

Semantics

Inference rules are useful for a variety of reasons. They provide concise and elegant representations of a certain class of algorithms such as transitive closure, Dijkstra shortest path, or the inside/outside algorithm. They provide provide a simple programming language in which one can write all and only polynomial time programs. They also provide a formal model of human reasoning that plays an important role in the philosophy of mind. It seems that some (but not all) aspects of cognition may be best modeled by inference processes.

In order to understand inference it is important to understand the distinction between provability (those assertions that can be deduced) and truth (those assertions that are true). The study of truth is part of the study of *semantics*. Semantics is the study of the meaning of expressions. In later sections of the notes we will explicitly discuss the semantics of sentences of natural languages such as English. In this section we start by studying semantics in the more formal, and much simpler, setting of artificial languages. We consider formal assertions such as those manipulated by inference rules. Rather than define an inference process, however, we define the notion of a "situation" and define a meaning function such that each assertion is either true or false in each situation. In the terminology of mathematical logic the situations are called *models* and the meaning of an expression is called the *denotation* of that expression. A language of assertions together with a set of models (situations) and a way of assigning a truth value to an assertion in a model is called a *logic*.

1 The Variety of Possible Logics

A logic consists of a language of possible assertions, a set of situations called models, and a way of determining whether a given assertion is true in a given model (situation).

> **Definition**: A *logic* is a triple of a set of *assertions*, a set of *models*, and a *meaning function* $\mathcal{V}$ such that, for any assertion Φ and model $\mathcal{M}$, we have that $\mathcal{V}(\Phi, \mathcal{M})$ (the meaning, or value, of Φ in $\mathcal{M}$) is either $\mathbf{T}$ or $\mathbf{F}$.

Table /reflogic-table lists five examples of logics. The table gives each logic a name, gives examples of assertions (constraints), describes the models (interpretations), and gives an example of applying the meaning function. The first logic corresponds to the CSP constraint transformation that would be used to solve a graph coloring problems. In a graph coloring problem one is given a graph and a number of colors. The problem is to assign a color to each node in the graph so that no arc of the graph connects two nodes of the same color. Each arc in the graph places a constraint on two nodes stating that those two nodes must have different colors. In table 1 such a constraint is a assertion of the form $n_1 \neq n_2$. A model, or interpretation, in the graph coloring logic is a mapping form nodes to colors, i.e., an assignment of colors to nodes. In a graph coloring problem we are given a set of assertions and must find a model that makes all those assertions true. This is a classical constraint satisfaction problem.

The second logic in table 1 is the logic that arises from the numerical constraint transformation. In this logic the assertions are numerical relationships between numerical variables. A model is an interpretation of the numerical variables as real numbers. The fourth example in the table is the logic that arises from the lifting transformation discussed in a later section of the notes. In this logic each assertion is an equation or disequation between expressions built out of constants and variables. A model is an interpretation of variables as expressions. The fourth logic in the table is the classical propositional calculus. In this logic each assertion is a Boolean formula constructed from

Logic Name	Assertions (Constraints)	Models (Interpretations)	Meaning Function
Graph Coloring Logic	$n_1 \neq n_2$	$\rho : Nodes \rightarrow Colors$	$\mathcal{V}(n_1 \neq n_2,\ \rho) = \mathbf{T}$ iff $\rho(n_1) \neq \rho(n_2)$
Numerical Constraint Logic	$x < 2y + 3z$	$\rho : Vars \rightarrow \mathcal{R}$	$\mathcal{V}(x < 2y + 3z, \rho) = \mathbf{T}$ iff $\rho(x) < 2\rho(y) + 3\rho(z)$
Expression Lifting Logic	$(\texttt{f}\ x)$ `=` `(f a)`	$\rho : Vars \rightarrow$ `Exps`	$\mathcal{V}((\texttt{f}\ x)\ \texttt{=}\ (\texttt{f a}),\ \rho) = \mathbf{T}$ iff $\rho(x) = \texttt{a}$
Propositional Calculus	$(p \wedge q) \vee w$	$\rho : Vars \rightarrow$ *Truth Values*	$\mathcal{V}((p \wedge q) \vee w,\ \rho) = \mathbf{T}$ iff either $\rho(p)$ and $\rho(q)$ or $\rho(w)$
First Order Logic	$\forall x(P(x) \rightarrow Q(x))$	A First Order Structure $\mathcal{M}$	$\mathcal{V}(\forall x(P(x) \rightarrow Q(x)),\ \mathcal{M}) = \mathbf{T}$ iff the set satisfying $\mathcal{M}(P)$ is a subset of the set satisfying $\mathcal{M}(Q)$

Table 1: Some Logics

Boolean variables using the standard Boolean connectives. A model is an interpretation of each Boolean variable as a truth value. The fifth logic in the table is the classical first order predicate calculus. This logic is discussed in detail in a later section. At this point we simply observe that a model of this logic provides an interpretation predicate symbols as predicates. The predicates that appear in the assertions of the logic should be viewed as variables. Each assertion can be viewed as a constraint on the interpretation of predicate variables.

We can now introduce some standard terminology that can be used in any logic.

> **Definition:** If $\mathcal{V}(\Phi, \mathcal{M})$ equals $\mathbf{T}$ then we say that the model $\mathcal{M}$ *satisfies* the assertion Φ, or that $\mathcal{M}$ *is a model of* Φ. If $\mathcal{M}$ is a model that satisfies every assertion in a set Σ of assertions, then we say that $\mathcal{M}$ satisfies Σ or that $\mathcal{M}$ is a model of Σ.

Definition: A assertion, or a set of assertions, is called *satisfiable* if there exists a model that satisfies it. A assertion is called *valid* if it is true in all models.

For any logic, it is possible to extend the set of assertions so that any Boolean combination of assertions is also a assertion (of course many logics already include all such Boolean combinations). The assertion $\neg\Phi$ is true just in case Φ is false. The assertion $\Phi \vee \Psi$ is true just in case either Φ is true or Ψ is true. Similar statements apply to other boolean operators such as $\wedge$ and $\rightarrow$. It is important to note that $\Phi \rightarrow \Psi$ is true in a model whenever Φ is false or Ψ is true. In other words $\Phi \rightarrow \Psi$ is equivalent to $\neg\Phi \vee \Psi$.

2 Entailment

Viewing assertions as constraints, one can observe that a set of constraints can imply a new constraint. For example, in the numerical logic $x < y$ and $y < z$ imply $x < z$. This notion of implication, or entailment, is captured in the following definition.

Definition: In any given logic, we say that a set of assertions Σ *entail* a assertion Φ, written $\Sigma \models \Phi$, if Φ is true in every model of Σ.

Entailment can be very difficult to determine. For example, consider the logic associated with the set of domain variables in a constraint satisfaction problem. Each constraint in a constraint satisfaction problem can be written as a Boolean formula $\Phi(x_1, \cdots, x_n)$ involving the domain variables $x_1, \cdots, x_n$. Now suppose we wish to determine if a given set of constraints Σ imply a particular assignment, i.e., we want to determine whether $\Sigma \models x = c$. This is equivalent to asking if there is some model of Σ in which $x = c$ is false. This is equivalent to asking if there exists a model of $\Sigma \cup \{x \neq c\}$. For an arbitrary constraint set Σ this is an NP-complete problem.

Observation: In any logic, $\Sigma \models \Phi$ if and only if $\Sigma \cup \{\neg\Phi\}$ is not satisfiable.

For many logics, including all of the logics discussed so far, the above observation implies that one can construct a procedure that determines whether or not $\Sigma \models \Phi$ by searching for a solution to the constraint set $\Sigma \cup \{\neg\Phi\}$. However, if inference is to be viewed primarily as a method of accelerating search, it is important to be able to construct fast (non-searching) inference procedures.

3 Inference Relations

In this section I will assume some arbitrary but fixed logic. An inference procedure (for the given logic) is a procedure that takes a set of assertions Σ and a given assertion Φ and attempts to determine whether $\Sigma \models \Phi$. We are particularly interested in inference procedures, such as constraint propagation, which terminate quickly. Because entailment can be difficult to determine, and because we want to construct inference procedures that always terminate quickly, it is important that the inference procedure not be required to always determine whether or not $\Sigma \models \Phi$.

An inference procedure defines an *inference relation*, written as $\vdash$. If the inference procedure applied to Σ and Φ determines that $\Sigma \models \Phi$ then we write $\Sigma \vdash \Phi$. If the procedure fails to determine that $\Sigma \models \Phi$ then we write $\Sigma \not\vdash \Phi$. Note that it is possible that $\Sigma \models \Phi$ and yet $\Sigma \not\vdash \Phi$.

Definition: An inference relation $\vdash$ is called *sound* if, for any assertion set Σ and any assertion Φ, if $\Sigma \vdash \Phi$ then $\Sigma \models \Phi$.

Definition: An inference relation $\vdash$ is called *complete* if, for any assertion set Σ and any assertion Φ, if $\Sigma \models \Phi$ then $\Sigma \vdash \Phi$.

As noted earlier, the entailment relation $\models$ is usually very difficult to compute. Depending on the logic involved, determining whether $\Sigma \models \Phi$ can be

NP-hard (or worse). However, there are many fast and useful incomplete inference procedures.

4 Inference Rules

Inference relations are often specified by inference rules rather than inference procedures. For example, suppose that the assertions of some logic include implications of the form $\Phi \rightarrow \Psi$ where Φ and Ψ are also assertions of the logic. The inference rule of *modus ponens* can be stated as follows.

$$\frac{\begin{array}{l}\Phi \\ \Phi \rightarrow \Psi\end{array}}{\Psi}$$

Intuitively, this rule says that if one can infer Φ, and one can infer $\Phi \rightarrow \Psi$, then one can infer Ψ. This inference rule is intended to be used for any assertions Φ and Ψ. A particular version of the rule, with particular formulas inserted for Φ and Ψ will be called an *instance* of the rule — the above rule of modus ponens is meant to abbreviate all of the infinitely many instances of the rule.

In general, an inference rule has the form

$$\frac{\begin{array}{l}A_1 \\ \vdots \\ A_n\end{array}}{C}$$

where each A_i is called an *antecedent* of the rule and C is called the *conclusion* of the rule. A set of inference rules defines a notion of "proof" or "derivation". Intuitively, a derivation is a sequence of "lines" where each line contains a assertion that can be derived from previous lines by the given rules of inference. Derivations of this type form the foundation of most high school geometry courses.

> **Definition:** If R is a set of inference rules, and Σ is a set of formulas, then a *derivation* of a formula Φ from the premises Σ is a sequence of assertions $\Psi_1, \Psi_2, \cdots, \Psi_n$ where Ψ_n is the derived assertion Φ and for each Ψ_i in the derivation, either Ψ_i is a member of Σ or there exists (an instance of) an inference rule in R whose conclusion is Ψ_i and all of whose antecedents occur prior to Ψ_i in the derivation.

> **Definition:** If there exists a derivation of Φ from Σ under rule set R then we write $\Sigma \vdash_R \Phi$. We call $\vdash_R$ the inference relation generated by the rule set R.

Defining an inference relation by a set of inference rules is very different from defining an inference relation by a procedure. A rule set R does not immediately provide any efficient way of determining whether or not $\Sigma \vdash_R \Phi$. For many rule sets, determining whether or not $\Sigma \vdash_R \Phi$ requires searching the space of derivations. Searching the space of derivations can be very expensive. If we are interested in using inference to accelerate difficult computation, then we are probably not interested in inference relations that require searching a space of derivations. Fortunately, for certain sets of inference rules, the generated inference relation is quickly decidable.

5 Sequent Systems and Efficient Boolean Inference

It has been noted that any logic can be extended so that any Boolean combination of assertions is a assertion. It is possible to construct a general purpose efficient inference procedure for Boolean combinations called *Boolean constraint propagation* or BCP. The truth value of a Boolean combination is, of course, related to the truth value of the combined expressions. Consider the combination $P \vee Q$. The truth of $P \vee Q$ is clearly related to the truth of P and the truth of Q. If we infer that P is true, then we can deduce that $P \vee Q$ true. Similarly, if we infer that $P \vee Q$ is true, and infer that P is false, then we can infer that Q is true. In Boolean constraint propagation, truth labels flow "locally" between the assertion $P \vee Q$ and the assertions P and Q. This local flow of truth labels between a Boolean formula and its parts can be described by inference rules. For example, we have the following rule.

$$\frac{\begin{array}{l} P \vee Q \\ \neg P \end{array}}{Q}$$

Boolean constraint propagation can be defined by a set of inference rules where each rule relates a Boolean formula to its subformulas.

Definition: BCP is defined to be the following set of inference rules.

- The two rules

  $$\frac{\neg\neg\Psi}{\Psi}$$

  and

  $$\frac{\Psi}{\neg\neg\Psi}$$

- The "contradiction rules"

  $$\frac{\begin{array}{l} \Phi \\ \neg\Phi \end{array}}{\mathbf{F}}$$

$$\frac{\mathbf{F}}{\Psi}$$

- All sound rules that relate a Boolean combination $\Phi \circ \Psi$, where $\circ$ is a Boolean operator, to the subformulas Φ and Ψ, i.e., all sound rules of the form

  $$\frac{\begin{array}{l}\pm(\Phi \circ \Psi) \\ \pm\Phi\end{array}}{\pm\Psi}$$

  $$\frac{\begin{array}{l}\pm(\Phi \circ \Psi) \\ \pm\Psi\end{array}}{\pm\Phi}$$

  $$\frac{\begin{array}{l}\pm\Psi \\ \pm\Phi\end{array}}{\pm(\Phi \circ \Psi)}$$

  where $\pm\Psi$ represents either Ψ or its negation.

For example, the inference rule of modus ponens,

$$\frac{\begin{array}{l}\Phi \\ \Phi \rightarrow \Psi\end{array}}{\Psi}$$

is included in the rule set BCP. Of course, BCP is restricted to only include sound rules. BCP contains enough inference rules for a large variety of simple inferences. For example, given P, $P \rightarrow R$, and $(P \wedge R) \rightarrow W$, the BCP inference rules can be used to derive W. To see this, note that the modus ponens inference rule can derive R. From P and R, one of the BCP inference rules for the operator $\wedge$ can be used to derive $P \wedge R$. Modus ponens can then be used again to derive W. Although the rule set BCP provides for a wide variety of useful inferences, the inference relation defined by BCP is linear time decidable.

> **BCP Tractability Theorem:** The inference relation $\vdash_{\mathrm{BCP}}$ generated by the rule set BCP is linear time decidable, i.e., for any set of assertions Σ, and any assertion Φ, one can determine in linear time in the size of Σ and Φ whether or not $\Sigma \vdash_{\mathrm{BCP}} \Phi$.

The proof of the above theorem is given in a later section.

Classical propositional logic is the logic whose assertions consist of arbitrary Boolean combinations of assertion symbols and whose models are assignments of truth values to the assertion symbols. In propositional logic, the entailment relation $\models$ is CoNP-complete — assuming $P \neq NP$ there can not be any polynomial time algorithm for determining whether or not $\Sigma \models \Phi$ in propositional logic. The statement that $\vdash_{\mathrm{BCP}}$ is linear time decidable has the immediate corollary (assuming $P \neq NP$) that $\vdash_{\mathrm{BCP}}$ is not complete for classical propositional logic. A simple example of this lack of completeness is the fact that from $P \rightarrow Q$, and $\neg P \rightarrow Q$, the inference rules in BCP can not be used to derive Q. This shows that $\vdash_{\mathrm{BCP}}$ fails to incorporate the following *case analysis principle.*

$$\frac{\begin{array}{l}\Sigma \cup \{\Psi\} \vdash \Phi \\ \Sigma \cup \{\neg\Psi\} \vdash \Phi\end{array}}{\Sigma \vdash \Phi}$$

An inference relation of the form $\Sigma \vdash \Phi$ is called a *sequent.* The case analysis rule is a sequent rule — it allows for the derivation of one sequent from another sequent. In general, any set of ordinary inference rules, plus any set of sequent rules, generates a well defined inference relation.

6 Sequent Proof Systems

A *sequent derivation* is like an ordinary derivation except that each step is a sequent rather than a assertion.

> **Definition:** A *sequent proof system* is a pair $<R, S>$ where R is a set of ordinary inference rules and S is a set of sequent rules.

> **Definition:** A *sequent derivation* in the sequent system $<R, S>$ is a series of sequents of the form
>
> $$\Sigma_1 \vdash \Phi_1$$
> $$\Sigma_2 \vdash \Phi_2$$
> $$\vdots$$
> $$\Sigma_n \vdash \Phi_n$$
>
> where, for each sequent $\Sigma_i \vdash \Phi_i$, either $\Sigma_i \ \vdash_R \ \Phi_i$, where $\vdash_R$ is the inference relation generated by R, or the sequent $\Sigma_i \vdash \Phi_i$ can be derived from previous sequents using one of the sequent rules in S.

> **Definition:** The inference relation $\vdash_{<R,S>}$ generated by the sequent system $<R, S>$ is defined so that $\Sigma \ \vdash_{<R,S>} \ \Phi$ if and only if there exists a sequent derivation based on R and S whose last sequent is $\Sigma \vdash \Phi$.

Intuitively, a derivation is a syntactic proof. In any proof system it should be possible to quickly check if a proposed proof is in fact a proof — is each step really derivable from previous steps? The notion of derivation (proof) for ordinary inference rules has the property that derivations can be quickly checked. For any sequence of formulas one can design a polynomial time procedure for determining whether each assertion is either a premise or derivable by a single rule from previous assertions. However, the above definition of a sequent derivation does not guarantee that proposed proofs are quickly checkable. The above definition allows for the (unjustified) introduction of a sequent $\Sigma \vdash \Phi$ provided that $\Sigma \vdash_R \Phi$. If R is a powerful rule set, then determining whether or not $\Sigma \vdash_R \Phi$ may require search, i.e., may not be solvable in polynomial time.

Definition: The sequent system $<R, S>$ will be called *acceptable* if the relation $\vdash_R$ is polynomial time decidable.

Lemma: If $<R, S>$ is acceptable then, given any series of sequents, one can determine in polynomial time whether that series is a valid derivation based on R and S.

Note that, because sequents of the form $\Sigma \vdash_R \Phi$ can be introduced without justification, sequent proofs can be shorter than proofs in which each step must be derivable by a single application of a single inference rule. In any acceptable sequent system, however, proposed derivations can be quickly verified.

7 Boolean Completeness

In this section we prove that the sequent system consisting of the ordinary rule set BCP plus the case analysis sequent rule is complete for Boolean inference.

Definition: We define B to be the sequent system composed of the ordinary rule set BCP and the single sequent rule of case analysis.

Theorem The inference relation $\vdash_B$ is complete for propositional logic.

Proof Recall that propositional logic is the logic whose models are truth assignments to assertion symbols and whose assertions are Boolean combinations of the assertion symbols. We must show that if $\Sigma \models \Phi$ in propositional logic, i.e., if every truth assignment that satisfies Σ also satisfies Φ, then $\Sigma \;\vdash_B\; \Phi$. A assertion symbol P will be called *BCP-assigned* by Σ if $\Sigma \;\vdash_{\mathrm{BCP}}\; P$ or $\Sigma \;\vdash_{\mathrm{BCP}}\; \neg P$. A assertion symbol P will be called a *residual* of Σ and Φ if P appears in some expression in Σ or Φ and P is not BCP-assigned by Σ. The proof of completeness is by induction on the number of residuals.

First suppose that there are no residuals, i.e., every assertion symbol that appears in Σ or Φ is BCP-assigned by Σ. We will show that in this case, if $\Sigma \models \Phi$ then $\Sigma \;\vdash_{\mathrm{BCP}}\; \Phi$. There are two cases. First suppose that the truth values assigned to the assertion symbols do not form a model of Σ. The inference rules of BCP allow the value of any formula to be derived from the truth value of the assertion symbols that appear in that formula. If the truth assignment is not a model of Σ then the inference rules of BCP allow one to derive the negation of some element of Σ. So we have $\Sigma \;\vdash_{\mathrm{BCP}}\; \mathbf{F}$, and consequently $\Sigma \;\vdash_{\mathrm{BCP}}\; \Phi$. Now suppose that the truth values assigned to the assertions do form a model of Σ. By assumption $\Sigma \models \Phi$, so the truth assignment must also be a model of Φ. The inference rules of BCP allow the value of Φ to be derived from the values of the assertion symbols that appear in Φ. So we again have $\Sigma \;\vdash_{\mathrm{BCP}}\; \Phi$.

Now suppose that $\vdash_B$ is complete for all inference problems with fewer residuals than Σ and Φ. Let P be a residual of Σ and Φ. If $\Sigma \models \Phi$ then we must have that $\Sigma \cup \{P\} \models \Phi$. But $\Sigma \cup \{P\}$ and Φ have fewer residuals than Σ and Φ, so by the induction hypothesis we must have $\Sigma \cup \{P\} \;\vdash_B\; \Phi$. Similarly, we must have $\Sigma \cup \{\neg P\} \;\vdash_B\; \Phi$. By the case analysis sequent rule we now have $\Sigma \;\vdash_B\; \Phi$.

8 Interning Expressions

The procedure for determining whether or not $\Sigma \vdash_{\mathrm{BCP}} \Phi$ is based on constraint propagation. Intuitively, constraint propagation operates on graph structure. By "graph structure" we do not necessarily mean a graph in the strict mathematical sense. The phrase graph structure is used here in an intuitive, informal, way to mean a collection of "nodes" and a collection of "connections". The connection can involve more than two nodes and both nodes and connections can have internal structure. In constraint satisfaction problems the nodes are domain variables and the connections are the constraints. A single constraint can involve more than one node.

In most logics, assertions are expressions. In general, it is possible to represent expressions by nodes in a graph structure — any algorithm that manipulates expressions can be converted to an algorithm that manipulates nodes. Under most schemes for representing expressions, such as representations based on list structure, the same expression can be represented many different times where each representation is located in different part of memory. In Lisp, this multiple representation corresponds to the fact that lists can be `EQUAL` without being `EQ`. Such representations do not support side effects — it is not possible to mark an expression in such a way that all representations of the expression "see" the mark. To convert expressions to nodes one can *intern* expressions. When expressions are interned each expression is represented by a unique data structure located in a unique place in memory. If s is an expression, perhaps represented by list structure, then we can let n_s be the interned data structure representing s.

We can assume that the interned data structure n_s that represents the expression s contains pointers to the interned data structures that represent the immediate subexpressions of s. For example, if s is the expression $(w_1, \cdots, w_n)$ where $w_1, \cdots, w_n$ are the immediate subexpressions of s, then the data structure n_s points to the data structures $n_{w_1}, \cdots, n_{w_n}$. If we think of the data structures of the form n_s as nodes in a graph structure, then the pointers between the node for an expression and the nodes for its subexpressions can be viewed as connections in this graph structure.

Let Υ be a set of expressions. We say that an expression s *appears in* Υ if s

is either a member of Υ or s is a subexpression of a member of Υ. For any set of expressions Υ one can construct a graph structure whose nodes are the interned data structures of the form n_s and whose connections are pointers that relate expressions to subexpressions. If w is an immediate subexpression of s, where s is an expression that appears in Υ, then n_s contains a pointer to n_w and n_w contains a pointer to n_s. In this case we call s a *parent expression* of w.

The mapping from s to the interned data structure n_s is analogous to the intern function I used to recognize previously visited nodes in a graph search problem. We use the notion n_s here rather than $I(s)$ because, unlike the graph search algorithms, the algorithms given below never compute the intern mapping — the algorithms take "predigested" graph structure as input.

9 Proof of the BCP Tractability Theorem

We now prove that $\vdash_{\mathrm{BCP}}$ is quickly decidable. We assume that we are given the graph structure representing all expressions that appear in Σ and Φ.[1] The graph structure representation can be viewed as a constraint graph in which, for each Boolean combination $\Psi_1 \circ \Psi_2$ that appears in Σ or Φ, there is a constraint that relates the truth of the node $n_{\Psi_1 \circ \Psi_2}$ to the truth of n_{Ψ_1} and the truth of n_{Ψ_2}. To determine if $\Sigma \vdash_{\mathrm{BCP}} \Phi$ one simply labels the assertions in Σ as true and then one runs constraint propagation (value propagation) on this graph structure. Since each constraint in the constraint network involves at most three Boolean variables, constraint propagation can be run to completion in linear time in the size of the graph structure. If the constraint propagation process either derives a contradiction, or labels Φ true, then $\Sigma \vdash_{\mathrm{BCP}} \Phi$.

To show that this is a decision procedure for $\vdash_{\mathrm{BCP}}$, we must must now show that if the constraint propagation process fails to derive a contradiction, and fails to label Φ true, then $\Sigma \not\vdash_{\mathrm{BCP}} \Phi$. The proof is by contradiction.

[1] Under the assumption that hash table operations take constant time, the graph structure representation of Σ and Φ can be computed from the list structure representation in linear time in the total size of Σ and Φ.

Suppose that $\Sigma \quad \vdash_{\mathrm{BCP}} \Phi$ but that the above constraint propagation fails to either derive a contradiction or label Φ true. Since $\Sigma \quad \vdash_{\mathrm{BCP}} \Phi$ there must exist some finite derivation of Φ from Σ using the inference rules in BCP. This implies that there is some (possibly very large) graph structure of nodes representing expressions such that constraint propagation on that structure, starting with true labels for the elements of Σ, either generates a contradiction or labels Φ true. We are given that constraint propagation on the graph structure representing Σ and Φ does not provide a derivation of Φ. Now consider two formulas Ψ_1 and Ψ_2 are represented by nodes in this small network. Consider the expanded network that includes the one new node $\Psi_1 \circ \Psi_2$ where $\circ$ is some Boolean operator. It is possible to show that constraint propagation on this expanded network will not generate any new labels on nodes of the original network. This expansion process can be continued to create arbitrarily large networks. This implies that for any finite network, constraint propagation on that network will not provide a derivation of Φ. But this contradictions the statement that there must exist some large network on which constraint propagation will provide a derivation of Φ.

10 Fast Equational Reasoning

A logic consists of assertions, a set of models, and a way of assigning a value to each assertion in each model. Many logics involve expressions other than assertions. For example, in a logic whose models contain numbers, a assertion might be of the form $s < t$ where s and t are expressions that denote numbers. Expressions that denote objects rather than truth values are called *terms*.

> **Definition:** A *logic with terms* consists of a set of models, a set of assertions, a set of terms, and a semantic evaluation function $\mathcal{V}$ such that for any model $\mathcal{M}$, any formula Φ and any term s, we have that $\mathcal{V}(\Phi, \mathcal{M})$ is either $\mathbf{T}$ or $\mathbf{F}$, and $\mathcal{V}(s, \mathcal{M})$ is some object called the *semantic value* of the term s in the model $\mathcal{M}$.

It is rather easy to define a variety of examples of logics with terms. For example, consider a set of variables $x_1 \cdots x_n$ that range over integers. We can define a logic such that a model is an assignment of an integer to each variable. We can define the set of terms to be the expressions that can be constructed from the binary operations $+$ and $\times$ and the given variables. The assertions will then be expressions of the form $s < t$ where s and t are terms. By treating the operators $+$ and $\times$ as addition and multiplication respectively, each term can be assigned a semantic value in each model. For example, $\mathcal{V}(x, \mathcal{M})$ is defined to be the number that the model $\mathcal{M}$ assigns to the variable x and $\mathcal{V}(s+t, \mathcal{M})$ is defined to be $\mathcal{V}(s, \mathcal{M}) + V(t, \mathcal{M})$. The assertion $s < t$ is true in a model if and only if the semantic value of s in that model is less than the semantic value of t.

We have previously observed that any logic can be extended so that any Boolean combination of assertions is also a assertion. Any logic with terms can be extended to include assertions of the form $s = t$ where s and t are either terms or assertions. An equation $s = t$ is true in a model $\mathcal{M}$ if and only if $\mathcal{V}(s, \mathcal{M})$ equals $\mathcal{V}(t, \mathcal{M})$.[2] This way of assigning truth values to equations immediately implies that the following set of inference rules is sound.

> **Definition:** We define E to be the set of inference rules given below and $\vdash_E$ to be the inference relation generated by these rules.

Reflexivity: $s = s$

Symmetry:

$$\frac{s = t}{t = s}$$

Transitivity:

$$\frac{\begin{array}{l} s = t \\ t = w \end{array}}{s = w}$$

[2]For assertions Φ and Ψ, the equation $\Phi = \Psi$ is true if Φ and Ψ have the same truth value. Equality on assertions is considered to be another Boolean operator and Boolean rules for equality are included in the BCP rule set.

Just as the rule set BCP is sound in any logic, the rule set E is sound in any logic with terms. Furthermore, the inference relation $\vdash_E$, like the inference relation $\vdash_{\mathrm{BCP}}$, can be computed quickly.

The inference relation $\vdash_E$ can be computed with a union-find procedure. For a fixed premise set Σ, the set of equations $s = w$ such that $\Sigma \ \vdash_E \ s = w$ determines an equivalence relation on terms. An equivalence relation on expressions can be represented by "coloring" the node representations of the expressions. The following procedure can be used to determine whether or no $\Sigma \ \vdash_E \ s = w$.

Procedure for Determining Whether $\Sigma \ \vdash_E \ s = w$.

1. Construct the graph structure representation of Σ and the equation $s = w$.
2. Assign colors to nodes in the graph structure such that each node has a separate color.
3. For each equation $t_1 = t_2$ that is a member of Σ, use the union procedure given below to merge the equivalence classes of t_1 and t_2.
4. If the node n_s has the same color as the node n_w then return "yes", else return "no".

Union Procedure on t_1 and t_2

> If node n_{t_1} has the same color as n_{t_2} then do nothing. Otherwise, if the equivalence class of n_{t_1} (the nodes with the same color as n_{t_1}) has fewer members than the equivalence class of n_{t_2} then recolor the equivalence class of n_{t_1} to the same color as n_{t_2}, otherwise recolor the equivalence class of n_{t_2} to the color of n_{t_1}.

Note that the procedure for merging equivalence classes always recolors the smaller of the two equivalence classes. This implies that every time a node is recolored the size of that node's equivalence class at least doubles — if a node is recolored k times then its equivalence class must have at least 2^k members. But this implies that a node can be recolored at most $\log_2 n$ times where n is the number of nodes in the graph. Thus the total number of recoloring operations performed by the above procedure can be no greater than $n \log_2 n$.

Union-find procedures can be found in any modern text on algorithms. There exists a procedure for the union-find problem that runs in order $nA^{-1}(n)$ where A^{-1} is the inverse Ackermann's function. The inverse Ackermann function grows extremely slowly. In fact, $A^{-1}(2^{65536}) = 5$. This implies that $A^{-1}(n) \leq 5$ for all practical values of n. The more efficient union-find procedure, although quite simple, is not presented here. Unlike the more efficient algorithm, the above $n \log n$ algorithm has a natural generalization that incorporates the additional inference rule of the substitution of equals for equals. This additional rule is discussed below.

11 Compositional Terms

In most logics with terms, the terms are expressions with internal structure. For example, in the numerical logic described above, terms were numerical expressions constructed from numerical variables, such as the expressions $x + (y \times z)$. A term often consists of an operator expression, such as $+$ or $\times$ and argument expressions that are themselves terms. The semantic value of a term can often be constructed by first taking the semantic value of the operator expression — the semantic value should be a function — and applying that value to the semantic values of the arguments.

> **Definition:** In a logic with terms, a term of the form $<t_1, \cdots t_n>$ is called *compositional* if there exists a "composition function" C such that for any model $\mathcal{M}$ we have
>
> $$\mathcal{V}(<t_1, \cdots, t_n>, \ \mathcal{M}) = C(\mathcal{V}(t_1, \ \mathcal{M}), \cdots, \mathcal{V}(t_n, \ \mathcal{M})).$$

Intuitively, a term is compositional if the meaning of the term is determined by the meaning of the subterms. For example, in the number logic all terms other than the number variables are compositional. The meaning of $<s + w>$ can be derived from the meaning of s, the meaning of $+$, and the meaning of w. The following "substitution" inference rule is sound for compositional terms.

$$\frac{\begin{array}{l} s_1 = w_1 \\ \vdots \\ s_n = w_n \end{array}}{<s_1, \cdots, s_n> = <w_1, \cdots, w_n>}$$

Many logics have terms that are not compositional. For example, the substitution rule can not be applied to Lisp expressions of the form `(LAMBDA (X) (F X X))` — one can not soundly derive that `(LAMBDA (X) (F X X))` equals `(LAMBDA (X) (G X))` from the assumptions that `X` equals 3 and `(F 3 3)` equals `(G 3)`. `LAMBDA` is a quantifier. In general, quantified expressions are not compositional. Other examples of quantified expressions are first order quantified formulas of the form $\forall x \Phi(x)$ or integral expressions of the form $\int_{x=0}^{\infty} f(x)$. Terms that do not involve quantifiers, however, are almost always compositional. We will assume that in any logic with terms some subset of the terms are designated as compositional.

The introduction of another inference rule leads to another inference relation.

> **Definition:** We define C (for *Congruence*) to be the set of inference rules consisting of the three equality rules in E plus the above substitution rule restricted to compositional terms.

> **Congruence Tractability Theorem:** Assuming that hash table operations take constant time, one can determine whether or not $\Sigma \ \vdash_C \ s = w$ in order $n \log n$ time where n is the written size of Σ, s and w.

To prove the congruence tractability procedure we must construct a procedure that determines whether $\Sigma \ \vdash_C \ s = w$ in order $n \log n$ time. For a fixed premise set Σ, the set of equations $s = w$ such that $\Sigma \ \vdash_C \ s = w$ determines an equivalence relation on expressions. This equivalence relation also respects the above substitution inference rule. An equivalence relation that respects the substitution rule is called a *congruence relation.* Now consider the graph structure representation of expressions. An equivalence relation on expressions can be represented by a coloring of the nodes in the graph structure representation. A coloring of the nodes will be called *congruence closed* if it is closed under the substitution inference rule, i.e., if $n_{s_1}, \cdots, n_{s_k}$ have the same colors as $n_{w_1}, \cdots, n_{w_k}$ respectively, then $n_{(s_1,\cdots,s_k)}$ has the same color as $n_{(w_1,\cdots w_k)}$. The process of merging equivalence classes, as justified by the substitution rule, until no further mergers are possible is called *congruence closure.*

We now show that, without loss of generality, we can assume that every compositional expression has at most three immediate subexpressions. Consider the problem of determining whether $\Sigma \ \vdash_E \ s = w$. Let P (for Pair) be a symbol that does not appear in Σ, s, or w. Now let Σ', s' and w' be the result of replacing each expression of the form $<s_1, s_2, \cdots s_{n-1}, s_n>$ by $<P, s_1, <P, s_2, \cdots, <P, s_{n-1}, s_n>>>$. We have that $\Sigma \ \vdash_C \ s = w$ if and only if $\Sigma' \ \vdash_C \ s' = w'$. Furthermore, every compositional expression in Σ', s', and w' has at most three immediate subexpressions and the written length of Σ', s' and w' is proportional to the written length of Σ, s and w.

To efficiently compute a congruence closed node coloring a new data structure is needed. In particular we introduce a hash table H which maps tuples of colors to nodes in the graph; for a tuple of colors $<c_1\, c_2, \cdots, c_k>$ for which the hash table contains an entry, the hash table value $H(<c_1\, c_2, \cdots, c_k>)$ is a node n_e. The significance of the table H involves the fact that a color represent an equivalence class of expressions. Therefore a tuple $<c_1\, c_2, \cdots, c_k>$ also represents a set of expressions: the set of expressions generated by choosing particular members of the equivalences classes represented by c_1, c_2, $\cdots$, c_k. The table H will always have the property that if H contains the entry $H(<c_1\, c_2, \cdots, c_k>) = n_e$ then all expression in the class represented by $<c_1\, c_2, \cdots, c_k>$ are equivalent to e.

Now consider two nodes $n_{<s_1\, s_2, \cdots, s_k>}$ and $n_{<w_1\, w_2, \cdots, w_k>}$ and consider a coloring of nodes such that the corresponding subexpressions of these two nodes are equal, i.e. the color of n_{s_i} equals the color of n_{w_i}. Let c_1, c_2, $\cdots$, c_n be the color of n_{s_1}, n_{s_2}, $\cdots$, n_{s_k} respectively. If the coloring is not congruence closed, the nodes $n_{<s_1\, s_2, \cdots, s_k>}$ and $n_{<w_1\, w_2, \cdots, w_k>}$ may have different colors even though the corresponding subnodes have the same color. Furthermore the value of $H(<c_1\, c_2, \cdots, c_k>)$ could be either node $n_{<s_1\, s_2, \cdots, s_k>}$ or node $n_{<w_1\, w_2, \cdots, w_k>}$.

Now consider a particular coloring and a particular state of the hash table H. A *congruence violator* is a node of the form $n_{<t_1\, t_2, \cdots, t_k>}$ where $<t_1\, t_2, \cdots, t_k>$ is a compositional expression where node n_{t_i} has color c_i such that either the hash table entry $H(<c_1\, c_2, \cdots, c_k>)$ is undefined or the node $H(<c_1\, c_2, \cdots, c_k>)$ has a different color from the node $n_{<t_1\, t_2, \cdots, t_k>}$. If $n_{<t_1\, t_2, \cdots, t_k>}$ is a congruence violator because the node $H(<c_1\, c_2, \cdots, c_k>)$ has a different color from $n_{<t_1\, t_2, \cdots, t_k>}$ then one is justified in unioning the equivalence classes of $n_{<t_1\, t_2, \cdots, t_k>}$ and $H(<c_1\, c_2, \cdots, c_k>)$. If there are no congruence violators then the coloring must be congruence closed.

The congruence closure procedure has one additional data structure: a list V of all the nodes which might be congruence violators; the procedure maintains the invariant that all congruence violators are members of the list V.

Procedure for Determining Whether $\Sigma \vdash_C s = w$:

1. Construct the graph structure representation of Σ and $s = w$.
2. Compute the union-find color labeling used to determine whether $\Sigma \vdash_E s = w$.
3. Initialize the list V to include all nodes that represent compositional terms.
4. Until V is empty do the following.
   (a) Let $n_{<t_1\, t_2, \cdots, t_k>}$ be an element of V, remove $n_{<t_1\, t_2, \cdots, t_k>}$ from V, and let c_1, c_2, $\cdots$, c_k be the colors of n_{t_1}, n_{t_2}, $\cdots$, n_{t_k} respectively.
   (b) If $H(<c_1\, c_2, \cdots, c_k>)$ is undefined set $H(<c_1\, c_2, \cdots, c_k>)$ equal to $n_{<t_1\, t_2, \cdots, t_k>}$.

(c) If $H(<c_1\, c_2, \cdots, c_k>)$ is a different color than $n_{<t_1\, t_2, \cdots, t_k>}$ then union the nodes $H(<c_1\, c_2, \cdots, c_k>)$ and $n_{<t_1\, t_2, \cdots, t_k>}$ (merge their equivalences classes). For every node n_v whose color is changed by this union operation, place all compositional parents of n_v on the list V. (A compositional parent of n_v is a node of the form $n_{<u_1\, , \cdots,\, v\, , \cdots,\, u_k>}$ where $<u_1\, , \cdots,\, v\, , \cdots,\, u_k>$ is a compositional expression.)

5. If the node n_s has the same color as the node n_w then return "yes", else return "no".

The algorithm maintains the invariant that at the beginning of step 3) all congruence violators are members of the list V. This is clearly true for the initial value of V. Now assume that at the start of some execution of step 3) all congruence violators are on V. I will show that all congruence violators are on V at the start of the next execution of step 3).

First consider the node $n_{<e_1\, e_2, \cdots, e_k>}$ which is removed from V at step 3). This node may be a congruence violator. If $n_{<e_1\, e_2, \cdots, e_k>}$ is not a congruence violator then steps 4) and 5) do nothing and every congruence violator is still on V at the start of the next execution of step 3). If $n_{<e_1\, e_2, \cdots, e_k>}$ is a congruence violator because $H(<c_1\, c_2, \cdots, c_k>)$ is undefined then step 4) will set $H(<c_1\, c_2, \cdots, c_k>)$ to $n_{<e_1\, e_2, \cdots, e_k>}$ and thus $H(<c_1\, c_2, \cdots, c_k>)$ will be a node with the same color as $n_{<e_1\, e_2, \cdots, e_k>}$ so $n_{<e_1\, e_2, \cdots, e_k>}$ will no longer be a congruence violator. If $n_{<e_1\, e_2, \cdots, e_k>}$ is a congruence violator because $H(<c_1\, c_2, \cdots, c_k>)$ is a node with a different color than $n_{<e_1\, e_2, \cdots, e_k>}$ then the union operation will ensure that $H(<c_1\, c_2, \cdots, c_k>)$ has the same color as $n_{<e_1\, e_2, \cdots, e_k>}$. If the union operation does not change the color of any of the nodes n_{e_i} then $n_{<e_1\, e_2, \cdots, e_k>}$ will not be a congruence violator at the start of next execution of step 3). If the union operation does change the color of one of the nodes n_{e_i} then the node $n_{<e_1\, e_2, \cdots, e_k>}$ will placed back on V before the start of next execution of step 3).

Now consider an arbitrary node $n_{<e_1\, e_2, \cdots, e_k>}$ which is not on V at the start of step 3) and thus is not a congruence violator at the start of step 3). We must show that if $n_{<e_1\, e_2, \cdots, e_k>}$ is a congruence violator at the start of the next execution of step 3) then $n_{<e_1\, e_2, \cdots, e_k>}$ is on V at that time. Note that once $H(<c_1\,c_2, \cdots, c_k>)$ becomes defined it is never changed and once two nodes have the same color they will always have the same color. Thus if $n_{<e_1\,e_2, \cdots, e_k>}$ is not a congruence violator it must continue not being a congruence violator unless one of the nodes n_{e_i} changes color. But if one of the nodes n_{e_i} changes color then $n_{<e_1\,e_2, \cdots, e_k>}$ will be placed on V during step 4).

Note that a given node can change colors at most $\log_2 n$ times and thus the total number of times that $n_{<e_1\,e_2, \cdots, e_k>}$ can be placed on the list V is $1 + k \log_2 n$ where n is the number of nodes. Summing over all nodes, the total number of times that nodes can be placed on the list V is $n + e \log_2 n$ where e is the number of edges in the graph structure representation (an edge is a pair of a node n_s and a node n_w where w is an immediate subexpression of s). We argued above that we could assume without loss of generality that every expression has at most three immediate subexpressions. This implies that the hash table operation can be performed in constant expected time and that the total running time of the procedure is order $n \log n$ where n is the number of nodes in the graph.

Let $\vdash$ be the inference relation defined by the above procedure, i.e., we write $\Sigma \vdash s = w$ if the procedure terminates with "yes" when applied to Σ and $s = w$. We must prove that the inference relation $\vdash$ implemented by the procedure is the same as the inference relation $\vdash_C$ defined by the rule set C. Since every class merger performed by the procedure is justified by the inference rules, if $\Sigma \vdash s = w$ then $\Sigma \vdash_C s = w$. The converse is more difficult to prove.

Suppose that $\Sigma \not\vdash s = w$. We must prove that that $\Sigma \not\vdash_C s = w$. This can be done by constructing an infinite set of equations such that Γ that includes all the equations in Σ, is closed under all the inference rules of C, and that does not contain the equation $s = w$. To define this equation set we first define a "normalization function" that maps arbitrary expressions to normal forms — two expressions will be provably equivalent if and only if they have the same normal form. For any expression t, the normal form of t, denoted $G(t)$ is defined below. In this definition, an expression is called *local* if it appears in Σ, s or w, i.e., if it represented by some node in the graph structure representation of Σ, s and w. For any term t, if t is provably equivalent to some local expression e then $G(t)$ is the color that the congruence closure algorithm assigns to n_e. If t is not equivalent to any local expression, then $G(t)$ is an expression that may contain colors in some places where symbols would normally appear. The normalization procedure is based on the hash table constructed by the congruence closure procedure. The normalization function G is defined as follows.

- If t is not a compositional expression (e.g., t is a constant symbol) then if t is local then $G(t)$ is the color of n_t, otherwise $G(t)$ equals t.
- To compute $G(<e_1\, e_2\,, \cdots,\, e_k>)$ where $<e_1, \cdots, e_k>$ is compositional, first compute the tuple $<G(e_1)\, G(e_2)\,, \cdots,\, G(e_k)>$. If this tuple is a tuple of colors, and if $H(<G(e_1), \cdots, G(e_k)>)$ has been defined, then $G(<e_1\, e_2\,, \cdots,\, e_k>)$ is the color of $H(<G(e_1), \cdots, G(e_k)>)$; otherwise $G(<e_1\, e_2\,, \cdots,\, e_k>)$ is the expression $<G(e_1)\, G(e_2)\,, \cdots,\, G(e_k)>$.

Let Γ be the set of equations $u = t$ such that $G(u)$ equals $G(t)$. It is possible to show that Γ contains all the equations in Σ, that Γ is closed under all of the inference rules of C, and that if $\Sigma \not\vdash s = w$ then Γ does not contain $s = w$. This shows that if $\Sigma \not\vdash s = w$, i.e., if the congruence closure procedure terminates with "no", then $\Sigma \not\vdash_C s = w$.

12 Grammar Representations of Expressions

Dynamic programming is one of the fundamental methods of accelerating difficult computations. Dynamic programming might be intuitively defined as the principle "avoid repeatedly solving the same subproblem". For example, it is possible to accelerate graph search by "marking" those nodes in the search graph that have been previously visited. The ability to mark previously visited nodes requires that the nodes be interned — different list structure representations of the same node must be mapped to a unique data structure at a unique location in memory.

Quotient acceleration is a related way of accelerating difficult computations. Quotient acceleration might be intuitively defined as "avoid repeatedly solving equivalent subproblems". For example, it is possible to accelerate graph search by defining an equivalence relation on nodes (satisfying a certain homomorphism condition). Once the equivalence relation is defined, one can search a graph of equivalence classes rather than the original graph of nodes. In searching the space of equivalence classes one avoids visiting a node if *an equivalent* node has already been visited. One can recognize when an equivalent node has been visited before by interning nodes relative to the equivalence relation. The interning process should map nodes represented by list structure to internal data structures so that *equivalent* nodes map to the same internal data structure (in the same location in memory).

Dynamic programming, and quotient acceleration, can be used to accelerate general purpose inference. We have already discussed the fact that expressions can be represented as nodes in a graph structure. The graph structure representation of expressions is analogous to the intern operation on nodes in a graph search problem. The ability to mark nodes in the graph structure provides a way of avoiding inferring the same same assertion twice. It is possible to accelerate the inference process further by quotienting the nodes representing expressions — a set of nodes that have all been proven equivalent can be replaced by a single data structure representing the equivalence class of expressions.

It turns out that for representations of expressions it is more natural to view the quotiented graph structure as a different kind of object — a context free grammar. Each equivalence class of expressions can be represented by a nonterminal symbol of the grammar. For example, consider the equation $<f\ a> = a$ (the function f applied to the constant a equals the constant a). The set of expressions that are provably equivalent to $<f\ a>$ from this single equation is the language generated by the nonterminal symbol X in the following grammar.

$$X \Rightarrow a$$

$$X \Rightarrow <f\ X>$$

This grammar states that, to generate an expression in the equivalence class, one can either generate the symbol a or one can generate the expression

$<fX>$ and then recursively replace X by a recursively generated expression.

Let Υ be a set of expressions and let Σ be a set of equations. Running the congruence closure procedure on the graph structure representation of Υ, using the equations in Σ, will result in some assignment of colors to nodes. Each color represents an equivalence class of expressions in Υ. These equivalences classes can be represented by a grammar.

> **Definition:** The *quotient structure* of the expression set Υ under the equation set Σ, denoted Υ/Σ, is the grammar defined by the following conditions.
>
> - For each expression t that appears (either as a member or a subexpression of a member) in Υ or Σ there is a nonterminal symbol X_t such that for any two expressions t and w that appear in Υ or Σ the nonterminal symbol X_t is the same as the nonterminal symbol X_w if and only if $\Sigma \ \vdash_C \ t = w$.
> - For each noncompositional expression t (e.g., constant symbol) that appears in Υ or Σ the grammar contains the production
> $$X_t \Rightarrow t.$$
> - For each compositional expression $<t_1, \cdots, t_k>$ that appears in Υ or Σ, the grammar contains the production
> $$X_{<t_1, \cdots, t_k>} \Rightarrow <X_{t_1}, \cdots, X_{t_n}>.$$
>
> **Lemma:** If t is a term that appears in either Υ or Σ then the nonterminal symbol X_t of the grammar Υ/Σ generates the language consisting of all expressions w such that $\Sigma \ \vdash_C \ t = w$.

The quotient structure Υ/Σ can be computed from Υ and Σ by construct the graph structure representation of all terms that appear in either Υ or Σ and then running the congruence closure procedure from Σ to color all of the nodes in this graph structure. For each color in the resulting colored graph structure one constructs a nonterminal symbol. Finally, one can construct productions according to the above definition where X_t is the nonterminal associated with the color assigned to the node n_t.

When one can derive many equivalences it is more efficient to manipulate grammars than to manipulate either expressions or graph structure representations of expressions. It is possible to define the congruence closure procedure as an operation that takes a grammar G, and an equation $X = Y$ between nonterminal symbols of the grammar G, and computes a new grammar $G/(X = Y)$ — the grammar that represents all of the equations implicit in the grammar G plus the new equation represented by $X = Y$.

Consider Boolean constraint propagation. It is possible to define Boolean constraint propagation as an inference process on a grammar. Each assertion is an expression and can therefore be represented by a nonterminal symbol of the grammar. The inference rules of Boolean constraint propagation can be phrased as inference rules on the grammar. For example, consider the inference rule of modus ponens.

$$\frac{\begin{array}{l}\Phi \\ \Phi \rightarrow \Psi\end{array}}{\Psi}$$

This inference rule can be phrased in terms of grammars as follows. If a nonterminal symbols X and Y have both been labeled true, and the grammar contains the production

$$X \Rightarrow \langle Y \rightarrow Z \rangle$$

then label the nonterminal Z true. In the grammar formulation this inference rule says that if X is true, and X is equivalent to the implication $Y \rightarrow Z$, and Y is true, then label Z true. Inference on the grammar provides quotient acceleration in the sense that one avoids repeating *equivalent* inferences.

13 Problems

1. Consider the propositional logic defined by n Boolean variables. A model of this logic is an assignment of values to these variables and each variable can be either true or false. How many models are there in the propositional logic generated by n Boolean variables?

2. Each assertion of simple propositional logic is some Boolean combination of the Boolean variables. For example, $\mathcal{V}(P \vee Q, \mathcal{M})$ equals $\mathbf{T}$ if either $\mathcal{V}(P, \mathcal{M})$ equals $\mathbf{T}$ or $\mathcal{V}(Q, \mathcal{M})$ equals $\mathbf{T}$. Two assertions Φ and Ψ are called *equivalent* if they are true in exactly the same models, i.e., for any model $\mathcal{M}$, we have that $\mathcal{V}(\Phi, \mathcal{M})$ equals $\mathcal{V}(\Psi, \mathcal{M})$. How many non-equivalent assertions are there in the propositional logic generated by n Boolean variables, i.e., how many equivalence classes of assertions are there? (Hint: Define each equivalence class in terms of models.)

3. In this problem we assume some fixed but arbitrary logic. Let $\vdash$ be a relation defined by an inference procedure. Given a set of assertions Σ and a assertion Φ, the procedure decides whether $\Sigma \vdash \Phi$. We define $\vdash^*$ to be another inference relation such that $\Sigma \vdash^* \Phi$ just in case $\Phi \in \Sigma$ or $\Sigma \vdash \Phi$ or there exists some Γ such that $\Sigma \vdash^* \Psi$ for every $\Psi \in \Gamma$ and $\Gamma \vdash \Phi$ ($\vdash^*$ can be though of as the "transitive closure" of $\vdash$). Is it possible that $\vdash$ is sound and yet $\vdash^*$ is different from $\vdash$? Suppose that $\vdash$ is an inference relation generated by inference rules. In this case can $\vdash^*$ be different from $\vdash$? Is it possible that $\vdash$ is both sound and complete and yet $\vdash^*$ is different from $\vdash$? Explain all your answers.

4. The relation $\vdash$ is called *monotonic* if for any assertion set Σ and any assertion Φ, if $\Sigma \vdash \Phi$ then $\Sigma \cup \Gamma \vdash \Phi$ for any set of assertions Γ. Is it possible that $\vdash$ is sound and yet nonmonotonic? Is it possible that $\vdash^*$ is nonmonotonic? Is it possible that $\vdash$ is sound, complete, and nonmonotonic? Explain all answers.

5. The inference relation $\vdash_B$ is defined using the case analysis sequent rule. Give an ordinary inference rule, i.e., a rule that derives assertions from assertions rather than sequents form sequents, that can be instead of the case analysis sequent rule to generate a semantically complete inference relation

when combined with the BCP rules. Give an argument that your ordinary rule generates the same inference relation as the case analysis sequent rule.

6. Consider the logic whose assertions consist of equations of the form $a = b$ where a and b are constant symbols. A model of this logic is a mapping from constant symbols to "colors" (it does matter what colors really are as long as there are as many colors as constant symbols in the logic.) $\mathcal{V}(a = b,\ \mathcal{M})$ equals $\mathbf{T}$ if and only if $\mathcal{M}(a)$ is the same as $\mathcal{M}(b)$. Prove that the inference relation $\vdash_E$ generated by the symmetry, reflexifity, and transitivity rules for equality is semantically complete for this logic.

7. Give a procedure for determining whether $\Sigma \vdash_{BCP+E} \Phi$ that runs in order $n \log n$ time. You may use the union procedure from the notes. Give an explicit argument that your procedure only takes order $n \log n$ time.

8. Give a proof that the procedure defined in problem 7 is actually a decision procedure for the relation $\vdash_{BCP+E}$. More specifically, let $\vdash$ be the relation defined by the procedure, i.e., $\Sigma \vdash \Phi$ if and only if the procedure terminates with the claim that Φ is provable from Σ. Show that $\Sigma \vdash \Phi$ implies $\Sigma \vdash_{BCP+E} \Phi$ and that $\Sigma \vdash_{BCP+E} \Phi$ implies $\Sigma \vdash \Phi$.

9. This problem is difficult. Consider a finite set of constant symols and function symbols. Suppose we define a *term* to be either a constant symbol or an application $f(t)$ where f is a function symbol and t is a term. Note that every term has the form $f_1(f_2(\cdots f_k(c)))$ where each f_i is a function symbol and c is a constant symbol. (We are only considering function symbols that take a single argument, but this problem can also be done without much more difficulty for function symbols of any number of arguments.) Consider a logic whose assertions are equations of the form $s = t$ where s and t are terms and where a model of this logic interprets each constant symbol as a "colors" and each function symbol as a function from colors to colors (it does not matter what the set of colors really is as long as the number of colors is infinite). We define $\mathcal{V}(c, \mathcal{M})$ to be $\mathcal{M}(c)$; we define $\mathcal{V}(f(t),\ \mathcal{M})$ to be $\mathcal{M}(f)(\mathcal{V}(t,\ \mathcal{M}))$; and we define $\mathcal{V}(s = t,\ \mathcal{M})$ to be $\mathbf{T}$ if and only if $\mathcal{V}(s,\ \mathcal{M})$ is the same as $\mathcal{V}(t,\ \mathcal{M})$. Prove that $\vdash_C$, the inference relation generated by the three basic rules for equality plus the congruence rule, is semantically complete for this logic. Hint: Generalize the solution for problem 6 but considering equivalence classes of terms rather than equivalence classes of

constants. Note that in this problem the set of terms is infinite.

10. In this problem we will consider the relationship between constraint propagation and inference rules. Recall that a constraint satisfaction problem consists of a set of domain variables where each domain variable has a finite set of possible values, plus a set of constraints on simultaneous assignments of values. Recall that an assignment is a assertion of the form $x = c$ and a disassignment is a assertion of the form $x \neq c$. We will consider inference rules that, in addition to assignments and disassignments, involve assertions of the following forms.

$$x \in \{d_1, \cdots, d_n\}$$

$$x = c \rightarrow y \in \{d_1, \cdots, d_k\}$$

A assertions of the first type states that the value of x must be one of $d_1, \cdots, d_n$. A assertion of the second type states that if x is c then y is one of the values $d_1, \cdots, d_k$.

We now translate a constraint satisfaction problem into a set of assertions of the above form. For each domain variable x there is a assertion of the form $x \in \{d_1, \cdots, d_k\}$ where $d_1, \cdots, d_k$ are the possible domain values for x. We assume that each constraint involves at most two variables. For each constraint $\Phi(x, y)$, and for each possible value c of x, we have a assertion of the form $x = c \rightarrow y \in \{d_1, \cdots, d_k\}$ where $d_1, \cdots, d_k$ is the set of all values of y that are consistent with $x = c$ under the constraint $\Phi(x, y)$. A constraint of the form $\Phi(x, y)$ also generates similar assertions of the form $y = d \rightarrow x \in \{c_1, \cdots, c_k\}$.

Let $q_1, \cdots, q_n$ be domain variables representing the queens on an eight by eight chess board. Each domain variable ranges over the numbers one through eight. For each pair of queen variables q_i and q_j there is a constraint $\Phi(q_i, q_j)$ which states that q_i and q_j do not attack each other. Describe the assertion set Σ generated, as described above, by these domain variables and the constraints of the eight queens problem.

11. Recall that constraint propagation can be done in various strengths. In this problem we consider generalized forward checking and arc consistency. Arc consistency is stronger, but slower, than generalized forward checking. Both generalized forward checking and arc consistency incorporate the fol-

lowing two inference rules.

$$\frac{\begin{array}{l} x \in \{d_1, \cdots, d_k\} \\ x \neq d_1, \cdots, x \neq d_{i-1}, x \neq d_{i+1}, \cdots, x \neq d_k \end{array}}{x = d_i}$$

$$\frac{x = c}{x \neq b}$$

where b is any value other than c.

Both generalized forward checking and arc consistency provide an additional way of deriving disassignments. In generalized forward checking a disassignment can be derived from a single constraint *and assignments*. In arc consistency, a disassignment can be derived from a single constraint *and disassignments*.

Give a third inference rule, capable of deriving disassignments, that when combined with the above two rules define generalized forward checking. Your rule may be associated with a condition, such as the condition on the second rule above, stating when the rule can be applied. Also give an inference rule, capable of deriving disassignments, that together with the two rules given above define arc consistency. (Hint: Again consider the eight queens example).

12. Consider a term of the form $x_1 + x_2 + \cdots + x_n$ where $+$ is a binary operator that is associative and commutative, i.e., $x_1 + x_2 = x_2 + x_1$ and $(x_1 + x_2) + x_3 = x_1 + (x_2 + x_3)$. Define a grammar that generates the set of all terms equivalent to $x_1 + x_2 + \cdots x_n$. How many nonterminal symbols does your grammar contain? How many productions does your grammar contain? Give both answers as a function of n and explain each answer. For $n = 5$, compare the number of productions in your grammar for representing the term $x_1 + x_2 + \cdots x_n$ with the number of distinct ways this term can be written (different parenthesizations count as distinct ways of writing the term.) You should give the exact number of productions and the exact

number of distinct ways the term can be written. (It is quite difficult to derive a general answer as a function of n for the number of distinct ways of writing the expression. However, if you are sufficiently clever, computing the exact value for $n = 5$ is not very arduous.)

13. Consider a set of predicate symbols $P_1, \cdots, P_k$. Let R be a set of inference rules where each rule has the form

$$\frac{\begin{array}{l} P_i(x) \\ P_j(y) \end{array}}{P_w(x+y)}$$

where any term can be substituted for x and y. Let $\vdash_R$ be the inference relation generated by R, plus the four basic properties of equality (reflexivity, transitivity, symmetry, and substitution), plus the associative and commutative properties of $+$. Describe a procedure based on the grammar representation of terms for determining whether or not $\Sigma \ \vdash_R \ \Phi$. (You do not have to give a proof that your procedure is correct, but try to make sure that it is.) Explain why the decision procedure based on grammars should be much more efficient than a similar procedure based on classical expressions (much more efficient, even, than a procedure based on DAG representations of expressions).