Universal Quantification in a Constraint-Based Planner

Keith Golden and Jeremy Frank
NASA Ames Research Center
Mail Stop 269-1
Moffett Field, CA 94035
{kgolden,frank}@ptolemy.arc.nasa.gov

Abstract

We present a general approach to planning with a restricted class of universally quantified constraints. These constraints stem from expressive action descriptions, coupled with large or infinite universes and incomplete information. The approach essentially consists of checking that the quantified constraint is satisfied for all members of the universe. We present a general algorithm for proving that quantified constraints are satisfied when the domains of all of the variables are finite. We then describe a class of quantified constraints for which we can efficiently prove satisfiability even when the domains are infinite. These form the basis of constraint reasoning systems that can be used by a variety of planners.

1 Introduction

Softbots (**soft**ware ro**bots**) are intelligent software agents that sense and act in an environment, such as a computer operating system. Since software environments are so rich, there is almost no limit to the kinds of tasks that softbots can perform, including on-line comparison shopping, managing email, scheduling meetings, and processing data. Planner-based softbots (Etzioni & Weld 1994; Golden 1997) accept goals from users and invoke a planner to find a sequence of actions (e.g., commands or program invocations) that will achieve the goal.

We are working on softbots for data processing, including image processing, managing file archives, and running scientific models. Due to the richness of softbot problem domains in general, and data processing domains in particular, the planner must handle a rich action representation. In particular, it must support:

- **Universal quantification**: Many commands and programs operate on sets of things, where membership in the set can be defined in terms of necessary and sufficient conditions. For example,
  - The Unix `ls` (or DOS `dir`) command lists all files in a given directory.
  - The "`tar x`" (or `unzip`) command extracts all files in a given archive.
  - The `grep` command returns all lines of text in a file matching a given regular expression.
  - Most image processing commands operate on all pixels in an image or in a given region of an image.
- **Incomplete information**: It is common for softbots to have only incomplete information about their environment. For example, a softbot is unlikely to know about all the files on the local file system, much less all the files available over the Internet.
- **Large or infinite universes**: The size of the universe is generally very large or infinite. For example, there are hundreds of thousands of files accessible on a typical file system and billions of web pages publicly available over the Internet. The number of *possible* files, file path names, *etc*., is effectively infinite. Given the presence of incomplete information and the ability to create new files, it is necessary to reason about these infinite sets.
- **Constraints**: As noted in (Chien *et al.* 1997; Lansky & Philpot 1993), data processing domains typically involve a rich set of constraints. By constraints, we mean any relations whose truth values can be computed.

The intersection of these features poses some interesting challenges. For example, the intersection of universal quantification and incomplete information means that standard approaches to dealing with universal quantification in planning (Penberthy & Weld 1992) don't work, and other approaches are needed (Golden 1998; Etzioni, Golden, & Weld 1997; Babaian & Schmolze 2000). This paper discusses the effect of universal quantification and large or infinite universes on constraint reasoning and proposes a way to accommodate universally quantified constraints into a constraint-based planner.

1.1 Universally quantified constraints

Universally quantified constraints can be exceedingly useful when representing image processing domains. For example, to represent an image-processing command that performs a horizontal flip of the pixels in a rectangular region of an image between (MINX, MINY) and (MAXX, MAXY), we might write something like:

$\forall x, y$ **when**(MINX$\leq x \leq$MAXX && MINY$\leq y \leq$MAXY)
output.value(x,y) := *input*.value(MAXX+MINX-x,y)

where *output*.value(x,y) is the pixel value of the image *output* at coordinates x,y, and similarly for *input*.value. The keyword **when** indicates a conditional effect. We might also want to specify spatial transforms of an image, such as scaling or projections, or changes to color values. All of these are convenient to represent using numeric constraints, quantified over the pixels in the image or in the specified region.

In describing commands that act on text files, it is useful to quantify over lines or characters of text. For example, the `grep` command outputs all lines of text contained in the input that match a given regular expression:

$\forall$ `line` **when** (*input*.containsLine(*line*)
&& *input*.matches(*regexp*))
output.containsLine(*line*)

Similarly, many commands operate on sets of files, which can often be expressed in terms of a regular expression satisfied by their path names. For example, the files recursively contained in directory "/foo/bar" all have a path name matching "/foo/bar/.+", where ".+" means "any string at least one character long."

In these examples, we see that it is necessary to reason about constraints on variables with either infinite or very large domains.

1.2 Road map

In the remainder of the paper, we discuss how universally quantified constraints arise in the planning process and how they are solved. In Section 2 we describe how universally quantified constraints arise as subgoals in the planning process. In Section 3 we present a general approach to solving universally quantified constraints in a constraint network and an algorithm for implementing this approach, and we prove that the algorithm is both sound and complete. The general approach is not always possible to instantiate when there are infinite domains. In Section 4 we describe how to efficiently handle constraints with infinite domains under certain restrictions. In Section 5, we discuss how these techniques apply to an Earth Science domain that we are working on, and in Section 6 we present a detailed example covering both planning and constraint reasoning. In Section 7 we describe related work, and in Section 8 we conclude and describe future work.

2 Planning with universal quantification

The traditional approach to planning with universal quantification, used by UCPOP (Penberthy & Weld 1992) and other planners works as follows:

1. Universally quantified goals are replaced with the equivalent universally ground conjunctive goal, which is called the *universal base*.
2. Universally quantified effects are *peeled* as needed; that is, given an effect

   $\forall x$ **when**(P(x)) Q(x)

   and a goal, Q(a), a new ground effect is "peeled off" the forall effect to satisfy the goal:

   when(P(a)) Q(a)

   The result is the subgoal P(a).

Replacing goals with their universal base depends on the Closed World Assumption (all objects must be known) and on the number of objects in the universe being relatively small. In softbot domains, neither assumption is likely to be valid. For example, not all files accessible to the softbot will be known, and the number of available files can easily be thousands or millions. To address the problem that not all files are known, the softbot can first achieve a subgoal of knowing all the relevant files and then proceed as above (Etzioni, Golden, & Weld 1997), but that still leaves the problem that the number of files may be large. For example, suppose the softbot has the goal of making all of the files in the user's home directory group readable. This goal could be achieved by identifying all the files (recursively) contained in the home directory "~user" and then ensuring that each one is group readable, but it would take some time just to identify all the files. It is much simpler and faster to handle them all at once with a single Unix command, which recursively makes all files in the directory group readable:

```
chmod -R g+r ~user
```

Such an approach is supported in the PUCCINI planner (Golden 1998) by directly linking from universally quantified goals to universally quantified effects. The approach used by PUCCINI presupposes that the goals and effects are all expressed in terms of predicates, like group-readable, for which entailment can be determined using simple unification. When conditions include constraints as well as predicates, determining entailment requires additional mechanisms, as we discuss in Section 2.2.

2.1 Restrictions on universally quantified expressions

Given the requirement to support universally quantified goals directly with universally quantified effects, it is important to specify exactly what kinds of expressions the language will allow, since the unrestricted case would require first-order theorem proving, which is undecidable. In a goal, the use of the keyword **when** indicates that the antecedent and consequent refer to different times. For example, the goal **when**($\Phi(\vec{x})$) $\Psi(\vec{x})$ means that for all $\vec{x}$ that satisfy $\Phi(\vec{x})$ *when the goal is given* (*i.e.*, in the initial state), we want $\Psi(\vec{x})$ to be true *when the goal is achieved* (*i.e.*, in the final state). Thus, we can specify goals like "paint all the blue chairs green" without contradiction:

$\forall c$: chair **when** (c.color = `blue`) c.color = `green`

The planner has no control of what is true in the initial state, so it will never try to achieve the goal by falsifying the antecedent. To borrow a term from contingency planning, the antecedent specifies the *context* in which the consequent should be achieved.

Effects All universally quantified effects are conditional effects, in which the antecedent specifies restrictions on the universe(s) of the quantified variable(s) and the consequent specifies what will become true for members of the specified universes. These effects are of the form

$\forall \vec{x}, \vec{y}$ (**when**($\Phi(\vec{x}, \vec{y}, \vec{w})$) $\Psi(\vec{x}, \vec{w})$).

where Φ and Ψ are conjunctive expressions and variables in $\vec{w}$ are *action parameters*, variables in action schemas that need to be instantiated in order to obtain concrete actions. Limiting Φ to a conjunction is not a real limitation, since an expression of the form

$$\textbf{when } (\Phi_1 \vee \Phi_2)\ \Psi$$

can be rewritten as the conjunction of "**when**(Φ_1) Ψ" and "**when**(Φ_2) Ψ."

Effects cannot contain existential quantifiers,[1] or anything equivalent to existentials, such as universal quantifiers nested within an antecedent or negation. Allowing existentials or disjunctive consequents in effects would make them non-deterministic. Given the lack of nesting and existentials, all universals can be treated as free variables. All quantified variables appearing in Ψ must also appear in Φ. This is just a sanity check, since the domain of any quantified variable that does not appear in Φ is completely unrestricted. Φ may contain additional quantified variables, $\vec{y}$, that don't appear in Ψ.

Goals and preconditions The syntax of universally quantified goals and action preconditions is the same as that of effects, except that existential quantifiers nested within the universal quantifiers are allowed in Ψ:

$$\forall \vec{x}, \vec{y}, \exists \vec{z}\ (\textbf{when}(\Phi(\vec{x},\vec{y},\vec{w}))\ \Psi(\vec{x},\vec{z},\vec{w})).$$

All universal quantifiers precede all existential quantifiers; this is simply the negation of Skolem Normal Form. Goals can also explicitly refer to time. For example, we can ask for data on last Tuesday's rainfall. Whereas effects are not really restricted compared to the commonly supported subset of ADL (Pednault 1989), the limitations on universally quantified goals are more restrictive. This particular set of restrictions was chosen to support the class of goals required for the softbot domains that interest us, while simplifying the inference procedures.

2.2 Goal regression with quantified variables

The subgoaling, or goal regression, procedure we use is similar to that used by PUCCINI. We use the peeling technique outlined above, with the addition that quantified variables in the effect can be replaced by quantified variables in the goal. Suppose we have a goal **when**$(\Phi_g)\Psi_g$ that we want to satisfy using an effect **when**$(\Phi_e)\Psi_e$. If the right-hand side (RHS) of a goal Ψ_g contains multiple conjuncts, they are solved independently, so subgoals are all of the form **when**$(\Phi_g)\psi_g$, where ψ_g is a single literal. We rely on a unification function MGU(ψ_e, ψ_g), which returns the most general unifier between the effect literal ψ_e and the goal literal ψ_g. If the literals don't unify, MGU returns $\perp$. Otherwise, it returns a set of pairs $\{\langle v_e, v_g\rangle\}$, whose interpretation is that ψ_e unifies with ψ_g if all the constraints $v_e = v_g$ are satisfied.

The Goal Regression Algorithm To determine the conditions required for {**when**$(\Phi_e)\Psi_e$} to satisfy the goal {**when**$(\Phi_g)\psi_g$}, ψ_g is matched against each of the literals $\psi_e \in \Psi_e$, using the following procedure.

```
1.  regress({when(Φe)ψe}, {when(Φg)ψg})
2.    β =MGU(ψe,ψg)
3.    C = {}
4.    Φn := copy(Φe)
5.    if β =⊥ then return failure
6.    for each ⟨ve,vg⟩ ∈ β
7.      if ve is quantified ∀
8.         then replace ve in Φn with vg.
9.      else if vg is quantified ∀
10.        then return failure.
11.     else C := C ∧ (ve = vg).
12.   end for
13.   for each ve ∉ β
14.     replace ∀ve in Φn with ∃v'e
15.   end for
15.   return {when(Φg)Φn}∧C
```

The reason that unmatched universally quantified variables can be replaced with existentials (line 14) is as follows: since the effect occurs for all v that satisfy Φ, and v isn't mentioned in the goal, it is only necessary to find *some* value of v that satisfies Φ. Any new $\exists$ variables are written inside the scope of all $\forall$ variables from the goal.[2]

Examples of Goal Regression We will now present some examples of goal regression. Suppose that we have an action to give a Mothers' Day card to all new mothers:

$$\forall p_1, p_2\text{:person } \textbf{when}(p_1 = \text{parent}(p_2)\ \&\&\ \text{sex}(p_1) = \text{F}\ \&\&\ \text{age}(p_2) < 1)\ \text{has-card}(p_1)$$

and our goal is to give a card to Mary (*i.e.*, has-card(Mary)). Applying this action to satisfy the goal will result in the subgoal

$$\exists p_2'\text{:person } (\text{Mary} = \text{parent}(p_2')\ \&\&\ \text{sex}(\text{Mary}) = \text{F}\ \&\&\ \text{age}(p_2') < 1)$$

That is, the action will achieve the goal if Mary is female and has a child less than one year old.

Now suppose our goal is to give a card to all mothers of newborn boys:

$$\forall m, s\text{:person } \textbf{when}(m = \text{parent}(s)\ \&\&\ \text{sex}(m) = \text{F}\ \&\&\ \text{sex}(s) = \text{M}\ \&\&\ \text{age}(s) = 0)\ \text{has-card}(m)$$

If we use the action to give a card to all new mothers, the subgoal then becomes

$$\forall m, s\text{:person } \textbf{when}(m = \text{parent}(s)\ \&\&\ \text{sex}(m) = \text{F}\ \&\&\ \text{sex}(s) = \text{M}\ \&\&\ \text{age}(s) = 0)\ \{m = \text{parent}(s);\ \text{sex}(m) = \text{F};\ \text{age}(s) < 1\}$$

[1] Effects *can* introduce the creation of new objects, through the **new** keyword, which is similar in some respects to an existential quantifier, but that is outside the scope this paper.

[2] For completeness, it is also necessary to determine whether two or more effects combine to achieve a universally quantified goal. A technique called goal partitioning (Golden 1997), provides this ability, but at a high computational cost. We are investigating a way to lower this cost, but that is outside the scope of this paper.

Note that the left hand side of this expression is just the left-hand side of the original goal, and the right hand side is the "peeled" left hand side (LHS) of the effect. All subgoals from conditional effects are generated the same way, so the same LHS expression is carried back through successive goal regressions.

The right-hand side (RHS) literals m =parent(s) and sex(m) = F are clearly entailed by the LHS, which we can determine by unification, using a slight variation on the regression procedure above. When the LHS entails a literal on the RHS, we say that the goal literal is *trivially satisfied*, and remove it without further subgoaling.

The remaining goal condition, a constraint, is not so straightforward. Although age(s) = 0 clearly entails age(s) < 1, the two do not unify. As we discuss below, the purpose of reasoning about universally quantified constraints is to answer the entailment question for constraints.

The Form of Subgoals Subgoals are just goals, and obey the same restrictions. However, since subgoals are generated through a specific process, outlined above, it is worth showing that the process maintains the restriction on the form of subgoals.

- Since the subgoaling process always copies the LHS of the goal to the LHS of the subgoal, all restrictions obeyed by the former are obeyed by the latter. In particular, the LHS is conjunctive and it must not contain existentials.
- The RHS of the subgoal comes from the (peeled) LHS of the effect. Since the latter is conjunctive, so is the former.
- Quantified variables appearing in the RHS but not in the LHS are existential. To see why, consider that every quantified variable that appears in the RHS either originated in the goal or is a copy of a variable from the effect.

1. If the variable appeared in the goal, then it cannot have been in the LHS of goal, since otherwise it would be in the LHS of the subgoal, contradicting our assumption. Since it was not in the LHS of the goal, it must be an existential.
2. If the variable came from the effect, then it must be an existential, since, as indicated in line 14 of the regression algorithm, all universals in the effect that aren't replaced by variables from the goal are replaced by existentials.

2.3 From planning to constraints

In the remainder of the paper, we discuss how to tell if the LHS of a universally quantified subgoal entails the RHS when both sides contain constraints. We will not concern ourselves further with the details of the planning algorithm. We can convert the whole planning problem into a constraint problem, but it would also be possible to use a causal-link planner like PUCCINI (Golden 1998), and perform constraint reasoning to answer questions about whether certain subgoals are trivially satisfied (*i.e.*, the LHS entails the RHS). In either case, we can separate the problem of solving constraints to check subgoal satisfaction from the rest of the planning problem.

We assume that the planner produces candidate plans that are complete except for the instantiation of some action parameters and are correct subject to a list of subgoals being "trivially" satisfied (*i.e.*, no more actions need to be inserted into the plan). The planner sends the constraint reasoner this list of subgoals, which are of the form

$$\forall \vec{x}, \vec{y}, \exists \vec{z}\ (\Phi(\vec{x}, \vec{y}, \vec{w}) \Rightarrow \Psi(\vec{x}, \vec{z}, \vec{w}))$$

along with some additional constraints on the parameters. The job of the constraint network is to either return an assignment to all of the unspecified parameters ($\vec{w}$) such that all of the subgoals are trivially satisfied, or return failure in case there is no such assignment. If the constraint network returns failure then the candidate plan is invalid, so the planner should continue searching. Otherwise, the candidate plan, instantiated with the values for $\vec{w}$ returned by the constraint network, is a valid plan.

3 Solving Quantified Constraints

In order to determine whether the subgoals are trivially satisfied, it is necessary to reason about the solutions to the CSPs induced by Φ and Ψ. Before proceeding, we review some standard CSP notation. Let X be a set of variables. Denote the domain of $x \in X$ as $d(x)$. Let D be the set of domains. Let $k = (x_1 \dots x_i \dots x_n; R)$ be a constraint; $x_i \in X$ and $R \subseteq d(x_1) \times \dots \times d(x_n)$ is a relation defining the permitted assignments to the variables. Let K be the set of constraints. Then $C(X) = (X, D, K)$ is a CSP. A *solution* to the CSP is an assignment of values to the variables such that all constraints are satisfied. Let $S(C)$ be the set of solutions to C. Let L be a relation on a set of variables U, and let $\pi_V(L)$ be the projection of the relation L onto the set $V \subseteq U$. A CSP is *k-consistent* if any consistent assignment to k-1 variables can be extended to an assignment to k variables (k=2 is arc consistency.) A CSP is *strongly k-consistent* if it is j-consistent for all j$\leq$k.

Having reviewed these definitions, we now formally define quantified constraints:

Definition 1 *Let* Φ, Ψ *be CSPs. We then refer to a subgoal* $\forall \vec{x}, \vec{y} \exists \vec{z} (\Phi(\vec{x}, \vec{y}, \vec{w}) \Rightarrow \Psi(\vec{x}, \vec{z}, \vec{w}))$ *as a* quantified constraint, *and refer to the constraints comprising* Φ, Ψ *as* primitive constraints. *A quantified constraint is* satisfied *for* $\vec{w} = \vec{\theta}$ *iff* $\pi_{\{\vec{x}\}} S(\Phi(\vec{x}, \vec{y}, \vec{\theta})) \subseteq \pi_{\{\vec{x}\}} S(\Psi(\vec{x}, \vec{z}, \vec{\theta}))$.

The general approach to solving quantified implications is straightforward. Given an expression of the form "all things that satisfy Φ also satisfy Ψ," we identify the set of things that satisfy Φ and check whether they also satisfy Ψ. We can think of this as an empirical proof technique: we're doing nothing more than checking the validity of the expression for all members of the universe.

Given a quantified constraint

$$\forall \vec{x}, \vec{y} \exists \vec{z} (\Phi(\vec{x}, \vec{y}, \vec{w}) \Rightarrow \Psi(\vec{x}, \vec{z}, \vec{w})),$$

the variables in $\vec{w}$ must be assigned values by a search procedure. As mentioned in Section 2, these variables represent the parameters of actions; the search over these values is a search over candidate plans. During this search, we can propagate the domains of the variables in $\vec{x}, \vec{y}$ based on Φ, but do not assign these variables. We do not propagate based on the constraints in Ψ, because these constraints do not hold if the domains of the variables in Φ are empty. Once all of these variables are assigned, we are left with the constraint

$$\forall\vec{x},\vec{y}\exists\vec{z}(\Phi(\vec{x},\vec{y}) \Rightarrow \Psi(\vec{x},\vec{z})),$$

where $\vec{x}$ represents one or more universally quantified variables common to Φ and Ψ. Again, as described above, the desired semantics of this implication is that everything satisfying Φ also satisfies Ψ. Thus, we must identify the set of tuples corresponding to the assignments to $\vec{x}$ that satisfy $\Phi(\vec{x},\vec{y})$, and check that each tuple also satisfies $\Psi(\vec{x},\vec{z})$. To do this, we solve both $\Phi(\vec{x},\vec{y})$ and $\Psi(\vec{x},\vec{z})$ for $\vec{x}$. We then check to see if $\pi_{\{\vec{x}\}}S(\Phi(\vec{x},\vec{y})) \subseteq \pi_{\{\vec{x}\}}S(\Psi(\vec{x},\vec{z}))$. Because the quantified constraint takes the form of an implication, if the set of solutions to Φ is empty, then the implication is satisfied vacuously, and there are no constraints on the values of the variables in $\vec{x}$. If there are solutions to Φ but $\pi_{\{\vec{x}\}}S(\Phi(\vec{x},\vec{y})) \not\subseteq \pi_{\{\vec{x}\}}S(\Psi(\vec{x},\vec{z}))$, then the quantified constraint is not satisfied, and some other assignment to the variables in $\vec{w}$ must be generated. Otherwise, the constraint is satisfied, and the domains of $\vec{x}$ are defined by the the restrictions imposed by Φ.

If the set of tuples satisfying Φ is finite, then enumerating them and checking that each one of them satisfies Ψ is relatively straightforward, though possibly time consuming. But what if the set is infinite? In the general case, there is nothing that can be done. However, as we will see, there are some useful classes of problems where it is possible to identify the infinite set of tuples satisfying $\Phi(\vec{x},\vec{y})$ and check that they all satisfy $\Psi(\vec{x},\vec{z})$ using efficient constraint propagation techniques.

It should be noted that the steps presented above can be done in a variety of ways. There is no need to assign all variables in $\vec{w}$ before beginning the process of identifying the domain of $\vec{x}$. It is also possible to fix the domains of $\vec{x}$ after solving Φ before solving Ψ and only check to see if any elements of these domains are eliminated during the solving of Ψ. These refinements are left as future work.

We present an algorithm for proving that quantified constraints are satisfied. The only assumptions are that there is a way of enumerating the variables in $\vec{w}$, and that there is some way of representing the values satisfying $\Phi(\vec{x},\vec{y})$ and $\Psi(\vec{x},\vec{y})$. In the following sections, we discuss specific techniques for performing these operations.

```
1. isSatisfied(γ)
2.  choose assignments for all variables w⃗.
3.  for (each (∀x⃗,y⃗,∃z⃗.Φ(x⃗,y⃗) ⇒ Ψ(x⃗,z⃗)) ∈ γ)
4.    if (S(Φ(x⃗,y⃗)) ≠ ∅)
5.       for (each α⃗ ∈ π_{x⃗} S(Φ(x⃗,y⃗)))
6.           if (α⃗ ∉ π_{x⃗} S(Ψ(x⃗,y⃗)))
7.              return failure.
8.       end for
9.  end for
10. return success.
```

We now prove that the algorithm is both sound and complete:

Theorem 1 *The algorithm for checking the satisfiability of quantified constraints is sound: it will not return success if, for any quantified constraint,* $\forall\vec{x},\vec{y},\exists\vec{z}.\Phi(x,\vec{y},\vec{w}) \Rightarrow \Psi(\vec{x},\vec{z},\vec{w})$*, there is some assignment* $\vec{\alpha}$ *to* $\vec{x}$ *such that* $\exists\vec{y},\forall\vec{z}.\Phi(\vec{\alpha},\vec{y},\vec{w}) \wedge \neg\Psi(\vec{\alpha},\vec{z},\vec{w})$.

Proof: Suppose otherwise. Then there is some some $\vec{\alpha}$ such that $\exists\vec{y},\forall\vec{z}.\Phi(\vec{\alpha},\vec{y},\vec{w}) \wedge \neg\Psi(\vec{\alpha},\vec{z},\vec{w})$. The algorithm will only return success if each $w_i \in \vec{w}$ is singleton, and line 7 is not reached. This happens if

1. There are no quantified constraints (line 3). This contradicts the assumption that there is such a constraint.
2. $S(\Phi(\vec{x},\vec{y},\vec{w})) = \emptyset$ (line 4). This is equivalent to saying Φ is false for all $\vec{x}$, contradicting our assumption that there was some $\vec{\alpha}$ for which Φ was true.
3. $S(\Phi(\vec{x},\vec{y},\vec{w})) \neq \emptyset$ and there is no $\vec{\alpha}$ such that $\vec{\alpha} \in \pi_{\{\vec{x}\}}S(\Phi(\vec{x},\vec{y},\vec{w}))$ and $\vec{\alpha} \notin \pi_{\{\vec{x}\}}S(\Psi(\vec{x},\vec{y},\vec{w}))$ (lines 5,6). That is, there is no $\vec{\alpha}$ such that $\exists\vec{y}.\Phi(\vec{\alpha},\vec{y},\vec{w})$ and $\forall\vec{z}.(\neg\Psi(\vec{\alpha},\vec{z},\vec{w}))$, contradicting the assumption that $\exists\vec{y},\forall\vec{z}.\Phi(\vec{\alpha},\vec{y},\vec{w}) \wedge \neg\Psi(\vec{\alpha},\vec{z},\vec{w})$.

Theorem 2 *The algorithm for checking the satisfiability of quantified constraints is complete: If, for all quantified constraints,* $\forall\vec{x},\vec{y},\exists\vec{z}.\Phi(x,\vec{y},\vec{w}) \Rightarrow \Psi(\vec{x},\vec{z},\vec{w})$*, then the algorithm returns success.*

Proof: Suppose the algorithm returns failure, but for all quantified constraints, $\forall\vec{x},\vec{y},\exists\vec{z}.\Phi(x,\vec{y},\vec{w}) \Rightarrow \Psi(\vec{x},\vec{z},\vec{w})$. The algorithm will return failure if there is some quantified constraint for which $S(\Phi(\vec{x},\vec{y},\vec{w})) \neq \emptyset$ and $\vec{\alpha} \in \pi_{\{\vec{x}\}}S(\Phi(\vec{x},\vec{y},\vec{w}))$ but $\vec{\alpha} \notin \pi_{\{\vec{x}\}}S(\Psi(\vec{x},\vec{z},\vec{w}))$ (line 6). But then $\pi_{\{\vec{x}\}}S(\Phi(\vec{x},\vec{y},\vec{w})) \not\subseteq \pi_{\{\vec{x}\}}S(\Psi(\vec{x},\vec{z},\vec{w}))$, which in turn violates the assumption that for all quantified constraints, $\forall\vec{x},\vec{y},\exists\vec{z}.\Phi(x,\vec{y},\vec{w}) \Rightarrow \Psi(\vec{x},\vec{z},\vec{w})$.

Complexity Let n_Φ be the number of variables in Φ and let d_Φ be the size of the largest domain of any variable in Φ. Denote n_Ψ and d_Ψ similarly. The complexity of the algorithm is $O((d_\Phi)^{n_\Phi} + (d_\Psi)^{n_\Psi})$, because checking the satisfiability of the constraints potentially requires enumerating the solution space for both CSPs Φ,Ψ.

4 Handling infinite universes

The general approach discussed in Section 3 works for relatively small, finite domains. To handle large or infinite domains efficiently, we need to employ special-case constraint propagation techniques. We describe one such technique in detail in this section. The technique depends on being able to represent infinite domains concisely. In Sections 4.1 and 4.2, we discuss concise representations of infinite domains for numbers and strings, and describe classes of constraints for which these concise representations can store the valid domains exactly. In Section 4.3, we describe further restrictions on the form of the quantified constraints that allow us to check the satisfiability of these quantified constraints efficiently, even if the variable domains are infinite.

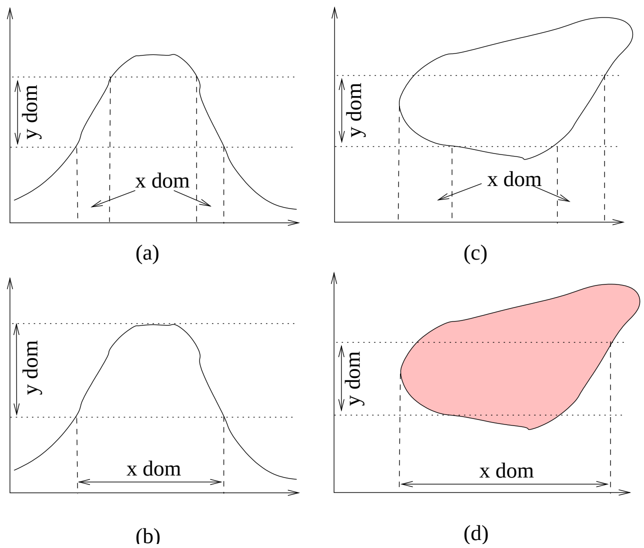

Figure 1: Reasoning about numeric functions and relations

4.1 Numeric domains

Large or infinite sets of numbers can be represented concisely using intervals. Additionally, we can determine whether one interval contains another efficiently. If we assume that all infinite numeric domains are represented as single intervals, the question of whether the domain of a numeric variable can represent exactly the possible values allowed by a constraint reduces to the question of whether the values for that variable allowed by the constraint can be represented as an interval. Assuming that the domains of the other variables in the constraint are also represented as intervals, the question then becomes whether the projection of an interval on one variable is an interval on another. We will consider both continuous (real) and discrete (integer) domains.

Continuous If the domain of x is continuous, then for every continuous function $y = f(x)$, if the domain of x is an interval, the domain of y will also be an interval. The converse is not necessarily true. However, the converse is true if f is either non-decreasing or non-increasing. If $f(x)$ increases and decreases in x, then there will be some y interval that corresponds to multiple x intervals (Figure 1a). However, if the y interval obeys certain restrictions, then the domain of x will still be an interval. In particular,

- neither of the horizontal lines representing the bounds of the y interval may cross f more than twice. Crossing twice corresponds to passing through one peak or trough in f.
- if one of the lines passes through a peak, the other line must be above the peak (Figure 1b), and if one line passes through a trough, then the other line must be below the trough.

We can apply the same sort of reasoning to relations (Figure 1c); however a special class of relations is worth noting. If any relation defines a convex region (Figure 1d), such that the relation is true for all points inside the region and false for all points outside it, then the projection of any interval on y will be an interval on x (or vice versa). Examples of convex regions are: $x < 10$, $y > 2x+1$, $x^2+y^2 \leq r^2$.

Continuous to discrete A function from a continuous (real) variable to a discrete (integer) variable is by definition not a continuous function. However, it may be regarded as a continuous function whose range is projected onto the integer number line. If such a description is valid, then the projection of any continuous interval on x will be a discrete interval on y. Going the other direction, intervals on y will map to intervals on x under the same circumstances as in the fully continuous case: non-decreasing functions, non-increasing functions, and relations defining convex regions.

Discrete A function whose domain is discrete will not, in general, project an interval onto another interval. For example, consider the simple case of $y = 2x$, where x and y are integers. The domain of y is the set of even numbers, which cannot be represented as an interval. However, when we consider relations defining convex regions, we again find that the projection of an interval is an interval. So although $y = 2x$ does not give an interval, $y \leq 2x$ does.

Other domain representations The decision to represent a numeric domain using a single interval has had a profound impact on the class of constraints that we can “solve” for particular variables. Another representation, such as a finite set of intervals, would allow additional constraints to be handled, though at the cost of additional complexity in constraint execution.

4.2 String domains

Just as infinite sets of numbers can be represented by intervals, infinite sets of strings can be represented by regular expressions. Regular expressions are a much more flexible representation than intervals, in that the set of regular expressions is closed under intersection, union and negation, whereas the set of intervals is only closed under intersection. Regular expressions (regexps) are equivalent to finite automata (FAs) in expressive power, and in fact we represent regexps as FAs, since the latter are easier to compute with. For example, deciding whether two FAs accept the same language can be done efficiently.

Concatenation The concatenation of two strings, x and y, yields another string, z. This constraint is represented as $z = x + y$. If the domains of x and y are regexps, the domain of z will simply be the regexp resulting from concatenating the regexps for x and y.

Less obviously, if the domains of x and z are regexps, the domain of y is a regexp. To construct an FA for y given FAs for x and z, we in effect traverse the FAs for z and x in parallel, exploring the cross-product of the nodes from the two FAs, starting with the pair of initial states and adding a transition $\{s_n, t_m\} \stackrel{lab}{\rightarrow} \{s_p, t_q\}$ from every node $\{s_n, t_m\}$ and every label lab such that the transitions $s_n \stackrel{lab}{\rightarrow} s_p$ and $t_m \stackrel{lab}{\rightarrow} t_q$ appear in the original FAs (see Figure 2). This is simply the operation that is performed when intersecting two FAs. Whenever

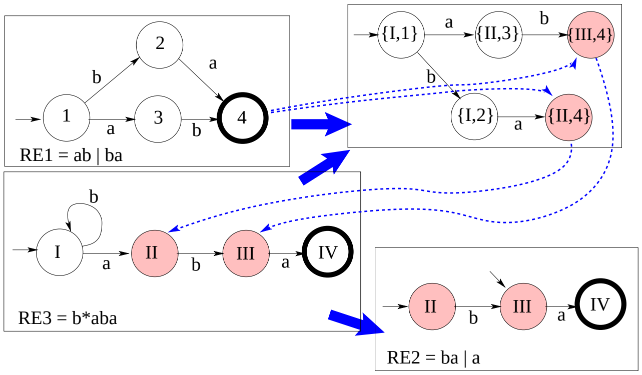

Figure 2: Given FAs for RE1 and RE3, find an FA for RE2 such that RE3 is concatenation of RE1 and RE2. First, traverse FAs for RE3 and RE1 in parallel, constructing cross-product FA (upper right). Then, identify states that are accept states for RE1 and mark the corresponding states in the FA for RE3 (shaded circles). Construct a new NFA (bottom) for RE2 by copying FA for RE3 and making marked nodes start nodes.

we reach a node $\{s,t\}$, such that node s is an accept state in the FA for x, we mark node t. After the traversal is complete, the marked nodes in the FA for z represent all of the states that can be reached by reading a string accepted by x.

A new nondeterministic FA (NFA) for y is constructed by copying the FA for z, making the start node a non-start node and making all the marked nodes new start nodes. The complexity of the whole operation is dominated by generating the cross-product FA (O(mn), where m and n are the number of nodes in the FAs for x and z, respectively). A similar procedure can be used to construct an NFA for x, given FAs for y and z.

Note that, in Figure 2, the FA for RE3 does not yet reflect the concatenation constraint. That is, RE3 accepts strings, such as bbaba, for which RE1 is not a prefix. When the constraint is enforced for all three variables, RE3 = aba | baba. It doesn't matter what order the variables are considered.

Containment The relation contains(a, b) means that string b is a substring of a. If the domain of b is a regexp r, then the domain of a is simply the regexp ".*r.*", where "." means "accept any character," so ".*" means "accept any string of zero or more characters." Less obviously, if the domain of a is a regexp, then so is the domain of b. Given an FA for a, we can construct an NFA for b by eliminating any dead-end nodes from a (that is, nodes from which it is impossible to reach an accept node), and then making all nodes in a both start and accept nodes.

4.3 Tractable Reasoning

In the previous sections we established that we can enforce consistency on a variety of constraints, even when the domains are infinite. We now show how to use these results to demonstrate that a quantified constraint is satisfied. In order to do this, we need some additional definitions. Let $C(X)$ be a CSP. Consider the hypergraph G_C, where the vertices of G_C are the variables of C and the hyperedges are the constraints. Assume we have imposed a total order o on the variables X. Freuder (Freuder 1982) defines the *width* of a variable $x \in X$ induced by ordering o as the number of variables earlier in the ordering that are in the scope of a constraint on x. The width of an ordering o is the maximum width of any variable induced by the ordering o, and the width of a CSP is the minimum width over all orderings.

We restate the following theorem from (Freuder 1982) without proof:

Theorem 3 *Let C be a CSP. If C is strongly k-consistent and the width of C is $< k$, then there is a variable order that will result in a backtrack-free search for a solution to C.*

We can now prove the following:

Corollary 1 *Let C be a CSP and assume C is strongly k-consistent and the width of C is $w < k$. Let x be the first variable in a search order inducing a width of $w < k$. Then $d(x) = \pi_x(S(C))$.*

Proof: We will show that each element of $d(x)$ can be extended to a solution to C. For each $\alpha \in d(x)$, make the assignment $x = \alpha$. Consider the assignment of any variable y. Now, since the width of C is $w < k$, we know that when we use a variable ordering that induces a width $w < k$, fewer than k variables sharing constraints with y are assigned before assigning y. Further, since we also know that C is strongly k-consistent, any consistent assignment of fewer than k variables can always be extended by one assignment. Thus, we can continue assigning variables without failure until all variables are assigned, regardless of the initial assignment to x.

Theorem 4 *Let $\forall \vec{x}, \vec{y}, \exists \vec{z}. \Phi(x, \vec{y}, \vec{w}) \Rightarrow \Psi(x, \vec{z}, \vec{w})$ be a quantified constraint such that:*

1. *Φ and Ψ share one universally quantified variable x whose domain is infinite, and x and any other infinite domain variables are only involved in constraints for which strong k-consistency can be enforced.*
2. *Φ and Ψ are strongly k-consistent.*
3. *There exists an ordering o_1 such that Φ has width $w < k$ induced by o_1 and x is the first variable in the order.*
4. *There exists an ordering o_2 such that Ψ has width $w < k$ induced by o_2 and x is the first variable in the order.*

Then the quantified constraint is satisfied if and only if $d_\Phi(x) \subseteq d_\Psi(x)$.

Proof: Since x is the only universally quantified variable shared between Φ and Ψ, we only need to check that $\pi_{\{x\}} S(\Phi(x, \vec{y}, \vec{w})) \subseteq \pi_{\{x\}} S(\Psi(x, \vec{z}, \vec{w}))$. Since we have assumed Φ and Ψ are k-consistent, and that each has an ordering that induces width less than k, the previous theorem allows us to conclude that all values of the first variable in the ordering are part of the solution space. But we have also assumed that, for both orderings, that variable is x. Thus, $\pi_{\{x\}} S(\Phi(x, \vec{y}, \vec{w})) = d_\Phi(x)$ and $\pi_{\{x\}} S(\Psi(x, \vec{z}, \vec{w})) = d_\Psi(x)$, and we are done.

We are now confronted with the problem of establishing strong k-consistency. For CSPs with variables with infinite domains, arc-consistency can be enforced on tree-structured (width 1) CSPs in polynomial time, but no stronger result is known. In the case of finite domains, Freuder (Freuder 1990) has shown that, for certain families of CSPs called k-trees, strong k-consistency can be established in polynomial time in the number of variables. Our current implementation maintains strong k-consistency for primitive k-ary constraints over infinite numeric or string domains but only maintains arc consistency globally. Thus, we limit our attention to tree-structured CSPs.

Universally quantified constraints with infinite domains can be solved in time polynomial in the number of variables, but it is also necessary to consider the cost of computing the domain for each variable. In the case of numeric constraints, this cost is generally trivial, consisting of a few arithmetic operations. In the case of string domains, the cost depends on the size of the regular expressions representing the domains. Given two domains represented by FAs of size m and n, intersection of the two domains is $O(mn)$, union is $O(m+n)$, negation is $O(m)$, and enforcement of the constraints discussed in Section 4.2 is at worst $O(mn)$. However, some of these operations produce NFAs as outputs, and others require deterministic FAs (DFAs) as inputs. Converting from an NFA to a DFA can result in an exponential increase in the size of the FA.

5 Applicability

We have implemented this approach in a constraint-based planner and are applying it to an Earth Science data processing domain that involves a mixture of image processing, text processing and other operations. Preliminary results indicate that the assumptions we make in this paper are valid for this domain. There are two main assumptions that potentially limit the applicability of our approach.

1. Constraints can be fully captured by the domain representation. This is really only a limitation for numeric constraints, since every string constraint in the domain can be captured fully using regexps. Most numeric constraints that appear in universally quantified expressions represent either convex regions of images or functions from real-valued measurements to integral pixel values. These constraints all obey this restriction.
2. The width of the constraint network defined by quantified constraints must less than the level of consistency enforced, and the left and right hand sides must share at most one quantified variable. This is a more serious limitation. Since the nature of the quantified constraints is dictated by quantified goals, it is possible to formulate goals that violate this restriction. Since the set of goals is open, we can't draw any conclusions about which goals are common without extensive user tests. On the other hand, in most goals we have looked at, quantified constraints result in tree-structured CSPs that trivially obey our assumptions.

6 An Image Processing Example

In this section, we illustrate the entire planning process, including generating subgoals through regression, determining entailment through unification and computing entailment for universally quantified constraints with infinite domains.

Suppose we have a grayscale image corresponding to the elevation over some region:

```
plot.xSize = XMAX;
plot.ySize = YMAX;
∀x,y: unsigned, el: real.
    when(x < XMAX && y < YMAX &&
          el=elevation(xProj(x),yProj(y)))
        plot.value(x, y) = hProj(el)
```

where words in ALL CAPS are constants, xProj and yProj are linear functions mapping the x, y coordinates of the image to the corresponding longitude, latitude that they represent, hProj is a linear function mapping elevation to pixel values in the image, with lower (blacker) values correspond to lower elevations, and elevation(x, y) is the elevation at longitude x, latitude y. The notation `plot`.xSize denotes the horizontal size of the image `plot`, and `plot`.value(x, y) means the pixel value at the coordinates x,y in the image `plot`.

Say we would like to produce a color image showing the same elevations, but highlighting particular ranges of elevation using different colors. For example, pixels corresponding to points below sea level should be blue and points above the snow line should be shades of gray.

One way to accomplish this would be by creating bitmaps or monochrome images corresponding to the the pixels of interest (*i.e.*, pixels above or below a particular value), and using these bitmaps to select the pixels on which particular operations, like coloring the pixels blue, will be performed. Suppose we have a `threshold` command, which takes an image, *in*, as input and has an argument specifying a threshold value, and outputs an image, *out*, the same size as the input, with a value of 255 for every pixel in the input whose value is above the threshold and a value of zero for every pixel below the threshold:

```
∀x,y: unsigned, v: pixelValue
  when ( x < in.xSize && y < in.ySize &&
         v = in.value(x, y) )
      when (v≤thresh) out.value(x,y) := 0;
      when (v>thresh) out.value(x,y) := 255;
```

where *thresh* is an action parameter of type pixelValue (*i.e.*, a variable from $\vec{w}$) denoting the threshold value, and a pixelValue is an integer in the range $[0,255]$. The use of nested **when** statements is merely a shorthand, where "**when** (Φ_1) {**when** (Φ_2) Ψ}" is equivalent to "**when** $(\Phi_1 \wedge \Phi_2)$ Ψ}." Here, we focus on a single subgoal that arises during planning: to generate a threshold map, *sea*, based on elevation at sea level:

```
∀x′,y′: unsigned, elev: real.
  when(x′<XMAX && y′<YMAX &&
        elev=elevation(xProj(x′),yProj(y′)))
      when (elev > 0) sea.value(x,y) = 255;
      when (elev ≤ 0) sea.value(x,y) = 0;
```

Regressing this subgoal through the `threshold` action, we get:

$\forall x', y'$: unsigned, *elev*: real, $\exists v'$:unsigned
when(x'<XMAX && y'<YMAX &&
elev=elevation(xProj(x'),yProj(y')))
x' < *in*.xSize;
y' < *in*.ySize;
v' = *in*.value(*x, y*);
when (*elev* > 0) v'>*thresh*;
when (*elev* ≤ 0) $v' \leq$*thresh*;

We try to satisfy this goal using the initial state; specifically, letting the image *in* be plot.

$\forall x', y'$: unsigned, *elev*: real $\exists v'$: unsigned $\exists el'$:real
when(x'<XMAX && y'<YMAX &&
elev=elevation(xProj(x'),yProj(y')))
x' < XMAX;
y' < YMAX;
v' = hProj(el');
el' =elevation(xProj(x'),yProj(y')));
in =plot;
when (*elev* > 0) v'>*thresh*;
when (*elev* ≤ 0) $v' \leq$*thresh*;

The subgoal el' =elevation(xProj(x'),yProj(y'))) is trivially satisfied by unification if $el' = elev$. The subgoals x' < XMAX and y' < YMAX are also trivially satisfied. This can be determined easily by quantified constraint reasoning: The domain of x' established by the LHS is [0,XMAX-1], and the same domain is established by the RHS. Removing the satisfied terms, we get:

$\forall x', y'$: unsigned, *elev*: real $\exists v'$: unsigned $\exists el'$:real
when(x'<XMAX && y'<YMAX &&
elev=elevation(xProj(x'),yProj(y')))
v' = hProj(el');
el'=elev;
when (*elev* > 0) v'>*thresh*;
when (*elev* ≤ 0) $v' \leq$*thresh*;

which, simplified to it essence, gives us the following two quantified constraints.

$\forall e_1$: real. $(e_1 > 0) \Rightarrow$(hProj(e_1)>*thresh*)
$\forall e_2$: real. $(e_2 \leq 0) \Rightarrow$(hProj(e_2)$\leq$*thresh*)

Recall that hProj is an increasing linear function. Assume hProj(e)=$0.05e+42$. Note that although the domain of hProj is unbounded, the range is $[0,255]$, so all values of e below -840 map to 0, and all values above 4260 map to 255. Since we map real values onto integers, we will always round up.

These constraints share the parameter *thresh*, which needs to be assigned a value. As discussed above, there are a number of possible variable ordering strategies we could employ, the default being to choose a value for *thresh* and then see if the quantified constraints are satisfied. Say we pick the value 43. Let's tackle the constraint on e_1 first. Enforcing the LHS constraint sets the domain of e_1 to the interval $(0,\infty)$. On the RHS, propagating the value of *thresh* sets the domain of hProj(e_1) to $[44,255]$. The domain of e_1 then becomes $(20,\infty)$. Since the domain of e_1 is not the same as it was according to the LHS, the constraint is violated, so 43 is not a valid assignment to *thresh*.

Now say we pick 42. Once again, the domain of e_1 is $(0,\infty)$. This time, propagating *thresh* in the RHS makes the domain of hProj(e_1) $[43,255]$, resulting in a domain for e_1 of $(0,\infty)$, which is consistent with the LHS, so we proceed to the other forall constraint. Enforcing the LHS sets the domain of e_2 to the interval $(-\infty,0]$. Propagating the value of *thresh* in the RHS sets the domain of hProj(e_2) to $[0,42]$, resulting in a domain of $(-\infty,0]$ for e_2. Both forall constraints are consistent.

An alternative to branching on values of *thresh* would be to leave it unassigned and see if we can narrow down the choices through propagation. Working on the constraint on e_1 first, we enforce the LHS constraint, setting the domain of e_1 to the interval $(0,\infty)$. Propagating the value of e_1, the domain of hProj(e_1) is then $[43,255]$ and the domain of *thresh* is $[42,255]$. Since enforcing the RHS constraints did not shrink the domain of e_1, the first implication is valid so far. Enforcing the LHS of the second constraint sets the domain of e_2 to the interval $(-\infty,0]$. Enforcing the RHS sets the domain of hProj(e_2) to $[0,42]$ and restricts the domain of *thresh* to the singleton 42. The domain of e_2 did not shrink, and the reduction of the domain of *thresh* did not shrink the domain of e_1, so both implications hold, and the only valid parameter choice is 42, which is hProj(0), the pixel value corresponding to sea level.

7 Previous Work

Other planners, including (Golden, Etzioni, & Weld 1994; Golden 1998; Babaian & Schmolze 2000) also support universal quantification. The universally quantified statements in PSIPLAN (Babaian & Schmolze 2000) can include inequality constraints, which are used to exclude individuals from the universe of discourse. However, no prior planning systems support the ability to determine the validity of universally quantified constraints that we discuss here.

The Amphion system (Stickel *et al.* 1994) was designed to construct programs consisting of calls to elements of a software library. Amphion is supported by a first-order theorem prover. The task of assembling a sequence of image processing commands is similar to the task Amphion was designed to solve. However, the underlying representation we present here is a subset of first-order logic, enabling the use of less powerful reasoning systems. The planning problem we address is considerably easier than general program synthesis in that action descriptions are not expressive enough to describe arbitrary program elements, and the plans themselves do not contain loops or conditionals.

Ginsberg and Parkes (Ginsberg & Parkes 2000) point out that the satisfiability encoding of many STRIPS planning problems requires creating multiple grounded instances for axioms of the form $\forall xyz.(a(x,y) \wedge b(y,z) \Rightarrow c(x,z)$, then performing search over the truth values for all of the grounded instances. They propose a formulation in which $a(x,y)), b(y,z)$ and $c(x,z)$ are constraints on variables x,y,z and use this formulation to either search for units or find good variables to flip in local search. This is a different restriction on first-order logic from that we use, and furthermore, the domains of x,y,z are implicitly assumed to be finite.

L'Homme (L'Homme 1993) and Marriott and Stuckey (Marriott & Stuckey 1998) both describe methods of preserving an interval representation of variables involved in arithmetic constraints while eliminating infeasible values. However, they explicitly assume that the interval representation is an unsound approximation to the domain of feasible values. Benhamou and Goualard (Benhamou & Goualard 2000) describe a method of sound but incomplete approximate propagation of infinite domains. Since we require both soundness and completeness in cases where that set may be infinite, we have made stronger restrictions on the types of reasoning performed.

8 Conclusions and Future Work

We have described a planning methodology for softbots that supports universal quantification, incomplete information, and constraints on variables with very large or infinite domains. We restrict the form of both goals and effects, while preserving the ability to express conditional effects and reason about incomplete information. Our approach uses a combination of unification and constraint reasoning to demonstrate entailment. We described an algorithm for proving or disproving entailment for constraints over finite domains, and identified a subclass of constraints for which the same algorithm can prove or disprove entailment for variables with infinite domains. This class of constraints has proven useful in the domains of planning for image processing and managing file archives.

When describing the algorithm to validate quantified constraints, we assumed that all parameters of the actions were assigned before validation occurs. As described in Section 6, there are times when it is worth deferring the decision about parameters to actions, because propagation will limit the possibilities. Exploiting these possibilities is the subject of future work.

We can potentially weaken the conditions on quantified constraints required to reason about variables with infinite domains. The condition that Φ and Ψ share only one variable can be relaxed when there is a procedure for checking the validity of the constraint without checking infinitely many values. One case is when all of the constraints describe linear equations or inequalities. In addition, it may be possible to generalize the conditions under which consistency enforcement allows us to conclude that all the values of a variable participate in solutions to a CSP. Finally, we can try to find more constraints on which we can enforce consistency when domains are infinite.

Acknowledgments We would like to thank Tania Bedrax-Weisss, Ari Jónsson, Wanlin Pang, Robert Morris and Ellen Spertus for their helpful comments and contributions to this work. This work was supported by the NASA Intelligent Systems program.

References

Babaian, T., and Schmolze, J. 2000. Psiplan: Open world planning with ψ-forms. In *Proceedings of the 5th Conference on Artificial Intelligence Planning and Scheduling*.

Benhamou, F., and Goualard, F. 2000. Universally quantified interval constraints. In *Proceedings of the 6th International Conference on the Principles and Practices of Constraint Programming*, 67–82.

Chien, S.; Fisher, F.; Lo, E.; Mortensen, H.; and Greeley, R. 1997. Using artificial intelligence planning to automate science data analysis for large image database. In *Proc. 1997 Conference on Knowledge Discovery and Data Mining*.

Etzioni, O., and Weld, D. 1994. A softbot-based interface to the Internet. *C. ACM* 37(7):72–6.

Etzioni, O.; Golden, K.; and Weld, D. 1997. Sound and efficient closed-world reasoning for planning. *J. Artificial Intelligence* 89(1–2):113–148.

Freuder, E. 1982. A sufficient condition for backtrack-free search. *Journal of the Association for Computing Machinery* 29(1):24–32.

Freuder, E. 1990. Complexity of k-tree structured constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artificial Intelligence*, 4–9.

Ginsberg, M., and Parkes, A. 2000. Satisfiability algorithms and finite quantification. In *Proceedings of the 7th Conference on Knowledge Representation*.

Golden, K.; Etzioni, O.; and Weld, D. 1994. Omnipotence without omniscience: Sensor management in planning. In *Proc. 12th Nat. Conf. AI*, 1048–1054.

Golden, K. 1997. *Planning and Knowledge Representation for Softbots*. Ph.D. Dissertation, University of Washington. Available as UW CSE Tech Report 97-11-05.

Golden, K. 1998. Leap before you look: Information gathering in the PUCCINI planner. In *Proc. 4th Intl. Conf. AI Planning Systems*.

Lansky, A. L., and Philpot, A. G. 1993. AI-based planning for data analysis tasks. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence for Applications (CAIA-93)*.

L'Homme, O. 1993. Consistency techniques for numeric csps. In *Proceedings of the 13th International Conference on Artificial Intelligence*.

Marriott, K., and Stuckey, P. 1998. *Programming with Constraints: An Introduction*. The MIT Press.

Pednault, E. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. Principles of Knowledge Representation and Reasoning*, 324–332.

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. Principles of Knowledge Representation and Reasoning*, 103–114. See also `http://www.cs.washington.edu/research/projects/ai/www/ucpop.html`.

Stickel, M.; Waldinger, R.; Lowry, M.; Pressburger, T.; and Underwood, I. 1994. Deductive composition of astronomical software from subroutine libraries. In *Proceedings of the 12th Conference on Automated Deduction*.