,\Pi'$, *then* $\Sigma.\,\Pi \mapsto \Sigma'.\,\Pi'$
2. *If* $\Sigma.\,\Pi \Mapsto \Sigma'.\,\Pi'$, *then* $\Sigma.\,\Pi \mapsto^* \Sigma'.\,\Pi'$

Relation to other Languages

- Classical multiset rewriting

$$!\forall(L \multimap L')$$

- Generative multiset rewriting

$$!\forall\vec{x}.\,(L \multimap \exists\vec{y}.\,L')$$

- Asynchronous π-calculus

$$\nu c.\,(x(m).\,(\bar{c}\langle m\rangle \parallel \bar{y}\langle c\rangle))$$

$$\rightsquigarrow \quad \exists c.\,\forall m.\,\underline{\mathrm{ch}}(x,m) \multimap [\underline{\mathrm{ch}}(c,m), \underline{\mathrm{ch}}(y,c)]$$

- Datalog

$$!\forall\vec{x}.\,(L \multimap !A)$$

Logical Foundations — Compositional Semantics

$$
\begin{array}{rcl}
\cdot & \leftrightsquigarrow & \mathbf{1} \\
A, I & \leftrightsquigarrow & \varphi \otimes \varphi \\
I \multimap P & \leftrightsquigarrow & \varphi \multimap \varphi \\
\forall x.\, R & \leftrightsquigarrow & \forall x.\, \varphi \\
\exists x.\, P & \leftrightsquigarrow & \exists x.\, \varphi \\
!R & \leftrightsquigarrow & !\varphi
\end{array}
$$

$$
\Sigma.\,(\Pi, \exists x.\, P) \mapsto (\Sigma, x).\,(\Pi, P) \quad \leftrightsquigarrow \quad \frac{\Delta, \varphi \longrightarrow_{\Sigma, x} \psi}{\Delta, \exists x.\, \varphi \longrightarrow_{\Sigma} \psi}\ \exists \mathsf{L}
$$

Theorem

If $\Sigma.\,\Pi \mapsto \Sigma'.\,\Pi'$, *then* $\ulcorner \Pi \urcorner \longrightarrow_{\Sigma} \exists \Sigma'.\,\ulcorner \Pi' \urcorner$

Logical Foundations — Atomic Semantics

$$\Sigma.\,\Pi, L\theta, \forall(L \multimap P) \Mapsto \Sigma.\,(\Pi, P\theta) \qquad \rightsquigarrow$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots \quad \overline{A\theta \Longrightarrow_\Sigma \boxed{A\theta}} \quad \cdots}{\cdots} \quad \Delta_2 \Longrightarrow_\Sigma \boxed{\varphi_L\theta} \qquad \dfrac{\Delta_1, \varphi_P\theta \Longrightarrow_\Sigma \psi}{\Delta_1, \boxed{\varphi_P\theta} \Longrightarrow_\Sigma \psi}\,\text{blurL}}{\Delta_1, \Delta_2, \boxed{\varphi_L\theta \multimap \varphi_P\theta} \Longrightarrow_\Sigma \psi}\,\multimap\text{L}}{\Delta_1, \Delta_2, \boxed{\forall(\varphi_L \multimap \varphi_P)} \Longrightarrow_\Sigma \psi}\,\forall\text{L(repeated)}}{\Delta_1, \Delta_2, \forall(\varphi_L \multimap \varphi_P) \Longrightarrow_\Sigma \psi}\,\text{focusL}$$

Theorem

1. *If* $\Sigma.\,\Pi \Mapsto \Sigma'.\,\Pi'$, *then* $\ulcorner\Pi\urcorner \Longrightarrow_\Sigma \exists\Sigma'.\ulcorner\Pi'\urcorner$
2. *If* $\Sigma.\,\Pi \Mapsto \Sigma'.\,\Pi'$, *then* $\ulcorner\Pi\urcorner \Longrightarrow_\Sigma \exists\Sigma'.\ulcorner\Pi'\urcorner$

$\mathcal{L}^{1.5}$ — a Mild Second-order Extension

$$
\begin{array}{rrcl}
\textit{Terms} & t & ::= & x \mid \cdots \mid X \mid p \\
\textit{Atoms} & A & ::= & p\ t \mid X\ t \\
\textit{LHS} & L & ::= & \cdot \mid A, L \\
\textit{Rules} & R & ::= & L \multimap P \mid \forall x.\, R \mid \forall X.\, R \\
\textit{Programs} & P & ::= & \cdot \mid P, P \mid A \mid R \mid\ !R \mid \exists x.\, P \mid \exists X.\, P
\end{array}
$$

Safety requirement:

- if $X\ t$ on LHS, then X must occur in earlier LHS term

Avoids

$$!\forall X.\, \forall x.\ \ X\ x \multimap \cdot$$

but accepts

$$!\forall X.\, \forall x.\ \ \mathsf{delete\_all}(X), X\ x \multimap \cdot$$

$\mathcal{L}^{1.5}$ — Semantics

- Compositional semantics

$$\Sigma.\,(\Pi, \forall X.\,R) \overset{\cdots}{\mapsto} \Sigma.\,[p/X]R \qquad \text{if } p \text{ in } \Sigma$$

- Atomic semantics

$$\Sigma.\,(\Pi, \exists X.\,P) \overset{\cdots}{\Mapsto} (\Sigma, X).\,(\Pi, P)$$

Compiles to $\mathcal{L}^{1}$

Outline

Example: Information Hiding

Adding two numbers

$$!\forall x.\, \forall y.\ \mathsf{add}(\mathsf{s}(x), y) \multimap \mathsf{add}(x, \mathsf{s}(y))$$

$$\begin{array}{rcl} \forall m.\, \forall n. \quad \cdots & \multimap & \cdots, \mathsf{add}(n, m) \\ \forall r.\ \mathsf{add}(\mathsf{z}, r), \cdots & \multimap & \cdots \end{array}$$

Example: Information Hiding

Adding two numbers

$$!\forall x.\,\forall y.\ \mathsf{add}(\mathsf{s}(x), y) \multimap \mathsf{add}(x, \mathsf{s}(y)) \qquad \text{“module code”}$$

$$\begin{array}{rll} \forall m.\,\forall n. & \cdots & \multimap \cdots, \mathsf{add}(n, m) \\ \forall r. & \mathsf{add}(\mathsf{z}, r), \cdots & \multimap \cdots \end{array} \qquad \text{“client code”}$$

Example: Information Hiding

Adding two numbers

$$!\forall x.\forall y.\ \mathsf{add}(\mathsf{s}(x), y) \multimap \mathsf{add}(x, \mathsf{s}(y)) \qquad \text{“module code”}$$

$$\begin{array}{rll} \forall m.\forall n. \quad \cdots & \multimap & \cdots, \mathsf{add}(n, m) \\ \forall r.\ \mathsf{add}(\mathsf{z}, r), \cdots & \multimap & \cdots \end{array} \qquad \text{“client code”}$$

Does not work with more than one client

Example: Information Hiding

Adding two numbers

$$!\forall x.\forall y.\ \mathsf{add}(\mathsf{s}(x), y) \multimap \mathsf{add}(x, \mathsf{s}(y)) \qquad \text{“module code”}$$

$$\begin{array}{rcl} \forall m.\forall n. \quad \cdots & \multimap & \cdots, \mathsf{add}(n, m) \\ \forall r.\ \mathsf{add}(\mathsf{z}, r), \cdots & \multimap & \cdots \end{array} \qquad \text{“client code”}$$

Does not work with more than one client

- whose result is $\mathsf{add}(\mathsf{z}, r)$??

Example: Information Hiding

Solution: generate the add predicate dynamically

$$\forall m.\, \forall n.\ \cdots \multimap \exists add.\ \left[\begin{array}{lll} !\forall x.\, \forall y.\ add(\mathsf{s}(x), y) & \multimap & add(x, \mathsf{s}(y)), \\ \cdots, add(n, m), & & \\ \quad \forall r.\ add(\mathsf{z}, r), \cdots & \multimap & \cdots \end{array}\right]$$

Safety requirement prevents interception

Example: Information Hiding

Solution: generate the add predicate dynamically

$$\forall m.\, \forall n.\ \cdots \multimap \exists add.\ \left[\begin{array}{lll} !\forall x.\, \forall y.\ add(\mathsf{s}(x), y) & \multimap & add(x, \mathsf{s}(y)), \\ \cdots, add(n, m), & & \\ \quad \forall r.\ add(\mathsf{z}, r), \cdots & \multimap & \cdots \end{array}\right]$$

Safety requirement prevents interception

Does not look like modular code

Example: Information Hiding

Solution: generate the add predicate dynamically

$$\forall m.\, \forall n.\ \cdots \multimap \exists add.\ \left[\begin{array}{lll} !\forall x.\, \forall y.\ add(\mathsf{s}(x), y) & \multimap & add(x, \mathsf{s}(y)), \\ \cdots, add(n, m), & & \\ \quad \forall r.\ add(\mathsf{z}, r), \cdots & \multimap & \cdots \end{array}\right]$$

Safety requirement prevents interception

Does not look like modular code

Can we do better?

Example: Information Hiding

- Pull it out
- Give it a public name adder
- Pass private predicate

$$!\forall add.\ \mathrm{adder}(add) \multimap \left[\ !\forall x.\forall y.\ add(\mathsf{s}(x), y) \multimap add(x, \mathsf{s}(y))\ \right]$$

$$\forall m.\forall n.\ \cdots \multimap \exists add.\left[\begin{array}{l} \mathrm{adder}(add), \\ \cdots, add(n, m), \\ \forall r.\ add(\mathsf{z}, r), \cdots \multimap \cdots \end{array}\right]$$

Example: Information Hiding

- Pull it out
- Give it a public name adder
- Pass private predicate

$$!\forall add.\ \mathrm{adder}(add) \multimap \left[\ !\forall x.\forall y.\ add(\mathsf{s}(x), y) \multimap add(x, \mathsf{s}(y))\ \right]$$

$$\forall m.\forall n.\ \cdots \multimap \exists add.\ \left[\begin{array}{l} \mathrm{adder}(add), \\ \cdots, add(n, m), \\ \forall r.\ add(\mathsf{z}, r), \cdots \multimap \cdots \end{array}\right]$$

- Next, provide syntactic sugar

Example: Information Hiding

Module:

module adder
  provide $add : \text{nat} \times \text{nat} \to o$
  $!\forall m.\, \forall n.\ add(\mathsf{s}(m), n) \multimap add(m, \mathsf{s}(n))$
end

Client code:

$$\forall m.\, \forall n.\ \cdots \multimap A \textbf{ as adder.} \left[\begin{array}{l} \cdots, A.add(n, m), \\ \forall r.\ A.add(\mathsf{z}, r), \cdots \multimap \cdots \end{array} \right]$$

Example: Capabilities

$$
\begin{array}{l}
\textbf{module}\ \text{adder}' \\
\quad \textbf{provide} \quad \textbf{out} \quad \mathit{add\_req} \quad : \text{nat} \times \text{nat} \to o \\
\qquad\qquad\quad \textbf{in} \quad \mathit{add\_res} \quad : \text{nat} \to o \\
\qquad !\forall x.\, \forall y.\ \mathit{add\_req}(\text{s}(x), y) \quad \multimap \quad \mathit{add\_req}(x, \text{s}(y)) \\
\qquad\quad \forall z.\ \ \mathit{add\_req}(\text{z}, z) \quad \multimap \quad \mathit{add\_res}(z) \\
\textbf{end}
\end{array}
$$

and the client code assumes the concrete form:

$$
\forall m.\, \forall n.\ \cdots \quad \multimap \quad A\ \textbf{as}\ \text{adder}'.\ \left[\begin{array}{l} \cdots, A.\mathit{add\_req}(n, m), \\ \forall r.\ A.\mathit{add\_res}(r), \cdots \end{array} \quad \multimap \quad \cdots \right]
$$

- **out**: request only on RHS
- **in**: result only on LHS

Abstract Data Types

interface QUEUE
  out $enq : \mathsf{nat} \to o$
  out $deq\_req : o$ **in** $deq : \mathsf{nat} \to o$
end

module queue
  provide QUEUE
  local $head : \iota \to o$ $tail : \iota \to o$ $data : \mathsf{nat} \times \iota \times \iota \to o$

$$\cdot \multimap \exists d.\, head(d), tail(d)$$

$$!\forall e.\forall d.\ enq(e), head(d) \multimap \exists d'.\, data(e, d', d), head(d')$$

$$!\forall e.\forall d.\forall d'.\ \begin{bmatrix} deq\_req, \\ tail(d'), \\ data(e, d, d') \end{bmatrix} \multimap deq(e), tail(d)$$

end

Sharing Private Names

$$
\begin{array}{l}
\textbf{module}\ \mathsf{cell}\ (v : \mathsf{nat}) \\
\quad \textbf{provide}\ \ \textbf{out}\ \ \mathit{get}\ : o \qquad\qquad \textbf{in}\ \ \mathit{got}\ : \mathsf{nat} \to o \\
\qquad\qquad\ \ \textbf{out}\ \ \mathit{set}\ : \mathsf{nat} \to o \\
\quad \textbf{local}\ \ \mathit{content}\ : \mathsf{nat} \to o \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdot \qquad \multimap \quad \mathit{content}(v) \\
\qquad\qquad\qquad\qquad\qquad\quad !\forall v.\ \ \mathit{get}, \mathit{content}(v) \qquad \multimap \quad \mathit{got}(v), \mathit{content}(v) \\
\qquad\qquad\qquad\qquad !\forall v.\, \forall v'.\ \mathit{set}(v'), \mathit{content}(v) \qquad \multimap \quad \mathit{content}(v') \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{rcl}
\cdot & \multimap & C\ \textbf{as}\ \mathsf{cell}(\mathsf{s}(\mathsf{z})).\ \left[\begin{array}{l} p(C.\mathit{set}), \\ q(C.\mathit{get}, C.\mathit{got}) \end{array}\right] \\
\forall \mathit{write}.\ \ p(\mathit{write}) & \multimap & \mathit{write}(\mathsf{z}) \\
\left[\begin{array}{r} \forall \mathit{read\_req}. \\ \forall \mathit{read}. \end{array}\right] q(\mathit{read\_req}, \mathit{read}) & \multimap & \left[\begin{array}{l} \mathit{read\_req}, \\ \forall r.\ \mathit{read}(r) \ \multimap\ s(r) \end{array}\right]
\end{array}
$$

Parametric Modules

module prodcons(Q: QUEUE)
  provide **in** *produce*: nat $\to o$
  in *consume_req*: o **out** *consume*: nat $\to o$

$$\cdot \multimap B \text{ as } Q.\ \left[\begin{array}{l} !\forall e.\ \ produce(e) \multimap B.enq(e) \\ \quad !\,consume\_req \multimap \left[\begin{array}{l} B.deq\_req, \\ \forall e.\ B.deq(e) \multimap consume(e) \end{array}\right] \end{array}\right]$$

end

$$\cdot \multimap B \text{ as prodcons(queue)}.\ \left[\begin{array}{l} p_1(B.produce), p_2(B.produce), \\ c_1(B.consume\_req, B.consume) \end{array}\right]$$

$\mathcal{L}^M$ — an Extension with Modules

Extends $\mathcal{L}^{1.5}$ with

- A convenience syntax for
  - Module definitions

module $p\,(\Sigma_{par})$	Module name and parameters
provide Σ_{export}	Exported names
local Σ_{local}	Local predicates and constructors
P	Module definition
end	

  - Module instantiation: N **as** $p\ t.\ P$
  - Use of exported names $N.p$

- Static restrictions on use (**in**, **out**)

Elaboration of $\mathcal{L}^M$ into $\mathcal{L}^{1.5}$

Compiles into $\mathcal{L}^{1.5}$

- **module** $p\,(\Sigma_{par})$
  provide Σ_{export}
  local Σ_{local}
  P
  end

$$\rightsquigarrow \quad \forall \Sigma_{par}.\, \forall \Sigma_{export}.\, p\,(\Sigma^*_{par}, \Sigma^*_{export}) \multimap \exists \Sigma_{local}.\, P$$

- N **as** $p\ t.\ P \quad \rightsquigarrow \quad \exists \Sigma_{export}.\, p(t, \vec{X}), [\vec{X}/N.\vec{X}]P$

which compiles into $\mathcal{L}^1$

Outline

Looking Back

- $\mathcal{L}^1$ — an foundational rule-based language
  - at the core of many existing languages
  - clear operational semantics
  - solid logical foundations
- Module system blueprint for rule-based languages
  - Featureful
    - name space separation
    - support for abstract data types
    - parametrization (functors, possibly higher-order and recursive)
    - selective sharing
  - Compiles into $\mathcal{L}^1$
  - Broadly applicable
    - forward-chaining logic programming languages
    - rewriting-based languages
    - process algebras

Future Work

- Implementation within CoMingle
  - Distributed logic programming language
    - For programming distributed mobile applications
    - Based on decentralized multiset rewriting with comprehension
  - Available at https://github.com/sllam/comingle
  - Show your support, please STAR CoMingle GitHub repository!
- Separate compilation
  - Comingle is compiled CHR-style
  - Essential for efficiency
  - Essential for team-based programming

Questions?

Bibliography

M.A. Nait Abdallah.

Procedures in Horn-clause Programming.
In *ICLP'86*, pages 433–447. Springer LNCS 225, 1986.

Dale Miller.

A Proposal for Modules in λProlog.
In *ELP'94*, pages 206–221. Springer LNCS 798, 1994.