The Relative Complement Problem for Higher-Order Patterns

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

Frank Pfenning
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.
fp@cs.cmu.edu

Abstract

We address the problem of complementing higher-order patterns without repetitions of free variables. Differently from the first-order case, the complement of a pattern cannot, in general, be described by a pattern, or even by a finite set of patterns. We therefore generalize the simply-typed λ-calculus to include an internal notion of *strict function* so that we can directly express that a term must depend on a given variable. We show that, in this more expressive calculus, finite sets of patterns without repeated variables are closed under complement and unification. Our principal application is the transformational approach to negation in higher-order logic programs.

Keywords: Relative complement, strict λ-calculus, higher-order logics, intensional negation.

1 Introduction

In most functional and logic programming languages the notion of a pattern, together with the requisite algorithms for matching or unification, play an important role in the operational semantics. Besides unification other problems such as generalization or complement also arise frequently. In this paper we are concerned with the problem of pattern complement in a setting where patterns may contain binding operators, so-called *higher-order patterns* [10]. Higher-order patterns have found

A version of this paper will appear in ICLP99

applications in logic programming [10], logical frameworks [13], term rewriting [12], and functional logic programming [5]. Higher-order patterns inherit many pleasant properties from the first-order case. In particular, most general unifiers [10] and least general generalizations [14] exist, even for complex type theories.

Unfortunately, the complement operation does not generalize as smoothly. Lugiez [8] has studied the more general problem of higher-order disunification and had to go outside the language of patterns and terms to describe complex constraints on sets of solutions. We can isolate one basic difficulty: a pattern such as $\lambda x.\ E\ x$ for an existential variable E matches any term of appropriate type, while $\lambda x.\ E$ matches precisely those terms $\lambda x.\ M$ where M does not depend on x. The complement then consists of all terms $\lambda x.\ M$ such that M *does* depend on x. However, this set cannot be described by a pattern, or even a finite set of patterns.

This formulation of the problem suggests that we should consider a λ-calculus with an internal notion of *strictness* so that we can directly express that a term must depend on a given variable. For reasons of symmetry and elegance we also add the dual concept of *invariance* expressing that a given term does not depend on a given variable. As in the first-order case, it is useful to single out the case of *linear patterns*, namely those where no existential variable occurs more than once.[1] We show that patterns in our calculus have the following properties:

1. The complement of a linear pattern is a finite set of linear patterns.
2. Unification of two patterns is decidable and leads to a finite set of most general unifiers.

Consequently, finite sets of linear patterns in the strict λ-calculus are closed under complement and unification. If we think of finite sets of linear patterns as representing the set of all their ground instances, then they form a boolean algebra under simple union, intersection (implemented via unification) and the complement operation.

The paper is organized as follows: Section 2 briefly reviews related work, while Section 3 introduces some preliminary definitions. In Section 4 we introduce a strict λ-calculus and note some basic properties such as the existence of canonical forms. Section 5 introduces a restriction of the language for which complementation is possible. The algorithm for negation is presented in Section 6. In Section 7 we give a unification algorithm for our fragment. We conclude in Section 8 with some applications and speculation on future research. For reasons of space, a number of lemmas and proofs are omitted here and can be found in [11].

2 Related Work

Complement problems have a number of applications in theoretical computer science (see [4] for a list of references). For example, they are used in functional programming to produce non-ambiguous function definitions by patterns and to improve

[1] This notion of linearity should not be confused with the eponymous concept in linear logic and λ-calculus.

their compilation and in rewriting systems to check whether an algebraic specification is sufficiently complete. They can also be employed to analyze communicating processes expressed by infinite transition systems. Other applications lie in the areas of machine learning and inductive theorem proving. In logic programming Kunen [6] used term complement to represent infinite sets of answers to negative queries. Our main motivation has been the explicit synthesis of the negation of higher-order logic programs, as discussed in Section 8.

Lassez and Marriot [7] proposed the seminal *uncover* algorithm for computing relative complements and introduced the now familiar restriction to linear terms. We quote the definition of the "$\neg$" algorithm for the (singleton) complement problem given in [1] which we generalize in Definition 11. Given a finite signature Σ and a linear term t we define:

$$\begin{array}{ll}
\neg_\Sigma(x) & = \emptyset \\
\neg_\Sigma(f(\overline{t_n})) & = \\
& \{g(\vec{x}) \mid \text{for all } g \in \Sigma \text{ distinct from } f\} \cup \\
& \{f(z_1, \ldots, z_{i-1}, s, z_{i+1}, \ldots, z_n) \mid s \in \neg_\Sigma(t_i), i \in [1, n]\}
\end{array}$$

An alternative solution to the relative complement problem is *disunification* (see [4] for a survey and [8] for an extension to the simply-typed λ-calculus): here operations on sets of terms are translated into conjunctions or disjunctions of equations and disequation under explicit quantifiers. Non-deterministic application of a few dozen rules eventually turns a given problem into a solved form. Though a reduction to a significant subset of the disunification rules is likely to be attainable for complement problems, control is a major problem. We argue that using disunification for this purpose is unnecessarily general. Moreover the higher-order case results in additional complications, such as restrictions on the occurrences of bound variables, which fall outside an otherwise clean framework. As we show in this paper, this must not necessarily be the case. We believe that our techniques can also be applied to analyze disunification, although we have not investigated this possibility at present.

3 Preliminaries

In this section we introduce some preliminary definitions and examples which guide our development. We write P for atomic types, c for term-level constants, and x for term-level variables.

$$\begin{array}{rrcl}
\textit{Simple Types} & A & ::= & P \mid A_1 \to A_2 \\
\textit{Terms} & M & ::= & c \mid x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \\
\textit{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, P{:}\mathrm{type} \mid \Sigma, c{:}A \\
\textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A
\end{array}$$

We require that signatures and contexts declare each constant or variable at most once. Furthermore, we identify contexts which differ only in their order and promote "," to denote disjoint set union. As usual we identify terms which differ only in the names of their bound variables. We restrict attention to well-typed terms, omitting the standard typing rules.

In applications such as logic programming or logical frameworks, λ-abstraction is used to represent binding operators in some object language. In such a situation the most useful notion of normal form are long $\beta\eta$-normal forms (which we call *canonical forms*), since the canonical forms are almost always the terms in bijective correspondence with the objects we are trying to represent. Every well-typed term in the simply-typed λ-calculus has a unique canonical form—a property which persists in the strict λ-calculus introduced in Section 4.

We denote existential variables of type A (also called logical variables, meta-variables, or pattern variables) by E_A, although we mostly omit the type A when it is clear from the context. We think of existential variables as syntactically distinct from bound variables or free variables declared in a context. A term possibly containing some existential variables is called a *pattern* if each occurrence of an existential variable appears in a subterm of the form $E\ x_1 \ldots x_n$, where the arguments x_i are distinct occurrences of free or bound variables (but not existential variables).

Semantically, an existential variable E_A stands for all canonical terms M of type A in the empty context with respect to a given signature. We extend this to arbitrary well-typed terms in the usual way, and write $\Gamma \vdash M \in \|N\| : A$ when a term M is a ground instance of a pattern N at type A. In this setting, unification of two patterns without shared existential variables corresponds to an intersection of the set of terms they denote [10, 14]. This set is always either empty, or can be expressed again as the set of instances of a single pattern. That is, patterns admit most general unifiers.

The class of higher-order patterns inherits many properties from first-order terms. However, as we will see, it is *not* closed under complement, but a special subclass is. We call a canonical pattern $\Gamma \vdash M : A$ *fully applied* if each occurrence of an existential variable E under binders $y_1, \ldots, y_m$ is applied to some permutation of the variables in Γ and $y_1, \ldots, y_m$. Fully applied patterns play an important role in functional logic programming and rewriting [5] because any fully applied existential variable $\Gamma \vdash E\ x_1 \ldots x_n$ denotes all canonical terms with free variables from Γ. It is this property which makes complementation particularly simple.

Example 1 *Consider the untyped λ-calculus.*

$$e \quad ::= \quad x \mid \lambda x\,.\,e \mid e_1\,e_2$$

We encode these expressions using the usual techniques of higher-order abstract syntax (see, for example, [9]) as canonical forms over the following signature.

$$\Sigma \quad = \quad exp : type, lam : (exp \to exp) \to exp, app : exp \to exp \to exp$$

The representation function is given by.

$$\begin{aligned} \ulcorner x \urcorner &= x \\ \ulcorner \lambda x\,.\,e \urcorner &= lam\ (\lambda x{:}exp.\ulcorner e \urcorner) \\ \ulcorner e_1\ e_2 \urcorner &= app\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner \end{aligned}$$

The representation of an object-language β-redex then has the form

$$\ulcorner (\lambda x\,.\,e)\ f \urcorner \quad = \quad app\ (lam\ (\lambda x{:}exp.\ulcorner e \urcorner))\ \ulcorner f \urcorner$$

where $\ulcorner e \urcorner$ *my have free occurrences of* x*. When written as a pattern with variables* $E_{exp \rightarrow exp}$ *and* F_{exp} *ranging over* closed *terms, this is expressed as* $app\ (lam\ (\lambda x : exp. (E\ x))\ F)$*. The complement of the right-hand side considered as a pattern with respect to the empty context contains every top-level* λ*-abstraction plus every application where the first argument is not an abstraction:*

$$\neg(app\ (lam\ (\lambda x{:}exp. (E\ x))\ F) = \{lam\ (\lambda x{:}exp. H\ x), app\ (app\ H_1\ H_2)\ H_3\}$$

For patterns which are not fully applied, the complement cannot be expressed as a finite set of patterns, as the following example illustrates.

Example 2 *The encoding of an* η*-redex takes the form*

$$\ulcorner \lambda x\,.\,e\ x \urcorner \quad = \quad lam(\lambda x{:}exp.\, app\ \ulcorner e \urcorner\ x)$$

where $\ulcorner e \urcorner$ *may contain no free occurrence of* x*. The side condition is expressed in a pattern by introducing an existential variable* E_{exp} *which* does not *depend on* x*. Hence, its complement with respect to the empty context should contain, among others, also all terms*

$$lam\ (\lambda x{:}exp.\, app\ (Ex)\ x)$$

where E does *depend on* x*.*

Note that there is no finite set of patterns which has as its ground instances exactly those terms M which depend on a given variable x. Following standard terminology, we call such terms *strict in* x and the corresponding function $\lambda x{:}A.\,M$ a *strict function*. As the example above shows, the complement of patterns which are not fully applied can therefore not be represented as a finite set of patterns. This failure of closure under complementation cannot be avoided similarly to the way in which left-linearization bypasses the limitation to linear patterns and it needs to be addressed directly.

One approach is taken by Lugiez [8]: he modifies the language of terms to permit occurrence constraints. For example $\lambda xyz.\ M\{1,3\}$ would denote a function which depends on its first and third argument. The technical handling of those objects then becomes awkward as they require specialized rules which are foreign to the issues of complementation.

Since our underlying λ-calculus is typed, we use typing to express that a function *must* or *must not* depend on its argument. In the next section we develop such a λ-calculus and generalize the complement algorithm to work on such terms.

4 A Strict λ-Calculus

As we have seen in the preceding section, the complement of some patterns in the simply-typed λ-calculus cannot be expressed in a finitary manner within the same calculus. We thus generalize our language to include *strict functions* of type $A \xrightarrow{1} B$ (which are guaranteed to depend on their argument) and *invariant functions* of type $A \xrightarrow{0} B$ (which are guaranteed **not** to depend on their argument). Of course, any concretely given function either will or will not depend on its argument, but in the presence of existential variables we still need the ability to remain uncommitted. Therefore our calculus also contains the full function space $A \xrightarrow{u} B$.

$$
\begin{array}{rrcl}
\textit{Labels} & k & ::= & 1 \mid 0 \mid u \\
\textit{Types} & A & ::= & P \mid A_1 \xrightarrow{k} A_2 \\
\textit{Terms} & M & ::= & c \mid x \mid \lambda x^k{:}A.\ M \mid M_1\ {M_2}^k \\
\textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A
\end{array}
$$

Note that there are three different forms of abstractions and applications, where the latter are distinguished by different labels on the argument. It is not really necessary to distinguish three forms of application syntactically, since the type of function determines the status of the application, but it is convenient for our purposes. A label u it is called *undetermined*; other labels are *determined* and denoted by d.

We use a formulation of the typing judgment with three zones, containing the undetermined, irrelevant and strict hypotheses, denoted by Γ, Ω, and Δ, respectively.

$$\Gamma; \Omega; \Delta \vdash M : A$$

We implicitly assume a fixed signature Σ which would otherwise clutter the presentation.

Our system is biased towards a bottom-up reading of the rules in that variables never disappear, i.e., they are always propagated from the conclusion to the premises, although their status might be changed.

Let us go through the typing rules in detail. The requirement for the strict context Δ to be empty in the Id^u and Id^1 rules expresses that strict variables must be used, while undetermined variables in Γ or irrelevant variables in Ω can be ignored. Note that there is no rule for irrelevant variables, which expresses that they cannot be used.

The introduction rules for undetermined, invariant, and strict functions simply add a variable to the appropriate context and check the body of the function.

The difficult rules are the three elimination rules. First, the undetermined context Γ is always propagated to both premises. This reflects that we place no restriction on the use of these variables.

Next we consider the strict context Δ. Recall that this contains the variables which should occur strictly in a term. An undetermined function $M : A \xrightarrow{u} B$ may or may not use its argument. An occurrence of a variable in the argument to such a function can therefore not be guaranteed to be used. Hence we must require in the rule $\xrightarrow{u} E$ for an application $M\ N^u$ that all variables in Δ occur strictly in M. This ensures at least one strict occurrence in M and no further restrictions on occurrences of strict variables in the argument are necessary. This is reflected in the rule by adding Δ to the undetermined context while checking the argument N. The treatment of the strict variables in the vacuous application $M\ N^0$ is similar.

In the case of a strict application $M\ N^1$ each strict variable should occur strictly in either M or N. We therefore split the context into Δ_M and Δ_N guaranteeing that each variable has at least one strict occurrence in M or N, respectively. However,

$$\frac{c{:}A \in \Sigma}{\Gamma; \Omega; \cdot \vdash c : A}\,Con$$

$$\frac{}{(\Gamma, x{:}A); \Omega; \cdot \vdash x : A}\,Id^u \qquad \textit{no } Id^0 \textit{ rule} \qquad \frac{}{\Gamma; \Omega; x{:}A \vdash x : A}\,Id^1$$

$$\frac{(\Gamma, x{:}A); \Omega; \Delta \vdash M : B}{\Gamma; \Omega; \Delta \vdash \lambda x^u{:}A.\ M : A \stackrel{u}{\to} B} \stackrel{u}{\to} I$$

$$\frac{\Gamma; (\Omega, x{:}A); \Delta \vdash M : B}{\Gamma; \Omega; \Delta \vdash \lambda x^0{:}A.\ M : A \stackrel{0}{\to} B} \stackrel{0}{\to} I$$

$$\frac{\Gamma; \Omega; (\Delta, x{:}A) \vdash M : B}{\Gamma; \Omega; \Delta \vdash \lambda x^1{:}A.\ M : A \stackrel{1}{\to} B} \stackrel{1}{\to} I$$

$$\frac{\Gamma; \Omega; \Delta \vdash M : A \stackrel{u}{\to} B \qquad (\Gamma, \Delta); \Omega; \cdot \vdash N : A}{\Gamma; \Omega; \Delta \vdash M\ N^u : B} \stackrel{u}{\to} E$$

$$\frac{\Gamma; \Omega; \Delta \vdash M : A \stackrel{0}{\to} B \qquad (\Gamma, \Omega, \Delta); \cdot; \cdot \vdash N : A}{\Gamma; \Omega; \Delta \vdash M\ N^0 : B} \stackrel{0}{\to} E$$

$$\frac{(\Gamma, \Delta_N); \Omega; \Delta_M \vdash M : A \stackrel{1}{\to} B \qquad (\Gamma, \Delta_M); \Omega; \Delta_N \vdash N : A}{\Gamma; \Omega; (\Delta_M, \Delta_N) \vdash M\ N^1 : B} \stackrel{1}{\to} E$$

Figure 1: Typing rules for $\lambda^{\stackrel{l}{\to}}$

strict variables can occur more than once, so variables from Δ_N can be used freely in M, and variables from Δ_M can occur freely in N. As before, we reflect this by adding these variables to the undetermined context.

Finally we consider the irrelevant context Ω. Variables declared in Ω cannot be used *except* in the argument to an irrelevant function (which is guaranteed to ignore its argument). We therefore add the irrelevant context Ω to the undetermined context when checking the argument of a vacuous application $M\ N^0$.

Our strict λ-calculus satisfies the expected properties, culminating in the existence of canonical forms which is critical for the intended applications. A full account can be found in [11].

Let us examine the structural properties of contexts. Exchange is directly built into the formulation. Weakening is allowed in the undetermined and in the irrelevant contexts. Contraction holds anywhere. We also have the possibility to lose track of occurrences, if we are so inclined. This follows by structural inductions on the typing judgment.

Theorem 3 (Loosening)

1. *If* $\Gamma; (\Omega, x{:}C); \Delta \vdash M : A$ *then* $(\Gamma, x{:}C); \Omega; \Delta \vdash M : A$.

2. *If* $\Gamma; \Omega; (\Delta, x{:}C) \vdash M : A$ *then* $(\Gamma, x{:}C); \Omega; \Delta \vdash M : A$.

The notions of reduction and expansion derive directly from the ordinary β and η rules.

$$\begin{array}{rcl} (\lambda x^k{:}A.\ M)\, N^k & \xrightarrow{\beta} & [N/x]M \\ M : A \xrightarrow{k} B & \xrightarrow{\bar{\eta}} & \lambda x^k{:}A.\ M\ x^k \end{array}$$

The subject reduction and expansion theorems are an immediate consequence of the structural and substitution properties generalized to this three-zoned calculus. The substitution properties again follow by straightforward structural inductions.

Theorem 4 (Subject Reduction and Expansion)

1. *If* $\Gamma; \Omega; \Delta \vdash M : A$ *and* $M \xrightarrow{\beta} M'$ *then* $\Gamma; \Omega; \Delta \vdash M' : A$.

2. *If* $\Gamma; \Omega; \Delta \vdash M : A$ *and* $M : A \xrightarrow{\bar{\eta}} M'$ *then* $\Gamma; \Omega; \Delta \vdash M' : A$.

The above results culminate in the canonical form theorem: for every term M such that $\Gamma; \Omega; \Delta \vdash M : A$, there exists a unique N in canonical (= β-normal η-long) form, denoted $\Gamma; \Omega; \Delta \vdash N \Uparrow A$, such that M is convertible to N. The proof employs the standard method of logical relations, adapted to the specific λ-calculus we consider.

The following lemma establishes a consistency property of the type system relevant to complementation.

Lemma 5 (Exclusivity) *It is not the case that both* $\Gamma; \Omega; (\Delta, x{:}A) \vdash M : C$ *and* $\Gamma; (\Omega, x{:}A); \Delta \vdash M : C$.

5 Embedding the Simply-Typed λ-Calculus

Now that we have developed a calculus which is potentially strong enough to represent the negation of linear patterns, we need to answer two questions: how do we embed the original λ-calculus, and is the calculus now closed under complement?

In this paper, we do not answer the second question conclusively. However, our algorithm is sound and complete for the fragment which results from the natural embedding of the original simply-typed λ-calculus which is sufficient for all applications we have presently considered (see Section 8).

Recall that we introduced strictness to capture occurrence conditions on variables in canonical forms. This means that constants (and by extension bound variables) should be considered *strict* functions of their argument, since these arguments will indeed occur in the canonical form. On the other hand, if we have a second order constant, we cannot restrict the argument function to be either strict or vacuous, since this would render our representations inadequate.

Example 6 *Continuing Example 1, we see*

$$\ulcorner \lambda x.\, \lambda y.\, x \urcorner = lam\ (\lambda x{:}exp.\, lam\ (\lambda y{:}exp.\, x))$$

so the argument to the first occurrence of 'lam' is a strict function, while the argument to the second occurrence is an invariant function. If we can give only one type to 'lam' it must therefore be $(exp \xrightarrow{u} exp) \xrightarrow{1} exp$.

Generalizing this observation means that positive occurrence of function types are translated to strict functions, while negative occurrences to undetermined functions. We can formalize this as an embedding of the simply-typed λ-terms into a fragment of strict terms via two mutually recursive functions $()^-$ and $()^+$. First, the definition on types.

$$\begin{array}{rcl}
(A \rightarrow B)^- & = & A^+ \xrightarrow{u} B^- \\
(A \rightarrow B)^+ & = & A^- \xrightarrow{1} B^+ \\
P^- = P^+ & = & P
\end{array}$$

We extend it to canonical terms (including existential variables), signatures, and contexts as follows:

$$\begin{array}{rcll}
(\lambda x{:}A.\, M)^- & = & \lambda x^u{:}A^+.\, M^- & \\
M^- & = & M^+ & \text{for } M \text{ of base type} \\
x^+ & = & x & \\
c^+ & = & c & \\
(E_A\ x_1 \ldots x_n)^+ & = & F_{A^-}\ x_1^u \ldots x_n^u & \\
(M\ N)^+ & = & M^+\ (N^-)^1 & \\
(\cdot)^+ & = & \cdot & \\
(\Gamma, x{:}A)^+ & = & \Gamma^+, x{:}A^+ & \\
(\Sigma, P{:}type)^+ & = & \Sigma^+, P{:}type & \\
(\Sigma, c{:}A)^+ & = & \Sigma^+, c{:}A^+ &
\end{array}$$

The image of the embedding of the canonical forms of the simply-typed λ-calculus gives rise to the following fragment, which we call *simple terms*.

$$\textit{Simple Terms} \quad M \ ::= \ \lambda x^u{:}A^+.\, M \mid h\ {M_1}^1 \ldots {M_n}^1 \mid E_{A^-}\ x_1^u \ldots x_n^u$$

We sometimes abbreviate $h\ {M_1}^1 \ldots {M_n}^1$ as $h\ \overline{M_n^1}$. Note that by the restriction to long normal forms such terms and applications of pattern variables must be of base type. For every judgment $\mathcal{J}$ on simple terms, we will shorten $\Gamma; \cdot; \cdot \vdash \mathcal{J}$ into $\Gamma \vdash \mathcal{J}$.

The correctness of the embedding is easily established by induction over the structure of canonical forms.

Theorem 7 *If* $\Gamma \vdash M \Uparrow A$ *then* $\Gamma^+ \vdash M^- \Uparrow A^-$.

From now on we drop the $\pm$ embedding annotations from terms, types and contexts. We may also hide the $()^1$ decoration from strict application of constants in examples.

Simple terms enjoy the tightening property which is related to loosening (Theorem 3) and exclusivity (Theorem 5) and is crucial for the correctness proof of our complement algorithm. It expresses that every closed *simple* term is either strict or vacuous in a given undetermined variable. After appropriate generalization, the result follows by induction on the given derivation.

Theorem 8 (Tightening) *Let M be a simple term without existential variables such that $(\Gamma, x{:}C); \Omega; \Delta \vdash M \Uparrow A$. Then $\Gamma; \Omega; (\Delta, x{:}C) \vdash M \Uparrow A$ or $\Gamma; (\Omega, x{:}C); \Delta \vdash M \Uparrow A$.*

It simplifies the presentation of the algorithms for complement and later unification if we extend every existentially quantified variable to be applied to all bound variables in their declaration order. This is possible for simple linear patterns without changing the set of its ground instances. We just insert vacuous applications, which guarantees that the extra variables are not used. We furthermore permute the argument to the standard order, which also does not affect the set of ground instances of linear patterns. In slight abuse of notation we call the resulting terms *fully applied.*

Example 9 *The simple term from Example 2,*

$$lam\ (\lambda x^u : exp.\, app\ E\ x)$$

, has fully applied form

$$lam\ (\lambda x^u : exp.\, app\ (Z\ x^0)\ x)$$

for a fresh existential variable Z of type $exp \xrightarrow{0} exp$.

The translation to fully applied terms is proven correct by induction over the structure of the given canonical form, using a few straightforward lemmas regarding variable occurrences in simple terms. We use the judgment $\Gamma \vdash M \in \|N\| : A$ to denote that a term M of type A is a ground instance of N in context GG.

Theorem 10 *Let N be a simple term and Q its fully applied translation. Then $\Gamma \vdash M \in \|N\| : A$ iff $\Gamma \vdash M \in \|Q\| : A$.*

From now on we tacitly assume that all simple terms are fully applied. We call a term $E\, x_1^{l_1} \ldots x_n^{l_n}$ (of base type) a *generalized variable.*

6 The Complement Algorithm

The idea of complementation for applications and abstractions is quite simple and similar to the first-order case. For generalized variables we consider each argument in turn. If an argument variable is undetermined it does not contribute to the negation. If an argument variable is strict then any term where this variable does not occur contributes to the negation. We therefore complement the corresponding label from 0 to 1 while all other arguments are undetermined. For vacuous argument variables we proceed dually. In preparation for the rules we define.

$$\mathrm{Not}(1) = 0 \quad \mathrm{Not}(0) = 1$$

If $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$, we write $E\ \Gamma^u$ for the application of $E\ {x_1}^u \ldots {x_n}^u$. Such an application represents the set of all terms without existential variables and free variables from Γ.

Definition 11 (Higher-Order Pattern Complement) *Fix a signature Σ. For a linear simple term M such that $\Gamma \vdash M : A$ define $\Gamma \vdash \mathrm{Not}(M) \Rightarrow N : A$ by the following rules. Note that a given M may be related to several terms N all of which belong to the complement of M.*

$$\frac{1 \le i \le n \quad k \in \{1,0\}}{\Gamma \vdash \mathrm{Not}(E\ x_1^{l_1} \ldots x_{i-1}^{l_{i-1}}\ x_i^k\ x_{i+1}^{l_{i+1}} \ldots x_n^{l_n}) \Rightarrow H\ x_1^u \ldots x_{i-1}^u\ x_i^{\mathrm{Not}(k)}\ x_{i+1}^u \ldots x_n^u : P}\ \mathrm{Not\_}Flx$$

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

$$\frac{g \in \Sigma \cup \Gamma, g : A_1 \xrightarrow{1} \ldots \xrightarrow{1} A_m \xrightarrow{1} P, m \ge 0, h \not\equiv g}{\Gamma \vdash \mathrm{Not}(h\ \overline{M_n}^1) \Rightarrow g\ (H_1 \Gamma^u)^1 \ldots (H_m \Gamma^u)^1 : P}\ \mathrm{Not\_}App^1$$

$$\frac{\Gamma \vdash \mathrm{Not}(M_i) \Rightarrow N : A_i \qquad 1 \le i \le n}{\Gamma \vdash \mathrm{Not}(h\ \overline{M_n}^1) \Rightarrow h\ (H_1 \Gamma^u)^1 \ldots (H_{i-1}\Gamma^u)^1\ N^1\ (H_{i+1}\Gamma^u)^1 \ldots (H_n \Gamma^u)^1 : P}\ \mathrm{Not\_}App^2$$

where the H's are fresh generalized variables, $h \in \Sigma \cup \Gamma$, and $\Gamma \vdash h : A_1 \xrightarrow{1} \ldots \xrightarrow{1} A_n \xrightarrow{1} P$. Finally we define $\Gamma \vdash \neg(M) = \mathcal{N} : A$ if $\mathcal{N} = \{N \mid \Gamma \vdash \mathrm{Not}(M) \Rightarrow N : A\}$.

Note that if E_A is a generalized variable considered in the empty context, it has the canonical form $\lambda \overline{x_n^u}.\, E\ \overline{x_n^u}$. Hence $\cdot \vdash \neg(E_A) = \emptyset : A$ as expected.

Example 12 *Let Γ be $x{:}B, y{:}C$:*

$$\begin{array}{rcl} \Gamma \vdash \neg(E\ x^u y^1) & = & \{F\ x^u y^0\} \\ \Gamma \vdash \neg(E\ x^0 y^1) & = & \{F\ x^1 y^u, G\ x^u y^0\} \end{array} \tag{1}$$

We remark that the members of a complement set are not mutually disjoint, due to the indeterminacy of u. We can achieve a representation by exclusive patterns if we resolve this indeterminacy, that is, by considering for every argument x^u the two possibilities x^1 and x^0. Thus, for example, the set from equation (1) may be transformed into

$$\{F\ x^1y^1, G\ x^1y^0, H\ x^0y^0\}$$

It is clear that in the worst case scenario the number of terms in a complement set is bound by 2^n; hence the usefulness of this further step needs to be pragmatically determined.

We can now revisit[2] Example 2:

$$\begin{aligned}
&\neg(lam(\lambda x^u\!:\!exp.\, app\ (E\ x^0)\ x)) = \\
&\quad \{lam(\lambda x^u\!:\!exp.\, app\ (H\ x^1)\ (H'\ x^u)), \\
&\quad lam(\lambda x^u\!:\!exp.\, app\ (H\ x^u)\ (app\ (H'\ x^u)\ (H''x^u)), \\
&\quad lam(\lambda x^u\!:\!exp.\, app\ (H\ x^u)\ (lam(\lambda y^u\!:\!exp.\, H'\ x^u\ y^u)), \\
&\quad lam(\lambda x^u\!:\!exp.\, lam(\lambda y^u\!:\!exp.\, H\ x^u\ y^u)), \\
&\quad lam(\lambda x^u\!:\!exp.\, x), \\
&\quad app\ H\ H'\}
\end{aligned}$$

We can show that simple terms are closed under complementation by induction over the construction of the complement.

Theorem 13 *If M is a simple term, so is N, for all $N \in \Gamma \vdash \neg(M) : A$.*

We address the soundness and completeness of the complement algorithm. Termination is obvious as the algorithm is syntax-directed and only finitely branching. We write $\Gamma \vdash M \in \|\neg(N)\| : A$ if $\Gamma \vdash \mathrm{Not}(N) \Rightarrow Q : A$ and $\Gamma \vdash M \in \|Q\| : A$.

Theorem 14 (Partition) *For a fixed signature Σ, let $\Gamma \vdash N : A$ be a simple linear term. Then:*

1. *(Exclusivity) It is not the case that both $\Gamma \vdash M \in \|N\| : A$ and $\Gamma \vdash M \in \|\neg(N)\| : A$.*
2. *(Exhaustivity) If $\Gamma \vdash M : A$ then either $\Gamma \vdash M \in \|N\| : A$ or $\Gamma \vdash M \in \|\neg(N)\| : A$.*

Note that exclusivity is based on Theorem 5, which holds for *any* strict term. In contrast, exhaustivity requires tightening (Theorem 8) which holds only for simple terms.

7 Unification of Simple Terms

We now address the issue of unification, i.e., intersection of simple linear higher-order patterns. We start by determining when two labels are compatible and define

[2]To avoid too many indices on existential variables, we adopt a convention that the scope of existential variables is limited to each member of a complement set.

their intersection as follows:

$$\begin{aligned} 1 \cap 1 = u \cap 1 = 1 \cap u &= 1 \\ 0 \cap 0 = u \cap 0 = 0 \cap u &= 0 \\ u \cap u &= u \end{aligned}$$

The remaining cases $1 \cap 0$ and $0 \cap 1$ are undefined since no variable can be both strict and vacuous in a given term.

We use Φ for sequences of labeled bound variables and write $\Phi(x) = k$ if $x^k \in \Phi$. We then extend the intersection operations to these contexts:

$$\begin{aligned} \cdot \cap \cdot &= \cdot \\ (x^k\ \Phi) \cap (x^{k'}\ \Phi') &= x^{k \cap k'}\ (\Phi \cap \Phi') \quad \text{if } k \cap k' \text{ is defined.} \end{aligned}$$

Following standard terminology with call atomic terms whose head is a free or bound variable *rigid*, while terms whose head is an existential variable is called *flexible*.

Definition 15 (Higher-Order Pattern Intersection) *Fix a signature Σ. For simple linear terms M and N without shared variables such that $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$, we define $\Gamma \vdash M \cap N \Rightarrow Q : A$ by the following rules.*

$$\frac{}{\Gamma \vdash E\ \Phi_1 \cap F\ \Phi_2 \Rightarrow H\ (\Phi_1 \cap \Phi_2) : P}\ \cap\_FlxFlx$$

no rule for flex-flex with same variable

$$\frac{c \in \Sigma \quad \Gamma \vdash H_1\ \Phi_1 \cap M_1 \Rightarrow N_1 : A_1 \cdots \Gamma \vdash H_n\ \Phi_n \cap M_n \Rightarrow N_n : A_n}{\Gamma \vdash E\ \Phi \cap c\ \overline{M_n^1} \Rightarrow c\ \overline{N_n^1} : P}\ \cap\_FlxRgd^c$$

$$\frac{y \in \Gamma \qquad \Gamma \vdash H_1\ \Phi_1 \cap M_1 \Rightarrow N_1 : A_1 \cdots \Gamma \vdash H_n\ \Phi_n \cap M_n \Rightarrow N_n : A_n}{\Gamma \vdash E\ \Phi \cap y\ \overline{M_n^1} \Rightarrow y\ \overline{N_n^1} : P}\ \cap\_FlxRgd^y$$

$$\frac{h \in \Gamma \cup \Sigma \quad \Gamma \vdash M_1 \cap N_1 \Rightarrow Q_1 : A_1 \cdots \Gamma \vdash M_n \cap N_n \Rightarrow Q_n : A_n}{\Gamma \vdash h\ \overline{M_n^1} \cap h\ \overline{N_n^1} \Rightarrow h\ \overline{Q_n^1} : P}\ \cap\_App$$

$$\frac{\Gamma, x{:}A \vdash M \cap N \Rightarrow Q : B}{\Gamma \vdash \lambda x^u{:}A.\, M \cap \lambda x^u{:}A.\, N \Rightarrow \lambda x^u{:}A.\, Q : A \to B}\ \cap\_Lam$$

where the H's are fresh variables of correct typing and $n \geq 0$.

We have omitted two rules $\cap\_RgdFlx^c$ and $\cap\_RgdFlx^y$ which are symmetric to $\cap\_FlxRgd^c$ and $\cap\_FlxRgd^y$. The rules $\cap\_FlxRgd^c$ and $\cap\_RgdFlx^c$ have the following proviso: for all $x \in \Phi$ and $1 \leq i \leq n$:

$$\begin{aligned} \forall x.\Phi(x) = 0 &\to \forall i\,.\,\Phi_i(x) = 0 \\ \forall x.\Phi(x) = u &\to \forall i\,.\,\Phi_i(x) = u \\ \forall x.\Phi(x) = 1 &\to \exists i\,.\,\Phi_i(x) = 1 \wedge \forall j, j \neq i.\Phi_j(x) = u \end{aligned}$$

The rules $\cap\_FlxRgd^y$ and $\cap\_RgdFlx^y$ are subject to the proviso

$$\forall x.\Phi(x) = 0 \to \forall i\,.\,\Phi_i(x) = 0$$

$$\forall x.\Phi(x) = u \rightarrow \forall i\,.\,\Phi_i(x) = u$$
$$\forall x.x \neq y \wedge \Phi(x) = 1 \rightarrow \exists i\,.\,\Phi_i(x) = 1 \wedge \forall j, j \neq i.\Phi_j(x) = u$$
$$\Phi(y) = u \vee (\Phi(y) = 1 \wedge \forall i.\Phi_i(y) = u)$$

Finally define $\Gamma \vdash M \hat{\cap} N : A = \mathcal{Q}$ *if* $\mathcal{Q} = \{Q \mid \Gamma \vdash M \cap N \Rightarrow Q : A\}$.

Some remarks are in order:

- In rule $\cap\_FlxFlx$ we can assume that the same list of variables, though possibly with different labeling, is the argument of E, F and H, since simple terms are fully applied and due to linearity we can always reorder the context to the same list. Since patterns are linear and M and N share no pattern variables, the flex-flex case arises only with distinct variables. This also means we do not have to apply substitutions or perform the customary occurs-check.
- In the flex/rigid and rigid/flex rules, the proviso enforces the typing discipline since each strict variable x must be strict in some premise. The modified condition on y takes into account that the head of an application constitutes a strict occurrence.

The following example illustrates how the Flex-Rigid rules, in this case $\cap\_FlxRgd^c$, make unification on simple terms finitary.

Example 16 *Consider the unification problem* $x{:}A \vdash E\ x^1 \hat{\cap}\ c\ (F\ x^u)^1\ (F'\ x^u)^1$. *Since* x *is strict in the LHS, there are two ways in which* Φ *can be split leading to the following sub-problems:*

$$\begin{array}{ll} x{:}A \vdash E'\ x^1 \cap\ F\ x^u \Rightarrow H\ x^1 & x{:}A \vdash E''\ x^u \cap\ F'\ x^u \Rightarrow H'\ x^u \\ x{:}A \vdash E'\ x^u \cap\ F\ x^u \Rightarrow H\ x^u & x{:}A \vdash E''\ x^1 \cap\ F'\ x^u \Rightarrow H'\ x^1 \end{array}$$

Hence the result:

$$x{:}A \vdash E\ x^1 \hat{\cap}\ c\ (F\ x^u)^1\ (F'\ x^u)^1 = \{c\ (H\ x^1)^1\ (H'\ x^u)^1, c\ (H\ x^u)^1\ (H'\ x^1)^1\}$$

Note that, similarly to complementation, intersection returns a solution with possible overlaps between the patterns. Again it is possible, in a post-processing phase, to transform the results into disjoint form.

As for complement, the termination of the algorithm is straightforward since the judgment is term-directed with finite branching. The adequacy proof for the unification algorithm on linear simple terms requires two implications which follow inductively, at the heart relying on properties of strict typing derivations.

Theorem 17 (Adequacy of Pattern Intersection) *Fix a signature* Σ. *For simple linear terms* N_1 *and* N_2 *without shared variables such that* $\Gamma \vdash N_1 : A$ *and* $\Gamma \vdash N_2 : A$, $\Gamma \vdash M \in \|N_1\| : A$ *and* $\Gamma \vdash M \in \|N_2\| : A$ *iff* $\Gamma \vdash M \in \|N_1 \hat{\cap} N_2\| : A$.

8 Conclusion

We have shown that the complement of linear higher-order patterns cannot be expressed as a finite union of such patterns. To close the language under complement we generalized to a λ-calculus in which strict and vacuous functions can be expressed in the type system. The original language can be embedded compositionally into *simple patterns*. On this language, both complement and intersection (that is, unification) can be expressed as a finite union of simple linear patterns. From this it is easy to see that finite sets of simple linear patterns form a boolean algebra. Indeed, we can trivially define the *relative complement* of two terms via complement and intersection; namely, extending as expected the latter operation to operate on sets of terms, we have:

$$\mathcal{M} - \mathcal{N} \quad = \quad \mathcal{M} \hat{\cap} \neg \mathcal{N}$$

Our main application lies in higher-order logic programming, where pattern complement is a necessary component in any algorithm to synthesize the negation of a given program. This synthesis includes two basic operations: negation to compute the complements of heads of clauses in the definition of a predicate, and intersection to combine results of negating individual clause heads. In this paper we have provided algorithms to compute both.

While this approach has been investigated for first-order Horn clauses [1] and CLP programs [2], the field has been little explored with respect to higher-order logics and logical frameworks. This is the most important future work we are considering; the algorithms presented here provide the central foundation.

We also plan to extend the results to dependent types to endow intentionally weak frameworks such as Twelf [15] with a logically meaningful notion of negation along the lines of [1]. Finally, we hope to address the general case of complementation and unification in the strict λ-calculus which is likely to be useful in linear logical frameworks as *LLF* [3].

Acknowledgments. We are grateful to Iliano Cervesato, Carsten Schürmann and Roberto Virga for useful comments on previous versions of this paper. This work has been partially supported by NSF Grant CCR-9619584.

References

[1] Roberto Barbuti, Paolo Mancarella, Dino Pedreschi, and Franco Turini. A transformational approach to negation in logic programming. *Journal of Logic Programming*, 8:201–228, 1990.

[2] P. Bruscoli, F. Levi, G. Levi, and M. C. Meo. Compilative constructive negation in constraint logic programs. In Sophie Tison, editor, *Proc. Trees in Algebra and Programming - CAAP'94, 19th International Colloquium*, volume 787, pages 52–76. Springer, 1994.

[3] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer*

Science, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[4] H. Comon. Disunification: a survey. In J-L. Lassez and G.Plotkin, editors, *Computational Logic*. MIT Press, Cambridge,MA, 1991.

[5] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. In *Proc. Seventh International Conference on Rewriting Techniques and Applications (RTA'96)*, pages 138–152. Springer LNCS 1103, 1996.

[6] K. Kunen. Answer sets and negation-as-failure. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming (ICLP '87)*, pages 219–228, Melbourne, Australia, May 1987. MIT Press.

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

[8] D. Lugiez. Positive and negative results for higher-order disunification. *Journal of Symbolic Computation*, 20(4):431–470, October 1995.

[9] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its metatheory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[10] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[11] Alberto Momigliano. *Elimination of negation in a logical framework*. PhD thesis, Carnegie Mellon University, Forthcoming.

[12] Tobias Nipkow. Orthogonal higher-order rewrite systems are confluent. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 306–317, Utrecht, The Netherlands, May 1993.

[13] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[14] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

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