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

Outiline

1. Logical Frameworks
2. Adding Negation
3. Elimination of negation vs. Negation-as-Failure
4. Clause Complement
5. Term Complement
6. Future Work

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:
  - 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. Applications as a teaching and research tool include:

  – Reasoning **within** a specified system (operational semantics of Mini-ML).

  – Implementation of algorithms (evaluation, type inference).

  – Reasoning **about** a deductive system (proof of type preservation).

- We work in a fragment of hereditary Harrop formulae (HHF), but our design decisions influeneced by LF.

Logical Framework's 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.
  - Efficient implementations.
- **Extending** a logical framework: balance benefits against complications in its meta-theory.
- Extensions: sub-typing, modularity, equational reasoning, linearity... What about having negation?

Negation in Specification 1/2

- 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 Peterson's 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.

Negation in Specification 2/2

- 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.

- Determinacy of rules application: a declarative `if-then-else` statement.

Negation in Lf's

- Lf's with logic programming interpretation have no (declarative) negation operator.
- Need to explicitly program the negative code (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.

Negation-as-Failure: Horn case 1/3

- *NF* is the usual answer in logical programming:

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

- Not declarative, but easy to implement.

- Possibly unsound and generally incomplete: semantics?

- **Closed World Assumption:** Horn Logic $\sim$ inductive definitions: everything which is not provable from a program 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).

- Lf's such as HHF, LF not inductive definitions, because of embedded implication and universal quantifiers in goals:

$$D' \leftarrow (D \rightarrow G) \qquad D' \leftarrow \forall x\!:\!A.\, G$$

- **Open World Assumption:** programs and signature are open-ended.

- *NF* and embedded implication is particularly problematic (Gabbay [1985]): failure of cut-elimination.

Between CWA and OWA

- Intrinsic tension between CWA and OWA.

- Both assumptions represent too extreme poles.

- Is there is a middle ground where negation is meaningful?

- We need to identify a class of “possible worlds” which are stable under extensions of the program and signature.

- *Regular World Assumption:* extensions are enforced to be regular.

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 Lf's.

Eliminating Negation

- Our approach is **transformational** (Sato & Tamaki [1984] Barbuti et al. [1990] for Horn logic and to CLP's [1994]):

  Given a clause $Q \leftarrow G \wedge \neg p \wedge G'$
  a definition of p
  synthesize a positive definition of non_p, such that

  $$\not\vdash p \text{ iff } \vdash non\_p$$

  Replace non_p in p: $Q \leftarrow G \wedge non\_p \wedge G'$

- Technically:

  – Bring the program in negation normal form.

  – Need an algorithm to complement terms (Lassez [1987]) to negate atoms.

A Simple Example: from `even` to `odd` 1/2

- Even numbers:

$$even(0) \wedge \forall Y : nat.\, even(s(s(Y))) \leftarrow even(Y)$$

- The complement:

$$\begin{aligned}&\mathsf{Not}_{\mathsf{D}}(even(0) \wedge \forall Y : nat.\, even(s(s(Y))) \leftarrow even(Y)) = \\ &\quad \mathsf{Not}_{\mathsf{D}}(even(0)) \vee \mathsf{Not}_{\mathsf{D}}(\forall Y : nat.\, even(s(s(Y))) \leftarrow even(Y))\end{aligned}$$

- Compute complement of an atom by computing the complement of the term:

$$\mathsf{Not}_{\mathsf{D}}(even(0)) = \bigwedge_{M \in \mathsf{Not}(O)} \neg even(M) = \forall X : nat.\, \neg even(s(X))$$

A Simple Example: from `even` to `odd` 2/2

- Complement of a rule is the conjunction of the complement of the head with the complement of the body

$$\mathsf{Not}_{\mathsf{D}}(\forall Y\!:\!nat.\, even(s(s(Y))) \leftarrow even(Y)) =$$

$$\bigwedge_{M \in \mathsf{Not}(s(s(Y)))} \neg even(M) \wedge$$

$$\forall Y\!:\!nat.\, \neg even(s(s(Y))) \leftarrow \mathsf{Not}_{\mathsf{G}}(even(Y)) =$$

$$\neg even(0) \wedge \neg even(s(0)) \wedge \forall Y\!:\!nat.\, \neg even(s(s(Y))) \leftarrow \neg(even(Y))$$

- Extend to clauses in general in a ‘boolean’ way (negation normal form).

- Compile away disjunctions in clauses via unification to recover goal-oriented proof-search.

Eliminating Negation: Benefits 1/2

- 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 floundering, no incompleteness.
- Proof terms provide evidence for (positivized) negative information.
- User level: adequacy theorem(s) unaltered.

Eliminating Negation: Problems 2/2

- Unfortunately, elimination of negation does not scale immediately to a logical framework as HHF:

  - Same friction between **Closed** and **Open** World Assumption.

  - Negation normal forms 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.

Our Solution: RWA

- Observation: in Lf's 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.

- **RWA:** restrict to programs whose dynamic assumptions extend the current database in a **regular** way, so that static and dynamic clauses never **overlap** (eigenvariable condition).

- A program is **complementable** if every assumption is parametric.

- Restriction to complementable program a simple way to ensure no overlap between dynamic and static clauses; other decidable conditions possible.

- Not allowed: encoding of implication introduction in natural deduction:

$$nd(A\ imp\ B) \leftarrow (nd(A) \rightarrow nd(B)).$$

- If implication is not parametric, need to explicitly manage assumptions as lists (see McDowell& Miller [1997]).

- Our restriction has shown to be both elegant and pragmatically adequate (and conservative w.r.t. Horn logic).

Example: Encoding the (untyped) λ-calculus

- The language of expressions:

$$e \ ::= \ x \mid (\mathbf{\Lambda} x \, . \, e) \mid e_1 \ e_2$$

- The signature:

$$\begin{array}{rcl} exp & : & \text{type} \\ lam & : & (exp \to exp) \to exp \\ app & : & exp \to exp \to exp \end{array}$$

- The representation function:

$$\begin{array}{rcl} \ulcorner x \urcorner & = & x : exp \\ \ulcorner \mathbf{\Lambda} x \, . \, e \urcorner & = & lam \ (\lambda x \!:\! exp . \ulcorner e \urcorner) \\ \ulcorner e_1 \ e_2 \urcorner & = & app \ \ulcorner e_1 \urcorner \ \ulcorner e_2 \urcorner \end{array}$$

Example: Encoding the linear functions in λ-calculus

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

$$
\begin{array}{rcl}
linlam & : & linear(lam\ \lambda x\,.\,E\ x) \\
 & & \quad \leftarrow linx(\lambda x\,.\,E\ x) \\
 & & \quad \leftarrow (\forall x\!:\!exp.\, linear(x) \rightarrow linear(E\ x)). \\
linapp & : & linear(app\ E_1\ E_2) \\
 & & \quad \leftarrow linear(E_1) \\
 & & \quad \leftarrow linear(E_2). \\
linxx & : & linx(\lambda x\,.\,x). \\
linxap1 & : & linx(\lambda x\,.\,app\ (E_1\ x)\ E_2) \\
 & & \quad \leftarrow linx(\lambda x\,.\,E_1\ x). \\
linxap2 & : & linx(\lambda x\,.\,app\ E_1\ (E_2\ x)) \\
 & & \quad \leftarrow linx(\lambda x\,.\,E_2\ x). \\
linxlm & : & linx(\lambda x\,.\,lam(\lambda y.\,E\ x\ y)) \\
 & & \quad \leftarrow (\forall y\!:\!exp.\, linx(\lambda x\,.\,E\ x\ y)).
\end{array}
$$

- 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.

$$\begin{aligned} \neg linapp \;:\; & \neg linear(app\ E_1\ E_2) \\ & \qquad \leftarrow \neg linear(E_1) \vee \neg linear(E_2). \end{aligned}$$

- 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 assumption allows to complement *around* the parameter, delaying as contraints dynamic instantiations.

- We synchronize clause, assumption and goal complementation together (*augmentation*).

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

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

- Augment the clause `linxlm`:

$$
\begin{aligned}
aug_{\mathsf{D}}(linxlm) \quad : \quad & linx(\lambda x\,.\,lam(\lambda y.\, E\ x\ y)) \\
& \qquad\qquad \leftarrow (\forall y\!:\!exp.\, \neg linx(\lambda x\,.\,y) \rightarrow linx(\lambda x\,.\,E\ x\ y)).
\end{aligned}
$$

- Negate:

$$
\begin{aligned}
\neg aug_{\mathsf{D}}(linxlm) \quad : \quad & \neg linx(\lambda x\,.\,lam(\lambda y.\, E\ x\ y)) \\
& \qquad\qquad \leftarrow (\forall y\!:\!exp.\, \neg linx(\lambda x\,.\,y) \rightarrow \neg linx(\lambda x\,.\,E\ x\ y)).
\end{aligned}
$$

Negation Normal Forms

- Operational semantics of embedded implication is intuitionistic.

- No negation normal forms available in intuitionistic logic.

- Strong negation normal form (Nelson [1949]) incompatible with semantic of failure:

$$\neg(D \to G) \;=\; D \to \neg G$$

- Valid for **complementable** programs.

Part I: Term Complement

- A term t with free variables as the intensional representation of its *ground* instances, denoted $\|t\|$..
- *Unification* of two terms $t = s$ as *intersection* $t \cap s$ of the (ground instances of) s and t.
- The *complement* of t (Not(t)) is the set of ground terms which are not instances of t, i.e. are not in $\|t\|$.
- Example: take the signature of numerals:

$$\mathrm{Not}(s(0)) = \{0, s(s(0)), s(s(s(0))) \ldots\} = \{0, s(s(Y))\}$$

More in general

- Notation: a term stands for the set of its instances, ',' represents union.
- 'Multiple Complement': by De Morgan rules:

$$\begin{aligned} \mathsf{Not}\{0, s(s(Y))\} &= \mathsf{Not}(0) \cap \mathsf{Not}(s(s(Y))) \\ &= s(X) \cap \{0, s(0)\} \\ &= \{s(0)\} \end{aligned}$$

- 'Relative Complement': all the ground instances of a given (finite) set of terms which are not instances of another given one.

$$\{t_1, \ldots, t_n\} - \{u_1, \ldots, u_m\}$$

- Example:

$$\begin{aligned} s(X) - \{0, s(s(Y)))\} &= s(X) \cap \mathsf{Not}\{0, s(s(Y))\} \\ &= s(X) \cap s(0) = s(0) \end{aligned}$$

Higher-Order Term Complement

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

  1. $\mathsf{Not}_{\mathsf{D}}(linx(\lambda x\,.\,x))$: Not problematic.

  2. $\mathsf{Not}_{\mathsf{D}}(linx(\lambda x\,.\,app\ (E_1\ x)\ \boxed{E_2}) \leftarrow linx(\lambda x\,.\,E_1\ x))$

- Note that the variable x *does* not occur in the argument E_2.

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

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

Partially Applied Patterns 1/3

- 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.

- $(\lambda x\,.\,x)$ is fully applied, $\lambda x\,.\,app\ (E_1\ x)\ E_2$ is not.

- Complement of fully applied terms is a simple extension of the first-order case, by keeping track of bound variables.

Partially Applied Patterns

- The complement of $\lambda x \,.\, app\ (E_1\ x)\ E_2$ contains every ground terms $\lambda x.app\ M_1\ M_2$ 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 modifies the language of terms to promote constraints to first-class objects. The technical handling of those objects then becomes 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 $A \stackrel{1}{\to} B$ (which are guaranteed to depend on their argument)

    2. *invariant functions* of type $A \stackrel{0}{\to} B$ (which are guaranteed **not** to depend on their argument)

    3. The full function space $A \stackrel{\mathsf{u}}{\to} B$

- Under the Howard-Curry isomorphism, this yields a relative of the implicational fragment of relevance logic **R**.

Strict Types

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

- 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.

Toward Term Complement

- We have a calculus which is potentially strong enough to represent the complement of higher-order patterns.

- Complement must satisfy the boolean rules for negation:

  1. (Disjointness) It is not the case that $M \in \|N\|$ and $M \in \|\mathsf{Not}(N)\|$.

  2. (Exhaustivity) $M \in \|N\|$ or $M \in \|\mathsf{Not}(N)\|$.

- Disjointness entailed by: not the case that both $\Gamma_1; \Omega_1; (\Delta_1, x{:}C) \vdash M : A$ and $\Gamma_2; (\Omega_2, x{:}C); \Delta_2 \vdash M : A$.

- Exhaustivity does not hold in general in the presence of intuitionistic application.

Simple Terms

- Complement algorithm is sound and complete for the natural embedding of the simply-typed λ-calculus:

$$\textit{Simple Terms} \quad M \quad ::= \quad \lambda x^u{:}A.M \mid (\ldots(h\ M_1)^1 \ldots M_n)^1 \\ \mid (\ldots(E\ x_1^{k_1}) \ldots x_n^{k_n})$$

- Simple terms are closed under complement.

- Crucial property: **(tightening)**. If $(\Gamma, x{:}C); \Omega; \Delta \vdash M : A$, then $\Gamma; \Omega; (\Delta, x{:}C) \vdash M : A$ or $\Gamma; (\Omega, x{:}C); \Delta \vdash M : A$.

- Now exhaustivity holds.

The Algebra of Simple Terms

- We extend higher-order pattern unification to simple terms in the form of intersection.

- Most general unifiers exists, but may be finite.

- 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:

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

- 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).

$\neg$ `linear`

$$
\begin{array}{rcl}
\neg linapp1 & : & \neg linear(app\ E_1\ E_2) \leftarrow \neg linear(E_1). \\
\neg linapp2 & : & \neg linear(app\ E_1\ E_2) \leftarrow \neg linear(E_2). \\
\neg linlam1 & : & \neg linear(lam\ \lambda x\,.\,E\ x) \leftarrow \neg linx(\lambda x\,.\,E\ x) \\
.\neg linlam2 & : & \neg linear(lam\ \lambda x\,.\,E\ x) \\
 & & \quad \leftarrow (\forall y\colon exp.\,(\neg linx(\lambda x\,.\,y) \wedge linear(y)) \rightarrow \neg linear(E\ y)). \\
\neg linxap0 & : & \neg linx(\lambda x\,.\,app\ (E_1\ x^1)\ (E_2\ x^1)). \\
\neg linxap1 & : & \neg linx(\lambda x\,.\,app\ (E_1\ x)\ (E_2\ x^1)) \leftarrow \neg linx(\lambda x\,.\,E_1\ x). \\
\neg linxap2 & : & \neg linx(\lambda x\,.\,app\ (E_1\ x^1)\ (E_2\ x)) \leftarrow \neg linx(\lambda x\,.\,E_2\ x). \\
\neg linxap3 & : & \neg linx(\lambda x\,.\,app\ (E_1\ x)\ (E_2\ x)) \\
 & & \quad \leftarrow \neg linx(\lambda x\,.\,E_1) \wedge \neg linx(\lambda x\,.\,E_2\ x). \\
\neg linxlm & : & \neg linx(\lambda x\,.\,lam(\lambda y.\,E\ x\ y)) \leftarrow \\
 & & \quad (\forall y\colon exp.\,\neg linx(\lambda x\,.\,y) \rightarrow \neg linx(\lambda x\,.\,E\ x\ y)).
\end{array}
$$

Conclusions and Contributions

- Negation is useful in Lf's, 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 RWA.
- 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 λProlog, and possibly to general purpose systems such as Isabelle.
- Investigate the transformational approach to negation in substrucural calculi and languages (Lolli, Forum, Linear LF).