Constructive Negation Using Typed Existence Properties

John G. Cleary and Lunjin Lu

Department of Computer Science
University of Waikato
Hamilton, New Zealand
{jcleary,lunjin}@cs.waikato.ac.nz
Phone: +64-7-8384627
Fax: +64-7-8384155

Abstract. A method for extracting positive information from negative goals is proposed. The method makes use of knowledge about the existence of solutions of predicates and about the types of variables. It strictly generalises earlier work in this area. It can be used as a construction rule or as a simplification rule. As a construction rule it does not involve an SLD-derivation of the negated sub-goal and so is applicable during compilation. As a simplification rule it prunes unsatisfiable goals without doing an explicit satisfiability test.
Keywords: constructive negation, *existence* properties, type system

1 Introduction

The "negation by failure" rule, NAF for short, may lead to floundering when negative goals contain variables. The reason is that NAF doesn't allow negative goals to bind variables. To overcome this limitation, Chan introduces the "constructive negation" rule which allows non-ground negative goals to bind variables in the same way as positive ones [2, 3]. The basic idea is that answers to $\neg Q$ are obtained by negating answers to Q. Given $\neg Q$, a frontier of a derivation tree for Q is first obtained. Answers to $\neg Q$ are then obtained from the frontier as first-order formulae which are interpreted in Clark's equality theory (CET). Chan's method was formulated for logic programs in the Herbrand universe and involves introducing disequality constraints over the Herbrand universe. An answer to a goal by Chan's operational semantics SLD-CNF is a set of equality and disequality constraints. Originally, Chan's method applied only to negative goals with finite sub-derivation trees and worked by negating answers to the negated sub-goal [2]. Chan later extended his method by negating a frontier of a derivation tree for the negated sub-goal [3]. The simplification procedure in Chan's method relies on the following property of the Herbrand universe.

$$\neg\exists \mathbf{y},\mathbf{z}.(x = s(\mathbf{y}) \wedge Q(\mathbf{y},\mathbf{z})) \leftrightarrow \forall \mathbf{y}.(x \neq s(\mathbf{y})) \vee \exists \mathbf{y}.(x = s(\mathbf{y}) \wedge \neg\exists \mathbf{z}.Q(\mathbf{y},\mathbf{z}))$$

where x is a free variable, bold letters denote vectors of variables, $\mathbf{y}$ and $\mathbf{z}$ are disjoint and variables in $\mathbf{z}$ don't occur in $\mathbf{y}$.

Foo et. al propose an approach for constructive negation for Datalog programs [10]. A Datalog program with negation is first transformed by adding typing goals before its negative goals. Each argument of a negated sub-goal corresponds to a typing goal, called the relevant type of the argument. The typing goal enumerates all possible values for the argument. In this way, the negative goals in the transformed program are always ground when it is selected and floundering is avoided. The relevant type for a program is a superset of the success set of the program and is derived from the program.

Małuszyński and Näslund put forward another approach to constructive negation which allows a negative goal to directly return fail substitutions, as its answers [14]. Since answers to negative goals can't in general be represented by a finite number of substitutions, Małuszyński and Näslund's approach sometimes needs to return an infinite number of fail substitutions.

Drabent defines SLDFA resolution over the Herbrand universe [8]. Chan's first method works only when the negated sub-goal has a finite number of answers. SLDFA overcomes this by constructing answers for the negative goal from a finite number of answers to the negated sub-goal.

Fage proposes a simple concurrent pruning mechanism over standard SLD derivation trees for constructive negation in constraint logic programs [9]. Two derivation trees are concurrently constructed. The computed answers from one of the trees are used to prune the nodes of the other. Fage's method admits an efficient implementation as it is not necessary to deal with complex goals with explicit quantifiers outside the constraint part.

Stuckey provides a constructive negation method for constraint logic programs over arbitrary structures [16]. Stuckey's method which is sound and complete with respect to the three-valued consequences of the completion of the program can be thought of as a generalisation of Chan's. Stuckey uses the following property of logic formulae in his simplification procedure.

$$\neg\exists \mathbf{y}.(c \wedge Q) \leftrightarrow \neg\exists \mathbf{y}.c \vee \neg\exists \mathbf{y}.(c \wedge Q)$$

where c is a constraint over a pre-defined structure and Q is a conjunction of goals. The method needs to do a satisfiability test when combining $\neg\exists \mathbf{y}.c$ with other constraints.

Cleary makes use of domain knowledge and *existence* properties of arithmetic constraints to construct answers to negative goals [4]. There are usually functional dependencies between arguments to an arithmetic constraint. Let $add(x, y, z)$ denote addition on the domain of integers, for any integers x and y, then there is a unique z such that $add(x, y, z)$ is true. This is called an *exists unique* property. It implies that $\neg\exists z.add(x, y, z)$ is unsatisfiable and that $\neg\exists z.(add(x, y, z) \wedge q(z))$ can be directly simplified to $add(x, y, z) \wedge \neg q(z)$. Another kind of property is called the *exists sometimes* property which corresponds to partial functional dependencies between arguments to an arithmetic constraint. Let $log(y, x)$ denote $y = 10^x$ on the domain of integers, for any integer y. Then there is at most one x such that $log(y, x)$ is true. So, we can directly simplify $\neg\exists x.(log(y, x) \wedge q(x))$ to $\neg\exists x.log(y, x) \vee log(y, x) \wedge \neg q(x)$. Yet another kind of property is called the *exists* property. For instance, for any integer x, there are integers y and z such that $add(x, y, z)$ is true. *Exists* properties can be used to derive falsity of negative goals such as $\neg\exists y, z.add(x, y, z)$. [4] offers a rewrite rule for each of these three kinds of property. The simplification procedure in [4] consists of rewrite rules for these kinds of property plus some miscellaneous rewrite rules such as $\neg(x > y) \leftrightarrow (x \leq y)$ provided x and y range over the domain of numbers.

Unlike other constructive negation methods [3, 8, 9, 14, 16], the method in [4] does not necessarily rely on SLD resolution to obtain a frontier of the negated sub-goal, and can be used both as a constructive rule and as a simplification rule. Though the method was proposed for arithmetic constraints, the basic principles behind the method carries over to other goals so long as they satisfy some available *existence* properties.

The prerequisite that a functional or partial functional dependency exists between arguments to a predicate (arithmetic constraints in [4]) is over restrictive. Consider $sq(x, y)$ in the domain of real numbers. For every x, there is a unique y such that $sq(x, y)$ is true. However, for every $y > 0$, there are two x's such that $sq(x, y)$ is true. The rewrite rule for *exists unique* properties in [4] doesn't apply directly when it comes to constructing answers to $\neg\exists x.(sq(x, y) \wedge b(x))$. [4] resolves this problem by inserting a tautology $(x \geq 0 \vee x < 0)$ into the negative goal and transforming $\neg\exists x.(sq(x, y) \wedge b(x))$ into $\neg\exists x_1.(sq(x_1, y) \wedge x_1 \geq 0 \wedge b(x_1)) \wedge \neg\exists x_2.(sq(x_2, y) \wedge x_2 < 0 \wedge b(x_2))$ and then applies the rewrite rule for *exists unique* properties to the two negative sub-goals. This causes difficulty because we need to have *exists unique* properties for complex constraints

$(sq(x_1, y) \wedge x_1 \geq 0)$ and $(sq(x_2, y) \wedge x_2 < 0)$. Moreover, inserting a correct tautology, say $(x \geq 0 \vee x < 0)$, into the negative goal before rewriting is involved and difficult to mechanise.

Types are essential in any practical implementation of the constructive negation method in [4] as it is necessary to constrain the range of input and output values in *existence* properties. Though [4] allows the use of types in *existence* properties in the context of arithmetic constraints, it does not provide a theoretical framework for doing so.

This paper generalises the constructive negation method in [4] by generalising the notion of an *existence* property and incorporating a type system. In a generalised *existence* property, an input value may now correspond to multiple output values provided each of the output values can be isolated into a subdomain. Thus, the generalised method is applicable to more negative goals than the original one. A typed version of *existence* properties and rewrite rules are formulated by expressing (sub-)domains as types. Types also allow *existence* properties to be more concise and precise. The generalised method can be used either as a construction rule or as a simplification rule. As a construction rule, it does not involve sub-SLD derivation of the negated sub-goal. As a simplification rule, it prunes unsatisfiable goals without doing an explicit satisfiability test. We also note that Chan's simplification rule can be fully characterised by a set of *existence* properties.

The rest of the paper is organised as follows. Section 2 informally presents the type system. Section 3 formulates the typed version of *existence* properties and rewrite rules and concludes with a number of examples. Section 4 concludes the paper.

2 Type System

This section informally describes a type system which supports inclusion, parametric and overloading polymorphism. The type system is expressive and decidable. A formal presentation can be found in the full version of the paper [5]. The reader is referred to [15] for more on type systems in logic programming.

A *type* is a finite expression denoting a possibly infinite set of ground terms. Specifically, a type denotes a set of ground terms which is a regular term language. We first introduce the notion of a type program. A type term is a term constructible from a denumerable set Para of *type parameters* β, β_i, a set Cons of *type constructors*. Cons is divided two groups: Cons_f containing type constants with fixed denotations and Cons_d containing type constructors whose denotations are determined by type definitions. Cons_f contains two special type constants none denoting the empty set of ground terms and all denoting the Herbrand universe $\mathcal{HU}$.

Example 1. The following will be used to illustrate the type system. Let

$$\mathsf{Cons}_f = \{\mathsf{real}, \mathsf{int}, \mathsf{none}, \mathsf{all}\}$$

$$\cup\{\mathsf{int}_{L,H} \mid L, H \text{ are integers} \wedge L \leq H\}$$
$$\cup\{\mathsf{real}_{L,H} \mid L, H \text{ are real numbers} \wedge L \leq H\}$$

The type constants in Cons_f have the following fixed denotations. real denotes $\{-\infty, \infty\}$ plus the set of reals, int denotes $\{-\infty, \infty\}$ plus the set of integers, $\mathsf{int}_{L,H}$ denotes the integer interval $[L..H]$ and $\mathsf{real}_{L,H}$ denotes the closed real interval $[L..H]$.

Following [11] and [13], we define types by type clauses. A type clause is either of the form $f(x_1, \cdots, x_n) : c(\beta_1, \cdots, \beta_m) \leftarrow x_1 : \mathcal{G}_1, \cdots, x_n : \mathcal{G}_n$ or of the form $x : c(\beta_1, \cdots, \beta_m) \leftarrow x : d(\beta'_1, \cdots, \beta'_l)$ where $m, n, l \geq 0$, $c/m \in \mathsf{Cons}_d$, $d/l \in \mathsf{Cons}$, f/n is a function symbol, $\beta_1, \cdots, \beta_m$ are different type parameters, $x_1, \cdots, x_n$ are different program variables, each β'_j is a type parameter, and each $\mathcal{G}_j$ is either a type parameter or a type constructor applied to type parameters. Type clauses are required to be *type preserving* [17] in that any type parameter occurring in the body also occurs in the head.

Example 2. Let $\mathsf{Cons}_d = \{\mathsf{nat}, \mathsf{man}, \mathsf{woman}, \mathsf{person}, \mathsf{list}\}$. The programmer may define types by the following.

$0 : \mathsf{nat} \leftarrow$	% $D1$	$peter : \mathsf{man} \leftarrow$	% $D5$
$s(x) : \mathsf{nat} \leftarrow x : \mathsf{nat}$	% $D2$	$john : \mathsf{man} \leftarrow$	% $D6$
$[\,] : \mathsf{list}(\beta_1) \leftarrow$	% $D3$	$mary : \mathsf{woman} \leftarrow$	% $D7$
$[h\|l] : \mathsf{list}(\beta_2) \leftarrow$	% $D4$	$x : \mathsf{person} \leftarrow x : \mathsf{man}$	% $S1$
$\quad h : \beta_2, l : \mathsf{list}(\beta_2)$		$x : \mathsf{person} \leftarrow x : \mathsf{woman}$	% $S2$

D1 says that 0 is a member of type nat. D2 means that if x has type nat then $s(x)$ has type nat. D3 states that $[\,]$ is a member of type $\mathsf{list}(\beta_1)$ for any type β_1 while D4 reads that, for any type β_2, if h is a member of type β_2 and l a member of type $\mathsf{list}(\beta_2)$ then $[h|l]$ is a member of type $\mathsf{list}(\beta_2)$. D5 and D6 mean that $peter$ and $john$ are members of man while D7 means that $mary$ is a member of woman. S1 and S2 declare that man and woman are subtypes of person.

User-defined type clauses induce a type program Φ as in [7]. Φ consists of the user-defined type clauses, $f(x_1, \cdots, x_n) : \mathsf{all} \leftarrow x_1 : \mathsf{all}, \cdots, x_n : \mathsf{all}$ for each function symbol f/n, and type clauses $t : c_f \leftarrow$ for any other $c_f \in \mathsf{Cons}_f$ and each constant t of type c_f.

A monomorphic type term $\mathcal{M}, \mathcal{N}$ is type term without any type parameter. The meaning $\mathfrak{F}_\Phi(\mathcal{M})$ of a monomorphic type term $\mathcal{M}$ with respect to a type program Φ is defined as the set of ground terms t such that $t : \mathcal{M}$ is a logical consequence of Φ.

$$\mathfrak{F}_\Phi(\mathcal{M}) \stackrel{def}{=} \{t \in \mathcal{HU} \mid \Phi \models t : \mathcal{M}\}$$

$\mathfrak{F}_\Phi(\mathcal{M})$ is a regular term language, according to lemma 1 in appendix. In the sequel, we shall assume a fixed Φ and drop Φ from $\mathfrak{F}_\Phi$.

We now introduce two mutually recursively defined notions of a typed term and a type. A term is typed by typing its variables. Let $x : \sigma$ denote that variable x is of type σ. $x : \sigma$ means that any term assigned to the variable x must be a member of σ. As its variables are constrained by types, possible instances of a term are restricted. For example, $f(a)$ is not a legitimate instance of $f(x : \mathsf{int})$ since a is not an integer. If a variable in a term is not typed, it can be thought of as being typed by all.

A special function value maps a typed term to a type, called a singleton type. $\mathsf{value}(T)$ with T being a typed term is treated as a set expression [1].

$$\gvertneqq(\mathsf{value}(x : \sigma)) \stackrel{def}{=} \gvertneqq(\sigma)$$

$$\gvertneqq(\mathsf{value}(f(T_1, \cdots, T_n))) \stackrel{def}{=} \{f(t_1, \cdots, t_n) \mid t_i \in \gvertneqq(\mathsf{value}(T_i))\}$$

$\gvertneqq(\mathsf{value}(T))$ is a regular term language of ground terms, see lemma 2 in appendix.

A type is an expression consisting of monomorphic type terms, singleton types, and boolean operators $\mathsf{and}, \mathsf{or}$ and not. Notice that the use of boolean operators are restricted. For instance, $\mathsf{list}(\mathsf{person\ or\ nat})$ is not a type. or, and and not are defined respectively as set union, set intersection and set complement.

$$\begin{aligned}
\gvertneqq(\sigma_1 \ \mathsf{and}\ \sigma_2) &\stackrel{def}{=} \gvertneqq(\sigma_1) \cap \gvertneqq(\sigma_2) \\
\gvertneqq(\sigma_1 \ \mathsf{or}\ \sigma_2) &\stackrel{def}{=} \gvertneqq(\sigma_1) \cup \gvertneqq(\sigma_2) \\
\gvertneqq(\mathsf{not}(\sigma)) &\stackrel{def}{=} \mathcal{HU} \setminus \gvertneqq(\sigma)
\end{aligned}$$

Define $\sigma \sqsubseteq \eta \stackrel{def}{=} \gvertneqq(\sigma) \subseteq \gvertneqq(\eta)$ and $\sigma \equiv \eta \stackrel{def}{=} \gvertneqq(\sigma) = \gvertneqq(\eta)$. Types are regular term languages since monomorphic type terms, singleton types are regular term languages which are are closed under set intersection, union and complement. The inclusion and equivalence problems of regular term languages are known to be decidable [12]. So, the validities of $\sigma \sqsubseteq \eta$ and $\sigma \equiv \eta$ can be effectively tested.

Example 3. Continue examples 1 and 2. We have

$$\begin{aligned}
\gvertneqq(\mathsf{person}) &= \{peter, john, mary\} \\
\gvertneqq(\mathsf{list}(\mathsf{none})) &= \{[\,]\} \\
\gvertneqq(\mathsf{list}(\mathsf{all})) &= \{[\,], \cdots, [x], \cdots, [y, s(0)], \cdots\}
\end{aligned}$$

$\mathsf{real} \equiv \mathsf{real}_{-\infty,\infty}$, $\mathsf{int} \equiv \mathsf{int}_{-\infty,\infty}$, and $\mathsf{int}_{0,\infty}$ denotes the set of non-negative integers. $[\,]$ is in $\gvertneqq(\mathsf{list}(\mathcal{M}))$ for any monomorphic type term $\mathcal{M}$. Note that $\mathsf{nat} \not\equiv \mathsf{int}_{0,\infty}$ as $\gvertneqq(\mathsf{nat})$ and $\gvertneqq(\mathsf{int}_{0,\infty})$ are different sets of terms. Open real intervals can be expressed by type expressions. For instance, $\mathsf{real}_{0,1}$ and $\mathsf{not}(\mathsf{value}(0))$ is a type expression denoting the set of real numbers from 0 to 1 but excluding 0.

In the next section, we shall use the following abbreviations for type expressions. $\mathsf{real}_{>0}$ stands for $(\mathsf{real}_{0,\infty}$ $\mathsf{and\ not}$ $\mathsf{value}(0))$ that denotes the set of real numbers greater than zero, $\mathsf{real}_{<0}$ for $(\mathsf{real}_{-\infty,0}$ $\mathsf{and\ not}$ $\mathsf{value}(0))$ that the set of real numbers less than zero, $\mathsf{real}_{\geq 0}$ for $\mathsf{real}_{0,\infty}$, $\mathsf{int}_{>0}$ for $\mathsf{int}_{1,\infty}$, and $\mathsf{int}_{<0}$ for $\mathsf{int}_{-\infty,-1}$.

3 Rewrite Rules

This section revises Cleary's rewrite rules for constructive negation [4]. One rewrite rule applies when it is known that for every input value a predicate holds for exactly one output value. Another applies when it is known that for every input value a predicate holds for at most one output value. It is not necessary to have the output value available in order to apply these two rewrite rules. What these two rewrite rules make use of is knowledge of whether for every input value a predicate holds for exactly one output value or for at most one output value.

We first revise these two rewrite rules by relaxing their application conditions. The rewrite rule making use of *exists unique* properties is revised as follows. For every input value, the predicate holds for a fixed number of output values each of which can be isolated into a sub-domain. For instance, each positive number has two square roots one of which is positive and the other is negative. The rewrite rule making use of *exists sometimes* properties is revised in the same manner, so that every input value has at most one output value on each of a fixed number of sub-domains. We then formalise a typed version of the revised rewrite rules.

3.1 Basic Rewrite Rules

Exists unique Cleary's rewrite rule for *exists unique* properties requires that, for every $\mathbf{u}$, there be exactly one $\mathbf{x}$ such that $A(\mathbf{ux})$ holds. In other words, "A" must be a function from the domain of $\mathbf{u}$ to that of $\mathbf{x}$. $\mathbf{u}$ and $\mathbf{x}$ can be viewed as input and output parameters respectively. The predicate "A" may satisfy other *exists unique* properties with different groups of input and output parameters.

The revised rewrite rule for *exists unique* properties requires that there be a fixed number of sub-domains such that, for any $\mathbf{u}$, there is exactly one $\mathbf{x}$ in each of those sub-domains such that $A(\mathbf{ux})$ holds. It also requires that, for any $\mathbf{u}$, any $\mathbf{x}$ such that $A(\mathbf{ux})$ holds lies in one of those sub-domains. Let $D_i(\mathbf{x}), i \in I$ be sub-domains (type constraints) where I is a finite set of indices. The above requirement is formally stated as follows.

$$\forall \mathbf{ux}.[A(\mathbf{ux}) \rightarrow \vee_{i \in I} D_i(\mathbf{x})] \tag{1}$$

$$\forall \mathbf{u}. \wedge_{i \in I} \exists ! \mathbf{x}_i.[A(\mathbf{ux}_i) \wedge D_i(\mathbf{x}_i)] \tag{2}$$

where $\exists!$ means "there is exactly one".

The revised rewrite rule is obtained as follows. From (1), we have $A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \leftrightarrow A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge [\vee_{i \in I} D_i(\tilde{\mathbf{x}})]$ where bold letters with tildes denote vectors of terms. Hence $A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge Q \leftrightarrow \vee_{i \in I} A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge D_i(\tilde{\mathbf{x}}) \wedge Q$. Distributing $\exists$ over $\vee$, renaming existentially quantified variables within their scopes and applying De Morgan's law result in the following.[1]

$$\neg \exists \tilde{\mathbf{x}}\tilde{\mathbf{y}}.[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge Q] \leftrightarrow \wedge_{i \in I} \neg \exists \tilde{\mathbf{x}}_i \tilde{\mathbf{y}}.[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}_i) \wedge D_i(\tilde{\mathbf{x}}_i) \wedge Q[\tilde{\mathbf{x}} := \tilde{\mathbf{x}}_i]]$$

[1] $\tilde{\mathbf{x}}$ has been renamed into $\tilde{\mathbf{x}}_i$ for each sub-domain D_i.

where $Q[\tilde{\mathbf{x}} := \tilde{\mathbf{x}}_i]$ is the result of substituting $\tilde{\mathbf{x}}_i$ for $\tilde{\mathbf{x}}$ in Q. From (2), we obtain the following revised rewrite rule where $\mathbf{V}_E$ is the set of variables in E.

Q	Given (1) and (2) and $\mathbf{V}_{\tilde{\mathbf{u}}} \cap (\tilde{\mathbf{x}} \cup \tilde{\mathbf{y}}) = \emptyset$
	$\neg\exists\tilde{\mathbf{x}}\tilde{\mathbf{y}}.[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge Q] \leftrightarrow \wedge_{i\in I}(A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}_i) \wedge D_i(\tilde{\mathbf{x}}_i) \wedge \neg\exists\tilde{\mathbf{y}}.Q[\tilde{\mathbf{x}} := \tilde{\mathbf{x}}_i])$

The condition $\mathbf{V}_{\tilde{\mathbf{u}}} \cap (\tilde{\mathbf{x}} \cup \tilde{\mathbf{y}}) = \emptyset$ ensures that $\tilde{\mathbf{u}}$ does not contain existentially quantified variables also called *local* variables. That it is necessary is shown as follows. Assume the *exists unique* property for integer addition in the introduction, $\neg\exists y.(add(x,y,y) \wedge q(y))$ cannot be simplified to $add(x,y,y) \wedge \neg q(y)$ because $\neg\exists y.add(x,y,y)$ holds for $x \neq 0$. The fact that the second argument y to add is a local variable invalidates the condition. The sub-domains needn't be disjoint so as to apply the revised rewrite rule though it is computationally more efficient to have disjoint sub-domains.

Example 4. The fact that, in the domain of real numbers, a positive number has exactly one negative square root and exactly one positive square root can be expressed as the following *exists unique* property.

$$\forall y > 0.\forall x.(sq(x,y) \rightarrow x < 0 \vee x > 0)$$
$$\forall y > 0.(\exists! x_1.(sq(x_1,y) \wedge x_1 < 0) \wedge \exists! x_2.(sq(x_2,y) \wedge x_2 > 0))$$

Suppose $y > 0$, the negative goal $\neg\exists x.(sq(x,y) \wedge b(x))$ is rewritten into $sq(x_1,y) \wedge x_1 < 0 \wedge \neg\, b(x_1) \wedge sq(x_2,y) \wedge x_2 > 0 \wedge \neg\, b(x_2)$.

Exists sometimes The revision of Cleary's rewrite rule for *exists sometimes* properties is similar to that for *exists unique* properties. (2) is now replaced by the following requirement.

$$\forall \mathbf{u}.\ \wedge_{i\in I}\ \exists?\mathbf{x}_i.[A(\mathbf{u}\mathbf{x}_i) \wedge D_i(\mathbf{x}_i)] \tag{3}$$

where $\exists?$ denotes "there is at most one". (3) requires that, for each $\mathbf{u}$, there is at most one $\mathbf{x}$ in each sub-domain such that $A(\mathbf{ux})$ holds. The revised rewrite rule for *exists sometimes* properties follows.

S	Given (1) and (3) and $\mathbf{V}_{\tilde{\mathbf{u}}} \cap (\tilde{\mathbf{x}} \cup \tilde{\mathbf{y}}) = \emptyset$
	$\neg\exists\tilde{\mathbf{x}}\tilde{\mathbf{y}}.[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge Q] \leftrightarrow$
	$\wedge_{i\in I}(\neg\exists\tilde{\mathbf{x}}_i.[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}_i) \wedge D_i(\tilde{\mathbf{x}}_i)] \vee A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}_i) \wedge D_i(\tilde{\mathbf{x}}_i) \wedge \neg\exists\tilde{\mathbf{y}}.Q[\tilde{\mathbf{x}} := \tilde{\mathbf{x}}_i])$

Example 5. The fact that, in the domain of integer numbers, a positive number has at most one negative square root and at most one positive square root can be expressed as the following *exists sometimes* property.

$$\forall y > 0.\forall x.(sq(x,y) \rightarrow x < 0 \vee x > 0)$$
$$\forall y > 0.(\exists? x_1.(sq(x_1,y) \wedge x_1 < 0) \wedge \exists? x_2.(sq(x_2,y) \wedge x_2 > 0))$$

Suppose $y > 0$, the negative goal $\neg\exists x.(sq(x,y) \wedge b(x))$ is rewritten into $(\neg\exists x_1.(sq(x_1,y) \wedge x_1 < 0) \vee sq(x_1,y) \wedge x_1 < 0 \wedge \neg\, b(x_1)) \wedge (\neg\exists x_2.(sq(x_2,y) \wedge x_2 > 0) \vee sq(x_2,y) \wedge x_2 > 0 \wedge \neg\, b(x_2))$.

Exists There is no revision to *exists* properties and corresponding rewrite rules at this stage as the knowledge of subdomains in which output values lie is irrelevant.

E	Given $\forall \mathbf{u}.\exists \mathbf{x}.A(\mathbf{ux})$ and $\mathbf{V}_{\tilde{\mathbf{u}}} \cap \tilde{\mathbf{x}} = \emptyset$
	$\neg\exists\tilde{\mathbf{x}}.A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \leftrightarrow \mathsf{false}$

Miscellaneous The miscellaneous rewrite rule makes use of properties $\forall \mathbf{u}.(\neg A(\mathbf{u}) \leftrightarrow B(\mathbf{u}))$ to rewrite a negative goal into a positive one.

R	Given $\forall \mathbf{u}.(\neg A(\mathbf{u}) \leftrightarrow B(\mathbf{u}))$
	$\neg A(\tilde{\mathbf{u}}) \leftrightarrow B(\tilde{\mathbf{u}})$

3.2 Typed Rewrite Rules

Types make concise the expression of an *existence* property as domains are types and sub-domains are subtypes.

Example 6. The *exists unique* property in example 4 can be expressed as follows. [2]

$$\forall y{:}\mathsf{real}_{>0}.\forall x.(sq(x,y) \rightarrow x \in \mathsf{real}_{>0} \vee x \in \mathsf{real}_{<0})$$
$$\forall y{:}\mathsf{real}_{>0}.(\exists! x_1{:}\mathsf{real}_{>0}.sq(x_1,y) \wedge \exists! x_2{:}\mathsf{real}_{<0}.sq(x_2,y))$$

Sub-domain constraints are expressed as types, which results in a concise expression of the property.

Types also allow precise *existence* properties to be expressed.

Example 7. The fact that the square of any real number is a positive real number is expressed as follows.

$$\forall x{:}\mathsf{real}.\forall y.(sq(x,y) \rightarrow y \in \mathsf{real}_{\geq 0})$$
$$\forall x{:}\mathsf{real}.\exists! y{:}\mathsf{real}_{\geq 0}.sq(x,y)$$

Note that we have restricted the domain of y into $\mathsf{real}_{\geq 0}$ rather than real, which helps avoid the introduction of local variables in some cases as explained later on.

---

[2] The type of a variable in a conjunctive formula follows just the first occurrence of the variable.

The typed version of an *exists unique* property is expressed by the following two equations.

$$\forall \mathbf{u}{:}\boldsymbol{\sigma}.\forall \mathbf{x}.[A(\mathbf{ux}) \rightarrow \vee_{i \in I} \mathbf{x} \in \boldsymbol{\theta}_i] \tag{4}$$

$$\forall \mathbf{u}{:}\boldsymbol{\sigma}.\ \wedge_{i \in I} \exists! \mathbf{x}_i{:}\boldsymbol{\theta}_i.A(\mathbf{ux}_i) \tag{5}$$

Domains in (1) and (2) are replaced by types in (4) and (5). Each $\boldsymbol{\theta}_i$ is a called a *solution subtype* of the output parameter $\mathbf{x}$. Furthermore, input parameters are typed, which uses a type expression to express the condition under which a specific property holds. The typed version of an *exists sometimes* property is expressed by (4) and the following equation.

$$\forall \mathbf{u}{:}\boldsymbol{\sigma}.\ \wedge_{i \in I} \exists ? \mathbf{x}_i{:}\boldsymbol{\theta}_i.A(\mathbf{ux}_i) \tag{6}$$

An example of typed *exists sometimes* properties can be found in example 10.

A typed *exists* property $\forall \mathbf{u}{:}\boldsymbol{\sigma}.\exists \mathbf{x}{:}\boldsymbol{\theta}.A(\mathbf{ux})$ states that for every $\mathbf{u}$ of type $\boldsymbol{\sigma}$ there are some $\mathbf{x}$ of type $\boldsymbol{\theta}$ such that $A(\mathbf{ux})$ holds. For instance, the *append*/3 program satisfies $\forall z{:}\mathsf{list}(\beta).\exists x{:}\mathsf{list}(\beta).y{:}\mathsf{list}(\beta).append(x,y,z)$ which states that every list z can be split into two lists x and y.

A typed miscellaneous property $\forall \mathbf{u}{:}\boldsymbol{\sigma}.(\neg A(\mathbf{u}) \leftrightarrow B(\mathbf{u}))$ states that, for every $\mathbf{u}$ of type $\boldsymbol{\sigma}$, $\neg A(\mathbf{u})$ can be replaced by $B(\mathbf{u})$. For instance, we have $\forall x{:}\mathsf{int}.y{:}\mathsf{int}.(\neg(x < y) \leftrightarrow (x \geq y))$.

As types and subtypes are associated with variables, the constraints that express sub-domains can be eliminated from the right-hand sides of rewrite rules for *existence* properties. So, more concise residual formulae can be obtained by making use of typed *existence* properties. As each variable is now associated with a type, it is necessary to make sure that a local variable is of the right type as an output argument in order to avoiding incorrect rewriting. This is guaranteed by requiring that if the output value lies in a solution subtype of an output parameter then the type of the local variable in the corresponding argument position is a supertype of that solution subtype.

Example 8. The following is an *exists unique* property in the domain of integers.

$$\forall x{:}\mathsf{int}.y{:}\mathsf{int}.\forall z.(add(x,y,z) \rightarrow z \in \mathsf{int})$$
$$\forall x{:}\mathsf{int}.y{:}\mathsf{int}.\exists! z{:}\mathsf{int}.add(x,y,z)$$

It states that, for any integers x and y, there is a unique integer z such that $add(x,y,z)$ is true. It would be wrong to use the *exists unique* property to rewrite $\neg\, \exists z{:}\mathsf{int}_{-\infty,10}.(add(10,y{:}\mathsf{int},z) \wedge b(z))$ into $add(10,y{:}\mathsf{int},z{:}\mathsf{int}_{-\infty,10}) \wedge \neg\, b(z)$. This is because z can take any value in int and $\mathsf{int}_{-\infty,10}$ is not a supertype of int.

The number of solutions to be negated is limited by the number of solution subtypes of the output parameter. Some solution subtypes are not relevant for a particular negative goal. A solution subtype is relevant if and only if it intersects with the type of the local variable in the negative goal. We call an index a *relevant index* if its corresponding solution subtype is relevant. We only need to consider relevant solution subtypes when rewriting the negative goal.

Example 9. Let the negative goal to rewrite be the following.

$$\neg\exists x{:}\mathsf{real}_{\geq 0}.(sq(x, y{:}\mathsf{real}_{>0}) \wedge b(x))$$

From example 6, $sq(x, y{:}\mathsf{real}_{>0})$ has two solutions for x, one of them is in $\mathsf{real}_{<0}$ and the other is in $\mathsf{real}_{>0}$. This suggests that there are two solutions to be negated. But, the type $\mathsf{real}_{\geq 0}$ of the local variable x doesn't intersect with $\mathsf{real}_{<0}$, that is, only solution subtype $\mathsf{real}_{>0}$ is relevant for the negative goal. The negative goal is rewritten to

$$sq(x_1{:}\mathsf{real}_{>0}, y{:}\mathsf{real}_{>0}) \wedge \neg\, b(x_1)$$

since the type $\mathsf{real}_{\geq 0}$ of x is a supertype of the relevant solution subtype $\mathsf{real}_{>0}$.

Let $iarg(\tilde{\mathbf{u}}, \boldsymbol{\sigma}, \tilde{\mathbf{z}}) \stackrel{def}{=} \tilde{\mathbf{u}} \in \boldsymbol{\sigma} \wedge \mathbf{V}_{\tilde{\mathbf{u}}} \cap \tilde{\mathbf{z}} = \emptyset$ where $\tilde{\mathbf{z}}$ is a set of variables which are local when $iarg(\tilde{\mathbf{u}}, \boldsymbol{\sigma}, \tilde{\mathbf{z}})$ is invoked. The above considerations lead to the following typed rewrite rules for *exists unique* and *exists sometimes* properties.

QT	Given (4), (5) and $iarg(\tilde{\mathbf{u}}, \sigma, (\tilde{\mathbf{x}} \cup \tilde{\mathbf{y}})) \wedge \forall i \in J.(\theta_i \sqsubseteq \eta)$ where $J = \{i \in I \mid (\eta \text{ and } \theta_i) \not\equiv \mathsf{none}\}$
	$\neg\exists\tilde{\mathbf{x}}{:}\eta\tilde{\mathbf{y}}{:}\tau.[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge Q] \leftrightarrow \wedge_{j\in J}[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}_j{:}\theta_j) \wedge \neg(\exists\tilde{\mathbf{y}}{:}\tau.Q[\tilde{\mathbf{x}} := \tilde{\mathbf{x}}_j])]$

ST	Given (4), (6) and $iarg(\tilde{\mathbf{u}}, \sigma, (\tilde{\mathbf{x}} \cup \tilde{\mathbf{y}})) \wedge \forall i \in J.(\theta_i \sqsubseteq \eta)$ where $J = \{i \in I \mid (\eta \text{ and } \theta_i) \not\equiv \mathsf{none}\}$
	$\neg\exists\tilde{\mathbf{x}}{:}\eta\tilde{\mathbf{y}}{:}\tau.[A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \wedge Q] \leftrightarrow \wedge_{j\in J}\begin{bmatrix} \neg\exists\tilde{\mathbf{x}}_j{:}\theta_j.A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}_j) \\ \vee \\ A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}_j{:}\theta_j) \wedge \neg(\exists\tilde{\mathbf{y}}{:}\tau.Q[\tilde{\mathbf{x}} := \tilde{\mathbf{x}}_j]) \end{bmatrix}$

The condition $iarg(\tilde{\mathbf{u}}, \boldsymbol{\sigma}, (\tilde{\mathbf{x}} \cup \tilde{\mathbf{y}}))$ in the above two rewrite rules ensures that an input argument is of the type of the corresponding input parameter and it doesn't contain any local variables. The rewrite rules only generate subformulae for relevant solution subtypes which are collected by $J = \{i \in I \mid (\boldsymbol{\eta} \text{ and } \boldsymbol{\theta}_i) \not\equiv \mathsf{none}\}$. For these relevant solution subtypes, the type of a local variable needs be a supertype of the type of the corresponding output parameter. This is guaranteed by $\forall i \in J.(\boldsymbol{\theta}_i \sqsubseteq \boldsymbol{\eta})$.

The following rewrite rule makes uses of typed *exists* properties. It verifies that an input argument is of the type of the corresponding input parameter and that the type of an output argument is a supertype of the type of the corresponding output parameter.

ET	Given $\forall\mathbf{u}{:}\sigma.\exists\mathbf{x}{:}\theta.A(\mathbf{ux})$ and $iarg(\tilde{\mathbf{u}}, \sigma, \tilde{\mathbf{x}}) \wedge \theta \sqsubseteq \eta$
	$\neg\exists\tilde{\mathbf{x}}{:}\eta.A(\tilde{\mathbf{u}}\tilde{\mathbf{x}}) \leftrightarrow \mathsf{false}$

The following miscellaneous rewrite rule verifies that an input argument is of the type of the corresponding input parameter.

RT	Given $\forall\mathbf{u}{:}\sigma.(\neg A(\mathbf{u}) \leftrightarrow B(\mathbf{u}))$ and $\tilde{\mathbf{u}} \in \sigma$
	$\neg A(\tilde{\mathbf{u}}) \leftrightarrow B(\tilde{\mathbf{u}})$

3.3 Variable Introduction

The typed rewrite rules for *exists unique* and *exists sometimes* properties require that input arguments are terms that do not contain any local variables and that are of type $\boldsymbol{\sigma}$. They also require that output arguments are local variables of type $\boldsymbol{\eta}$ such that $\boldsymbol{\theta}_i \sqsubseteq \boldsymbol{\eta}$ for each relevant solution subtype $\boldsymbol{\theta}_i$. If the requirement on output arguments is not met then it is necessary to introduce new local variables so as to apply these rewrite rules. The method in [4] for introducing local variables carries over. To facilitate presentation, let us first reformulate (4), (5) and (6) by splitting $\mathbf{x}$ into $\mathbf{sr}$ where $\mathbf{r}$ are output parameters corresponding to output arguments that satisfy the above requirement while $\mathbf{s}$ are other output parameters. An algorithm for partitioning output parameters $\mathbf{x}$ in an *existence* property into $\mathbf{s}$ and $\mathbf{r}$ so as to match a particular goal is omitted due to limited space.

$$\forall \mathbf{u}{:}\boldsymbol{\sigma}\mathbf{sr}.[A(\mathbf{usr}) \rightarrow \vee_{i\in I}(\mathbf{s} \in \boldsymbol{\psi}_i \wedge \mathbf{r} \in \boldsymbol{\omega}_i)] \tag{7}$$

$$\forall \mathbf{u}{:}\boldsymbol{\sigma}.\ \wedge_{i\in I}\ \exists!\mathbf{s}_i{:}\boldsymbol{\psi}_i\mathbf{r}_i{:}\boldsymbol{\omega}_i.A(\mathbf{us}_i\mathbf{r}_i) \tag{8}$$

$$\forall \mathbf{u}.\ \wedge_{i\in I}\ \exists?\mathbf{s}_i{:}\boldsymbol{\psi}_i\mathbf{r}_i{:}\boldsymbol{\omega}_i.A(\mathbf{us}_i\mathbf{r}_i) \tag{9}$$

Let $\tilde{\mathbf{w}}$ being the set of local variables in $\tilde{\mathbf{s}}$ that do not occur in $\tilde{\mathbf{r}}$. The corresponding typed rewrite rules for *exists unique* and *exists sometimes* properties with introduction of local variables follow.

QVT	Given (7), (8) and $iarg(\tilde{\mathbf{u}}, \sigma, (\tilde{\mathbf{w}} \cup \tilde{\mathbf{r}} \cup \tilde{\mathbf{y}})) \wedge \forall i \in J.(\omega_i \sqsubseteq \gamma)$ where $J = \{i \in I \mid (\gamma \text{ and } \omega_i) \not\equiv \text{none}\}$,
	$\neg\exists\tilde{\mathbf{w}}{:}\zeta\tilde{\mathbf{r}}{:}\gamma\tilde{\mathbf{y}}{:}\tau.[A(\tilde{\mathbf{u}}\tilde{\mathbf{s}}\tilde{\mathbf{r}}) \wedge Q] \leftrightarrow$ $\wedge_{j\in J}\begin{pmatrix} A(\tilde{\mathbf{u}}\tilde{\mathbf{z}}_j{:}\psi_j\tilde{\mathbf{r}}_j{:}\omega_j) \wedge \neg\exists\tilde{\mathbf{w}}_j{:}\zeta.(\tilde{\mathbf{s}}[\tilde{\mathbf{r}}\tilde{\mathbf{w}} := \tilde{\mathbf{r}}_j\tilde{\mathbf{w}}_j] = \tilde{\mathbf{z}}_j) \\ \vee \\ A(\tilde{\mathbf{u}}\tilde{\mathbf{z}}_j{:}\psi_j\tilde{\mathbf{r}}_j{:}\omega_j) \wedge (\tilde{\mathbf{s}}[\tilde{\mathbf{r}}\tilde{\mathbf{w}} := \tilde{\mathbf{r}}_j\tilde{\mathbf{w}}_j] = \tilde{\mathbf{z}}_j) \wedge \neg\exists\tilde{\mathbf{y}}{:}\tau.Q[\tilde{\mathbf{r}}\tilde{\mathbf{w}} := \tilde{\mathbf{r}}_j\tilde{\mathbf{w}}_j] \end{pmatrix}$

SVT	Given (7), (9) and $iarg(\tilde{\mathbf{u}}, \sigma, (\tilde{\mathbf{w}} \cup \tilde{\mathbf{r}} \cup \tilde{\mathbf{y}})) \wedge \forall i \in J.(\omega_i \sqsubseteq \gamma)$ where $J = \{i \in I \mid (\gamma \text{ and } \omega_i) \not\equiv \text{none}\}$,
	$\neg\exists\tilde{\mathbf{w}}{:}\zeta\tilde{\mathbf{r}}{:}\gamma\tilde{\mathbf{y}}{:}\tau.[A(\tilde{\mathbf{u}}\tilde{\mathbf{s}}\tilde{\mathbf{r}}) \wedge Q] \leftrightarrow$ $\wedge_{j\in J}\begin{pmatrix} \neg\exists\tilde{\mathbf{z}}_j{:}\psi_j\tilde{\mathbf{r}}_j{:}\omega_j.A(\tilde{\mathbf{u}}\tilde{\mathbf{z}}_j\tilde{\mathbf{r}}_j) \\ \vee \\ A(\tilde{\mathbf{u}}\tilde{\mathbf{z}}_j{:}\psi_j\tilde{\mathbf{r}}_j{:}\omega_j) \wedge \neg\exists\tilde{\mathbf{w}}_j{:}\zeta.(\tilde{\mathbf{s}}[\tilde{\mathbf{r}}\tilde{\mathbf{w}} := \tilde{\mathbf{r}}_j\tilde{\mathbf{w}}_j] = \tilde{\mathbf{z}}_j) \\ \vee \\ A(\tilde{\mathbf{u}}\tilde{\mathbf{z}}_j{:}\psi_j\tilde{\mathbf{r}}_j{:}\omega_j) \wedge (\tilde{\mathbf{s}}[\tilde{\mathbf{r}}\tilde{\mathbf{w}} := \tilde{\mathbf{r}}_j\tilde{\mathbf{w}}_j] = \tilde{\mathbf{z}}_j) \wedge \neg\exists\tilde{\mathbf{y}}{:}\tau.Q[\tilde{\mathbf{r}}\tilde{\mathbf{w}} := \tilde{\mathbf{r}}_j\tilde{\mathbf{w}}_j] \end{pmatrix}$

In the above (QVT) and (SVT) rewrite rules, $\tilde{\mathbf{r}}_j$, $\tilde{\mathbf{z}}_j$ and $\tilde{\mathbf{w}}_j$ are variables that do not occur in the left-hand sides of the rewrite rules. $\tilde{\mathbf{r}}_j\tilde{\mathbf{z}}_j$ are typed with j^{th} solution subtype while $\tilde{\mathbf{w}}_j$ inherits the type of $\tilde{\mathbf{w}}$. Both (QVT) and (SVT) introduce disequality constraints $\neg\exists\tilde{\mathbf{w}}_j{:}\boldsymbol{\zeta}.(\tilde{\mathbf{s}}[\tilde{\mathbf{r}}\tilde{\mathbf{w}} := \tilde{\mathbf{r}}_j\tilde{\mathbf{w}}_j] = \tilde{\mathbf{z}}_j)$ which are dealt with by augmenting Chan's method with types.

Example 10. The following is the typed version of the *exists sometimes* property in example 5.

$$\forall y{:}\mathsf{int}_{>0}.\forall x.(sq(x,y) \rightarrow x \in \mathsf{int}_{<0} \vee x \in \mathsf{int}_{>0})$$
$$\forall y{:}\mathsf{int}_{>0}.(\exists ?x_1{:}\mathsf{int}_{<0}.sq(x_1,y) \wedge \exists ?x_2{:}\mathsf{int}_{>0}.sq(x_2,y))$$

The local variable x in the negative goal $\neg\exists x{:}\mathsf{int}_{0,20}.(sq(x, y{:}\mathsf{int}_{>0}) \wedge b(x))$ has a type $\mathsf{int}_{0,20}$ which is not a supertype of the sole relevant solution subtype $\mathsf{int}_{>0}$ of the corresponding output parameter. Therefore, a new variable z_2 of type $\mathsf{int}_{>0}$ is introduced and the negative goal is rewritten to the following.

$$\begin{aligned}
&\neg\exists z_2{:}\mathsf{int}_{>0}.sq(z_2, y{:}\mathsf{int}_{>0})\\
\vee\ & sq(z_2{:}\mathsf{int}_{>0}, y{:}\mathsf{int}_{>0}) \wedge\ \neg\exists x{:}\mathsf{int}_{0,20}.(x = z_2)\\
\vee\ & sq(z_2{:}\mathsf{int}_{>0}, y{:}\mathsf{int}_{>0}) \wedge (x{:}\mathsf{int}_{0,20} = z_2) \wedge \neg\ b(z_2)
\end{aligned}$$

Solving the typed equality and the typed disequality constraints rewrites the above formula to the following.

$$\begin{aligned}
&\neg\exists z_2{:}\mathsf{int}_{>0}.sq(z_2, y{:}\mathsf{int}_{>0})\\
\vee\ & sq(z_2{:}\mathsf{int}_{21,\infty}, y{:}\mathsf{int}_{>0})\\
\vee\ & sq(z_2{:}\mathsf{int}_{1,20}, y{:}\mathsf{int}_{>0}) \wedge \neg\ b(z_2)
\end{aligned}$$

There is no rewrite rule with introduction of local variables for *exists* properties because introducing local variables won't lead to simplification. Introduction of local variables is irrelevant to the miscellaneous rewrite rule as miscellaneous properties have no output parameters.

The above rewrite rules can be used to extract positive information from negative goals.

Example 11. The *append*/3 program satisfies the following *exists unique* property.

$$\forall\beta. \begin{bmatrix} \forall x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta).\forall z.(append(x,y,z) \rightarrow z \in \mathsf{list}(\beta)) \\ \wedge \\ \forall x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta)\exists! z{:}\mathsf{list}(\beta).append(x,y,z) \end{bmatrix}$$

which satisfy (7) and (8). Using (QVT) allows

$$\neg\exists z{:}\mathsf{list}(\beta).(append(x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta), z), p(z))$$

to be rewritten as

$$append(x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta), z{:}\mathsf{list}(\beta)), \neg\ p(z)$$

Both Chan's method and Stuckey's first construct an SLD derivation tree of $append(x,y,z), p(z)$ and collect a frontier of the SLD derivation, say,

$$\{(x = [], y = z, p(z)), (x = [h|x'], y = y', z = [h|z'], append(x', y', z'), p(z))\}$$

Then the negation of this frontier is simplified and put into its disjunctive normal form. This gives rise to the following four conjunctive formulae.

(1) $x \neq [], \forall h, x'.(x \neq [h|x'])$
(2) $x \neq [], x = [h|x'], \neg\exists z'.(append(x', y, z'), p([h|z']))$
(3) $x = [], \forall h, x'.(x \neq [h|x']), \neg p(y)$
(4) $x = [], x = [h|x'], \neg p(y), \neg\exists z'.(append(x', y, z'), p([h|z']))$

Stuckey's method derives (2) and (3) because the constraint parts of (1) and (4) are unsatisfiable. Chan's method derives (1),(2) and (3) as it only tests satisfiability of atomic constraints. The constraint part of (4) is failed by unification in Chan's method as [] is not unifiable with $[h|x']$. Neither of these methods is effective as (2) is as complex as the original goal. The *exists unique* property allows us to obtain a simpler derived goal without making use of SLD derivation, and to eliminate unsatisfiable derived goals without satisfiability tests.

As well as rewriting the original goal, our rules can also be used for simplification. This will result in a more efficient simplification procedure since unsatisfiable goals are pruned without doing an explicit satisfiability test.

Example 12. We have $\forall y{:}\mathsf{all}.x.(x = s(y) \rightarrow x{:}\mathsf{all})$ and $\forall y{:}\mathsf{all}.\exists! x{:}\mathsf{all}.(x = s(y))$ in the Herbrand universe. Consider the following program.

```
p(y).
r(y) :- x=s(y),q(x).
```

The goal $p(y{:}\mathsf{all}), \neg\, r(y)$ is reduced to $p(y), \neg\exists x{:}\mathsf{all}.(x = s(y{:}\mathsf{all}), q(x))$ which is then simplified directly into $x{:}\mathsf{all} = s(y), p(y{:}\mathsf{all}), \neg\, q(x)$ using the above property. Without using this property, $\neg\exists x{:}\mathsf{all}.(x = s(y{:}\mathsf{all}), q(x))$ is simplified to

$$\forall x{:}\mathsf{all}.(x \neq s(y{:}\mathsf{all})) \vee (x{:}\mathsf{all} = s(y{:}\mathsf{all}), \neg\, q(x))$$

and a satisfiability test is then used to eliminate $\forall\, x{:}\mathsf{all}.(x \neq s(y{:}\mathsf{all}))$. In this sense, the satisfiability test is pushed into the simplification procedure by the *exists unique* property.

Chan's simplification rule can be formalised by a set of *exists sometimes* properties as follows.

$$\forall x{:}\mathsf{all}.y_1{:}\mathsf{all} \cdots y_n{:}\mathsf{all}.(x = s(y_1, \cdots, y_n) \rightarrow y_1 \in \mathsf{all} \wedge \cdots \wedge y_n \in \mathsf{all})$$
$$\forall x{:}\mathsf{all}.\exists ? y_1{:}\mathsf{all} \cdots y_n{:}\mathsf{all}.(x = s(y_1, \cdots, y_n))$$

These satisfy (7) and (9) and allow (SVT) to be applied.

The constructive negation method is sound as it rewrites a formula to another logically equivalent formula. The issue of completeness doesn't arise when the method is used as a simplication rule in other constructive methods [3, 16, 9]. Used stand-alone, the completeness of the method depends on the completeness of the available typed existence properties. It is part of our ongoing work to develop an algorithm for checking the completeness of the method with respect to a given program and a set of typed existence properties which can be either declared by the programmer or inferred from the program by static analysis [6].

4 Conclusion

The use of *existence* properties in [4] for the constructive negation of arithmetic goals has been generalised. This was done by generalising the notion of *existence* properties and by explicitly using types in generalised *existence* properties. This in turn allows negation of predicates that have multiple possible solutions. It also allows application of the rules to any user-defined predicates whose typed *existence* properties are known. The method need not do an SLD-derivation of the negated sub-goal in order to extract positive information. Because of this, it can be applied at compile time. It can also be used to simplify a frontier of an SLD-derivation tree. When used for simplification, it removes the need for an explicit satisfiability test.

Appendix

Lemma 1. *Given a type program Φ, $\gneqq_{\Phi}(\mathcal{M})$ is a regular term language for any monomorphic type term $\mathcal{M}$.*

The proof is done by constructing a regular term grammar G for $\mathcal{M}$ [12].

The type program Φ is partially evaluated with respect to $\mathcal{M}$ to obtain another program Ψ and Ξ such that $\gneqq_{\Phi}(\mathcal{M}) = \gneqq_{\Psi}(\mathcal{M})$, Ψ is monomorphic in that its type clauses do not contain any type parameter, and Ξ is the set of all monotypes in Ψ. Initially, $\Psi = \emptyset$ and $\Xi = \{\mathcal{M}\}$. Then monotype clauses and monotypes are repeatedly added into Ψ and Ξ respectively until they are saturated. Let $\mathcal{N} \in \Xi$. If $f(X_1, \cdots, X_n) : \mathcal{N} \leftarrow X_1 : \mathcal{N}_1, \cdots, X_n : \mathcal{N}_n$ is an instance of a clause in Φ then $f(X_1, \cdots, X_n) : \mathcal{N} \leftarrow X_1 : \mathcal{N}_1, \cdots, X_n : \mathcal{N}_n$ is added into Ψ and $\mathcal{N}_1, \cdots, \mathcal{N}_n$ are added into Ξ. If $X : \mathcal{N} \leftarrow X : \mathcal{R}$ is an instance of a clause in Φ then $X : \mathcal{N} \leftarrow X : \mathcal{R}$ is added into Ψ and $\mathcal{R}$ into Ξ. Ξ and Ψ are finite as they are only Cons is finite and the height of a monotype in the body of a clause in Ψ is no more than that of the monotype in the head of the same clause. By induction on the number of steps in the resolution of $\leftarrow t : \mathcal{M}$ with Ψ and Φ, it follows that $\gneqq_{\Phi}(\mathcal{M}) = \gneqq_{\Psi}(\mathcal{M})$.

The reguar term grammar G for $\gneqq_{\Psi}(\mathcal{M})$ is $\langle \Xi, \mathsf{Func}, \emptyset, P, \mathcal{M} \rangle$ where Func is the set of function symbols and P consists of the rules obtained from Ψ as follows. If $f(X_1, \cdots, X_n) : \mathcal{N} \leftarrow X_1 : \mathcal{N}_1, \cdots, X_n : \mathcal{N}_n$ is in Ψ then $\mathcal{N} \rightarrow f(\mathcal{N}_1, \cdots, \mathcal{N}_n)$ is in P. If $X : \mathcal{N} \leftarrow X : \mathcal{R}$ is in Ψ then $\mathcal{N} \rightarrow \mathcal{R}$ is in P. The nonterminals of G is Ξ, terminals Func, start symbol $\mathcal{M}$. G has an empty frontier alphabet [12]. By induction, we can prove that $\mathcal{M} \Rightarrow_G^* t$ iff $t \in \gneqq_{\Psi}(\mathcal{M})$. This concludes the proof of the lemma.

Lemma 2. *For any typed term T, if the types of variables in T are regular term languages then $\gneqq(\mathsf{value}(T))$ is a regular term language.*

Let X be the set of variables occurring in T and α be the mapping which maps every variable $x \in X$ into its associated type in T. $\{T\}$ can be thought of as a regular term language with its frontier alphabet being X and its initial assignment being α. $\gneqq(\mathsf{value}(T))$ is obtained from T and α by replacing each *occurrence of a variable x in T with each ground term in $\supsetneq(\alpha(x))$. As $\supsetneq(\alpha(x))$ is a regular term language for each x in T, $\supsetneq(\mathsf{value}(T))$ is also a regular term language [12].*

References

1. A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proceedings of the Seventh IEEE Symposium on Logic in Computer Science*, pages 329–340. The IEEE Computer Society Press, 1992.
2. D. Chan. Constructive negation based on the completed database. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 111–125. The MIT Press, 1988.
3. D. Chan. An Extension of Constructive Negation and its Application in Coroutining. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 477–496, 1989.
4. J.G. Cleary. Constructive negation of arithmetic constraints using data-flow graphs. To appear in Constraints.
5. J.G. Cleary and L. Lu. Constructive negation using typed existence properties. Technical report, Department of Computer Science, University of Waikato, 1997.
6. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.
7. P.W. Dart and J. Zobel. Efficient run-time type checking of typed logic programs. *Journal of Logic Programming*, 14(1-2):31–69, 1992.
8. W. Drabent. What is failure? An approach to constructive negation. *Acta Informatica*, 32:27–59, 1995.
9. F. Fage. Constructive negation by pruning. *Journal of Logic Programming*, 32(2):85–118, 1997.
10. N. Foo, A. Rao, A. Taylor, and A. Walker. Deduced relevant types and constructive negation. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the fifth International Conference and Symposium on Logic Programming*, pages 126–139. The MIT Press, 1988.
11. T. Fruhwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309. The IEEE Computer Society Press, 1991.
12. F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, 1984.
13. M. Kifer and J. Wu. A first-order theory of types and polymorphism in logic programming. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 310–321. The IEEE Computer Society Press, 1991.
14. J. Małuszyński and T. Näslund. Fail Substitutions for Negation as Failure. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 461–476. The MIT Press, 1989.
15. Frank Pfenning, editor. *Types in logic programming.* The MIT Press, Cambridge, Massachusetts, 1992.
16. P.J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118:12–33, 1995.
17. J. Tiuryn. Type inference problems: A survey. In B. Roven, editor, *Proceedings of the Fifteenth International Symposium on Mathematical Foundations of Computer Science*, pages 105–120. Springer-Verlag, 1990.