Implementing Type-Based Constructive Negation

Lunjin Lu
Department Of Computer Science and Engineering
Oakland University
lunjin@acm.org

ABSTRACT

This paper presents an implementation of a constructive negation method. The constructive negation method makes use of type dependencies between arguments of a predicate to rewrite negative goals in a logic program. The constructive negation method is first reformulated as a derivation rule. Then an algorithm for efficiently implementing the derivation rule is presented and its complexity is analyzed.
Keywords: Constructive negation, Logic programs, Types

1. INTRODUCTION

A challenging issue in logic programming is how to find answers to negative goals. The "negation by failure" rule that is adopted in majority of logic programming systems is problematic: it does not allow negative goals to bind variables. Any use of a negative goal other than as a test may lead to non-monotonicity of the system in the sense that the system produces less output information given more input information. To solve this problem, various "constructive negation" methods have been proposed that find solutions to negative goals.

One of the proposed methods makes use of *existence* properties to construct answers to negative goals. It was originally proposed in [9] where *existence* properties are functional dependencies between arguments to a predicate. Let $sq(x, y)$ denote square relation in the domain of real numbers. The predicate $sq(x, y)$ holds iff $x^2 = y$ holds. It has this property.

For any real number x, there is a unique y such that $sq(x, y)$ is true. (P)

Using property (P), $\neg\exists y.(sq(x, y) \wedge q(y))$ is rewritten to $sq(x, y) \wedge \neg q(y)$. Note that the atom $sq(x, y)$ has been extracted out of the scope of negation. Existentially quantified variable are called local and other variables global. The prerequisite is restrictive that a functional dependency exists between arguments to a predicate. Cleary et. al generalize the notion of an *existence* property [10]. An input may now correspond to multiple outputs provided that each output can be isolated into a sub-domain, which make the generalized method applicable to more negative goals. Domain constraints are expressed as type constraints in [10]. Examples of generalized *existence* properties can be found in section 2. The method developed in [9, 10] offer two advantages over other proposed methods. Firstly, the method can be applied in program transformation because it extracts an atom from a negative goal without executing the atom. Other methods execute the atom in order to construct solutions to the negative goal. Secondly, the method may avoid satisfiability tests when used as a simplification procedure (cf. example 7). See section 7 for more detailed discussion about related work.

This paper continues work in [10] and addresses two key issues in implementing the rewrite rules. A crucial task of any implementation of [10] is to introduce new local variables into an atom inside a negative goal so that it satisfies a given *existence* property. Consider $\neg(sq(x, 16) \wedge q(16))$. The atom $sq(x, 16)$ does not satisfy property (P) and hence cannot be extracted. This is because the unique y such that $sq(x, y)$ holds is not necessarily 16. However, $\neg(sq(x, 16) \wedge q(16))$ can be transformed to $\neg\exists y'.(sq(x, y') \wedge 16 = y' \wedge q(16))$ by introducing a new local variable y'. The transformed goal can then be rewritten to $sq(x, y') \wedge \neg(16 = y' \wedge q(16))$ since $sq(x, y')$ satisfies property (P). In general cases, the task of introducing new local variables is much more complicated. We present an algorithm in this paper that tests if an *existence* property can be used to extract an atom by introducing zero or more new local variables.

Another essential issue is how to find quickly an extractable atom inside a negative goal. Let G_i be $sq(x_{i-1}, x_i)$ and G be $\neg\exists x_1.\exists x_2.\cdots\exists x_n.\exists x_{n+1}.[G_n \wedge G_{n-1} \cdots G_2 \wedge G_1]$. By repeatedly using property (P), we can extract from the negative goal G atoms G_1, G_2 to G_{n-1} in order and obtain $G_1 \wedge G_2 \cdots G_{n-1} \wedge \neg\exists x_{n+1}.G_n$. Observe that G_j becomes extractable after and only after x_j becomes global upon extraction of G_{j-1}. We use a digraph to represent a negative goal. The digraph links an atom to a local variables iff the local variable occurs in the atom. This data structure allows efficient identification of extractable atoms. We present an implementation of [10] based on the digraph representation of negative goals and the above mentioned algorithm. This is the main contributions of the paper. In addition, rewrite rules in [10] have been substantially reformulated.

The rest of the paper is organized as follows. Section 2 reformulates the rewrite rules in [10]. Section 3 describes digraphs for representing negative goals and section 4 presents the algorithm for introducing new variables. Section 5 describes the implementation and section 6 analyzes its complexity. Section 7 discusses related work and concludes. Proofs are omitted due to space limits.

1.1 Notations

We assume that variables are typed. Expression $y{:}\eta$ indicates that variable y has type η. A type is a finite expression denoting a possibly infinite set of terms. We use $\mathbf{1}$ to denote the set of all terms and $\mathbf{0}$ to denote the empty set of terms. $\mathcal{R}$ and $\mathcal{Z}$ denote the set of real numbers and the set of integer numbers respectively. $\mathcal{R}$ and $\mathcal{Z}$ with subscripts denote their subtypes. A subscript is either an interval or a logical formula. Relation $\sigma \sqsubseteq \theta$ holds iff σ is a subtype of θ; and relation $\sigma \equiv \theta$ holds iff σ is equivalent to θ. The intersection of two types θ and σ is denoted as $\theta \sqcap \sigma$.

Both existence properties and rewrite rules partition the argument list in an atom into several selections. For an example, let add(x,y,z) denote $x + y = z$ where x, y and z range over the domain of real numbers. For given x and y, there is exactly one z such that add(x,y,z) holds. The input selection π_i consists of the first two arguments x and y and the output selection π_o consists of the third argument z. Formally, a selection is a partial function whose domain is a set of argument positions (positive integers). Thus, $\pi_i = \{1 \mapsto x, 2 \mapsto y\}$ and $\pi_o = \{3 \mapsto z\}$. The domain of a selection π is denoted $dom(\pi)$. The projection of π onto $D \subseteq dom(\pi)$ is denoted $\pi \downarrow D$. Then $(\pi \downarrow D)(i) = \pi(i)$ if $i \in D$. Otherwise, $(\pi \downarrow D)(i)$ is undefined. We call $\pi \downarrow D$ a sub-selection of π and accordingly π is a super-selection of $\pi \downarrow D$. The empty selection is denoted by ϵ. We have $\pi \downarrow \emptyset = \epsilon$ for any selection π. By an element of a selection π, we mean $\pi(i)$ for some $i \in dom(\pi)$. We use $diff(\pi)$ to indicate that elements in π are pair wise different, that is, $diff(\pi)$ is true iff $\pi(p_1) \neq \pi(p_2)$ for any $p_1 \in dom(\pi)$ and any $p_2 \in dom(\pi)$ such that $p_1 \neq p_2$. In the sequel, a letter $\bar{u}$ with an over bar denotes a selection of different variables, a letter with a tilde $\tilde{u}$ denotes a selection of terms and a Greek letter with an over bar denotes a selection of types. A selection of types is also called a type. When there is no ambiguity from the context, $\bar{u}$ is also used to denote the set of variables occurring in $\bar{u}$. By juxtaposition $\pi_1\pi_2$, we mean that π_1 and π_2 have disjoint domains and $\pi_1\pi_2 = \pi_1 \cup \pi_2$. For instance, $\pi_i\pi_o = \pi_o\pi_i = \{1 \mapsto x, 2 \mapsto y, 3 \mapsto z\}$. Let A be of arity n. By $A(\pi)$, we mean that $dom(\pi) = \{1..n\}$ and $A(\pi) = A(\pi(1), \cdots, \pi(n))$. For instance, $add(\pi_i\pi_o)$ stands for $add(x, y, z)$. When it is clear from context, a selection is simplify written as a sequence with positions omitted.

By $\bar{u} : \bar{\sigma}$, we mean that $dom(\bar{u}) = dom(\bar{\sigma})$ and $\bar{u}(i){:}\bar{\sigma}(i)$ for any $i \in dom(\bar{u})$. By $\bar{\sigma} \sqsubseteq \bar{\eta}$, we mean that $dom(\bar{\eta}) = dom(\bar{\sigma})$ and $\bar{\sigma}(i) \sqsubseteq \bar{\eta}(i)$ for all $i \in dom(\bar{\sigma})$. We say that $\bar{\sigma}$ and $\bar{\eta}$ intersect iff $\bar{\sigma}(i) \sqcap \bar{\eta}(i) \not\equiv \mathbf{0}$ for all $i \in dom(\bar{\sigma})$. Let E be an expression. We use $\mathbf{V}_E$ to denote the set of variables in E and $type(E)$ the type of E. As each variable in a term is typed, the type of E is defined. The letters L, W and Y denote sets of variables.

2. REWRITE RULES

This section reformulates the rewrite rules for constructive negation in [10]. The reformulated rewrite rules avoid introducing new local variables whenever possible.

2.1 Exists unique

An *exists unique* property states that for any input $\bar{u}$ in a particular domain $\bar{\sigma}$, there is exactly one output $\bar{x}$ in each of a fixed number of domains $\bar{\theta}_i$ such that $A(\bar{u}\bar{x})$ holds. Formally,

$$\forall \bar{u}{:}\bar{\sigma}.\forall \bar{x}.[A(\bar{u}\bar{x}) \rightarrow \vee_{i \in I} \bar{x} \in \bar{\theta}_i] \qquad (1)$$

$$\forall \bar{u}{:}\bar{\sigma}. \wedge_{i \in I} \exists! \bar{x}_i{:}\bar{\theta}_i.A(\bar{u}\bar{x}_i) \qquad (2)$$

where I is a set of indices and $\exists! x$ stands for "there is exactly one x". Each $\bar{\theta}_i$ is called an output subtype for $\bar{x}$. The type of a variable is annotated with its first occurrence in a formula.

Example 1. *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.*

$$\forall y{:}\mathcal{R}_{>0}.\forall x.(sq(x, y) \rightarrow x \in \mathcal{R}_{>0} \vee x \in \mathcal{R}_{<0}) \qquad (1')$$

$$\forall y{:}\mathcal{R}_{>0}.(\exists! x_1{:}\mathcal{R}_{>0}.sq(x_1, y) \wedge \exists! x_2{:}\mathcal{R}_{<0}.sq(x_2, y)) \qquad (2')$$

The formulas $(1')$ *and* $(2')$ *are respectively instances of (1) and (2) with* $I = \{1, 2\}$, $\bar{u} = y$, $\bar{\sigma} = \mathcal{R}_{>0}$, $\bar{x} = x$, $\bar{\theta}_1 = \mathcal{R}_{>0}$ *and* $\bar{\theta}_2 = \mathcal{R}_{<0}$.

We now consider how an *exists unique* property can be used in solving negative goals of the form

$$\neg\exists \mathsf{L}.[A(\tilde{u}\tilde{x}) \wedge Q] \qquad (g1)$$

Assume that $\tilde{u}$ is of type $\bar{\sigma}$ (I.e. $type(\tilde{u}) \sqsubseteq \bar{\sigma}$) and that variables in L do not occur in $\tilde{u}$ (I.e. $\mathbf{V}_{\tilde{u}} \cap \mathsf{L} = \emptyset$). Suppose that (1) and (2) hold. Then $\exists \bar{x}.(A(\tilde{u}\bar{x}) \wedge (\bar{x} = \tilde{x}) \wedge Q)$ is equivalent to $\vee_{i \in I} \exists \bar{x}_i.(A(\tilde{u}\bar{x}_i{:}\bar{\theta}_i) \wedge (\bar{x}_i = \tilde{x}) \wedge Q)$ from (1). Goal (g1) is equivalent to $\neg\exists \mathsf{L}.\exists \bar{x}.[A(\tilde{u}\bar{x}) \wedge (\tilde{x} = \bar{x}) \wedge Q]$ and hence is equivalent to $\neg\exists \mathsf{L}.[\vee_{i \in I} \exists \bar{x}_i.(A(\tilde{u}\bar{x}_i{:}\bar{\theta}_i) \wedge (\bar{x}_i = \tilde{x}) \wedge Q)]$. Distributing $\exists$ over $\vee$, applying De Morgan's law and using (2), we deduce that goal (g1) is equivalent to

$$\wedge_{i \in I} [A(\tilde{u}\bar{x}_i : \bar{\theta}_i) \wedge \neg\exists \mathsf{L}.((\tilde{x} = \bar{x}_i) \wedge Q)] \qquad (g2)$$

provided that (1), (2), $type(\tilde{u}) \sqsubseteq \bar{\sigma}$ and $\mathbf{V}_{\tilde{u}} \cap \mathsf{L} = \emptyset$ hold.

Example 2. *Let the* exists unique *property be expressed by (1') and (2') and the negative goal be the following.*

$$\neg\exists z'{:}\mathcal{Z}, x'{:}\mathcal{R}_{\geq 20}.(sq(x', y'{:}\mathcal{R}_{>10}) \wedge Q(x', z')) \qquad (g1')$$

Goal (g1') is an instance of (g1). We have $\mathsf{L} = \{z'{:}\mathcal{Z}, x'{:}\mathcal{R}_{\geq 20}\}$, $\tilde{u} = y'$ *and* $\tilde{x} = x'$. *It holds that* $y' \in \mathcal{R}_{>0}$ *since* $y' \in \mathcal{R}_{>10}$ *and* $(\mathcal{R}_{>10} \sqsubseteq \mathcal{R}_{>0})$. *It also holds that* $\mathbf{V}_{\tilde{u}} \cap \mathsf{L} = \{y'\} \cap \{z', x'\} = \emptyset$. *Therefore, (g1') rewrites to (g2') that follows.*

$$sq(x_1{:}\mathcal{R}_{>0}, y'{:}\mathcal{R}_{>10}) \wedge \neg\exists z'{:}\mathcal{Z}, x'{:}\mathcal{R}_{\geq 20}.(x' = x_1 \wedge Q(x', z'))$$
$$\wedge$$
$$sq(x_2{:}\mathcal{R}_{<0}, y'{:}\mathcal{R}_{>10}) \wedge \neg\exists z'{:}\mathcal{Z}, x'{:}\mathcal{R}_{\geq 20}.(x' = x_2 \wedge Q(x', z'))$$

If $type(\tilde{x})$ does not intersects with $\bar{\theta}_k$ then $A(\tilde{u}\bar{x}_k{:}\bar{\theta}_k) \wedge \neg\exists \mathsf{L}.((\tilde{x} = \bar{x}_k) \wedge Q)$ can be removed from (g2) because $(\tilde{x} = \bar{x}_k)$ is unsatisfiable and any further instantiation of $\bar{x}_k$ has no effect on the variables of the original goal. An output subtype is relevant for an atom iff it intersects with the type of the output argument of the atom. We need to consider only relevant output subtypes when rewriting the negative goal. The set of the indices of relevant output subtypes is $J = \{i \in I \mid type(\tilde{x}) \sqcap \bar{\theta}_i \not\equiv \mathbf{0}\}$. Let W be the set of those elements of L that occur in $\tilde{x}$ and $\mathsf{Y} = \mathsf{L} \setminus \mathsf{W}$. Then $\neg\exists \mathsf{L}.((\tilde{x} = \bar{x}_j) \wedge Q)$ is equivalent to

$$\neg\exists \mathsf{W}_j.(\tilde{x}[\mathsf{W}/\mathsf{W}_j] = \bar{x}_j) \vee (\tilde{x}[\mathsf{W}/\mathsf{W}_j] = \bar{x}_j) \wedge \neg\exists \mathsf{Y}.Q[\mathsf{W}/\mathsf{W}_j] \qquad (g3)$$

where W_j is a renaming of W and $Q[\mathsf{W}/\mathsf{W}_j]$ is the result of substituting W_j for W in Q. The disequality constraint

$\neg\exists \mathsf{W}_j.(\tilde{x}[\mathsf{W}/\mathsf{W}_j] = \bar{x}_j)$ can be dealt with by augmenting Chan's simplification procedure with types.

Example 3. *Continue with example 2. We have* $\mathsf{W} = \{x'{:}\mathcal{R}_{\geq 20}\}$, $\mathsf{Y} = \{z'{:}\mathcal{Z}\}$ *and* $J = \{1\}$. *The output subtype* $\mathcal{R}_{<0}$ *is not relevant since* $(type(x') \sqcap \mathcal{R}_{<0}) \equiv \mathbf{0}$. *The subformula* $\neg\exists z'{:}\mathcal{Z}, x'{:}\mathcal{R}_{\geq 20}.(x' = x_1 \wedge Q(x', z'))$ *in (g2') can be rewritten to*

$$\left(\begin{array}{c} \neg\exists w_1{:}\mathcal{R}_{\geq 20}.(w_1 = x_1) \\ \vee \\ (w_1{:}\mathcal{R}_{\geq 20} = x_1) \wedge \neg\exists z'{:}\mathcal{Z}.Q(w_1, z') \end{array}\right) \qquad (g3')$$

A new local variable is introduced for each output argument in (g3). As the cost of simplifying $\neg\exists \mathsf{W}_j.(\tilde{x}[\mathsf{W}/\mathsf{W}_j] = \bar{x}_j)$ increases with the number of equations it contains, it is desirable to avoid introducing new local variables whenever possible. No new local variable need be introduced for an output argument r if r is a local variable, its type is a supertype of all relevant output subtypes and it does not appear in an argument at any other output argument position.

Example 4. *Continue with example 2. Variable* x' *is a local variable. Its type is* $\mathcal{R}_{\geq 20}$. *The only relevant output subtype is* $\mathcal{R}_{>0}$. *A new local variable was introduced because* $\mathcal{R}_{\geq 20}$ *is not a supertype of* $\mathcal{R}_{>0}$.

The above considerations lead to the rewrite rule (QVT) for *exists unique* properties in table 1. Variables in $\bar{z}_j\bar{r}_j$ and W_j do not occur in the lefthand side of the rewrite rule. The selection $\bar{z}_j\bar{r}_j$ is typed with $\bar{\theta}_j$ while W_j inherits the type of W. The selection $\bar{r}$ consists of different variables; and it is a sub-selection of $\tilde{x}$ for which no new local variables need be introduced.

Example 5. *Continue with examples 1 and 2. Using* $(1')$ *and* $(2')$, *(QVT) rewrites (g1') directly to the following.*

$$\begin{array}{c} sq(x_1{:}\mathcal{R}_{>0}, y'{:}\mathcal{R}_{>10}) \wedge \neg\exists w_1{:}\mathcal{R}_{\geq 20}.(w_1 = x_1) \\ \vee \\ sq(x_1{:}\mathcal{R}_{>0}, y'{:}\mathcal{R}_{>10}) \wedge (w_1{:}\mathcal{R}_{\geq 20} = x_1) \wedge \neg\exists z'{:}\mathcal{Z}.Q(w_1, z') \end{array}$$

Example 6. *The append/3 program satisfies the following* exists unique *property.*

$$\begin{array}{c} \forall x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta).z.(append(x, y, z) \to z{:}\mathsf{list}(\beta)) \\ \forall x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta)\exists! z{:}\mathsf{list}(\beta).append(x, y, z) \end{array}$$

Goal $\neg\exists z{:}\mathsf{list}(\beta).(append(x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta), z), p(z))$ *is rewritten to* $append(x{:}\mathsf{list}(\beta), y{:}\mathsf{list}(\beta), z{:}\mathsf{list}(\beta)), \neg\ p(z)$ *by (QVT).*

As well as rewriting the original goal, (QVT) can also be used as a simplification rule. This will result in a more efficient simplification procedure since it will prune unsatisfiable goals without doing a satisfiability test.

Example 7. *We have* $\forall y : \mathbf{1}.x.(x = s(y) \to x : \mathbf{1})$ *and* $\forall y{:}\mathbf{1}.\exists! x{:}\mathbf{1}.(x = s(y))$ *in the domain of Herbrand universe. Consider the following program.*

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

The goal $p(y{:}\mathbf{1}), \neg\ r(y)$ *is reduced to* $p(y), \neg\exists x{:}\mathbf{1}.(x = s(y{:}\mathbf{1}), q(x))$ *which is then simplified directly into* $x{:}\mathbf{1} = s(y), p(y{:}\mathbf{1}), \neg\ q(x)$ *using the above property. Without using this property,* $\neg\exists x{:}\mathbf{1}.(x = s(y{:}\mathbf{1}), q(x))$ *is simplified to*

$$\forall x{:}\mathbf{1}.(x \neq s(y{:}\mathbf{1})) \vee (x{:}\mathbf{1} = s(y{:}\mathbf{1}), \neg\ q(x))$$

and a satisfiability test is then used to eliminate $\forall\ x{:}\mathbf{1}.(x \neq s(y{:}\mathbf{1}))$. *In that sense, the satisfiability test is pushed into the simplification procedure by the* exists unique *property.*

2.2 Exists sometimes

Another rewrite rule applies when it is known that for any input $\bar{u}$ in a particular domain, there is at most one output $\bar{x}$ in each of a fixed number of domains such that $A(\bar{u}\bar{x})$ holds where A is a predicate. Such a property is called an *exists sometimes* property and is expressed by (1) and

$$\forall \bar{u}{:}\bar{\sigma}.\ \wedge_{i \in I} \exists ? \bar{x}_i{:}\bar{\theta}_i.A(\bar{u}\bar{x}_i) \qquad (3)$$

where $\exists ?$ denotes "there is at most one". The same considerations as in the case for *exists unique* properties lead to the rewrite rule (SVT) for *exists sometimes* properties in table 2.

Example 8. *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.*

$$\begin{array}{c} \forall y{:}\mathcal{Z}_{>0}.\forall x.(sq(x, y) \to x \in \mathcal{Z}_{<0} \vee x \in \mathcal{Z}_{>0}) \\ \forall y{:}\mathcal{Z}_{>0}.(\exists ? x_1{:}\mathcal{Z}_{<0}.sq(x_1, y) \wedge \exists ? x_2{:}\mathcal{Z}_{>0}.sq(x_2, y)) \end{array}$$

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

$$\begin{array}{ll} & \neg\exists z_2{:}\mathcal{Z}_{>0}.sq(z_2, y{:}\mathcal{Z}_{>0}) \\ \vee & sq(z_2{:}\mathcal{Z}_{>0}, y{:}\mathcal{Z}_{>0}) \wedge\ \neg\exists x{:}\mathcal{Z}_{[0,20]}.(x = z_2) \\ \vee & sq(z_2{:}\mathcal{Z}_{>0}, y{:}\mathcal{Z}_{>0}) \wedge (x{:}\mathcal{Z}_{[0,20]} = z_2) \wedge \neg\ b(z_2) \end{array}$$

3. DIGRAPH

The rewrite rules (QVT) and (SVT) can be applied repeatedly to extract positive information from a negative goal $\neg\exists \mathsf{W}.G_n, \cdots, G_2, G_1$. A naive implementation would repeatedly scan a conjunction of goals and check if an atom is extractable. After an atom is extracted, some local variables become global. This makes it necessary to check if other atoms are extractable. That would result in an inefficient implementation because most of those checks would fail.

A previously inextricable atom becomes extractable only after some of its local variables become global or some of its global variables are given a value or a stronger type. However, none of (QVT) and (SVT) rules change the type of global variables, nor will it assign any value to them. So, after an atom is extracted, it is only necessary to check those other atoms that share with the extracted atom some variables that have become global. For that reason, we use a list Φ consisting of atoms to be checked and a digraph $\mathcal{D}$ which links each atom with the local variables it contains. The method repeatedly removes one atom from Φ and checks for its extractability until Φ becomes empty. Digraph $\mathcal{D}$ is used in order to quickly retrieve the local variables an atom contains and the atoms containing a particular local variable. After an atom is extracted, it is moved out of the scope of the negation and the local variables it contains become global. This is done by removing the atom and the local variables from $\mathcal{D}$. Before the removal of the local variables, other atoms linked to them are added to Φ as their extractability needs to be checked for again. Initially, every atom needs to be checked.

Given (1), (2), $type(\tilde{u}) \sqsubseteq \bar{\sigma}$ and $\mathbf{V}_{\tilde{u}} \cap \mathsf{L} = \emptyset$ hold	
$\neg\exists \mathsf{L}.[A(\tilde{u}\tilde{x}) \wedge Q] \leftrightarrow$	$\left(\begin{array}{l} \mathit{let}\ J = \{i \in I \mid type(\tilde{x}) \sqcap \theta_i \not\equiv \mathbf{0}\} \\ \quad \nu \subseteq \{p \mid p \in dom(\tilde{x}) \wedge \tilde{x}(p) \in \mathsf{L} \wedge \forall j \in J.(\bar{\theta}_j(p) \sqsubseteq type(\tilde{x}(p)))\}\ \mathit{such\ that}\ \mathit{diff}(\tilde{x} \downarrow \nu)\ \mathit{holds} \\ \quad \mu = dom(\tilde{x}) \setminus \nu \\ \quad \bar{r} = \tilde{x} \downarrow \nu \\ \quad \tilde{s} = \tilde{x} \downarrow \mu \\ \quad \mathsf{W} = (\mathsf{L} \cap \mathbf{V}_{\tilde{s}}) \setminus \bar{r} \\ \quad \mathsf{Y} = \mathsf{L} \setminus \mathsf{W} \\ \mathit{in} \\ \wedge_{j \in J} \left(\begin{array}{c} A(\tilde{u}(\bar{z}_j\bar{r}_j){:}\bar{\theta}_j) \wedge \neg\exists \mathsf{W}_j.(\tilde{s}[\bar{r}/\bar{r}_j, \mathsf{W}/\mathsf{W}_j] = \bar{z}_j) \\ \vee \\ A(\tilde{u}(\bar{z}_j\bar{r}_j){:}\bar{\theta}_j) \wedge (\tilde{s}[\bar{r}/\bar{r}_j, \mathsf{W}/\mathsf{W}_j] = \bar{z}_j) \wedge \neg\exists \mathsf{Y}.Q[\bar{r}/\bar{r}_j, \mathsf{W}/\mathsf{W}_j] \end{array}\right) \end{array}\right)$

Table 1: Rewriting Rule QVT

Given (1), (3), $type(\tilde{u}) \sqsubseteq \bar{\sigma}$ and $\mathbf{V}_{\tilde{u}} \cap \mathsf{L} = \emptyset$ hold	
$\neg\exists \mathsf{L}.[A(\tilde{u}\tilde{x}) \wedge Q] \leftrightarrow$	$\left(\begin{array}{l} \mathit{let}\ J = \{i \in I \mid type(\tilde{x}) \sqcap \theta_i \not\equiv \mathbf{0}\} \\ \quad \nu \subseteq \{p \mid p \in dom(\tilde{x}) \wedge \tilde{x}(p) \in \mathsf{L} \wedge \forall j \in J.(\bar{\theta}_j(p) \sqsubseteq type(\tilde{x}(p)))\}\ \mathit{such\ that}\ \mathit{diff}(\tilde{x} \downarrow \nu)\ \mathit{holds} \\ \quad \mu = dom(\tilde{x}) \setminus \nu \\ \quad \bar{r} = \tilde{x} \downarrow \nu \\ \quad \tilde{s} = \tilde{x} \downarrow \mu \\ \quad \mathsf{W} = (\mathsf{L} \cap \mathbf{V}_{\tilde{s}}) \setminus \bar{r} \\ \quad \mathsf{Y} = \mathsf{L} \setminus \mathsf{W} \\ \mathit{in} \\ \wedge_{j \in J} \left(\begin{array}{c} \neg\exists(\bar{z}_j\bar{r}_j){:}\bar{\theta}_j.A(\tilde{u}\bar{z}_j\bar{r}_j) \\ \vee \\ A(\tilde{u}(\bar{z}_j\bar{r}_j){:}\bar{\theta}_j) \wedge \neg\exists \mathsf{W}_j.(\tilde{s}[\bar{r}/\bar{r}_j, \mathsf{W}/\mathsf{W}_j] = \bar{z}_j) \\ \vee \\ A(\tilde{u}(\bar{z}_j\bar{r}_j){:}\bar{\theta}_j) \wedge (\tilde{s}[\bar{r}/\bar{r}_j, \mathsf{W}/\mathsf{W}_j] = \bar{z}_j) \wedge \neg\exists \mathsf{Y}.Q[\bar{r}/\bar{r}_j, \mathsf{W}/\mathsf{W}_j] \end{array}\right) \end{array}\right)$

Table 2: Rewriting rule SVT

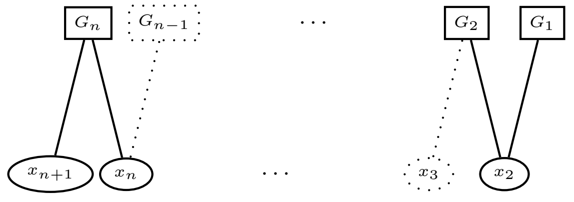
Let us first consider the case where an *existence* property has one output subtype for its output parameter. When an atom is extracted by (QVT) or (SVT) without introducing any new local variable, it is moved out of the scope of the negation and the local variables in it are promoted to being global. The atom is deleted from Φ and $\mathcal{D}$. The other atoms that are linked to the local variables are then added into Φ and the local variables are deleted from $\mathcal{D}$. The method continues with the updated $\mathcal{D}$ and Φ.

EXAMPLE 9. *Let p be of arity 2 with the following* exists unique *property.*

$$\forall x{:}\mathbf{1}.\forall y.(p(x,y) \rightarrow y \in \mathbf{1})$$

$$\forall x{:}\mathbf{1}.\exists! y{:}\mathbf{1}.p(x,y)$$

Let $G_i = p(x_i, x_{i+1})$. The negative goal $\neg\exists x_2{:}\mathbf{1}.\cdots x_{n+1}{:}\mathbf{1}.(G_n, \cdots, G_i, \cdots, G_1)$ is such that extracting G_i makes G_{i+1} extractable. A naive implementation of (QVT) does $\frac{n(n-1)}{2}$ tests by testing G_n for n times, G_{n-1} for $n-1$ times and so on. The negative goal has the following graph.



The implementation technique works as follows. Initially, Φ contains G_n, $G_{n-1}, \cdots, G_2$ and G_1 that are removed from Φ and tested in that order until G_1 is extracted. At that point, only G_2 is added to Φ, it is then immediately removed and tested. Extracting G_2 adds G_3 into Φ. This process continues until G_n is tested and extracted, proving the falsity of the original negative goal. A total of $(2n-1)$ tests are performed with G_1 being tested once and each G_i for $2 \leq i \leq n$ being tested twice.

When an atom is extracted by (QVT) or (SVT) by means of introducing local variables, only some local variables become global and the derived goals are more complex. However, the residual negative subgoals can be obtained in the same way as above.

When the output parameter of an *existence* property has more than one output subtype, several complex goals may be derived from the negative goal. Each of these complex goals may contain a number of residual negative subgoals to which (QVT) or (SVT) may be applicable. However, these residual negative subgoals differ only in the names and types of newly promoted global variables. So, the digraph and the checklist for each of these residual negative subgoals are obtained in the same way.

4. EXTRACTABILITY

Given an atom inside a negation and an *existence* property, (QVT) and (SVT) have to decide if the atom satisfies the *existence* property and, if so, decide for which output arguments new local variables need be introduced. The rules (QVT) and (SVT) differ only in that (SVT) has an extra disjunct $\neg\exists(\bar{z}_j\bar{r}_j){:}\bar{\theta}_j.A(\tilde{u}\bar{z}_j\bar{r}_j)$ for each relevant output subtype. Otherwise, they are the same. The common functionality of (QVT) and (SVT) is factored out to a function *sqvt*. It tests if an atom satisfies an *existence* property, introduces new local variables, decides if an output subtype is relevant, and renames and types local variables.

```
function $sqvt(P, G, \mathsf{L})$
begin
(01) Let $G$ be $B(\cdots, t_u, \cdots, t_x, \cdots)$ and $P$ be
     $\langle A(\cdots, \mathbf{i}(\sigma), \cdots, \mathbf{o}(\bar{\Theta}), \cdots), I\rangle$;
(02) if $B = A$ and $(type(t_u) \sqsubseteq \sigma) \wedge (\mathbf{V}_{t_u} \cap \mathsf{L}) = \emptyset$ for each
     $t_u$ matching an $\mathbf{i}(\sigma)$
(03) then
    (04) $\bar{r} := \epsilon; \bar{x}_m := \epsilon; \bar{z} := \epsilon; \tilde{s} := \epsilon; \mathsf{W} := nil; J := I$;
    (05) for each $t_x$ at position p matching an $\mathbf{o}(\bar{\Theta})$ do
        (06) $J := J \cap \{k \mid (type(t_x) \sqcap \bar{\Theta}(k)) \not\equiv \mathbf{0}\}$;
        (07) if $t_x \in (\mathsf{L} \setminus \bar{r}) \wedge \forall j \in J.(\bar{\Theta}(j) \sqsubseteq type(t_x))$
        (08) then
            (09) $\bar{r} := \bar{r}[p \mapsto t_x]$;
            (10) $\bar{x}_m := \bar{x}_m[p \mapsto (t_x, \bar{\Theta})]$;
        (11) else
            (12) $\tilde{s} := \tilde{s}[p \mapsto t_x]$;
            (13) for each $v \in ((\mathbf{V}_{t_x} \cap \mathsf{L}) \setminus (\bar{r} \cup \mathsf{W}))$
                 do $\mathsf{W} := v :: \mathsf{W}$ od;
            (14) $z := newv(\mathbf{1})$; $\bar{z} := \bar{z}[p \mapsto z]$;
                 $G := G[t_x/z]$;
            (15) $\bar{x}_m := \bar{x}_m[p \mapsto (z, \bar{\Theta})]$
        (16) fi;
    (17) od;
    (18) $\bar{x} := map(fst, \bar{x}_m)$;
    (19) $\bar{x}_{cs} := \bigcup_{j \in J}\{map(\lambda e.newv((snd(e))(j)), \bar{x}_m)\}$;
    (20) return $(G, \bar{x}, \bar{x}_{cs}, \tilde{s}, \bar{z}, \bar{r}, \mathsf{W})$
(21) else return $nil$
(22) fi
end;
```

Figure 1: The *sqvt* function where $x :: L$ is a list with head x and tail L.

An *exists unique* property is represented as follows. Each input parameter $u{:}\sigma$ in $\bar{u}{:}\bar{\sigma}$ is represented by $\mathbf{i}(\sigma)$. Each output parameter x in $\bar{x}$ with output subtypes $\{\theta_k \mid k \in I\}$ is represented by $\mathbf{o}(\bar{\Theta})$ where $\bar{\Theta}$ is a mapping which maps k in I to θ_k. An *exists unique* property has the following representation where input and output parameters may be interspersed.

$$\langle A(\cdots, \mathbf{i}(\sigma), \cdots, \mathbf{o}(\bar{\Theta}), \cdots), I\rangle$$

The set of *exists unique* properties is denoted by $\Gamma_!$. We use the same representation for an *exists sometimes* property and denote the set of *exists sometimes* properties by $\Gamma_?$.

EXAMPLE 10. *The* exists sometimes *property in example 8 is represented by this item in* $\Gamma_?$*:* $\langle sq(\mathbf{o}(\{1 \mapsto \mathcal{Z}_{<0}, 2 \mapsto \mathcal{Z}_{>0}\}), \mathbf{i}(\mathcal{Z}_{>0})), \{1, 2\}\rangle$.

Figure 1 defines *sqvt* with the following auxiliary functions. A call to $newv(T)$ creates a new variable of type T. Given a pair, the function *fst* returns the first component while *snd* returns the second component. The high order function *map* applies a function f to a selection π point-wise: $map(f, \pi)(i) = f(\pi(i))$ for each $i \in dom(\pi)$ and $dom(map(f, \pi)) = dom(\pi)$.

Given an *existence* property P of the form $\langle A(\cdots, \mathbf{i}(\sigma), \cdots, \mathbf{o}(\bar{\Theta}), \cdots), I\rangle$ and an atom G of the form $B(\cdots, t_u, \cdots, t_x, \cdots)$ and a set L of local variables, *sqvt* first checks if it is possible to replace some output arguments in G with newly introduced local variables so as to make G satisfy P. This is tested in line (02). Function *sqvt* returns *nil* from line (21) if this test fails. Otherwise, *sqvt* classifies every output argument according to whether a new local variable needs be introduced for it or not. The variable $\bar{r}$ holds the selection of output arguments for which no new local variables need be introduced, $\tilde{s}$ is the selection of other output arguments and $\bar{z}$ is the selection of corresponding newly introduced local variables. Whenever a new local variable z is introduced for an output argument t_x, *sqvt* substitutes z for t_x in G. The function *sqvt* collects the list W of the local variables that occur in $\tilde{s}$ but not in $\bar{r}$. It also builds up the selection $\bar{x}_m$ of the new output arguments each of which is associated with a mapping from indices in I to types and collects the set J of relevant indices for G. Line (04) initializes these selections and sets. The (05)-(17) loop iterates through all output arguments. Line (06) narrows the set J of relevant indices. Line (07) determines if it is necessary to introduce a new local variable for the output argument t_x under consideration. If not, line (09) adds t_x into $\bar{r}$ and line (10) adds to $\bar{x}_m$ a pair consisting of t_x and the mapping for the corresponding output parameter in P. Otherwise, line (12) adds t_x to $\tilde{s}$, line (13) adds to W the local variables in t_x that do not occur in $\bar{r}$ or W, line (14) introduces a new local variable z of type $\mathbf{1}$, adds z to $\bar{z}$ and substitutes z for t_x in G, and line (15) adds to $\bar{x}_m$ a pair consisting of z and the mapping for the corresponding output parameter in P. The newly introduced local variable z in line (14) will be renamed and attached with an appropriate type from the mapping paired with it in $\bar{x}_m$. Line (18) extracts the selection $\bar{x}$ of the new output arguments of G. Line (19) makes, for each relevant index in J, a new copy of $\bar{x}$ and types the copy with an appropriate type, and collects the set $\bar{x}_{cs}$ of all the copies made. For a fixed index $j \in J$, line (19) does the following for each pair in $\bar{x}_m$. It first takes the second component of the pair which is a mapping from indices to types, then finds the type for the index j, and creates a new variable of that type. Line (20) returns with required information.

EXAMPLE 11. *Continue with example 10. Let* $G = sq(x{:}\mathcal{Z}_{[0,20]}, y{:}\mathcal{Z}_{>0})$ *and* $\mathsf{L} = \{x{:}\mathcal{Z}_{[0,20]}\}$*. Then* $sqvt(P, G, \mathsf{L}) = (G', \bar{x}, \bar{x}_{cs}, \tilde{s}, \bar{z}, \bar{r}, \mathsf{W})$ *with* $G' = sq(z{:}\mathbf{1}, y{:}\mathcal{Z}_{>0})$*,* $\bar{x} = z{:}\mathbf{1}$*,* $\bar{x}_{cs} = \{z_2 : \mathcal{Z}_{>0}\}$*,* $\tilde{s} = x : \mathcal{Z}_{[0,20]}$*,* $\bar{z} = z : \mathbf{1}$*,* $\bar{r} = \epsilon$*, and* $\mathsf{W} = \{x{:}\mathcal{Z}_{[0,20]}\}$.

LEMMA 12. *The time complexity of the test for the extractability of an atom with respect to an* exists unique *or* exists sometimes *property is linear in the size of the atom.*

The following theorem gives the correctness of *sqvt*. In addition, it states that *sqvt* introduces a new variable only when it is necessary.

THEOREM 13. *Let P be an* exists sometimes *(resp.* exists unique *) property, G an atom, Q a conjunction of goals and L a set of variables. G can be extracted from $\exists\mathsf{L}.(G \wedge Q)$ by SVT (resp. QVT) using P iff $sqvt(P, G, \mathsf{L}) \neq nil$. Furthermore, letting $sqvt(P, G, \mathsf{L}) = (G', \bar{x}, \bar{x}_{cs}, \tilde{s}, \bar{z}, \bar{r}, \mathsf{W})$,*

- $\bar{r}$, $\tilde{s}$, $\bar{z}$ *and* W *are as in (SVT) (resp. (QVT)). Furthermore,* $\bar{r}$ *is maximal in the sense that any proper superselection of* $\bar{r}$ *will include at least one output argument of* G *for which a new variable must be introduced.*

- $G' = G[\tilde{s}/\bar{z}]$

- $\bar{x}$ *is the selection of the output arguments of* G'.

- $\bar{x}_{cs}$ *is a set of selections with each being a fresh copy of* $\bar{x}$ *typed by an output subtype of* P *that is relevant to* G.

5. IMPLEMENTATION

5.1 Derivation Rule

With a negative goal being represented by $neg(\Phi, \mathcal{D})$ where Φ is the checklist and $\mathcal{D}$ is the digraph, rewrite rules (QVT) and (SVT) are implemented a derivation rule $\hookrightarrow_{sqvt}$. Let $loc(\mathcal{D})$ be the set of local variables in $\mathcal{D}$, $delete(Ns, \mathcal{D})$ be the result of deleting nodes in Ns from $\mathcal{D}$, $link(N, Ns, \mathcal{D})$ be true iff $\mathcal{D}$ links node N with some node in Ns.

- $\boldsymbol{\alpha}, neg(\{G\} \cup \Phi, \mathcal{D}), \boldsymbol{\beta} \hookrightarrow_{sqvt} \boldsymbol{\alpha}, N_l, \boldsymbol{\beta}$ for each $1 \leq l \leq k$ if $\exists P \in \Gamma_{!}.sqvt(P, G, loc(\mathcal{D})) = (G', \bar{x}, \bar{x}_{cs}, \tilde{s}, \bar{z}, \bar{r}, \mathsf{W})$ and $N_1 \vee N_2 \vee \cdots \vee N_k$ is a disjunctive normal form of

$$\wedge_{\bar{x}' \in \bar{x}_{cs}} \left[\begin{array}{c} let\ \mathsf{W}' = map(newv \circ type, \mathsf{W})\ in \\ \left(\begin{array}{c} (G' \wedge \neg\exists \mathsf{W}'.(\tilde{s}[\mathsf{W}/\mathsf{W}'] = \bar{z}))[\bar{x}/\bar{x}'] \\ \vee \\ (G' \wedge (\tilde{s}[\mathsf{W}/\mathsf{W}'] = \bar{z}))[\bar{x}/\bar{x}'] \\ \wedge \\ neg(\Phi', \mathcal{D}')[\mathsf{W}/\mathsf{W}'][\bar{x}/\bar{x}'] \end{array} \right) \end{array} \right]$$

  where $\Phi' = \Phi \cup \{N \mid link(N, \bar{r} \cup \mathsf{W}, \mathcal{D})\} \setminus \{G\}$ and $\mathcal{D}' = delete(\bar{r} \cup \mathsf{W} \cup \{G\}, \mathcal{D})$. The above formula corresponds to the righthand side of (QVT) in that $\bar{x}'$ corresponds to $\bar{z}_j \bar{r}_j$ and W' to W_j. Note that $\bar{x}'$ and W' are typed when they are created.

- $\boldsymbol{\alpha}, neg(\{G\} \cup \Phi, \mathcal{D}), \boldsymbol{\beta} \hookrightarrow_{sqvt} \boldsymbol{\alpha}, N_l, \boldsymbol{\beta}$ for each $1 \leq l \leq k$ if $\exists P \in \Gamma_{?}.sqvt(P, G, loc(\mathcal{D})) = (G', \bar{x}, \bar{x}_{cs}, \tilde{s}, \bar{z}, \bar{r}, \mathsf{W})$ and $N_1 \vee N_2 \vee \cdots \vee N_k$ is a disjunctive normal form of

$$\wedge_{\bar{x}' \in \bar{x}_{cs}} \left[\begin{array}{c} let\ \mathsf{W}' = map(newv \circ type, \mathsf{W})\ in \\ \left(\begin{array}{c} \neg\exists \bar{x}'.G'[\bar{x}/\bar{x}'] \\ \vee \\ (G' \wedge \neg\exists \mathsf{W}'.(\tilde{s}[\mathsf{W}/\mathsf{W}'] = \bar{z}))[\bar{x}/\bar{x}'] \\ \vee \\ (G' \wedge (\tilde{s}[\mathsf{W}/\mathsf{W}'] = \bar{z}))[\bar{x}/\bar{x}'] \\ \wedge \\ neg(\Phi', \mathcal{D}')[\mathsf{W}/\mathsf{W}'][\bar{x}/\bar{x}'] \end{array} \right) \end{array} \right]$$

  where $\Phi' = \Phi \cup \{N \mid link(N, \bar{r} \cup \mathsf{W}, \mathcal{D})\} \setminus \{G\}$ and $\mathcal{D}' = delete(\bar{r} \cup \mathsf{W} \cup \{G\}, \mathcal{D})$.

- $\boldsymbol{\alpha}, neg(\{G\} \cup \Phi, \mathcal{D}), \boldsymbol{\beta} \hookrightarrow_{sqvt} \boldsymbol{\alpha}, neg(\Phi, \mathcal{D}), \boldsymbol{\beta}$ if $\forall P \in \Gamma_{!} \cup \Gamma_{?}.sqvt(P, G, loc(\mathcal{D})) = nil$. This rule removes from the checklist an atom which doesn't satisfy any *existence* property.

- $\boldsymbol{\alpha}, neg(\emptyset, \Lambda), \boldsymbol{\beta} \hookrightarrow_{sqvt}$ false where Λ is the empty digraph. Note that $neg(\emptyset, \Lambda)$ represents $\neg\mathsf{true}$.

```
neg_dig(GT,VT,[Gid|CHK])     :-                               %1
  lookup(GT,Gid,G),                                            %2
  qvt(G,VT,G1,Xs,Xscs,Ss,Zs,Rs,Ws),                            %3
  new_dig(GT,VT,CHK,Rs,Ws,Gid,GTn,VTn,CHKn),                   %4
  neg_dig_qvt(Xscs,Xs,Ss,Zs,Rs,Ws,G1,GTn,VTn,CHKn).            %5

neg_dig_qvt([],_Xs,_Ss,_Zs,_Rs,_Ws,_G1,_GT,_VT,_CHK).%6
neg_dig_qvt([Xsc|Xscs],Xs,Ss,Zs,Rs,Ws,G1,GT,VT,CHK):-%7
  copy_vars(Ws,Wsc),          % make a copy of Ws              %8
  replace(Ws,Wsc,(Ss,GT),(Ss0,GT0)), %rename Ws and            %9
  replace(Xs,Xsc,(G1,Ss0,Zs,GT0),(Gn,Ssn,Zsn,GTn)),            %10
  (                              % Xs in Ss,Zs,G1,GT           %11
      call(Gn),                                                %12
      neg_cet(Ssn,Zsn,Wsc)                                     %13
  ;                                                            %14
      call(Gn),                                                %15
      Ssn=Zsn,                                                 %16
      neg_dig(GTn,VT,CHK)                                      %17
  ),                                                           %18
  neg_dig_qvt(Xscs,Xs,Ss,Zs,Rs,Ws,G1,GT,VT,CHK).               %19
```

Figure 2: Implementation of the first rule for $\hookrightarrow_{sqvt}$

5.2 ECLiPSe Implementation

We have implemented in ECLiPSe [1] a prototype constructive negation system that also implements Chan's constructive negation rule. A type is associated with a variable as an attribute [3]. Due to lack of space, we can only sketch how the first rule for $\hookrightarrow_{sqvt}$ is translated into Prolog code. Other rules are translated similarly.

A node N in a digraph is represented by a triple $\langle id(N), N, linked(N)\rangle$ where $id(N)$ is an integer that uniquely identifies N and $linked(N)$ is the list of the identifiers of the nodes that are linked to N. The checklist is represented as a list of identifiers of subgoals while the digraph is represented as a pair consisting of a list GT of triples for subgoals and a list VT of triples for local variables.

The clause for $neg\_dig/3$ in figure 2 (lines 1-5) implements the first rule for $\hookrightarrow_{sqvt}$. Line 2 looks up the subgoal G identified by the Gid. Line 3 succeeds iff there is an *exists unique* property P such that $sqvt(P, G, VT) = (G1, Xs, Xscs, Ss, Zs, Rs, Ws)$. The call to $new\_dig/9$ in line 4 computes a new digraph (GTn,VTn) by removing G (identified by Gid), Ws and Rs from (GT,VT) and a new checklist CHKn by adding subgoals linked to any variable in Ws or Rs. Line 5 calls $neg\_dig\_qvt/10$ (lines 6-19) to compute the disjunctive normal form as specified in the first rule for $\hookrightarrow_{sqvt}$.

Procedure $neg\_dig\_qvt/10$ does the following for each Xsc in $Xscs$. It makes a fresh copy Wsc of Ws (line 8) and renames Ws and Xs in $G1, Ss, Zs, GT$ into Wsc and Xsc respectively (lines 9-10) resulting in Gn, Ssn, Zsn, GTn. Lines 12-13 compute the first conjunctive subformula inside the parentheses in the first rule for $\hookrightarrow_{sqvt}$ while lines 15-17 compute the second. VT in line 17 needn't be renamed because variables in VT remain local after moving G outside the scope of negation. Call $neg\_cet(Ssn, Zsn, Wsc)$ simplifies the inequality $\neg\exists Wsc.(Ssn = Zsn)$ by augmenting the simplification procedure in [7] with types.

The top-level of the constructive negation system is negx/2. $negx(G, Vs)$ is true iff $\neg\exists\, Vs.G$ is true. It constructs a digraph representation for $\neg\exists\, Vs.G$ and repeatedly invokes $neg\_dig/3$ that applies $\hookrightarrow_{sqvt}$ repeatedly until no rewriting can be done. It then delays. The set of all delayed goals constitutes a frontier of $\neg\exists Vs.G$.

EXAMPLE 14. *This example illustrates a session with the prototype. The prototype implements* existence *properties of arithmetic constraints [9] as well as other commonly used library predicates. Term $real(l,u)$ encodes type $\mathcal{R}_{[l,u]}$.*

```
[eclipse 2]: negx((sq(X:real(-0.5,0.5),U),
                   sq(Y:real(-1,1),V),
                   add(U,V,W:real(0,1))),[U,V]).

sq(Y:real(-1, 1), V1:real),
add(Z:real, V1:real, W:real(0, 1)),
sq(X:real(-0.5, 0.5), U1:real),
neg_cet(Z:real, U1:real, []);

no (more) solution.
[eclipse 3]:
```

This states that $\neg\exists U:\mathbf{1}.V:\mathbf{1}.(sq(X{:}\mathcal{R}_{[-0.5,0.5]},U), sq(Y{:}\mathcal{R}_{[-1,1]},V), add(U,V,W{:}\mathcal{R}_{[0,1]}))$ rewrites to $sq(Y{:}\mathcal{R}_{[-1,1]},V1{:}\mathcal{R}), add(Z{:}\mathcal{R},V1,W{:}\mathcal{R}_{[0,1]}), sq(X{:}\mathcal{R}_{[-0.5,0.5]},U1{:}\mathcal{R}), Z\neq U1$.

6. TIME COMPLEXITIES

Given a negative goal, a $\hookrightarrow_{sqvt}$ derivation step extracts an atom out of a negation and produces several residual negative goals which are then processed in subsequent derivation steps. The time complexity of $\hookrightarrow_{sqvt}$ with respect to a negative goal is measured by the time spent on all possible derivations from the negative goal.

Consider extracting an atom using an *exists unique* property with k relevant output subtypes for the atom. The disjunctive normal form of the righthand side of the (QVT) rewrite rule has at most 2^k disjuncts. Each of k different residual negative goals occurs in 2^{k-1} of the 2^k disjuncts. Therefore, the negative goal spawns $k\times 2^{k-1}$ occurrences of residual negative goals. If we take $\neg\exists(\bar{z}_j\bar{r}_j):\bar{\theta}_j.A(\tilde{u}\bar{z}_j\bar{r}_j)\vee A(\tilde{u}(\bar{z}_j\bar{r}_j){:}\bar{\theta}_j)\wedge\neg\exists\mathsf{W}_j.(\tilde{s}[\bar{r}/\bar{r}_j,\mathsf{W}/\mathsf{W}_j]=\bar{z}_j)$ as atomic when normalizing the righthand side of the (SVT) rewrite rule then the negative goal also spawns $k\times 2^{k-1}$ occurrences of residual negative goals when an atom is extracted using an *exists sometimes* property with k relevant output subtypes for the atom.

When an *existence* property is used to extract an atom, the number of the relevant output subtypes of the *existence* property for the atom may vary. For simplicity, we assume that the number of the relevant output subtypes is a fixed number k for all the atoms and all the *existence* properties. The relationship between negative goals and the residual negative goals they spawn forms a tree called a spawning tree.

Let $\mathcal{D}$ consist of n atoms with non-decreasing sizes $s_i, 1\leq i\leq n$. Consider the time complexity of $\hookrightarrow_{sqvt}$. We weight the i^{th} atom in $\mathcal{D}$ by the number w_i of those atoms that share local variables with the i^{th} atom and are smaller in size than the i^{th} atom. The following lemma gives the time complexity of the extractability tests performed along a path in the spawning tree for $\mathcal{D}$.

LEMMA 15. *The time complexity of the extractability tests performed along a path in the spawning tree for $\mathcal{D}$ is $\mathcal{O}(\Sigma_i(w_i+1)\times s_i)$.*

The above lemma states that in the worst case the n^{th} atom will be tested at levels $n,(n-1),\cdots,n-w_n$, the $(n-1)^{th}$ atom at levels $(n-1),(n-2),\cdots,(n-1)-w_{n-1}$, and so on. This gives rise to the following result.

THEOREM 16. *The time complexity of $\hookrightarrow_{sqvt}$ is*

$$\mathcal{O}(\Sigma_j s_j\times[\Sigma_{j-w_j\leq i\leq j}(k\times 2^{k-1})^{i-1}])$$

7. CONCLUSION AND DISCUSSION

We have presented an implementation of the constructive negation method in [10] and analyzed its complexity. The implementation uses a digraph and a worklist to represent a negative goal so as to avoid futile extractability tests of atoms in the negative goal. An algorithm is presented that does the extractability test given an atom and an *existence* property and introduces new local variables into the atom to make it satisfy the *existence* property. The complexity of the algorithm is linear in the size of the atom.

The constructive negation method developed in [9, 10] and this paper was not intended to replace any other method. Nevertheless, the method is correct in that its rewrite rules preserve logical equivalence. The method is also complete in the sense that it does not throw away any answer to a negative goal; it simply suspends when none of its rewrite rules is applicable. Rewrite rules in the implementation use *existence* properties to extract atoms from negative goals. How these existence properties are obtained is beyond the scope of this paper. The current implementation includes *existence* properties for arithmetic constraints and some library predicates.

There have been much research into constructive negation [2, 4, 5, 7, 8, 11, 12, 13, 16, 17, 18, 19, 20, 21, 22, 23]. Constructive intensional negation was studied in [2, 5, 4, 21]. Marchiori [17] addresses the termination of logic programs with respect to constructive negation. Lobo[15] studies constructive negation for disjunctive logic programs. Ramírez and Falashi [22] and Moreno-Navaro [18, 19, 20] extend constructive negation for functional logic programs. Foo et. al propose a constructive negation approach for Datalog [13].

Chan introduced constructive negation which allows non-ground negative goals to bind variables in the same way as positive ones [7, 8]. Answers to $\neg\ Q$ are obtained by negating answers to Q. 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 subgoal [7]. Chan later extended his method by negating a frontier of a derivation tree for the negated subgoal [8]. Małuszyński and Näslund put forward an approach which allows a negative goal to directly return its answers [16]. Drabent defines SLDFA resolution over the Herbrand universe that constructs answers for the negative goal from a finite number of answers to the negated subgoal [11].

Stuckey provides a constructive negation method for constraint logic programs over arbitrary structures [23]. The method is a generalization of Chan's; it is sound and complete with respect to the three-valued consequences of the completion of the program. Stuckey uses the following property of logic formulae in his simplification procedure.

$$\neg\exists \mathsf{Y}.(c \wedge Q) \leftrightarrow \neg\exists \mathsf{Y}.c \vee \neg\exists \mathsf{Y}.(c \wedge Q)$$

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 $\neg\exists \mathsf{Y}.c$ with other constraints. Fages proposes a concurrent pruning mechanism over standard SLD derivation trees for constructive negation in constraint logic programs [12]. Two derivation trees are concurrently constructed. The computed answers from one of the trees are used to prune the nodes of the other.

We now conclude this discussion using example 6 to compare our method with Chan's and Stuckey's. Rule (QVT) rewrites $\neg\exists z\!:\!\mathsf{list}(\beta).(append(x\!:\!\mathsf{list}(\beta), y\!:\!\mathsf{list}(\beta), z), p(z))$ to $append(x\!:\!\mathsf{list}(\beta), y\!:\!\mathsf{list}(\beta), z\!:\!\mathsf{list}(\beta)), \neg\ p(z)$. 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,

$$\left\{ \begin{array}{c} (x = [], y = z, p(z)), \\ (x = [h|x'], y = y', z = [h|z'], \\ append(x', y', z'), p(z)) \end{array} \right\}$$

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. Similar comparison can be made between our's and methods in [11, 12, 16] since they all construct a frontier of an SLD derivation tree for $append(x, y, z), p(z)$.

Acknowledgement

This work was funded by NSF grants CCR-0131862 and INT-0327760.

8. REFERENCES

[1] A. Aggoun et. al. *ECL^iPS^e 3.5 User Manual.* ECRC Munich, Germany, December 1995.

[2] A. Bossi, M. Fabris, and M.C. Meo. A bottom-up semantics for constructive negation. In [6], pages 520–534.

[3] P. Brisset et. al. *ECL^iPS^e 3.4 Extensions User Manual.* ECRC Munich, Germany, July 1994.

[4] P. Bruscoli, A. Dovier, E. Pontelli, and G. Rossi. Compiling intensional sets in CLP. In [6], pages 647–661.

[5] P. Bruscoli, F. Levi, G. Levi, and M.C. Meo. Compilative constructive negation in constraint logic programs. *Lecture Notes in Computer Science*, 787:52–67, 1994.

[6] M. Bruynooghe, editor. *Proceedings of the Eleventh International Conference on Logic Programming.* The MIT Press, 1994.

[7] D. Chan. Constructive negation based on the completed database. In [14], pages 111–125.

[8] D. Chan. An Extension of Constructive Negation and its Application in Coroutining. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 477–496. The MIT Press, 1989.

[9] J.G. Cleary. Constructive negation of arithmetic constraints using data-flow graphs. *Constraints*, 2:131–162, 1997.

[10] J.G. Cleary and L. Lu. Constructive negation using typed existence properties. *Lecture Notes in Computer Science*, 1490:411–426, 1998.

[11] W. Drabent. What is failure? An approach to constructive negation. *Acta Informatica*, 32:27–59, 1995.

[12] F. Fages. Constructive negation by pruning. *Journal of Logic Programming*, 32(2):85–118, 1997.

[13] N. Foo, A. Rao, A. Taylor, and A. Walker. Deduced relevant types and constructive negation. In [14], pages 126–139.

[14] R. A. Kowalski and K. A. Bowen, editors. *Proceedings of the Fifth International Conference and Symposium on Logic Programming.* The MIT Press, 1988.

[15] Jorge Lobo. On constructive negation for disjunctive logic programs. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 704–718, The MIT Press, 1990.

[16] J. Małuszyński and T. Näslund. Fail Substitutions for Negation as Failure. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 461–476. The MIT Press, 1989.

[17] E. Marchiori. On termination of general logic programs w.r.t. constructive negation. *Journal of Logic Programming*, 26(1):69–89, 1996.

[18] J. J. Moreno-Navaro. Default rules: An extension of constructive negation for narrowing-based languages. In [6], pages 535–549.

[19] J. J. Moreno-Navarro. Extending constructive negation for partial functions in lazy functional-logic languages. *Lecture Notes in Artificial Intelligence*, 1050:213–228, 1996.

[20] J. J. Moreno-Navarro and S. Muñoz-Hernández. How to incorporate negation in a Prolog compiler. *Lecture Notes in Computer Science*, 1753:124–139, 2000.

[21] S. Muñoz-Hernández, J. Mariño, and J.J. Moreno-Navarro. Constructive intensional negation. *Lecture Notes in Computer Science*, 2998:39–54, 2004.

[22] M. J. Ramírez and M. Falaschi. Conditional Narrowing with Constructive Negation. *Lecture Notes in Artificial Intelligence*, 660:59–79, 1993.

[23] P.J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118:12–33, 1995.