Elimination of Negation in a Logical Framework

Alberto Momigliano

Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.
mobile@cs.cmu.edu

Abstract. Logical frameworks with a logic programming interpretation such as hereditary Harrop formulae (HHF) [12] cannot express directly negative information, although negation is a useful specification tool. Since negation-as-failure does not fit well in a logical framework, especially one endowed with hypothetical and parametric judgments, we adapt the idea of *elimination* of negation introduced in [17] for Horn logic to a fragment of higher-order HHF. This entails finding a middle ground between the Closed World Assumption usually associated with negation and the Open World Assumption typical of logical frameworks; the main technical idea is to isolate a set of programs where static and dynamic clauses do not overlap.

1 Introduction

Deductive systems consist of axioms and rules defining derivable judgments; they can be used to specify logics and aspects of programming languages such as operational semantics or type systems. A *logical framework* is a meta-language for the specification, implementation and verification of deductive systems and possibly their meta-theory. A logical framework must provide tools which make encodings as simple and direct as possible. One well known example is higher-order abstract syntax, which moves renaming and substitution principles to the meta-language. Logical frameworks should be by design as weak as possible to simplify proofs of adequacy of encodings, effective checking of the validity of derivations and proof-search as well as unification. Many logical framework have been proposed in the literature (see [16] for an overview) and many extensions are also under consideration. However, we must carefully balance the benefits that any proposed extension can bring against the complications its meta-theory would incur.

This paper discusses the introduction of a logically justified notion of *negation* in logical frameworks with a logic programming interpretation such as hereditary Harrop formulae (HHF) [12] and its implementation in λProlog [15]. We intend this to form the basis for type-theoretic frameworks such as LF [9] and its implementation *Twelf* [19]. Those systems do not provide a *primitive* negation operator. Indeed, constructive logics usually implement negative information as $\neg A \equiv A \rightarrow \perp$, where $\perp$ denotes absurdity and the Duns Scoto Law is the elimination rule. Thus negative predicates have no special status; that would correspond to explicitly code negative information in a program, which is entirely consistent with the procedural interpretation of hypothetical judgments available in logical frameworks with a logic programming interpretation. However, this would not only significantly complicate goal-oriented proof search, but providing negative definitions seems to be particularly error-prone, repetitive and not particularly interesting; more importantly, in a logical framework we have also to fulfill the *proof obligation* that the proposed negative definition does behave as the complement (of its positive counterpart). Automating the synthesis of negative information has not only an immediate practical relevance in the logic programming sense, but it may also have a rather dramatic effect on the possibility of implementing deductive systems that would prove to be too unwieldy to deal with otherwise. The synthesis of the negation of predicates such as *typable, well-formed, canonical form, subsort, value* etc.–as well as Prolog-like predicates such as equality, set membership and the like–will increase the amount of meta-theory that can be formalized.

Traditionally, negation-as-failure (*NF*) [5] has been the overwhelmingly used approach in logic programming (see [2] for a recent survey): that is, infer $\neg A$ if every proof of A fails finitely. The operational nature of this rule motivates the lack of a unique semantics and some of its related troublesome features: possible unsoundness, incompleteness and floundering. Furthermore, even if we manage to isolate a well-behaved logical fragment, such as acyclic normal programs, allowing *NF* in a logical framework would make adequacy theorems more difficult to prove, as both provability and unprovability must now be considered. The situation is even further complicated when we step to frameworks with hypothetical judgments; as recognized first by Gabbay [6], the unrestricted combination of *NF* and embedded implication is particularly problematic, since it leads to the failure of basic logic principles such as cut-elimination.

The approach to negation that we adopt is *transformational*, also known as *intensional negation*, initiated in [17] and developed in Pisa [3] for Horn logic with negation. Roughly, given a clause with occurrences of negated predicates, say $Q \leftarrow G, \neg P, G'$, where P is an already defined atom, the aim is to derive a *positive* predicate, say non_P which implements the complement of P, preserving operational equivalence; then, it is merely a question of replacement, yielding the negation-less clause $Q \leftarrow G, non\_P, G'$. This has the neat effect that negation and its problems are *eliminated*, i.e. we avoid any extension to the (meta) language. Technically, we can achieve this by transforming a Horn program into negation normal form and then by negating atoms via complementing terms, a problem first addressed in [10] for first-order terms. A final issue, which we do not tackle here, is dealing with *local* variables, which, during the transformation, become (extensionally) universally quantified [1].

Unfortunately, this approach does not scale immediately to logical frameworks such as HHF, for three main reasons:

1. The simply-typed λ-calculus is not closed under term complement.
2. Negation normal forms are incompatible with the operational semantics required by HHF.
3. There is an intrinsic tension between the *Closed World Assumption* (CWA), which is associated with negation, and the *Open World Assumption* (OWA) typical of languages with embedded implication.

The first problem has been solved in [14], by introducing a *strict* λ-calculus where term complement in the simply typed λ-calculus can be embedded and performed.

The second issue is orthogonal and requires an operational notion of normal form. The third one is rooted in the fundamental difference between Horn and HHF formulae: as well known, a Horn predicate definition can be seen as an inductive definition of the same predicate. The *minimality* condition of inductive definitions excludes anything else which is not allowed by the base and step case(s). This corresponds in Horn logic to the existence of the least model and to the consistency of the CWA and its finitary approximation, the *completion* of a program [5]: every atom which is not provable from a program is assumed to be false. Languages which provide embedded implication and universal quantification are instead *open-ended* and thus require the OWA; in fact, dynamic assumptions may, at run-time, extend the current signature and program in a totally unpredictable way. This makes it in general impossible to talk about the closure of such a program. In the literature the issue has been addressed in essentially three ways:

1. By enforcing a strict distinction between CWA and OWA predicates and applying *NF* only to the former [8], where the latter would require minimal negation.
2. By switching to a modal logic, which is able to take into account *arbitrary extensions* of the program as possible worlds (see the completion construction in [7] for N-Prolog and [4] for Hypothetical Datalog).
3. By embracing the idea of *partiality* in inductive definitions and using the rule of *definitional reflection* to incorporate a proof-theoretical notion of closure analogous to the completion [11].

None of those approaches are satisfactory for our purposes: most of the predicates we want to negate are open-ended; similarly, definitional reflection is not well-behaved (for example cut is not eliminable) for that very class of programs we are interested in. Moreover, we need to express the negation of a predicate in the same language where the predicate is formulated. Our solution is to restrict the set of programs we deem deniable in a novel way, so as to enforce a *Regular Word Assumption* (RWA): we define a class of programs whose dynamic assumptions extend the current database in a specific regular way. This constitutes a reasonable middle ground between the CWA which allows no dynamic assumption but is amenable to negation and the OWA, where assumptions are totally unpredictable. The RWA is also a promising tool in the study of the meta-logical frameworks [18]. Technically, this regularity under dynamic extension is calibrated so as to ensure that static and dynamic clauses never *overlap*. This property extends to the negative program; in a sense, we maintain a distinction between static and dynamic information, but at a much finer level, i.e. *inside* the definition of a predicate. The resulting fragment is very rich, as it captures the essence of the usage of hypothetical and parametric judgments in a logical framework; namely, that they are intrinsically combined to represent *scoping* constructs in the object language. This is why we contend that this class of programs is adequate for the practice of logical frameworks.

It is clear that elimination of negation makes sense only when negation is *stratified*, i.e. the negative predicates ultimately refers (in the call graph) to a positive one. While there may be a place in logic programming for non-stratified negation, this does not seem to be the case for a logical framework. Another difference from traditional logic programming is that negation applies only to *terminating* programs;

thus it refers not to finite failure but to unprovability tout court, as we refrain from negating programs whose negation is not recursively axiomatizable. We will thus identify negation with a *complement* operation.

The rest of the paper is organized as follow: in Sect. 2 we give an informal view of the complement algorithm by means of examples, while Sect. 3 introduces the language. Section 4 describe term and clause complementation. We conclude in Sect. 5 with some remarks on future work. We refer to [13] for more details and proofs omitted here for reasons of space.

2 A Motivating Example

Consider the expressions of the untyped λ-calculus:

$$e ::= x \mid \Lambda x \,.\, e \mid e_1\, e_2$$

We encode these expressions as terms in (labeled) HHF via the usual techniques of higher-order abstract syntax as canonical forms over the following signature:

$$\Sigma = exp : \text{type}, lam : (exp \to exp) \to exp, app : exp \to (exp \to exp)$$

The representation function is given by:

$$\ulcorner x \urcorner = x \qquad \ulcorner \Lambda x \,.\, e \urcorner = lam\ (\lambda x : exp. \ulcorner e \urcorner) \qquad \ulcorner e_1\ e_2 \urcorner = app\, \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner$$

A term is *linear* if every functional subterm uses each argument exactly once: in particular, we check for linearity of a function making sure that the latter is linear in its first argument and then recurring on the rest of the expression.

$$\begin{aligned} linapp &: linear(app\ E_1\ E_2) \leftarrow linear(E_1) \wedge linear(E_2). \\ linlam &: linear(lam(\lambda x \,.\, E\ x)) \\ &\qquad\qquad \leftarrow linx(\lambda x \,.\, E\ x) \wedge (\forall y : exp.\, linear(y) \to linear(E\ y)). \end{aligned}$$

$$\begin{aligned} linxx &: linx(\lambda x \,.\, x). \\ linxap1 &: linx(\lambda x \,.\, app\ (E_1\ x)\ E_2) \leftarrow linx(\lambda x \,.\, E_1\ x). \\ linxap2 &: linx(\lambda x \,.\, app\ E_1\ (E_2\ x)) \leftarrow linx(\lambda x \,.\, E_2\ x). \\ linxlm &: linx(\lambda x \,.\, lam(\lambda y \,.\, E\ x\ y)) \leftarrow (\forall y : exp.\, linx(\lambda x \,.\, E\ x\ y)). \end{aligned}$$

This is clearly a decision procedure, which can be complemented; an expression is *not* linear if there is some function which either does not use its argument or uses it more than once. First, the complement of `linapp` does not pose any problem, as it is a Horn clause: an application is not linear if either the first element or the second is not linear. Next, a lambda expression is not linear in two cases: one, if it is not linear in its first argument:

$$\neg linlam1 : \neg linear(lam\ \lambda x \,.\, E\ x) \leftarrow \neg linx(\lambda x \,.\, E\ x).$$

Secondly, if its body is not linear. Now, this poses a new problem, as we have to negate a hypothetical and parametric goal. Let us reason by example and suppose we are given, in the empty context, a goal $linear(lam(\lambda x \,.\, lam(\lambda y \,.\, x)))$, which is unprovable, since the second lambda term is not linear in y; the proof tree yields the failure leaf $linx(\lambda x \,.\, z)$, for a new parameter z, in the context $z{:}exp; linear(z)$. Our guiding intuition is that we want to mimic a failure derivation so as to provide a successful derivation from the negative definition, i.e. a proof of $\neg linx(\lambda x \,.\, z)$ from $z{:}exp; linear(z)$; this shows one prominent feature of complementation of an HHF formula: negation 'skips' over $\forall$ and $\rightarrow$, since it needs to mirror failure *from* assumptions. Now, let us examine clause `linxlm` and reconsider the above failure leaf; in a first attempt, according to the idea above, the complement would be:

$$\overset{?}{\neg}\, linxlm : \neg linx(\lambda x \,.\, lam(\lambda y \,.\, E\ x\ y)) \leftarrow (\forall y{:}exp.\, \neg linx(\lambda x \,.\, E\ x\ y)).$$

However, there is no way to obtain a proof of $\neg linx(\lambda x \,.\, z)$ from the current context. Indeed, the `linxlm` clause does not carry enough information so that its complement can mimic the failure proof. In a sense, the clause is not *assumption-complete*: once it has introduced a new parameter, the clause only specifies how to use it in a positive context. It is up to us to synthesize its dynamic negative definition, in this case simply $\forall y{:}exp.\, \neg linx(\lambda x \,.\, y)$. More in general, it is a characteristic of HHF that the negation of a clause is not strong enough to determine the behavior of a program under complementation. We will have to insert (via a source-to-source transformation) additional structure in a predicate definition, in order to completely determine the provability or failure of goals which mention *parameters*. By observing the structure of all possible assumption that a predicate definition can make, we will *augment* those assumptions with their negative definition. In particular, we first augment the clause `linxlm`:

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

so that, by complementation, we obtain:

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

Unfortunately, the procedure we have outlined is not possible in general. Consider a clause encoding the introduction rule for implication in natural deduction, which can be used to check whether an implicational formula trivially holds:

$$\Sigma = form : type, imp : form \rightarrow (form \rightarrow form), a : form, b : form, c : form$$
$$impi \ : \ nd(A\ imp\ B) \leftarrow (nd(A) \rightarrow nd(B)).$$

Following our earlier remark its complement would be:

$$\begin{aligned} &\neg impi1 : \neg nd(a). \\ &\neg impi2 : \neg nd(b). \\ &\neg impi3 : \neg nd(c). \\ &\overset{?}{\neg}\, impi : \neg nd(A\ imp\ B) \leftarrow (nd(A) \rightarrow \neg nd(B)). \end{aligned}$$

This specification is clearly incorrect since both $nd(a\ imp\ a)$ **and** $\neg nd(a\ imp\ a)$ are derivable from the empty context. We can isolate one major problem: in clause `impi` the assumption $nd(A)$ which is dynamically added to the (static) definition of the `nd` predicate *overlaps* with the head of the clause. A symmetrical problem can occur when dynamic and static clause do differ but their complements do not. We have thus isolated two main issues:

1. Exhaustivity: we need to enrich clauses so that every (ground) goal or its negation is provable.
2. Exclusivity: we need to isolate a significant fragment where it is not the case that both a goal and its negation are provable.

We will achieve exhaustivity (Theorem 2) by *augmenting* the program with the complement of assumptions; moreover, we will achieve exclusivity (Theorem 1) with the restriction to *complementable* programs. To anticipate the idea, a clause is complementable if every assumption contains some eigenvariable at execution time.

3 Provability and Unprovability

We will use the following somewhat unusual language:

$$
\begin{array}{rl}
\textit{Simple Types} & A ::= a \mid A_1 \rightarrow A_2 \\
\textit{Terms} & M ::= c \mid x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \\
\textit{Atoms} & Q ::= q\ \overline{M_n} \mid \neg q\ \overline{M_n} \\
\textit{Clauses} & D ::= \top \mid \bot \mid Q \leftarrow G \mid D_1 \wedge D_2 \mid D_1 \vee D_2 \mid \forall x : A.\, D \\
\textit{Goals} & G ::= Q \mid \top \mid \bot \mid \overline{M_n} \doteq \overline{N_n} \mid \overline{M_n} \mathrel{\dot{\neq}} \overline{N_n} \mid \\
& \quad G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \rightarrow G \mid \forall x : A.\, G \\
\textit{Signatures} & \Sigma ::= \cdot \mid \Sigma, a{:}\mathrm{type} \mid \Sigma, c{:}A \\
\textit{Parameter Contexts} & \Gamma ::= \cdot \mid \Gamma, x{:}A \\
\textit{Assumptions} & \mathcal{D} ::= \top \mid \mathcal{D} \wedge D
\end{array}
$$

There is a distinguished type $\mathbf{o}$ for propositions which can occur only as the target of some A. We remark that '$\neg$' is *not* a connective, but a name constructor for atomic formulae; 'facts' are represented, for convenience, by $Q \leftarrow \top$, although in examples we will omit to mention $\top$. We assume that existential variables occur only once in the head of program clauses (i.e. clauses are *left-linear*); this can always be achieved by introducing disequations in the body. In this paper we restrict ourselves to programs such that all assumptions are Horn and which can be proven to be terminating under some well-founded ordering. We introduce the uniform proofs system [12] for (immediate) provability and denial in Fig. 1. For terminating programs, we can prove that the failure to achieve a proof of G translates into (a derivation of) the denial of G. Note also that due to the presence of disjunction as a clause constructor, uniform proofs are *not* complete for our language. We will remedy this situation in Sect. 4.

$\Gamma; \mathcal{D} \vdash_{\mathcal{P}} G$	Program $\mathcal{P}$ and assumption $\mathcal{D}$ uniformly entail G.
$\Gamma; \mathcal{D} \nvdash_{\mathcal{P}} G$	Program $\mathcal{P}$ and assumption $\mathcal{D}$ uniformly deny G.
$\Gamma; \mathcal{D} \vdash_{\mathcal{P}} D \gg Q$	Clause D from $\mathcal{P}$ and $\mathcal{D}$ immediately entails atom Q.
$\Gamma; \mathcal{D} \nvdash_{\mathcal{P}} D \gg Q$	Clause D from $\mathcal{P}$ and $\mathcal{D}$ immediately denies atom Q.

$$\frac{}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}}\top}\vdash\top \qquad \frac{}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}}\bot}\not\vdash_{\bot}$$

$$\frac{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} G_1 \qquad \Gamma;\mathcal{D}\vdash_{\mathcal{P}} G_2}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} G_1\wedge G_2}\vdash\wedge \qquad \frac{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} G_1 \qquad \Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} G_2}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} G_1\vee G_2}\not\vdash\vee$$

$$\frac{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} G_i}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} G_1\vee G_2}\vdash\vee_i \qquad \frac{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} G_i}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} G_1\wedge G_2}\not\vdash\wedge_i$$

$$\frac{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} [t/x]G}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} \exists x:A.\,G}\vdash\exists \qquad \frac{\text{for all } n\ \Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} [n/x]G}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} \exists x:A.\,G}\not\vdash\exists$$

$$\frac{\Gamma;(\mathcal{D}\wedge D)\vdash_{\mathcal{P}} G}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} D\rightarrow G}\vdash\rightarrow \qquad \frac{\Gamma;(\mathcal{D}\wedge D)\not\vdash_{\mathcal{P}} G}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} D\rightarrow G}\not\vdash\rightarrow$$

$$\frac{(\Gamma,y{:}A);\mathcal{D}\vdash_{\mathcal{P}} [y/x]G}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} \forall x:A.\,G}\vdash\forall^y \qquad \frac{(\Gamma,y{:}A);\mathcal{D}\not\vdash_{\mathcal{P}} [y/x]G}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} \forall x:A.\,G}\not\vdash\forall^y$$

$$\frac{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} (\mathcal{P}\wedge\mathcal{D})\gg Q}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} Q}\vdash\mathrm{At} \qquad \frac{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} (\mathcal{P}\wedge\mathcal{D})\gg Q}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} Q}\not\vdash\mathrm{At}$$

$$\frac{}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}}\bot\gg Q}\gg\bot \qquad \frac{}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}}\top\gg Q}\not\gg\top$$

$$\frac{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} [t/x]D\gg Q}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} \forall x:A.\,D\gg Q}\gg\forall \qquad \frac{\text{for all } n\ \Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} [n/x]D\gg Q}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} \forall x:A.\,D\gg Q}\not\gg\forall$$

$$\frac{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} D_i\gg Q}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} D_1\wedge D_2\gg Q}\gg\wedge_i \qquad \frac{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} D_i\gg Q}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} D_1\vee D_2\gg Q}\not\gg\vee_i$$

$$\frac{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} D_1\gg Q \qquad \Gamma;\mathcal{D}\vdash_{\mathcal{P}} D_2\gg Q}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} D_1\vee D_2\gg Q}\gg\vee$$

$$\frac{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} D_1\gg Q \qquad \Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} D_2\gg Q}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} D_1\wedge D_2\gg Q}\not\gg\wedge$$

$$\frac{\overline{N_n}=\overline{M_n} \qquad \Gamma;\mathcal{D}\vdash_{\mathcal{P}} G}{\Gamma;\mathcal{D}\vdash_{\mathcal{P}} (q\ \overline{N_n}\leftarrow G)\gg q\ \overline{M_n}}\gg\rightarrow$$

$$\frac{\overline{N_n}\neq\overline{M_n}}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} (q\ \overline{N_n}\leftarrow G)\gg q\ \overline{M_n}}\not\gg\rightarrow_1 \qquad \frac{\overline{N_n}=\overline{M_n} \qquad \Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} G}{\Gamma;\mathcal{D}\not\vdash_{\mathcal{P}} (q\ \overline{N_n}\leftarrow G)\gg q\ \overline{M_n}}\not\gg\rightarrow_2$$

Fig. 1. (Immediate) Provability and Denial

Some brief comments are in order: the (in)equalities rules simply mirror the object logic symbols $\doteq, \not\doteq$ as meta-level (in)equalities and are omitted. $\not\gg\forall$ and $\nvdash \exists$ are infinitary rules, given the meta-linguistic extensional universal quantification on all terms. Rules $\vdash \forall$, $\nvdash \forall$ are instead parametric in y, where the $()^y$ superscript reminds us of the eigenvariable condition. The denial rules for implication and universal quantification reflect the operational semantics of unprovability that we have discussed earlier.

We start by putting every program in a *normalized* format w.r.t. assumptions, so that every goal in the scope of an universal quantifier is guaranteed to depend on some assumption, possibly the trivial clause $\top$. This has also the effect of 'localizing' the trivial assumption to its atom, a property will be central while complementing assumptions; for example we re-write `linxlm` as follows:

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

For the sake of this paper, we also need to modify the source program so that every term in a clause head is *fully applied*, i.e. it is a lambda term where every variable mentioned in the binder occurs in the matrix; this makes term complementation (Sect. 4) much simpler. For example clause `linxap1` is rewritten as:

$$linxap1 : linx(\lambda x\,.\,app\ (E_1\ x)\ (E_2\ x)) \leftarrow linx(\lambda x\,.\,E_1\ x) \wedge vac(\lambda x\,.\,E_2\ x).$$

where $vac(\lambda x\,.\,E_2\ x)$ enforces that x does not occur in $E_2\ x$. Its definition is type-directed, but we have shown in [14] how to internalize these occurrence constraints in a strict type theory, so that this further transformation is not needed.

We now discuss context schemata. As we have argued in Sect. 2, we cannot obtain closure under clause complementation for the full logic of HHF, but we have to restrict ourselves to a smaller (but significant) fragment. This in turn entails that we have to make sure that during execution, whenever an assumption is made, it remains in the fragment we have isolated. Technically, we proceed as follows:

- We extract from the static definition of a predicate the general 'template' of a legal assumption.
- We require dynamic assumptions to conform to this template.

We thus introduce the notion of *schema satisfaction*, which uses the following data structure: a *context schema* abstracts over all possible instantiations of a context during execution. To account for that, we introduce a quantifier-like operator, say SOME $\Phi.\mathcal{D}$, which takes a clause and existentially bounds its free variables, if any, i.e. $\Phi = FV(\mathcal{D})$. The double bar '$\|$', not to be confused with the BNF '$|$' that we informally use in the meta-language, denotes schema alternatives, while '$\circ$' stands for the empty context schema.

$$\textit{Contexts Schemata}\ \mathcal{S} ::= \circ \mid \mathcal{S} \| (\Gamma; \mathrm{SOME}\ \Phi.\mathcal{D})$$

The `linear` predicate yields this (degenerate) example of context schema:

$$\mathcal{S}_{linear} = \circ \mid \mathcal{S}_{linear} \| x{:}exp; linear(x) \mid \mathcal{S}_{linear} \| x{:}exp; \top_{linx}$$

We extract a context schema by collecting all negative occurrences in a goal; this is achieved by simulating execution until an atomic goal is reached and the current list of parameters and assumptions is returned, with their correct existential binding. Different clauses may contribute different schema alternatives for a given predicate definition. A *run-time* context consists of a set of *blocks*, each of which is an instance of the context schema, for example:

$$y_1{:}exp, y_2{:}exp, x_1{:}exp; \top_{linx} \wedge \top_{linx} \wedge linear(x_1)$$

We will need to disambiguate blocks in run-time contexts; overlapping may indeed happen when the alternatives in a context schema are not disjoint. Intuitively, a block is complete when an atomic conclusion is reached during the deduction. Any bracketing convention will do:

$$\lceil y_1{:}exp\rceil, \lceil y_2{:}exp\rceil, \lceil x_1{:}exp\rceil; \lceil \top_{linx}\rceil \wedge \lceil \top_{linx}\rceil \wedge \lceil linear(x_1)\rceil$$

We then define when a formula satisfies a schema. We start by saying that a completed block *belongs* to a schema when the block is an alphabetic variant of some instantiation of one of the alternatives of the schema. Then, the empty run-time context is an *instance* of every schema. Secondly, if Γ' and $\mathcal{D}'$ are completed blocks which belong to $\mathcal{S}$, and $\Gamma; \mathcal{D}$ in an instance of $\mathcal{S}$, then $(\Gamma, \lceil \Gamma' \rceil); (\mathcal{D} \wedge \lceil \mathcal{D}' \rceil)$ is an *instance* of $\mathcal{S}$, provided that $\mathcal{D}'$ is a valid clause. The latter holds when each of its subgoals satisfies the schema. This is achieved by mimicking the construction on the run-time schema until in the base case we check whether the resulting context is an instance of the given schema.

We can prove that if a context schema is extracted from a program, then any instance of the latter satisfies the former. Moreover, execution preserves contexts, i.e. every subgoal which arises in any given successful or failed (immediate and non-immediate) sub-derivation satisfies the context schema. See [13] for the formal development.

4 Clause Complementation

We restrict ourselves to programs with:

– Goals where every assumption is parametric, i.e. it is in the scope of a positive occurrence of a universal quantifier and the corresponding parameter occurs in head position in the assumption.
– Clauses $Q \leftarrow G$ such that the head of every term in Q is rigid.

Note that the rigidity restriction applies only to non-Horn predicate definitions and can be significantly relaxed; see [13] for a detailed account.

The first ingredient is *higher-order pattern* complement, $\mathrm{Not}(M)$, investigated in the general case in [14]; we give here the rules for complementing fully applied patterns:

$$\frac{}{\Gamma \vdash \mathrm{Not}(E\ \overline{x_n}) \Rightarrow \emptyset}\ \mathrm{Not\_}Flx$$

$$\frac{\Gamma, x{:}A \vdash \mathrm{Not}(M) \Rightarrow N : B}{\Gamma \vdash \mathrm{Not}(\lambda x : A.\, M) \Rightarrow \lambda x : A.\, N : A \to B}\ \mathrm{Not\_}Lam$$

$$\frac{g \in \Sigma \cup \Gamma, g : A_1 \to \ldots \to A_m \to a, m \geq 0, h \not\equiv g}{\Gamma \vdash \mathrm{Not}(h\ \overline{M_n}) \Rightarrow g\ (Z_1 \Gamma) \ldots (Z_m \Gamma) : a}\ \mathrm{Not\_}App^1$$

$$\frac{\exists i : 1 \leq i \leq n \qquad \Gamma \vdash \mathrm{Not}(M_i) \Rightarrow N :}{\Gamma \vdash \mathrm{Not}(h\ \overline{M_n}) \Rightarrow h\ (Z_1 \Gamma) \ldots (Z_{i-1} \Gamma)\ N\ (Z_{i+1} \Gamma) \ldots (Z_n \Gamma) : a}\ \mathrm{Not\_}App^2$$

where the Z's are fresh variables which may depend on the domain of Γ, $h \in \Sigma \cup \Gamma$, and $\Gamma \vdash h : A_1 \to \ldots \to A_n \to a$. $\Gamma \vdash \mathrm{Not}(M) = \mathcal{N} : A$ iff $\mathcal{N} = \{N \mid \Gamma \vdash \mathrm{Not}(M) \Rightarrow N : A\}$. For example:

$$\cdot \vdash \mathrm{Not}(\lambda x \,.\, x) = \{\lambda x \,.\, lam(\lambda y \,.\, E\ x\ y), \lambda x \,.\, app\ (E_1\ x)\ (E_2\ x)\}$$

If we write $\Gamma \vdash M \in \|N\| : A$ when M is a ground instance of a pattern N at type A, we can show that Not behaves as the complement on sets of ground terms, i.e.

1. (Exclusivity) Not ($\Gamma \vdash M \in \|N\| : A$ and $\Gamma \vdash M \in \|\mathrm{Not}(N)\| : A$).
2. (Exhaustivity) Either $\Gamma \vdash M \in \|N\| : A$ or $\Gamma \vdash M \in \|\mathrm{Not}(N)\| : A$.

Complementing goals is immediate: we just put the latter in negation normal form, respecting the operational semantics of failure.

$$\frac{}{\mathrm{Not_G}(\top) = \bot}\ \mathrm{Not_G}\top \qquad \frac{}{\mathrm{Not_G}(\bot) = \top}\ \mathrm{Not_G}\bot \qquad \frac{}{\mathrm{Not_G}(Q) = \neg Q}\ \mathrm{Not_G}At$$

$$\frac{}{\mathrm{Not_G}(\overline{M_n} \doteq \overline{N_n}) = (\overline{M_n} \not\doteq \overline{N_n})}\ \mathrm{Not} \doteq \qquad \frac{}{\mathrm{Not_G}(\overline{M_n} \not\doteq \overline{N_n}) = (\overline{M_n} \doteq \overline{N_n})}\ \mathrm{Not} \not\doteq$$

$$\frac{\mathrm{Not_G}(G) = G'}{\mathrm{Not_G}(\forall x{:}A.\, G) = \forall x{:}A.\, G'}\ \mathrm{Not_G}\forall \qquad \frac{\mathrm{Not_G}(G) = G'}{\mathrm{Not_G}(D \to G) = D \to G'}\ \mathrm{Not_G} \to$$

$$\frac{\mathrm{Not_G}(G_1) = G_1' \qquad \mathrm{Not_G}(G_2) = G_2'}{\mathrm{Not_G}(G_1 \wedge G_2) = G_1' \vee G_2'}\ \mathrm{Not}\wedge$$

$$\frac{\mathrm{Not_G}(G_1) = G_1' \qquad \mathrm{Not_G}(G_2) = G_2'}{\mathrm{Not_G}(G_1 \vee G_2) = G_1' \wedge G_2'}\ \mathrm{Not}\vee$$

Clause complementation is instead more delicate: given a rule $q\ \overline{M_n} \leftarrow G$, its complement must contain a 'factual' part motivating failure due to clash with the head; the remainder $\mathrm{Not_G}(G)$ expresses failure in the body, if any. Clause complementation must discriminate whether (the head of) a rule belongs to the static or dynamic definition of a predicate. In the first case all the relevant information is already present in the head of the clause and we can use the term complementation algorithm. This is accomplished by the rule $\mathrm{Not_D} \to$, where a set of negative facts is built via term complementation $\mathrm{Not}(\overline{M_n})$, namely $\bigwedge_{\overline{N_n} \in \mathrm{Not}(\overline{M_n})} \forall(\neg q\ \overline{N_n} \leftarrow \top)$, whose fresh free variables are universally closed; moreover the negative counterpart of the source clause is obtained via complementation of the body. The original quantification is retained thanks to rule $\mathrm{Not_D}\forall$.

$$\frac{}{\mathrm{Not_D}(\top) = \bot}\,\mathrm{Not_D}\top \qquad \frac{}{\mathrm{Not_D}(\bot) = \top}\,\mathrm{Not_D}\bot$$

$$\frac{\mathrm{Not_G}(G) = G'}{\mathrm{Not_D}(q\ \overline{M_n} \leftarrow G) = (\bigwedge_{\overline{N_n} \in \mathrm{Not}(\overline{M_n})} \forall(\neg(q\ \overline{N_n}) \leftarrow \top)) \wedge (\neg q\ \overline{M_n} \leftarrow G')}\,\mathrm{Not_D} \leftarrow$$

$$\frac{\mathrm{Not_D}(D) = D'}{\mathrm{Not_D}(\forall x\!:\!A.\,D) = \forall x\!:\!A.\,D'}\,\mathrm{Not_D}\forall$$

$$\frac{\mathrm{Not_D}(D_1) = D_1' \qquad \mathrm{Not_D}(D_2) = D_2'}{\mathrm{Not_D}(D_1 \wedge D_2) = D_1' \vee D_2'}\,\mathrm{Not}\wedge$$

$$\frac{\mathrm{Not_D}(D_1) = D_1' \qquad \mathrm{Not_D}(D_2) = D_2'}{\mathrm{Not_D}(D_1 \vee D_2) = D_1' \wedge D_2'}\,\mathrm{Not}\vee$$

Otherwise, we can think of the complement of an atomic assumption $(q\ M_1 \ldots x\ \ldots M_n)$, which is by definition parametric in some x, as static clause complementation w.r.t. x, i.e. $\mathrm{Not_D}(q_x\ M_1 \ldots M_{i-1}\ M_{i+1} \ldots M_n)$. However, most of those M_i, which at compile-time are variables, will be instantiated at run-time: therefore it would be incorrect to compute their complement as empty. Since we cannot foresee this instantiation, we achieve clause complementation via the introduction of disequations. This is realized by the judgment $\Gamma \vdash \mathrm{Not}_\alpha(D)$. We need the following notion: a parameter x:a is *relevant* to a predicate symbol q (denoted xR^iq) if $\Sigma(q) = A_1 \to \cdots \to A_n \to \mathbf{o}$ and for some $1 \leq i \leq n$ the target type of A_i is a.

$$\frac{}{\Gamma \vdash \mathrm{Not}_\alpha(\top_q) = \bigwedge_{x \in dom(\Gamma)} (\bigwedge_{xR^iq} \mathrm{Not}_x^i(\top_q))}\,\mathrm{Not}_\alpha\top$$

$$\frac{\mathrm{Not_G}(G) = G'}{\Gamma \vdash \mathrm{Not}_\alpha(Q \leftarrow G) = (\bigwedge_{x \in dom(\Gamma)} (\bigwedge_{xR^iq} \mathrm{Not}_x^i(Q))) \wedge (\neg Q \leftarrow G')}\,\mathrm{Not}_\alpha \leftarrow$$

Both rules refer to an auxiliary judgment $\mathrm{Not}_x^i(D)$:

$$\frac{\Sigma(q) = A_1 \to \cdots \to A_n \to \mathbf{o} \qquad \cdot \vdash sh(x, A_i) = e_x}{\mathrm{Not}_x^i(\top_q) = \forall Z_1\!:\!A_1. \ldots \forall Z_n\!:\!A_n. \neg(q\ \overline{Z_{e_x}^i}) \leftarrow \top}\,\mathrm{Not}_x^i\top$$

$$\frac{\Sigma(q) = A_1 \to \cdots \to A_n \to \mathbf{o} \qquad \cdot \vdash sh(x, A_i) = e_x}{\mathrm{Not}_x^i(q\ \overline{M_n}) = \bigwedge_{1 \leq j \leq n, j \neq i} (\forall Z_1\!:\!A_1. \ldots \forall Z_n\!:\!A_n. \neg(q\ \overline{Z_{e_x}^i}) \leftarrow M_j \dot{\neq} Z_j)}\,\mathrm{Not}_x^i\mathrm{At}$$

The idea is to:

– Pivot on $x{:}a \in \Gamma$.
– Locate a type A_i such that $x{:}a$ is relevant to q at i.
– Complement D w.r.t. x and i.
– Repeat for every A_i and for every x.

The rest of the rules for $\Gamma \vdash \mathrm{Not}_{\alpha}(D)$ are completely analogous to the ones for $\mathrm{Not}_{\mathrm{D}}(D)$ and are omitted. Both simply recur on the program respecting the duality of conjunction and disjunction w.r.t. negation. Notice the different treatment of the trivial clause $\top$ by rules $\mathrm{Not}_{\mathrm{D}}\top$ and $\mathrm{Not}_{\alpha}\top$: if no parameter has been assumed, then $\top$ truly stands for the empty predicate definition and its complement is the universal definition $\bot$. If, on the other hand Γ is *not* empty, it means that $\top_q$ has been introduced during the $\top$-normalization preprocessing phase and has been localized to the predicate q. The rule $\mathrm{Not}^i_x\top$ allows to build a new negative assumption w.r.t. q, x, i in case $\top_q$ is the only dynamic definition of q. As $\top_q$ carries no information at all concerning q, the most general negative assumption is added; the notation $\overline{Z^i_{e_x}}$ abridges $Z_1 \ldots Z_{i-1}\ e_x\ Z_{i+1} \ldots Z_n$, where the Z's are fresh logic variables and e_x is a term built prefixing a parameter x by an appropriate number of lambda's, according to the type of its position; this is specified by the $\Gamma \vdash sh(x, A)$ judgment, omitted here (but see it in action in Example 1).

Now that we have discussed how to perform clause, assumption and goal complementation, we synchronize them together in a phase we call *augmentation*, which simply inserts the correct assumption complementation into a goal and in turn into a clause. This is achieved by a judgment $\Gamma; \mathcal{D} \vdash_{\Phi} aug_{\mathrm{D}}(D)$, again omitted here for reasons of space.

For example, consider the `copy` clause on λ-terms:

$$\begin{aligned} &cplam : copy\ (lam\ E)\ (lam\ F) \\ &\qquad\qquad \leftarrow (\forall x{:}exp.\, copy\ x\ x \rightarrow copy\ (E\ x)\ (F\ x)). \end{aligned}$$

The augmentation procedure collects $x{:}exp; copy\ x\ x$ and calls $x{:}exp \vdash \mathrm{Not}_{\alpha}(copy\ x\ x)$. First $\mathrm{Not}^x_1(copy\ x\ x) = (\forall E'{:}exp.\, \neg copy\ E'\ x \leftarrow x \not\doteq E')$, secondly $\mathrm{Not}^x_2(copy\ x\ x) = (\forall F'{:}exp.\, \neg copy\ x\ F' \leftarrow x \not\doteq F')$, yielding:

$$\begin{aligned} aug_{\mathrm{D}}(cplam) : copy\ &(lam\ E)\ (lam\ F) \\ &\leftarrow (\forall x{:}exp. \\ &\qquad (\forall E'{:}exp.\, \neg copy\ E'\ x \leftarrow x \not\doteq E') \wedge \\ &\qquad (\forall F'{:}exp.\, \neg copy\ x\ F' \leftarrow x \not\doteq F') \rightarrow \\ &\qquad\quad (copy\ x\ x \rightarrow copy\ (E\ x)\ (F\ x))). \end{aligned}$$

Let us see how rule $\mathrm{Not}^i_x\top$ enters the picture; recall the normalized `linxlm` clause. From $\cdot \vdash sh(y, exp \rightarrow exp) = \lambda x\,.\,y$ we have $\mathrm{Not}^y_1(\top_{linx}) = \neg linx\ (\lambda x\,.\,y)$:

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

Let us apply the complement algorithm to the `linx` predicate definition. The `strict` predicate is simply the complement of the `vac` predicate previously introduced. Again, these annotations can be internalized in a strict type theory [14].

$$
\begin{aligned}
&\mathrm{Not_D}(\mathit{def}(\mathit{linx})) = \\
&\mathrm{Not_D}(\mathit{linxx}) \vee \mathrm{Not_D}(\mathit{linxap}1) \vee \mathrm{Not_D}(\mathit{linxap}2) \vee \mathrm{Not_D}(\mathit{linxlm}) = \\
&\quad (\neg \mathit{linx}(\lambda x\,.\,\mathit{app}\ (E_1\ x)\ (E_2\ x)) \wedge \neg \mathit{linx}(\lambda x\,.\,\mathit{lam}(\lambda y\,.\,(E\ x\ y)))) \vee \\
&\quad (\neg \mathit{linx}(\lambda x\,.\,x) \wedge \neg \mathit{linx}(\lambda x\,.\,\mathit{lam}(\lambda y\,.\,(E\ x\ y))) \wedge \\
&\quad \neg \mathit{linx}(\lambda x\,.\,\mathit{app}\ (E_1\ x)\ (E_2\ x)) \leftarrow \mathit{strict}(\lambda x\,.\,E_2\ x) \\
&\quad \wedge \neg \mathit{linx}(\lambda x\,.\,\mathit{app}\ (E_1\ x)\ (E_2\ x)) \leftarrow \neg \mathit{linx}(\lambda x\,.\,E_1\ x)) \vee \\
&\quad (\neg \mathit{linx}(\lambda x\,.\,x) \wedge \neg \mathit{linx}(\lambda x\,.\,\mathit{lam}(\lambda y\,.\,(E\ x\ y))) \wedge \\
&\quad \neg \mathit{linx}(\lambda x\,.\,\mathit{app}\ (E_1\ x)\ (E_2\ x)) \leftarrow \mathit{strict}(\lambda x\,.\,E_1\ x) \\
&\quad \wedge \neg \mathit{linx}(\lambda x\,.\,\mathit{app}\ (E_1\ x)\ (E_2\ x)) \leftarrow \neg \mathit{linx}(\lambda x\,.\,E_2\ x)) \vee \\
&\quad (\neg \mathit{linx}(\lambda x\,.\,x) \wedge \neg \mathit{linx}(\lambda x\,.\,\mathit{app}\ (E_1\ x)\ (E_2\ x)) \\
&\quad \wedge \neg \mathit{linx}(\lambda x\,.\,\mathit{lam}(\lambda y\,.\,(E\ x\ y))) \leftarrow (\forall y\!:\!\mathit{exp}.\,\neg \mathit{linx}(\lambda x\,.\,y) \to \neg \mathit{linx}(\lambda x\,.\,E\ x\ y))).
\end{aligned}
$$

We can now establish exclusivity and exhaustivity of clause complementation. Let $\mathrm{Not_D}(\mathcal{P}) = \mathcal{P}^-$:

Theorem 1 (Exclusivity). *For every run-time context $\Gamma;\mathcal{D}$ instance of a schema $\mathcal{S}$ extracted from an augmented program $\mathcal{P}$:*

1. *It is not the case that $\Gamma;\mathcal{D} \vdash_{\mathcal{P}} G$ and $\Gamma;\mathcal{D} \vdash_{\mathcal{P}^-} \mathrm{Not_G}(G)$.*
2. *It is not the case that $\Gamma;\mathcal{D} \vdash_{\mathcal{P}} (\mathcal{P} \wedge \mathcal{D}) \gg Q$ and $\Gamma;\mathcal{D} \vdash_{\mathcal{P}^-} (\mathcal{P}^- \wedge \mathcal{D}) \gg \neg Q$.*

Proof. (Sketch) By mutual induction on the structure of the derivation of $\Gamma;\mathcal{D} \vdash_{\mathcal{P}} G$ and $\Gamma;\mathcal{D} \vdash_{\mathcal{P}} (\mathcal{P} \wedge \mathcal{D}) \gg Q$. The proof goes through as there is no 'bad' interaction between the static and dynamic definition of a predicate; namely there is no overlap between a clause from $\mathcal{P}$ and from $\mathcal{D}$ since in every atomic assumption there must be an occurrence of an eigenvariable *and* every corresponding term in a program clause head must start with a constructor. If both clauses are dynamic, it holds because an appropriate disequation is present; this approximates what happens in the static case, which is based on term exclusivity.

The denial system comes in handy in the following proof.

Theorem 2 (Exhaustivity). *For every substitution θ, σ and run-time context $\Gamma; [\theta]\mathcal{D}$ instance of a schema $\mathcal{S}$ extracted from an augmented program $\mathcal{P}$:*

1. *If for all θ $\Gamma; [\theta]\mathcal{D} \not\vdash_{\mathcal{P}} [\theta]G$, then there is a σ such that $\Gamma; [\sigma]\mathcal{D} \vdash_{\mathcal{P}^-} [\sigma]\mathrm{Not_G}(G)$.*

2.1 *If, for all θ $\Gamma; [\theta]\mathcal{D} \not\vdash_{\mathcal{P}} [\theta]\mathcal{P} \gg [\theta]Q$, then here is a σ such that $\Gamma; [\sigma]\mathcal{D} \vdash_{\mathcal{P}^-} [\sigma]\mathrm{Not_D}(D) \gg [\sigma]\neg Q$.*

2.2 *If, for all θ $\Gamma; [\theta]\mathcal{D} \not\vdash_{\mathcal{P}} [\theta]\mathcal{D} \gg [\theta]Q$, then here is a σ such that $\Gamma; [\sigma]\mathcal{D} \vdash_{\mathcal{P}^-} [\sigma]\mathrm{Not}_{\alpha}(D) \gg [\sigma]\neg Q$.*

The proof is by mutual induction on the structure of the given derivations. As a corollary, we are guaranteed that clause complementation satisfies the boolean rules of negation.

Finally, we show how to eliminate from clauses the '$\vee$' operator stemming from the complementation of conjunctions, while preserving provability; this will recover uniformity in proof-search.

The key observation is that in this context '$\vee$' can be restricted to a program constructor *inside* a predicate definition; therefore it can be eliminated by simulating unification in the definition, that is $(Q_1 \leftarrow G_1) \vee (Q_2 \leftarrow G_2) \equiv \theta(Q_1 \leftarrow G_1 \wedge G_2)$, where $\theta = mgu(Q_1, Q_2)$.

However, the (strict) higher-order unification problem is quite complex, even more so due to the mixed quantifier structure of HHF; since we have already parameter (dis)equations introduced by the augmentation procedure, as well as variable-variable (dis)equations stemming from left-linearization, we first compile clauses in an intermediate language which keeps the unification problems explicit and then we perform constraint simplification as in *Twelf.* Continuing with our example and simplifying the constraints:

$$\begin{aligned}
&\mathrm{Not_D}(linxx) \vee \mathrm{Not_D}(linxap1) = \\
&\quad \neg linx(\lambda x\,.\,app\ (E_1\ x)\ (E_2\ x)) \leftarrow strict(\lambda x\,.\,E_2\ x) \wedge \\
&\quad \neg linx(\lambda x\,.\,app\ (E_1\ x)\ (E_2\ x)) \leftarrow \neg linx(\lambda x\,.\,E_1\ x) \wedge \\
&\quad \neg linx(\lambda x\,.\,lam(\lambda y\,.\,E\ x\ y)).
\end{aligned}$$

The final definition of $\neg$`linear` and in turn $\neg$`linx` is:

$$\begin{aligned}
\neg linapp &: \neg linear(app\ E_1\ E_2) \\
&\qquad \leftarrow \neg linear(E_1) \vee \neg linear(E_2). \\
\neg linlam1 &: \neg linear(lam(\lambda x\,.\,E\ x)) \\
&\qquad \leftarrow \neg linx(\lambda x\,.\,E\ x) \\
&\qquad \vee (\forall y{:}exp.\,(\neg linx(\lambda x\,.\,y) \wedge linear(y)) \rightarrow \neg linear(E\ y)). \\
\neg linxap0 &: \neg linx(\lambda x\,.\,app\ (E_1\ x)\ (E_2\ x)) \leftarrow strict(\lambda x\,.\,E_1\ x) \wedge strict(\lambda x\,.\,E_2\ x). \\
\neg linxap1 &: \neg linx(\lambda x\,.\,app\ (E_1\ x)\ (E_2\ x)) \leftarrow strict(\lambda x\,.\,E_2\ x) \wedge \neg linx(\lambda x\,.\,E_1\ x). \\
\neg linxap2 &: \neg linx(\lambda x\,.\,app\ (E_1\ x)\ (E_2\ x)) \leftarrow strict(\lambda x\,.\,E_1\ x) \wedge \neg linx(\lambda x\,.\,E_2\ x). \\
\neg linxap3 &: \neg linx(\lambda x\,.\,app\ (E_1\ x)\ (E_2\ x)) \leftarrow \neg linx(\lambda x\,.\,E_1\ x) \wedge \neg linx(\lambda x\,.\,E_2\ x). \\
\neg linxlm &: \neg linx(\lambda x\,.\,lam(\lambda y\,.\,E\ x\ y)) \\
&\qquad \leftarrow (\forall y{:}exp.\,\neg linx(\lambda x\,.\,y) \rightarrow \neg linx(\lambda x\,.\,E\ x\ y)).
\end{aligned}$$

5 Conclusions and Future Work

We have presented elimination of negation in a fragment of higher-order HHF; our next task is to overcome some of the current restrictions, to begin with the extension to *any* order, which requires a more refined notion of context. The issue of local variables is instead more challenging. The proposal in [1] is not satisfactory and robust enough to carry over to logical frameworks with intensional universal quantification. Our approach will be again to *synthesize* a HHF definition for the clauses with local variables which during the transformations has became *extensionally* quantified. Our final goal is to achieve negation elimination in LF.

Acknowledgments. I would like to thank Frank Pfenning for his continuous help and guidance. The notion of context schema is inspired by Schürmann's treatment of analogous material in [18].

References

[1] D. P. A. Brogi, P. Mancarella and F. Turini. Universal quantification by case analysis. In *Proc. ECAI-90*, pages 111–116, 1990.

[2] K. Apt and R. Bol. Logic programming and negation. *Journal of Logic Programming*, 19/20:9–72, May/July 1994.

[3] R. Barbuti, P. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *Journal of Logic Programming*, 8:201–228, 1990.

[4] A. Bonner. Hypothetical reasoning with intuitionistic logic. In R. Demolombe and T. Imielinski, editors, *Non-Standard Queries and Answers*, volume 306 of *Studies in Logic and Computation*, pages 187–219. Oxford University Press, 1994.

[5] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[6] D. M. Gabbay. N-Prolog: An extension of Prolog with hypothetical implications II. Logical foundations and negation as failure. *Journal of Logic Programming*, 2(4):251–283, Dec. 1985.

[7] L. Giordano and N. Olivetti. Negation as failure and embedded implication. *Journal of Logic Programming*, 36(2):91–147, August 1998.

[8] J. Harland. *On Hereditary Harrop Formulae as a Basis for Logic Programming.* PhD thesis, Edinburgh, Jan. 1991.

[9] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.

[10] J.-L. Lassez and K. Marriot. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3(3):301–318, Sept. 1987.

[11] R. McDowell and D. Miller. A logic for reasoning with higher-order abstract syntax: An extended abstract. In G. Winskel, editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, pages 434–445, Warsaw, Poland, June 1997.

[12] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[13] A. Momigliano. *Elimination of Negation in a Logical Framework.* PhD thesis, Carnegie Mellon University, 2000. Forthcoming.

[14] A. Momigliano and F. Pfenning. The relative complement problem for higher-order patterns. In D. D. Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming (ICLP'99)*, pages 389–395, La Cruces, New Mexico, 1999. MIT Press.

[15] G. Nadathur and D. Miller. An overview of λProlog. In K. A. Bowen and R. A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, Aug. 1988. MIT Press.

[16] F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning.* Elsevier Science Publishers, 2000. In preparation.

[17] T. Sato and H. Tamaki. Transformational logic program synthesis. In *International Conference on Fifth Generation Computer Systems*, 1984.

[18] C. Schürmann. *Automating the Meta-Theory of Deductive Systems.* PhD thesis, Carnegie-Mellon University, 2000. forthcoming.

[19] C. Schürmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In C. Kirchner and H. Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300, Lindau, Germany, July 1998. Springer-Verlag LNCS 1421.