

Elimination of Negation in a Logical Framework

Alberto Momigliano

Department of Philosophy
Carnegie Mellon University

Thesis Defense

Thesis Committee:

Frank Pfenning, Chair

Dana Scott

Dale Miller, Pennsylvania State University

Outline

1. Logical Frameworks
2. Adding Negation
3. Negation-as-Failure vs. Elimination of Negation
4. Elimination of Negation in a Logical Framework
 - (a) Clause Complement
 - (b) Term Complement
5. Contributions and Future Work

1. Logical frameworks:

A brief summary of their features and use.

Deductive Systems

- **Deductive System:** Calculus of axioms and inference rules which define derivable judgments. Used in the presentation of logics and aspects of programming languages such as:
 - Type systems.
 - Operational semantics.
 - Abstract machines and compilation.
 - Algorithms (e.g. type inference, unification, theorem proving).

Logical Frameworks

- **Logical Framework:** Meta-language for the specification, implementation and verification of (the meta-theory of) deductive systems.
 - Reasoning **within** a specified system (operational semantics of Mini-ML).
 - Reasoning **about** a deductive system (proof of type preservation).
- **Formalism:** hereditary Harrop formulae (HHF), dependently typed λ -calculi (LF), labeled deductive systems, inductive definitions, rewrite logics. . .
- We work in a fragment of HHF, but our design decisions are influenced by LF.

Logical Framework Design

- Logical frameworks are **intentionally weak** by design to facilitate:
 - Informal proofs of adequacy of representation.
 - Effective checking of the validity of derivations.
 - Proof-search and unification.
- **Extending** a logical framework: need tools to make specifications concise, but balance benefits against complications in its meta-theory.
- (Conservative) extensions: sub-typing, modularity, equational reasoning, linearity. . . What about having negation?

2. Adding Negation:

Why negation is a useful notion in specifications.

- Formal definition of a programming language in the style of natural semantics: Progress lemma (well-typed expressions cannot go wrong):
 - if $e : \tau$ and e is **not** a value, there exists e' such that $e \mapsto e'$
- Process algebra: *safety* in an algorithm for mutual exclusion: for any transition, if the initial state is *safe*, so is the final one.
 - A state is *safe* if processes are **not** in their critical section at the same time.

- Type inference: an expression is **not** well-typed.
- Functional program analysis: a function is **not** linear.
- Operational semantics: an expression is **not** in normal form.
- Algorithmic sub-typing: a sort is **not** a subsort (of another).
- Prolog style: two elements are **different**, an element is **not** a member of a list.

Negation in Logical Frameworks

- Logical frameworks with logic programming interpretation have no (declarative) negation operator, to preserve goal-oriented proof search.
- Need to explicitly program the negative code: for example, `define non-value(e)`. Tedious and error-prone.
- Need to **prove** that this code implements the negation of the intended predicate.
- **Our contribution:** automation of this synthesis of negative information, proven correct once and for all.

3. Negation-as-Failure vs. Elimination of Negation:

- Problems with NF
- Elimination of Negation in the Horn case
- Benefits of Elimination of Negation

- *NF* (Clark [1978]) is the usual answer in LP:

\mathcal{P} infers $\neg A$ if A finitely fails from \mathcal{P}

- Not declarative, but easy to implement (no change to the interpreter).
- Possibly unsound and generally incomplete: semantics?
- **Closed World Assumption:** Horn Logic \sim inductive definitions: what is not provable is assumed to be false.
- Allowing negation in clauses may spoil their (monotone) inductive nature.

- Even in a well-behaved fragment, NF in a logical framework makes the meta-theory much harder (provability and unprovability).
- Logical frameworks such as HHF cannot in general be seen as (monotone) inductive definitions, because they allow hypothetical and parametric judgments:

$$Q \leftarrow (D \rightarrow G) \quad Q \leftarrow \forall x : A. G$$

- **Open World Assumption:** programs and signature are open-ended.
- NF and embedded implication is particularly problematic (Gabbay [1985]): failure of cut-elimination.

Related Work

- Strict distinction between CWA and OWA predicates; *NF* for the former, minimal negation for the latter (Harland [1991]).
- A modal approach to account **arbitrary extensions** of the program as possible worlds (Olivetti & Giordano [1998] for N-Prolog, Bonner [1994] for Hypothetical Datalog).
- Embrace partiality of inductive definitions and use *definitional reflection* to obtain closure (Schroeder-Heister [1994], McDowell & Miller [1997]).
- Unsatisfactory (for our needs): need to negate OWA predicates, while not extending the language; PID's technically inadequate for logical frameworks.

- Our approach is **transformational** (Sato & Tamaki [1984] Barbuti et al. [1990] for first-order Horn logic and CLP's):
 - Given a clause $Q \leftarrow G \wedge \neg p(\overline{t_n}) \wedge G'$
 - a definition of $p(\overline{x_n})$

\Rightarrow **synthesize** a positive definition of non_p , such that for all ground terms $\overline{s_n}$:

$$\not\models p(\overline{s_n}) \quad \text{iff} \quad \vdash non_p(\overline{s_n})$$

- By replacing $\neg p$ with non_p , obtain the **negation-less** clause:

$$Q \leftarrow G \wedge non_p(\overline{t_n}) \wedge G'$$

- Use classical transformation, such as contraposition, to bring the (completion of the) program to a sort of negation normal form.
- Use an algorithm to complement first-order terms to negate atoms.

- Sketch: the negating the definition of `even`:

$$\text{Not}_D(\text{even}(0) \wedge \forall Y : \text{nat}. \text{even}(s(s(Y))) \leftarrow \text{even}(Y))$$

requires the use of term complement $\text{Not}(0)$:

$$\text{Not}_D(\text{even}(0)) = \bigwedge_{M \in \text{Not}(0)} \neg \text{even}(M) = \forall X : \text{nat}. \neg \text{even}(s(X))$$

- Compile-time source-to-source transformation: no run-time overhead.
- No need to extend the language; meta-theory unaltered, since negative information are 'positivized'.
- No new semantics problems, no incompleteness.
- Proof terms provide evidence for (positivized) negative information.
- User level: adequacy theorem(s) unaltered.

- Unfortunately, elimination of negation does not scale immediately to a logical framework such as HHF:
 - Intrinsic friction between **Closed** and **Open** World Assumption.
 - Negation normal forms (and completion-based approaches) are incompatible with the (constructive) operational semantics associated to HHF.
 - HHF terms language based on the simply-typed λ -calculus, which is **not** closed under term complement.

4. Elimination of Negation in a Logical Framework:

Part (a) Clause Complement.

- An Example
- The Regular World Assumption
- HHF Clause Complement

Example: Encoding the (untyped) λ -calculus

- The language of expressions:

$$e ::= x \mid (\lambda x. e) \mid e_1 e_2$$

- The signature:

$$exp : \text{type}$$

$$lam : (exp \rightarrow exp) \rightarrow exp$$

$$app : exp \rightarrow exp \rightarrow exp$$

- The representation function:

$$\llbracket x \rrbracket = x : exp$$

$$\llbracket \lambda x. e \rrbracket = lam (\lambda x : exp. \llbracket e \rrbracket)$$

$$\llbracket e_1 e_2 \rrbracket = app \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket$$

Example: (object-level) linear terms

- A lambda term is *linear* if every functional subterm uses each of its arguments exactly once:

$$\begin{aligned} \text{linapp} &: \text{linear}(\text{app } E_1 \ E_2) \\ &\quad \leftarrow \text{linear}(E_1) \\ &\quad \leftarrow \text{linear}(E_2). \\ \text{linlam} &: \text{linear}(\text{lam}(\lambda x. E \ x)) \\ &\quad \leftarrow \text{linx}(\lambda x. E \ x) \\ &\quad \leftarrow (\forall x : \text{exp}. \text{linear}(x) \rightarrow \text{linear}(E \ x)). \\ \text{linxx} &: \text{linx}(\lambda x. x). \\ \text{linxapp1} &: \text{linx}(\lambda x. \text{app} \ (E_1 \ x) \ E_2) \\ &\quad \leftarrow \text{linx}(\lambda x. E_1 \ x). \\ \text{linxapp2} &: \text{linx}(\lambda x. \text{app} \ E_1 \ (E_2 \ x)) \\ &\quad \leftarrow \text{linx}(\lambda x. E_2 \ x). \\ \text{linxlm} &: \text{linx}(\lambda x. \text{lam}(\lambda y. E \ x \ y)) \\ &\quad \leftarrow (\forall y : \text{exp}. \text{linx}(\lambda x. E \ x \ y)). \end{aligned}$$

Complementable Clauses

- Observation: in logical frameworks embedded implications represent **scoping constructs**, such as abstractions, quantifiers and rules with hypothetical premises.
- Implications paired with intensional universal quantifier to express the parametricity of the assumption.
- Here dynamic assumptions extend the current database in a **regular** way, so that static and dynamic clauses never **overlap** (eigenvariable condition).
- Idea: **Restrict** to programs (called **complementable**) where every assumption is parametric.

- A manifestation of the **Regular World Assumption**: we establish in advance the class of “possible worlds” which programs can end up into.
- A middle ground between programs that cannot make any extension (CWA) and those whose extensions are completely unpredictable (OWA).
- Not covered: for example, the encoding of implication introduction in natural deduction:

$$nd(A \text{ imp } B) \leftarrow (nd(A) \rightarrow nd(B)).$$

- We have shown that this restriction is both elegant and pragmatically adequate (and conservative w.r.t. Horn logic).

- An expression is **not** linear if there is some function which either does *not* use its argument or uses it *more than once*.
- An application is not linear if either the first element or the second is not linear: no new problems.

$$\neg linear : \neg linear(app\ E_1\ E_2) \\ \leftarrow \neg linear(E_1) \vee \neg linear(E_2).$$

- A lambda expression is not linear if it is not linear in its first argument or if its body is not linear: how to negate a hypothetical and parametric goal?

- We mimic a failure derivation to provide a derivation from the negative definition; it needs to mirror failure **from** assumptions.
- Assumption complementation more complicate than static clause complementation: not all information is available at compile-time.
- Restriction to parametric assumptions allows to complement *around* parameters, delaying dynamic instantiations as constraints.
- We synchronize clause, assumption and goal complementation (*augmentation*).

$$\begin{aligned} \text{linxlm} &: \text{linx}(\lambda x. \text{lam}(\lambda y. E \ x \ y)) \\ &\leftarrow (\forall y: \text{exp}. \boxed{\top_{\text{linx}}} \rightarrow \text{linx}(\lambda x. E \ x \ y)). \end{aligned}$$

- Compute $y:\text{exp} \vdash \text{Not}_\alpha(\top_{\text{linx}}) = \neg \text{linx}(\lambda x. y)$.

- Augment the clause linxlm with $y:\text{exp} \vdash \text{Not}_\alpha(\top_{\text{linx}})$:

$$\begin{aligned} \text{augD}(\text{linxlm}) &: \text{linx}(\lambda x. \text{lam}(\lambda y. E \ x \ y)) \\ &\leftarrow (\forall y: \text{exp}. \boxed{\neg \text{linx}(\lambda x. y)} \rightarrow \text{linx}(\lambda x. E \ x \ y)). \end{aligned}$$

- Negate the body:

$$\begin{aligned} \neg \text{augD}(\text{linxlm}) &: \neg \text{linx}(\lambda x. \text{lam}(\lambda y. E \ x \ y)) \\ &\leftarrow (\forall y: \text{exp}. \neg \text{linx}(\lambda x. y) \rightarrow \neg \text{linx}(\lambda x. E \ x \ y)). \end{aligned}$$

4. Elimination of Negation in a Logical Framework:

Part (b) Term Complement.

- Higher-Order Term Complement
- Strict Types
- Algebra of Simple Terms

Term Complement

- Recall: complement of the head of a clause requires **term** complement.
- A term M with free variables as the intensional representation of its *ground* instances, denoted $\|M\|$.
- *Unification* of two terms ' $N = M$ ' as *intersection* of their ground instances: $\|N\| \cap \|M\|$.
- The *complement* of M , $\text{Not}(M)$, is the set of ground terms which are not instances of M , i.e. are not in $\|M\|$.
- 'Relative Complement': all the ground instances of a given (finite) set of terms which are not instances of another given one:

$$\{M_1, \dots, M_n\} - \{N_1, \dots, N_m\}$$

Higher-Order Term Complement

- Need to extend term complement to the simply-typed λ -calculus, to compute, for example:

1. $\text{Not}_D(\text{linx}(\lambda x. \boxed{x}))$: not problematic.

2. $\text{Not}_D(\text{linx}(\lambda x. \text{app} (E_1 x) \boxed{E_2}) \leftarrow \text{linx}(\lambda x. E_1 x))$

- Note that the bound variable x does not occur in the (pattern variable) argument E_2 .

- Common technique: encoding a η -redex, where $x \notin FV(e)$:

$$\ulcorner \lambda x. (e x) \urcorner = \text{lam}(\lambda x : \text{exp}. (\boxed{\text{app } E}) x)$$

- Problem: a pattern $(\lambda x.E\ x)$ matches any term of appropriate type; $(\lambda x.E)$ matches only ground terms $(\lambda x.M)$ where M does **not depend** on x .
- The crux of the matter is whether a pattern is *fully applied*: all bound variables appearing in the binder are mentioned in the scope of a logic variable.
- In $(\lambda x.app\ (E_1\ x)\ E_2)$, the first argument $(E_1\ x)$ is fully applied, while the second one E_2 is not.
- Complement of fully applied terms is a simple extension of the first-order case, by keeping track of bound variables.

- The complement of $(\lambda x. app (E_1 x) E_2)$ contains every ground terms $(\lambda x. app (M_1 x) (M_2 x))$, such that M_2 **must** use x .
- The simply typed λ -calculus theory is not strong enough to directly represent the complement of partially applied patterns.
- Lugiez [1995] modifies the language of terms to promote constraints to first-class objects. The technical handling of those objects is awkward.
- We choose to have a notion of *strictness* in the language to express variable occurrence constraints.

- We *internalize* strictness as a type system, generalizing the language of simple types to include:
 1. *strict functions* of type $A \xrightarrow{1} B$ (which are guaranteed to depend on their argument)
 2. *invariant functions* of type $A \xrightarrow{0} B$ (which are guaranteed **not** to depend on their argument)
 3. The full function space $A \xrightarrow{\cup} B$
- Under the Howard-Curry isomorphism, this yields a relative of the implicational fragment of relevance logic **R**.

$$\text{Labels } k ::= 1 \mid 0 \mid u$$
$$\text{Types } A ::= a \mid A_1 \xrightarrow{k} A_2$$
$$\text{Terms } M ::= c \mid x \mid \lambda x^k:A. M \mid (M \ N)^k$$
$$\text{Contexts } \Gamma ::= \cdot \mid \Gamma, x:A$$

- Three labels: 1 ‘must occur’, 0 ‘must not occur’, u ‘undetermined’, three different forms of abstraction and application.
- Typing judgment: $\Gamma; \Omega; \Delta \vdash M : A$.
- Calculus is well-behaved.

Simple Terms

- We give a complement algorithm that is sound and complete for the natural embedding of the simply-typed λ -calculus:

$$\textit{Simple Terms } M ::= \lambda x^u. A.M \mid (h \ \overline{M_n^1}) \mid (E \ \overline{x_n^k})$$

- Partially applied terms are made fully applied with **vacuous** variables:

$$(\lambda x. app \ (E \ x^u) (\boxed{F \ x^0} \))$$

- Complement requires complementing the labels.

$$\begin{aligned} \text{Not}(\lambda x. app \ (E \ x^u) (\boxed{F \ x^0} \)) = \\ \{ \lambda x. x, \lambda x. lam(\lambda y. (E' \ x^u \ y^u)) \}, \\ \lambda x. app \ (H \ x^u) (\boxed{G \ x^1} \) \} \end{aligned}$$

The Algebra of Simple Terms

- We extend higher-order pattern unification to simple terms in the form of intersection.
- **Finite** sets of most general unifiers exist.
- From complement and intersection we can define any boolean operations: in particular, the *relative* complement operation, for \mathcal{M} and \mathcal{N} finite sets of terms of the same type:

$$\mathcal{M} - \mathcal{N} = \mathcal{M} \cap (\text{Not}(\mathcal{N}))$$

- The set of *finite* sets of simple terms (over a signature) as a *boolean* algebra under set union, pattern intersection and complementation and extensional identity (on sets of ground terms).

Complete definition of $\neg linear$

$$\begin{aligned}
\neg linapp1 & : \neg linear(app\ F\ G) \leftarrow \neg linear(F). \\
\neg linapp2 & : \neg linear(app\ F\ G) \leftarrow \neg linear(G). \\
\neg linlam1 & : \neg linear(lam\ \lambda x. E\ x) \leftarrow \neg linx(\lambda x. E\ x). \\
\neg linlam2 & : \neg linear(lam\ \lambda x. E\ x) \\
& \quad \leftarrow (\forall y : exp. (\neg linx(\lambda x. y) \wedge linear(y)) \\
& \quad \quad \rightarrow \neg linear(E\ y)). \\
\neg linxapp1 & : \neg linx(\lambda x. app\ (F\ x^1)\ (G\ x^1)). \\
\neg linxapp2 & : \neg linx(\lambda x. app\ (F\ x^0)\ (G\ x^0)). \\
\neg linxapp3 & : \neg linx(\lambda x. app\ (F\ x^1)\ (G\ x^0)) \leftarrow \neg linx(\lambda x. F\ x^1). \\
\neg linxapp4 & : \neg linx(\lambda x. app\ (F\ x^0)\ (G\ x^1)) \leftarrow \neg linx(\lambda x. G\ x^1). \\
\neg linxlm & : \neg linx(\lambda x. lam(\lambda y. E\ x\ y)) \leftarrow \\
& \quad (\forall y : exp. \neg linx(\lambda x. y) \rightarrow \neg linx(\lambda x. E\ x\ y)).
\end{aligned}$$

5. Contributions and Future Work

- A complement algorithm for a useful class of third-order hereditary Harrop formulae
- A relative complement algorithm for higher-order patterns internalized into a strict type theory

Conclusions

- Negation is useful in logical frameworks such as Hereditary Harrop Formulae, but it is difficult to allow.
- We have approached the problem of adding negation by a source-to-source **transformation**.
- We have found a middle ground between CWA and OWA in the Regular World Assumption.
- We have isolated a significant fragment of HHF, where clause complementation is possible.
- We have formulated a strict λ -calculus, which gives an independent contribution to the understanding of sub-structural calculi.

Future work

- Lift some of the current restrictions (no local variables).
- Extend to dependently typed λ -calculi.
- Implement this pre-compiler to languages as L_λ , and possibly to general purpose systems such as Isabelle.
- Investigate the transformational approach to negation in sub-structural calculi and languages (Lolli, Forum, Linear LF).