Implementing Type Theory in Higher Order Constraint Logic Programming

F. Guidi, C. Sacerdoti Coen, E. Tassi

University of Bologna & INRIA Sophia-Antipolis

Paris (FR), 16/12/2016

Plan of the Talk

HOCLP

HOCLP = CLP + HOLP

(Constraint Logic Programming + Higher Order Logic Programming)

Outline

Higher Order Logic Programming (HOLP)

Higher Order Logic Programming (HOLP)

- Dale Miller and Gopalan Nadathur.
  Higher-order logic programming.
  In *3rd Int. Conf. Logic Programming*, volume 225 of *LNCS*, pages 448 – 462. Springer-Verlan, 1986.

- Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov.
  Uniform proofs as a foundation for logic programming.
  Annals of Pure and Applied Logic, 51, 1991.

- Dale Miller and Gopalan Nadathur.
  Programming with Higher-Order Logic.
  Cambridge University Press, 1st edition, 2012.

Higher Order Logic Programming (HOLP)

HOLP (1st characterization)

Maximal fragment of (polarized) intuitionistic natural deduction complete w.r.t. uniform proofs.

HOLP (2nd characterization)

Minimal extension of FOLP (PROLOG) to reason inductively over syntax with binders.

Catherine Belleannée, Pascal Brisset, and Olivier Ridoux.
A Pragmatic Reconstruction of Lambda-Prolog.
In *Journal of Logic Programming*, 1998.

This talk: the above is true only when the terms are ground.

The Hello-World of HOLP (λProlog)

Representation of Simply Typed λ-calculus

```
kind term,typ type.

type arr typ -> typ -> typ.

type app term -> term -> term.
type lam (term -> term) -> term.
```

Example: $\lambda x.\lambda y.yxx$

```
lam x \ lam y \ app (app y x) x
```

Properties for free

- α-equivalence
- capture avoiding substitution
- displacing

Requirements on the language

- higher order unification under a mixed prefix
- efficient representation of variables and scopes

The Hello-World of HOLP (λProlog)

Type-Checking for Simply Typed λ-calculus

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \qquad \frac{\Gamma, x : A \vdash F\ x : B}{\Gamma \vdash \lambda x.F\ x : A \to B} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

Type-Checking/Inference in λProlog

```
type of term -> typ -> o.

of (app M N) B :- of M (arr A B), of N A.
of (lam F) (arr A B) :-
  pi x\ of x A => of (F x) B.
```

Requirements on the language

- $\forall$ (`pi`) in queries
  generation of fresh names
- $\Rightarrow$ in queries
  to avoid passing Γ around

The Hello-World of HOLP (λProlog)

Execution on ground term

```
   { ⊢ of (lam x \ lam y \ app y x) A }
A ← (arr B C)
↦ { of x B ⊢ of (lam y \ app y x) C }
C ← (arr D E)
↦ { of x A, of y D ⊢ of (app y x) E }
↦ { of x A, of y D ⊢ of y (arr F E),
     of x A, of y D ⊢ of x F }
D ← (arr F E), F ← A,
↦ ∅
```

Application Domains

Ground terms with binders

- formulae
- programming language syntax
- dependent types

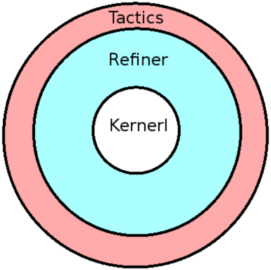
HOLP

- interpreters, compilers
- type and certificate/proof checkers
- animated operational semantics
- hypothetical reasoning

Is it fast enough?

Outline

Coq, Agda, Matita, ...



The Kernel

- Works on ground terms/proofs
- Syntax directed judgements + heuristics to speed up
  - $\Gamma \vdash t : T$
  - $\Gamma \vdash t_1 \equiv t_2$
  - $\Gamma \vdash t \triangleright t'$
- Reduction/conversion via reduction machines
- Needs to be FAST

Example: a reduction machine in λProlog/ELPI

$(\mathcal{E}, MN, \mathcal{S}) \longrightarrow$
$(\mathcal{E}, M, [N|\mathcal{S}])$

```
whd1 (app M N) S K :-
  K [] M [N|S].
```

$(\mathcal{E}, \lambda x.F, [N|\mathcal{S}]) \longrightarrow$
$(\mathcal{E}[x \mapsto N], F, \mathcal{S})$

```
whd1 (lam T F) [N|NS] K :-
  pi x \ val x T N _NF =>
    K [x] (F x) NS.
```

Observations

1. "$\Rightarrow$" is logically scoped: continuations "`K`" required (CPS style); "`[x]`" to read-back the machine state at the end
2. Teyjus is too restrictive about CPS: use ELPI
3. Call-by-need: `_NF` will be instantiated on-demand with the normal form of `N`

Evaluation

Achievements

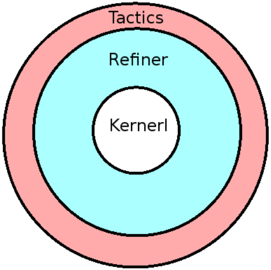
A kernel that is almost equivalent to the one of Matita 0.9 but way more readable.

Comparison with OCaml code

- + no logic-independent clutter (De Bruijn indexes, lifting, etc.)
- + simple code, very close to pen&paper judgements
  (a kernel for Martin-Löf TT by a student of math)
- + modular:
  - + add new rules later to extend the language
  - = accumulate alternative implementations
- - slow: from 8x to 15x slower (ELPI vs interpreted OCaml)
  (Teyjus is even slower)

Outline

Coq, Agda, Matita, ...



The Elaborator (Refiner)

- Works on non-ground terms/proofs
- On ground terms: should (...) behave as the kernel
- Tons of heuristics, fragile, ever changing, obscure code
  - $\Sigma : \Gamma \vdash t : T \rightsquigarrow \Sigma' : t' : T'$
  - $\Sigma : \Gamma \vdash t_1 \approx t_2 \rightsquigarrow \Sigma'$
- Deterministic solutions to higher order unification problems
- Needs to be INTELLIGENT
- user-provided heuristics in LP style

Representing open terms

Open proofs/terms as non-ground terms

$$\frac{A, \quad A \to B \vdash \qquad\qquad A}{\vdash \qquad\qquad\qquad\qquad\qquad\qquad A \to (A \to B) \to B}$$

Representing open terms

Open proofs/terms as non-ground terms

$$\frac{x : A, f : A \to B \vdash P\,x\,f : A}{\vdash \lambda x : A.\lambda f : A \to B.P\,x\,f : A \to (A \to B) \to B}$$

Representing open terms

Open proofs/terms as non-ground terms

$$\frac{x : A, f : A \to B \vdash P\,x\,f : A}{\vdash \lambda x : A.\lambda f : A \to B.P\,x\,f : A \to (A \to B) \to B}$$

Proof progress = metavariable instantiation

$$P\,x\,f := f\,(Q\,x)$$

$$\frac{\dfrac{\dfrac{}{x : A, f : A \to B \vdash f : A \to B} \qquad x : A, f : A \to B \vdash Q\,x : A}{x : A, f : A \to B \vdash f\,(Q\,x) : A}}{\vdash \lambda x : A.\lambda f : A \to B.f\,(Q\,x) : A \to (A \to B) \to B}$$

Type checking = partial correctness

The proof is correct so far

The Hello-World of HOLP (λProlog)

Type-Checking/Inference in λProlog

```
of (app M N) B :- of M (arr A B), of N A.
of (lam F) (arr A B) :-
  pi x\ of x A => of (F x) B.
```

Divergence on non-ground terms

```
   { ⊢ of (lam x \ app W x) A }
A ← (arr B C)
↦ { of x B ⊢ of (app W x) C }
↦ { of x B ⊢ of W (arr D C), of x B ⊢ of x D }
W ← (app W1 W2), ...
↦ { of x B ⊢ of W1 (arr E (arr D C)),
  of x B ⊢ of W2 E, of x B ⊢ of x D }
W1 ← (app W11 W12), ...
...
```

The Hello-World of HOCLP (ELPI)

Expected behaviour in HOCLP

A recursive predicate on a flexible term is turned into a constraint added to a constraint store.

Example behaviour on non-ground terms

```
   { ⊢ of (lam x \ app W x) A }, ∅
A ← (arr B C)
↦ { of x B ⊢ of (app W x) C }, ∅
↦ { of x B ⊢ of W (arr D C), of x B ⊢ of x D }, ∅
↦ { of x B ⊢ of x D }, { of x B ⊢ of W (arr D C) }
D ← B
↦ ∅, { of x B ⊢ of W (arr B C) }
```

The Hello-World of HOCLP (ELPI)

Expected behaviour in HOCLP

A recursive predicate on a flexible term is turned into a constraint added to a constraint store.

Example behaviour on non-ground terms

$$
\begin{array}{l}
\quad \{\ \vdash \texttt{of (lam x \textbackslash\ app W x) A}\ \},\ \emptyset \\
\texttt{A} \leftarrow \texttt{(arr B C)} \\
\mapsto \{\ \texttt{of x B} \vdash \texttt{of (app W x) C}\ \},\ \emptyset \\
\mapsto \{\ \texttt{of x B} \vdash \texttt{of W (arr D C)},\ \texttt{of x B} \vdash \texttt{of x D}\ \},\ \emptyset \\
\mapsto \{\ \texttt{of x B} \vdash \texttt{of x D}\ \},\ \{\ \texttt{of x B} \vdash \texttt{of W (arr D C)}\ \} \\
\texttt{D} \leftarrow \texttt{B} \\
\mapsto \emptyset,\ \{\ \texttt{of x B} \vdash \texttt{of W (arr B C)}\ \}
\end{array}
$$

Non-pattern-unification problems

Teyjus/ELPI: flex/flex case also delayed

E.g. $X\ (f\ x) = Y\ x$

Outline

High-Level Syntax and Semantics

Syntax (example)

```
delay (of X T) on X.
of (app M N) B :- of M (arr A B), of N A.
of (lam F) (arr A B) :-
  pi x\ of x A => of (F x) B.
```

Semantics of `delay`

If X is flexible (e.g. $Y\ t_1\ \ldots\ t_n$), succeeds adding it to the constraint store.

Resume the constraint when a query Q instantiates X to a rigid term In case of failure, backtrack on Q.

Representing open terms

Open proofs/terms as non-ground terms

$$\frac{x : A, f : A \to B \vdash P\,x\,f : A}{\vdash \lambda x : A.\lambda f : A \to B.P\,x\,f : A \to (A \to B) \to B}$$

Representing open terms

Open proofs/terms as non-ground terms

$$\frac{x : A, f : A \to B \vdash P\,x\,f : A}{\vdash \lambda x : A.\lambda f : A \to B.P\,x\,f : A \to (A \to B) \to B}$$

Proof progress = metavariable instantiation

$$P\,x\,f := f\,(Q\,x)$$

$$\frac{\overline{x : A, f : A \to B \vdash f : A \to B} \qquad x : A, f : A \to B \vdash Q\,x : A}{x : A, f : A \to B \vdash f\,(Q\,x) : A}$$

Accumulation of constraints

One metavariable, two typing constraints

$$\frac{f : \mathbb{N} \to \mathbb{B} \to \mathbb{N} \vdash X : \mathbb{N} \qquad f : \mathbb{N} \to \mathbb{B} \to \mathbb{N} \vdash X : \mathbb{B}}{f : \mathbb{N} \to \mathbb{B} \to \mathbb{N} \vdash f\ X\ X : \mathbb{N}}$$

- When X becomes ground, the constraint is checked twice.
- The two constraints together are unsatisfiable

Constraint Programming

- Declare constraints
- Propagate constraints
  - to detect early inconsistency
  - to do forward reasoning (beyond uniform proofs!)

CHR-style HOCLP

Constraint Handling Rules (CHR)-style propagation rules:

```
constraint list_of_constraint_heads {
  S1 \ S2 > A | G <=> S3.
}
```

Example: unicity of typing

```
constraint of {
  rule (G1 ?- of X T1) \ (G2 ?- of Y T2) > X ~ Y
       | (equiv G1 G2) <=> (?- T1 = T2).
}
```

Operational semantics

1. match (via σ) S_1, S_2 against the sintactic representation of constraints in the store S (up to the alignment mode A)
2. execute the guard $G\sigma$ in a meta-interpreter
3. $S := S \setminus S_2\sigma \cup S_3\sigma$

CHR-style HOCLP

What propagation rules can do

Rules compute in a meta-intepreter on the syntax of the logic below

- Meta-context `Gamma` is a list of formulae
- Meta-meta-variables are unified as usual
- Meta-variables are frozen (can only be matched via `??`, `(?? as X)`, `(?X L)`)
- `??` and `?? as X` match flexible terms
- `(?X L)` also decomposes the flexible term into its head `X` and the list `L` of its arguments
- Instantiation of meta-variables can only be triggered when back in the interpreter

Names and Alignments

CHR-like syntax (example)

```
constraint of {
  rule (G1 ?- of X T1) \ (G2 ?- of Y T2) > X ~ Y
       | (equiv G1 G2) <=> (G1 => T1 = T2).
}
```

Alignments

- The free variables of `G1 ?- of X T1` are disjoint from those of `G2 ?- of X T2` and need to be aligned using "equivariant matching"
- Example:
  `{ (of x ℤ ?- of (X x) ℤ),`
  `(of y ℤ ?- of (X y) ℤ) }`
- Equivariant matching: NP-complete
- Solution:
  1. manual matching in the guard
  2. pre-defined alignments: `X x ~ X y` matches `x` with `y`

Towards Certified HOCLP

Constraint Handling Rules (CHR)-style propagation rules

A CHR-style rule is sound and complete w.r.t. the semantics on ground terms iff

$$S_1 \wedge G \Rightarrow (S_2 \Leftrightarrow S_3)$$

CHR-style rule soundness

Every CHR-style rule should be proved sound and correct, i.e. we need to prove that a meta-theorem holds on small steps (...)λProlog executions.

Abella: an Interactive Theorem Prover for λProlog

Andrew Gacek, Dale Miller, Gopalan Nadathur
A two-level logic approach to reasoning about computations.
In *Journal of Automated Reasoning*, 49:241–273, 2012.

Towards Certified HOCLP

Is Abella the right tool?

Constraint propagation rules	=	meta-level computation
Abella	=	meta-level reasoning

- meta-meta-meta-... level computations are possible but Abella's reasoning logic $\neq$ Abella's object logic
- meta-level computations on intermediate execution states but those are invisible to Abella's big step semantics

Future work: a small-step Abella

Immediate Syntax

Frequent case (especially for heuristics)

A query Q is delayed to be immediately propagated by unary rules of the form $\emptyset \setminus Q \mid G \Leftrightarrow Q'$

Costly: delay, quote, match, start meta-runtime, execute G, unquote Q', stop meta-runtime

Immediate syntax

Immediate syntax to apply the rule before delaying the goal.

- Example: `mode (of i o).`
- Semantics: use matching (i) on the first argument of `of`
- Match and propagate flexible terms via `??`, `(?? as X)`, ...
- Delay flexible terms explicitely if not propagated

Semantically, it can be translated to CHR-style rules.

Immediate Syntax

Immediate syntax example: narrowing

```
mode (comp i i i i i).

% Case (T1 a1 ...) ≈ m2
% Solution: T1 :- λ x. F
% β-step triggered before recursion
comp (?? as T1) [A|AS] M T2 L2 :-
 of A TYA,
 T1 = lam TYA F,
 pi x  val x TYA A _NF => comp (F x) AS M T2 L2.

% Heuristic: try PROJECTION first
% Case V1 ≈ m2
% V1 := n-th argument X of the application
comp (?? as V1) [] M T2 S2 :-
 val X _ _ _,
 X = V1,
 comp V1 [] M T2 S2, !.
```

Outline

Higher Order Constraint Logic Programming (HOCLP)

HOCLP (1st characterization)

HOCLP = HOLP + Constraint Handling Rules (CHR)
Or how to exploit forward reasoning to reduce the search space

HOCLP (2nd characterization)

Minimal extension of HOLP to handle non-ground terms.

Application Domains

Ground terms with binders

- formulae
- programming language syntax
- dependent types

Non-ground terms with binders

- partial terms (user input)
- partial (dependent) types
- partial proofs

HOLP

- interpreters, compilers
- type and certificate/proof checkers
- animated operational semantics
- hypothetical reasoning

HOCLP

- type inference algorithm
- interactive theorem provers
- ???

Higher Order Constraint Logic Programming (HOCLP)

HOCLP (3rd characterization)

The best high-level language to implement interactive theorem provers for dependently typed languages (Type Theory).

Recipe for a certified modular elaborator (work in progress)

1. declare **delay**/**modes** for recursive predicates
2. accumulate the kernel (FULL REUSE, NO CODE DUPLICATION!)
3. add (immediate or not) propagation rules
4. prove all propagation rules to be sound (and some complete too)
5. let the user tamper the heuristics with his own additional rules
6. run some static analysis on the user augmented code

Outline

Advertisement

Open Post-Doc Position in Bologna

Looking for a 1 year Post-Doc on one of the following topics

- implementation and optimization of HOCLP
- semantics of HOCLP
- static analysis of HOCLP code
- formal verification of HOCLP propagation rules
- implementation of type theory in HOCLP

Contact: `claudio.sacerdoticoen@unibo.it`