A Theory of Abstraction

Toby Walsh

PhD
University of Edinburgh

1990

You who have robbed my heart,
depriving it of everything,
who have demanded my soul in delirium,
dearest, accept my gift ...

Mayakovsky

Declaration

I declare that this thesis has been composed by myself. Although some of the work described in this thesis is the result of collaboration with Fausto Giunchiglia, I declare that I have made a substantial contribution to this work. Whilst is it difficult to identify exactly the individual contributions, much of the research described in Chapters 3, 4, 6, and 7 is solely my own work.

Acknowledgements

In writing this thesis, I acquired enumerable debts; it's therefore impossible to mention everyone by name. But you know, I hope, who you are. I will, however, explicitly thank:

Alan Bundy for giving me the opportunity, providing the very best criticism, and guiding me to the end.

Fausto Giunchiglia to whom I owe the most. His ideas and enthusiasm brought this research into being. He is a great teacher, and, just as importantly, a very fine friend. My life is richer for knowing him.

The DReaMers past, present and future. Alan, Andrew and Andrew, Carole, Christian, Colin, Dave and Dave, Geraint, Jane, Mandy, Paul, Seán, ... For providing the enjoyable environment we all work in. And for putting up with my sartorial excesses.

The Mechanised Reasoners. Alessandro, Alex, Carola, Luciano, Paolo, Peck, and Lorenza. For putting up with me the rest of the time.

My family for giving me your love and support. Although you are often a long way away, you are always in my heart.

My friends for making it all worthwhile.

Abstract

Abstraction is the process of mapping one representation of a problem onto a simpler, more abstract representation; the abstract solution can then used to guide the search for a solution to the original, more complex problem. By providing a global control of the search, abstraction can greatly improve our problem solving ability. Unfortunately, the use of abstraction has in general lacked sound and theoretical foundations causing many problems. This thesis therefore proposes a general purpose **theory of abstraction**. We use this theory to classify the various types of abstraction, to investigate their formal properties, to analyse and criticise previous work in abstraction, to find methods for building abstractions automatically, and to explore how to use abstractions.

Publications

Some of this research has already been appeared in print; any prior publication will be indicated at the appropriate point in the text. To summarise, the following papers contain description of research which is reported in this thesis:

[BGW90] A. Bundy, F. Giunchiglia, and T. Walsh. Building abstractions. In *Proceedings of the AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions*, pages 221–232, American Association for Artificial Intelligence, 1990.

[GW89a] F. Giunchiglia and T. Walsh. Abstract Theorem Proving. In *Proceedings of the 11th IJCAI*, International Joint Conference on Artificial Intelligence, 1989. Also available as DAI Research Paper No 430, Dept. of Artificial Intelligence, Edinburgh.

[GW89b] F. Giunchiglia and T. Walsh. *Abstract theorem proving: mapping back*. Research Paper 460, Dept. of Artificial Intelligence, University of Edinburgh, 1989. An updated version of this paper has been submitted to IJCAI-91.

[GW89c] F. Giunchiglia and T. Walsh. Abstracting into inconsistent spaces (or the false proof problem). In *Proceedings of AI*IA 89*, Associazione Italiana per l'Intelligenza Artificiale, 1989. Also available as DAI Research Paper, Dept. of Artificial Intelligence, Edinburgh.

[GW89d] F. Giunchiglia and T. Walsh. Theorem Proving with Definitions. In *Proceedings of AISB 89*, Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1989. Also available as DAI Research Paper No 429, Dept. of Artificial Intelligence, Edinburgh.

[GW90a] F. Giunchiglia and T. Walsh. Abstraction in AI. *AISB Quaterly*, (73):22–26, 1990.

[GW90b] F. Giunchiglia and T. Walsh. Abstraction in automatic inference. In *Proceedings of the UK-IT 90 Conference*, pages 365–370, 1990.

Contents

Notation

<u>Sets, functions, and relations</u>

$\emptyset$	the empty set
$\{x \mid p(x)\}$	the set of all objects which satisfy $p(x)$
2^A	the power set of A
$\in$	set membership
$\cup, \cap$	set union, and intersection
$\subseteq$	subset
$\times$	cartesian product, $A^2 = A \times A$
$f : A \mapsto B$	a function, f with domain A and image B
$\circ$	composition of functions
$\lambda x \,.\, t(x)$	a lambda abstraction
$\langle A, ...\rangle$	an ordered list
$\sim$	an equivalence relation
$[x]$	the equivalence class of x with respect to $\sim$
$A/\!\sim$	the quotient set of A with respect to $\sim$

<u>Logic</u>

$\varphi, \alpha, \beta, ...$	well formed formulae (wffs)
$p(x), q(x), ...$	predicates
$\top, \bot$	true and false
$a, b, ...$	constants
$x, y, ...$	object-level variables
$X, Y, ...$	meta-level variables
$\{a/x\}$	substitution of a for x
$\theta, ...$	substitutions
$\underline{x}, ...$	vectors, often the n arguments to a predicate
$\vee, \wedge, \rightarrow, \leftrightarrow$	logical connectives
$\neg$	logical negation
$\forall, \exists$	logical quantifiers
$\Sigma, \Sigma_1, ...$	axiomatic formal systems, $\Sigma = \langle \Lambda, \Omega, \Delta \rangle$
$\Lambda, \Lambda_1, ...$	languages (usually sets of wffs)
$\Omega, \Omega_1, ...$	axioms (sets of wffs)
$\Delta, \Delta_1, ...$	deductive machineries (sets of inference rules)
$\subseteq$	containment (of a language within another, of a formal system within another, ...)
$\langle \mathcal{A}, \mathcal{T}, \mathcal{W} \rangle$	a language consisting of an alphabet, and sets of rules defining the well formed terms, and the well formed formulae
$\langle \mathcal{L}, \mathcal{V}, \mathcal{F}, \mathcal{P} \rangle$	an alphabet consisting of sets of logical symbols, variables, function symbols, and predicate symbols
$\mathrm{TH}(\Sigma)$	the theorems of Σ
$\mathrm{NTH}(\Sigma)$	the negation of the theorems of Σ

$\vdash_\Sigma$	derivable in Σ
$\mathcal{MT}_0$, $\mathcal{MT}_1$, ...	classes of meta-theoretic statements
s_0, s_1, ...	meta-theoretic statements
$A => B$	sequent
$\alpha \Rightarrow \beta$	rewrite rule

<u>Abstractions</u>

$f : \Sigma_1 \Rightarrow \Sigma_2$	an abstraction
f, g, h	mapping functions
$\mathcal{ABS}$	the set of all abstractions
$\mathcal{ABS}_\Sigma$	the set of all abstractions with the same ground space Σ
$\circ$	composition of abstractions
$\leq$, $\preceq$	partial orders on abstractions
$<$, $\prec$	strict orders on abstractions
$\equiv$, $\cong$	equivalences of abstractions
$\bowtie$, $\asymp$	incomparability of abstractions
$\mathcal{F}$	an abstraction schema, $\mathcal{F} : 2^{\Lambda_1} \mapsto \mathcal{ABS}$
$\mathrm{f} : \Sigma_1 \rightsquigarrow \Sigma_2$	an analogy

<u>Algebraic theories</u>

$T, T_1, ...$	algebraic theories, $T = \langle S, E \rangle$
$S, S_1, ...$	signatures, $S = \langle \mathcal{S}, \mathcal{O} \rangle$
$E, E_1, ...$	sets of equations
$\overline{E}, \overline{E_1}, ...$	the deductive closures of the sets of equations
$\mathcal{S}$, $\mathcal{S}_1$, ...	sorts of algebraic languages
$\mathcal{O}$, $\mathcal{O}_1$, ...	operators (constants and function names) of algebraic languages
$\sigma : S_1 \Rightarrow S_2$	signature morphism
$\sigma : T_1 \Rightarrow T_2$	theory morphism
T/σ, ...	quotient theories

<u>Trees</u>

$\Pi, \Pi_1, \Gamma, \Gamma_1, ...$	formulae trees
$b, b_1, ...$	branches of a tree
$hd(b), tl(b)$	root and tail of a branch
$\|\Pi\|$	the depth of Π
$\|\|\Pi\|\|$	the weight of Π
$\sqsubseteq$	subtree relation
$\sqsubset$	strict subtree relation
$\subseteq$	tree subsumption
$\simeq$	tree isomorphism
$\mathcal{N}(\varphi, \Pi)$	number of occurrences of the wff φ in Π

Model theory

$\mathcal{I}, \mathcal{I}_1, ...$	interpretations, $\mathcal{I} = \langle \mathcal{D}, \Phi, \Psi \rangle$
$\mathcal{D}, \mathcal{D}_1, ...$	domains of interpretations
$\Phi, \Phi_1, ...$	interpretation of function symbols
$\Psi, \Psi_1, ...$	interpretation of predicate symbols

Search spaces

$c(b, l, t)$	cost function (time to prove a theorem)
$b, b_0, ...$	branching rare of a search space
$l, l_0, ...$	length of proof
$t, t_0, ...$	time to perform one inference
τ_n	time to prove theorem using n levels of abstraction
$\tau(m, n)$	time to prove the theorem at the m-th level using levels m to n
$a_0, a_1, ...$	time to abstract a wff
$u_0, u_1, ...$	time to unabstract a wff
$g_0, g_1, ...$	size of gaps in abstract proof plans

Chapter 1

Introduction

This Chapter introduces the problem that concerns the rest of this thesis: what is abstraction ? We motivate why this is an interesting question, and outline the goals we hope to achieve.

1.1 Abstraction

Much research in AI is directed at tackling the complexity inherent in solving interesting problems; abstraction is one very general purpose heuristic that can help to attack this problem. It has been used in many areas of AI: theorem proving, planning, commonsense reasoning, learning, *etc.* Indeed its use goes back to some of the very earliest AI systems like GPS [NS72] and ABSTRIPS [Sac74].

Abstraction can be thought of as the mapping of one representation of a problem, the **ground** representation onto a new but simpler representation, the **abstract** representation that can help solve the original problem; the abstract representation is simpler because the mapping usually throws away details. The overall aim may be to improve the efficiency of reasoning, or alternatively to increase the number of derivable facts. For example, learning systems sometimes construct abstract versions of rules, making them easier to match and allowing them to fire in new situations. To be of any use, the abstract representation must be closely related to the ground representation; certain properties of the ground representation (*eg.* which rules fire) must be preserved by the mapping.

1.2 An Example

Consider the following fragment of a Prolog program for planning journeys between cities:

```
route(A,A,[]).
route(A,B,[train(A,C)|Rest]):-
       train(A,C),
       route(C,B,Rest).
route(A,B,[plane(A,C,Airline)|Rest]):-
       plane(A,C,Airline),
       route(C,B,Rest).
```

```
train(london,edinburgh).
train(edinburgh,london).
train(london,paris).
etc.

plane(moscow,milan,al).
plane(london,moscow,ba).
plane(london,milan,ba).
etc.
```

The program might include a huge database of train and plane connections. To search this database naively using Prolog's depth-first and left-to-right search strategy would not be very practical. For example, to travel from Edinburgh to Milan, the program might suggest a route via Moscow:

```
[ train(edinburgh,london), plane(london,moscow,ba),
              plane(moscow,milan,al) ]
```

A breadth-first or iterative deepening search would return a more sensible answer but at great computational cost. A better solution is to abstract the problem onto a more manageable one.

The first abstraction we could make is to change the "grain size". Instead of planning a route between individual cities, we plan a route between the different countries. The problem of finding a route from Edinburgh to Milan becomes the simpler problem of finding a route between Scotland and Italy. It would be much easier to discover then that there are no direct flights, and that we need to change planes in England (either at London or Manchester). We can build a program that represents this abstraction by mapping any constant in the database representing a town onto a constant representing its country. The language of the abstracted database is considerably less complex than the original language. Instead of discussing connections from every town in a country, it just has those between countries. Additionally, the size of the database could be considerably reduced. For example, the facts `plane(london,milan,ba)` and `plane(london,rome,ba)` can be represented by just one atomic fact, `plane(england,italy,ba)`:

```
route(A,A,[]).                              train(england,scotland).
route(A,B,[train(A,C)|Rest]):-              train(scotland,england).
       train(A,C),                          train(england,france).
       route(C,B,Rest).                     etc.
route(A,B,[plane(A,C,Airline)|Rest]):-
       plane(A,C,Airline),                  plane(ussr,italy,al).
       route(C,B,Rest).                     plane(england,ussr,ba).
                                            plane(england,italy,ba).
                                            etc.
```

It is now much easier to find the routing:

```
[ train(scotland,england), plane(england,italy,ba) ]
```

Of course, this routing between Scotland and Italy may not be of much use. It might correspond to a journey that starts in Glasgow, ends in Rome and leaves us stranded in Manchester with a connecting flight out of Heathrow. An abstract plan therefore needs to be refined by putting back the details which the abstraction threw away.

A further abstraction we could make is to ignore the actual routing. That is, we want to know simply if there is a route between two countries, not how we travel or where we make connections. A program to achieve this would have much simpler rules. Since we ignore the routing, we do not need to make any distinction between journeys using trains and those using planes. We also do not need an accumulator to record connections.

```
route(A,A).                     connect(england,scotland).
route(A,B):-                    connect(scotland,england).
        connect(A,C),           connect(england,france).
        route(C,B).             etc.

                                connect(ussr,italy).
                                connect(england,ussr).
                                connect(england,italy).
                                etc.
```

Again the size of the database can shrink considerable. For example, the facts `plane(england,france,ba)`, and `train(england,france)` can all be represented by the one abstract fact `connect(england,france)`.

The purpose of this thesis is to describe and to understand such abstractions as these.

1.3 Motivation

Unfortunately the use of abstraction in AI has in general lacked sound and theoretical foundations. Plaisted's work [Pla81] is the one honourable exception to this criticism. However, this work is not very general since it is limited to resolution theorem proving and one particular class of abstractions. The lack of a theoretical foundation to the use of abstraction has led to many problems. First, the number of different abstractions has grown immensely without a corresponding increase in understanding about the different classes of abstractions that exist. Second, abstractions have been used without a sufficient understanding of the reasons why they work, why they don't work, or of the trust we can place in their answers. It seems, in some cases, more a matter of luck than of design that they do work. Indeed, large pitfalls await the unwary; for example, by abstracting away detail, we can throw away just the information that kept the problem representation from becoming contradictory. Third, there seem almost as many different ways to use abstractions as there are different abstractions. Given a particular abstraction, it is unclear how you decide to use it, or why it should be used in a particular way. And fourth, abstractions have in general been constructed by hand. Without a comprehensive theory of abstraction, it has been impossible to determine automatically what to abstract for a given problem representation. This has greatly restricted the usefulness of abstraction as a general purpose problem solving heuristic. To conclude, it seems that a theoretical understanding of abstraction is long overdue.

1.4 Aims

Our main goal is to develop a general **theory of abstraction**. This theory should be both descriptive and prescriptive; we want to be able both to describe previous work in abstractions and to suggest new abstractions and uses of abstraction. Our aims are:

- to classify the various forms of abstraction;
- to investigate the formal properties of abstractions;
- to define operations that can be performed on abstractions, and relations for comparing them;
- to analyse and criticise past work;
- to explore how to build useful abstractions;
- to study how abstract problem solving can help solve the original problem.

This thesis discusses each of these topics in varying detail; each corresponds to a separate Chapter.

1.5 Structure of thesis

The thesis is organised as follows. Chapter 2 presents our basic **theory of abstraction**; this theory is developed and investigated throughout the rest of the thesis. In Chapter 3, we explore some of the consequences of our theory, identifying various properties possessed by abstractions. Chapter 4 tests the expressive adequacy of our theory by describing various abstractions that have been proposed in the past. In Chapter 5, we look at a very common problem that occurs with many of these abstractions, the introduction of inconsistency. Chapters 6 and 7 then consider how we might actually use abstraction. In Chapter 6, we study how to build abstractions automatically for new problems, and in Chapter 7 we study how an abstract solution can help us find a solution to the original problem. Finally, we end in Chapter 8 with a summary of the achievements of this thesis, its limitations and some suggestions for future work.

1.6 Mathematical preliminaries

The rest of this thesis will assume various basic notions concernings sets, relations and functions.

Sets

A **set** A is a collection of objects; each object a in the collection is called a **member** of the set, written $a \in A$. The set having no member, *ie.* the set A for which $\neg(a \in A)$ for any object a, is called the **empty set**, 0. The set of all objects which satisfy the property $p(x)$ is represented by $\{x \mid p(x)\}$. A set A is a **subset** of another set B, written $A \subseteq B$ iff $a \in A$ implies $a \in B$. By definition, the empty set is a subset of any set. The **union** of the sets A and B, $A \cup B$ is the set whose members belong either to A or to B. The **intersection** of the sets A and B, $A \cap B$ is the set whose members belong to both A and B. Two sets are **disjoint** iff their intersection is the empty set. The **power set** of A, written 2^A is the set of all subsets of A. If the set A has n elements, then its power set, 2^A

has 2^n elements. The **cartesian product** of the sets A, B, written $A \times B$ is the set of all pairs, $\langle a, b\rangle$ for which $a \in A$ and $b \in B$. The cartesian product can be extended from two sets to any number of sets. By A^2 we mean $A \times A$. Similarly for A^n where $n \geq 2$.

Relations

A binary **relation** R on the sets A and B is the set of pairs, $\langle a, b\rangle \in A \times B$ for which $R(a, b)$ holds. For the sake of brevity, we will normally use relation to mean binary relation. A relation R is **reflexive** iff for all x, $R(x, x)$ holds. A relation R is **irreflexive** iff for all x, $\neg R(x, x)$ (that is, $R(x, x)$ never holds). A relation R is **symmetric** iff $R(x, y)$ implies $R(y, x)$. A relation R is **transitive** iff $R(x, y)$ and $R(y, z)$ implies $R(x, z)$. And a relation R is **antisymmetric** iff $R(x, y)$ and $R(y, x)$ implies $x = y$.

A relation is a **preorder** iff it is transitive, and reflexive. A relation is a **weak partial order** iff it is transitive, antisymmetric, and reflexive. A relation is a **strict partial order** iff it is transitive and irreflexive. A relation is an **equivalence relation** iff it is reflexive, symmetric and transitive. If R is an equivalence relation then the **equivalence class** of x with respect to R, often written $[x]$ is the set $\{y \mid R(x, y)\}$ and the **quotient set** of a set A, written A/R is the set of all equivalence classes of A.

Functions

A **function** $f : A \mapsto B$ is a rule which assigns a member of B to every member of A. The expression $f(a)$ is used to represent the member of B to which a is assigned. We will say that $f(a)$ is the **application** of the function, f to a; we also use the notation, fa. If $f : A \mapsto B$ is a function then A is called the **domain** and B the **image** of f. We can extend a function to a mapping on sets in the obvious way: $f(A) = \{f(a) \mid a \in A\}$. The composition of two functions, $f : A \mapsto B$ and $g : B \mapsto C$ is the function, $f \circ g : A \mapsto C$ for which $f \circ g(a) = g(f(a))$. A function $f : A \mapsto B$ is **surjective** iff $f(A) = B$. A function is **injective** iff $a \neq b$ implies

$f(a) \neq f(b)$. A function is **total** if it is defined for all its domain, and **partial** otherwise. All the functions we will use in this thesis are total unless otherwise indicated. A function is **computable** if it can be computed by a Turing machine, or equivalently if it is a general recursive function. The **lambda abstraction**, $\lambda x \,.\, t(x)$ is the function which on application with a gives the value, $t(a)$. The rule for rewriting $(\lambda x \,.\, t(x))a$ as $t(a)$ is called **beta reduction**.

1.7 Summary

This thesis describes a general theory of abstraction. We will use this theory to investigate the formal properties of abstractions. The purpose of this investigation is to classify and criticise previous informal work, and to explore how to build and how to use abstractions. The result is a comprehensive understanding of abstraction, how it has been used (and misused), and how it should be used.

Chapter 2

A Theory of Abstraction

This Chapter presents the beginnings of a **theory of abstraction**. *We define abstraction as a mapping between representations of a problem, illustrating our formal definition by means of an example. We then consider the desirable properties that should be preserved by such a mapping.*

2.1 Introduction

This thesis adopts the "logicist" approach to Artificial Intelligence advocated by McCarthy and Hayes [McC77]; we attempt to model reasoning with abstraction in terms of logical reasoning. Minsky strongly criticises the logicist approach [Min81], arguing that it is inflexible, infeasible, and impractical. We shall not offer any defence to the first two criticisms, except to remark that much research has been devoted to increasing the flexibility of the logicist approach (*eg.* capturing nonmonotonic reasoning [McC80]) and to representing real world knowledge within a logical framework (*eg.* capturing commonsense knowledge [Hay79, Hay85]). This thesis should, however, help to answer Minksy's third criticism – abstraction can reduce search and make logical reasoning more practical.

Most of the work described in this Chapter first appeared in [GW89a].

2.2 Formal Systems

In the last Chapter, we informally described abstraction as a mapping between representations of a problem that throws away details but preserves certain desirable properties. Thus, in presenting a formal *theory* of abstraction, we begin by giving a very general method for describing representations of a problem. In line with our logicist approach, we choose formal systems for this purpose.

Following Kleene [Kle52], a **formal system** is a formal description of a theory; this theory may be Peano arithmetic, or the planning world of STRIPS. In this thesis, we will restrict our attention to **axiomatic formal systems**. This is not a significant restriction since we can represent most problems with axiomatic formal systems. Moreover, most of our analysis could be easily generalised to formal systems which are not axiomatic. Unless we explicitly state to the contrary, we will use "formal system" to mean "axiomatic formal system".

> **Definition 1 (Axiomatic formal system)** *: An* **axiomatic formal system** Σ *is a triple* $\langle \Lambda, \Omega, \Delta \rangle$*, where* Λ *is the* **Language***,* Ω *is the* **set of axioms** *and* Δ *is the* **Deductive Machinery** *of* Σ*.*

An example of an axiomatic formal system is the theory of groups with axioms for group identity, inverse and associativity. Usually a language is defined by giving the alphabet, and (rules for constructing) the set of well formed terms and the set of well formed formulae. That is, $\Lambda = \langle \mathcal{A}, \mathcal{T}, \mathcal{W} \rangle$. To simplify matters, we will normally forget about the alphabet and well formed terms and just say that the language is the set of well formed formulae (wffs from now on). The alphabet and well formed terms are given implicitly by providing the set of wffs. The axioms are the basic wffs which are accepted as theorems and from which, by application of the inference rules, all other theorems are derived. Thus $\Omega \subseteq \Lambda$. The deductive machinery is the set of inference rules which allows new theorems to be derived from existing ones. We shall not address the problem of representing inference rules, and such difficult issues as side-conditions and assumptions. Instead, we will adopt the standard Natural Deduction conventions and terminology of Prawitz [Pra65].

For the sake of simplicity, we will usually restrict ourselves to formal systems in which the axioms and the inference rules are a subset of those of first order logic. Even with such a restriction, we are able to capture most previous work in abstraction. Additionally, most of our arguments would remain true if we dropped this restriction.

2.3 Properties of Formal Systems

We briefly define some basic properties of axiomatic formal systems that will be used in the rest of the thesis.

One formal system, $\Sigma_1 = \langle \Lambda_1, \Omega_1, \Delta_1 \rangle$ is **contained** within another formal system, $\Sigma_2 = \langle \Lambda_2, \Omega_2, \Delta_2 \rangle$, written $\Sigma_1 \subseteq \Sigma_2$ iff $\Lambda_1 \subseteq \Lambda_2$, $\Omega_1 \subseteq \Omega_2$ and $\Delta_1 \subseteq \Delta_2$. The set of **theorems** of Σ, written $\mathbf{TH}(\Sigma)$ is the minimal set of wffs containing the axioms and closed under the inference rules. The theorems are the statements in the language which can be proven to be true. We say that Σ is **syntactically incomplete** if there is a formula α such that $\alpha \notin TH(\Sigma)$ and $\neg\alpha \notin TH(\Sigma)$. We call it **syntactically complete** otherwise. Notice that this is not the same as the idea of **completeness**, in which every formulae valid in a model is provable. A formal system, Σ is **absolutely inconsistent** iff for any wff α, $\alpha \in TH(\Sigma)$, and **inconsistent** iff there exists a wff α such that $\alpha \in TH(\Sigma)$ and $\neg\alpha \in TH(\Sigma)$. For an absolutely inconsistent system, negation need not be part of the language. In classical first order logic, an absolutely inconsistent system is inconsistent, and vice versa. A formal system, $\Sigma_1 = \langle \Lambda_1, \Omega_1, \Delta_1 \rangle$ is **monotonic** iff for any formal system $\Sigma_2 = \langle \Lambda_1, \Omega_2, \Delta_1 \rangle$ with $\Omega_1 \subseteq \Omega_2$, $\alpha \in TH(\Sigma_1)$ implies $\alpha \in TH(\Sigma_2)$.

In **proof systems**, we are interested in those wffs that can proven true. By comparison, in **refutation systems**, we are interested in those wffs that can be proven not to be true; with such systems, we show that certain (sometimes negated) wffs are inconsistent with the axiom and therefore cannot be true. Thus, we also define the set containing wffs which are the **negation of the theorems** of Σ, written $\mathbf{NTH}(\Sigma)$. This is the set of the wffs each of which, if added to the axioms, makes Σ absolutely inconsistent. $TH(\Sigma)$ and $NTH(\Sigma)$ are obviously related. For instance in classical first order logics, $\alpha \in NTH(\Sigma)$ iff $\neg\alpha \in TH(\Sigma)$; Σ is inconsistent iff $TH(\Sigma) = NTH(\Sigma) = \Lambda$, or iff $TH(\Sigma) \cap NTH(\Sigma) \neq \emptyset$; outside $TH(\Sigma)$ and $NTH(\Sigma)$ but inside the language are all those formulae α such that neither α nor $\neg\alpha$ belongs to TH(Σ); if a theory is syntactically complete then no formula is outside the union of TH(Σ) and NTH(Σ).

2.4 A Definition of Abstraction

Formal systems provide a very general method for describing the ground and the abstract representations of a problem. We can now define abstraction simply as a mapping between formal systems. Our definition of abstraction has two parts: the ground and abstract representations of a problem, and a mapping which relates the two representations. It is not sufficient just to give the two representations; we also need some way of relating (the parts of) one representation to (the parts of) the other.

> **Definition 2 (Abstraction)** *: An* **abstraction** *is a triple consisting of the formal systems Σ_1, Σ_2 and a total and computable function f which maps from the language of Σ_1 onto that of Σ_2.*

We will use the notation "$f : \Sigma_1 \Rightarrow \Sigma_2$" to represent an abstraction. This should be read as *syntactic sugar* for the triple $\langle \Sigma_1, \Sigma_2, f \rangle$. It should not be read as a function f from Σ_1 to Σ_2; it is simply a pair of formal systems, each being a representation of a problem and a mapping between them which relates wffs in one representation to wffs in the other. Following historical convention [Sac74], we call Σ_1 the **ground space** and Σ_2 the **abstract space**. In logic texts, "ground" is usually used to mean that an expression has no free or bound variables. To avoid confusion, we will always use "ground" to mean belonging to the representation we abstract, and "variable-free" to describe an expression with no free or bound variables. The function f is called the **mapping function**; we require that this function be total since we want to be able to "translate" any wff in the ground space into the abstract space. We require that it be computable because we actually want to be able to use such abstractions. However, most of our arguments would stand if we dropped this last requirement. We shall use $\mathcal{ABS}$ to stand for the infinite set of possible abstractions, and $\mathcal{ABS}_\Sigma$ to represent those with the common ground space, Σ. Where there is no ambiguity we will occasionally shorten "$f : \Sigma_1 \Rightarrow \Sigma_2$" simply to "$f$".

This notion of abstraction is very weak and is more general than those previously used. Indeed, it captures many types of mappings which are not usually thought of as abstractions; we also expect the properties of the ground and abstract spaces to be related, and the abstract space to be "simpler" than the ground space. Although we will only consider mappings between axiomatic formal systems, this definition of abstraction could equally well describe a mapping between *arbitrary* formal systems and not just between those that are axiomatic or logical.

2.5 An Example

We illustrate our definition of abstraction by means of an example. Consider the propositional abstraction proposed by Plaisted [Pla80] and used in [GW89d] to plan the unfolding of definitions. We can describe this as an abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ between a first order theory, Σ_1 and a propositional theory, Σ_2. The mapping function abstracts first order wffs onto propositional wffs by keeping the connective structure and throwing away quantifiers and arguments. Atomic wffs are mapped as follows:

$$f(p(\underline{x})) = p$$

All atomic formulae with the same predicate symbol map onto the same propositional constant. Thus "$a =_{set} b \rightarrow a \subseteq_{set} b$" in the ground language maps onto "$=_{set} \ \rightarrow \ \subseteq_{set}$" in the abstract language where "$=_{set}$" and "$\subseteq_{set}$" are propositional constants of the abstract language. The axioms of Σ_2 are formed by applying this same mapping function to the axioms of Σ_1.

This abstraction maps one representation of a problem onto a more abstract representation of the same problem. The abstract space is simpler than the ground space; indeed, this abstraction maps an undecidable ground space onto a decidable abstract space. Additionally, inference is cheaper in the abstract space than in the ground space since we need not perform full blown unification. This abstraction "throws away details" by dropping the quantifiers and the arguments to formulae. It is useful as certain "desirable properties" are preserved by the mapping. For instance, the definitions that must be unfolded in the ground space to prove a theorem are the same as the definitions unfolded in a proof of the abstract theorem. Thus theorem proving in the abstract space can be used to plan the unfolding of definitions in the ground space.

Consider, for example, the theorem that if two sets are equal then one is a subset of the other, "$a =_{set} b \rightarrow a \subseteq_{set} b$". This has a proof in the ground space which unfolds both the definitions of set equality and subset:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{a =_{set} b \qquad a =_{set} b \ \leftrightarrow\ \forall x.x \in a \leftrightarrow x \in b}{\forall x.x \in a \leftrightarrow x \in b}}{h \in a \leftrightarrow h \in b}}{h \in a \rightarrow h \in b}}{\forall x.x \in a \rightarrow x \in b \qquad a \subseteq_{set} b \ \leftrightarrow\ \forall x.x \in a \rightarrow x \in b}}{a \subseteq_{set} b}}{a =_{set} b \rightarrow a \subseteq_{set} b}$$

There is a proof of the abstraction of this theorem which unfolds the same definitions. Note that this abstract proof is shorter than the ground proof:

$$\cfrac{\cfrac{\cfrac{\cfrac{=_{set} \qquad =_{set} \leftrightarrow (\in \leftrightarrow \in)}{\in \leftrightarrow \in}}{\in \rightarrow \in \qquad \subseteq_{set} \leftrightarrow (\in \rightarrow \in)}}{\subseteq_{set}}}{=_{set} \rightarrow \subseteq_{set}}$$

The abstract proof should be easier to find than the ground proof; it could therefore be used to plan the unfolding of definitions in the ground space. Such a use of abstraction was proposed in [GW89d]. Notice also the similarity between the structure of the ground and the abstract proofs. We shall return to this topic in Chapter 7 when we show how to map abstract proofs back onto ground proofs.

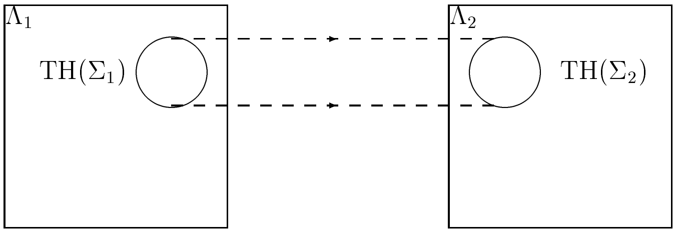
2.6 Preserving Provability

We informally described abstraction as a mapping between representations of a problem that was useful because it preserved certain "desirable properties". We now consider what these desirable properties might be.

A central notion in theorem proving is that of provability. We are therefore interested in how abstractions affect provability. This idea will play a central rôle in the rest of the thesis. Preserving provability is actually only a very weak property to demand of an abstraction; there are other desirable properties which must also be captured. The main idea underlying the use of abstractions is to prove the theorem in the abstract space (which, supposedly, should be simpler than in the ground space) and then to use the structure of the proof in the abstract space to guide the search for a proof in the ground space. This assumes that the structure of the abstract proof is "similar" to the structure of the ground proof; this is discussed in more detail in Chapter 7. Preserving provability is a necessary but not a sufficient condition for the structure of the abstract proof to be similar to that of the ground proof. It is, therefore, a very important property for an abstraction to possess. Indeed, we would claim that, even under the weak assumption that an abstraction simply preserves provability, we are able to prove some very powerful results. Our first step towards capturing abstractions that preserve certain desirable properties is therefore to classify abstractions by the way they preserve provability:

Definition 3 (T*-abstractions) *: An abstraction $f : \Sigma_1 \Rightarrow \Sigma_2$ is said to be a*

1. **TC-abstraction** *iff, for any wff, α, $\alpha \in TH(\Sigma_1)$ implies and is implied by $f(\alpha) \in TH(\Sigma_2)$;*
2. **TD-abstraction** *iff, for any wff, α, if $f(\alpha) \in TH(\Sigma_2)$ then $\alpha \in TH(\Sigma_1)$;*
3. **TI-abstraction** *iff, for any wff, α, if $\alpha \in TH(\Sigma_1)$ then $f(\alpha) \in TH(\Sigma_2)$.*



This is a graphical representation of a TC-abstraction. In this (and the five following figures) the two boxes represent the sets of wffs belonging to the two languages. The dashed lines show the behaviour of the abstraction mapping.

Figure 2.1: TC-abstractions

"T" stands for "theorem", "C" for "constant", "D" for "decreasing" and "I" for "increasing". TI-abstractions have also been called **truthful** abstractions [GW89a]. An abstraction is a **T*-abstraction** iff it is a TC-abstraction, a TD-abstraction or a TI-abstraction. A TC-abstraction is both a TI- and a TD-abstraction.

Theorem 1 *: A TC-abstraction is both a TD-abstraction and a TI-abstraction.*

Proof: Immediate from Definition 3. □

A TC-abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ maps all the members of TH(Σ_1) onto members of TH(Σ_2) and these are the only members of TH(Σ_2). This is represented graphically in figure 2.1. Many examples of TC-abstractions can be found; for example, any mapping which changes the representation of a problem without throwing away any information can be described as a TC-abstraction. Reduction methods in decision theory can be seen as TC-abstractions since we map some (syntactically defined) class of formulae onto a decidable class of formulae, and show that a formulae in the original class is provable iff its mapping is.

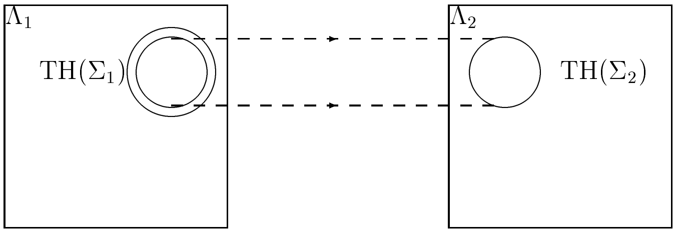


Figure 2.2: TD-abstractions

A TD-abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ maps a subset of the members of $\mathrm{TH}(\Sigma_1)$ onto $\mathrm{TH}(\Sigma_2)$; this subset generates all the members of $\mathrm{TH}(\Sigma_2)$. This is represented graphically in figure 2.2. TD-abstractions have been used by Tenenberg [Ten87, Ten88] and by Wos [WRD65] (for a good reference to this work, see [Bun83]).

A truthful or TI-abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ maps all the members of $\mathrm{TH}(\Sigma_1)$ onto a subset of $\mathrm{TH}(\Sigma_2)$. This is represented graphically in figure 2.3. In may ways, TI-abstractions are dual to TD-abstractions; we explore this duality in greater detail in Chapter 3. As far as we are aware, the majority of abstractions used in the past (*eg.* those proposed in [Pla80, Pla81]) are truthful; many examples of truthful abstractions are given in Chapter 4.

It is not just the theorems of the ground and abstract spaces that are related; often there is also a great similarity in the shape of a proof of a theorem in the ground space and of a proof of the abstraction of the theorem in the abstract space. The steps of the abstract proof can thus be used to guide theorem proving in the ground space, providing the major "islands" that we need to reach; we move between these islands by filling in details or by applying the inference rules thrown away by the abstraction. Truthfulness is a first step to guaranteeing a correspondence between proof steps in the ground and the abstract spaces.

Not all previously proposed abstractions are truthful or TI-abstractions. Why do we argue for the use of TI-abstractions and not the use of TD- or TC-abstractions? Tenenberg, for instance, puts forward TD-abstractions (and others

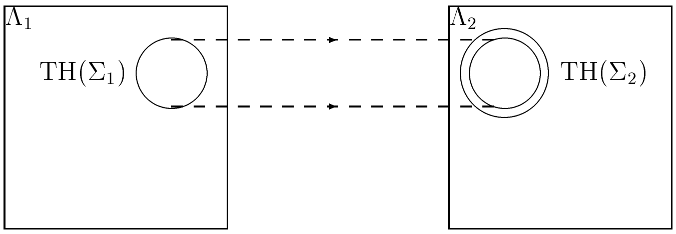


Figure 2.3: TI-abstractions (Truthful abstractions)

with similar properties) as a way to avoid the problem of abstractions mapping consistent ground spaces onto inconsistent abstract spaces [Ten87, Ten88].

In general, TC-abstractions are too strong and do not give "simpler" proofs. For example, if $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TC-abstraction, Σ_1 is undecidable, and the inverse of the mapping function is computable then Σ_2 cannot be decidable. Of course, this does not mean that TC-abstractions are useless. They are very useful, for instance, in changing the representation of a problem to one with which we can reason more efficiently.

We would also argue against the use of TD-abstractions which are not TC-abstractions as completeness is lost; there are theorems of the ground space whose abstractions are not theorems of the abstract space. If the abstract space is going to be used to help find a proof in the ground space, we consider completeness one property that you do not want to lose. We do not wish to take a great stand on the issue of completeness versus efficiency. We simply mean that there is no *a priori* reason for loosing completeness, and even less reason for losing it in an uncontrolled fashion.

There are even a few abstractions proposed in the past which do not preserve provability in any direction. One such example is the abstraction used in "gazing" [Plu87], a heuristic for planning the unfolding of definitions. Unlike the abstraction we briefly described in section 2.5, this abstraction gave neither a complete nor a sound strategy for deciding when to unfold definitions. We cannot be certain when it will suggest the appropriate definitions to unfold, nor when it will fail to identify the appropriate definitions to unfold. For this reason, we would criticise this abstraction (and other abstractions which do not preserve provability in a precise way) as being too *ad hoc*.

2.7 Preserving Deducibility

We have characterised abstractions by the way they preserve provability. A more general property than provability is deducibility. A deducibility relation is a set of ordered pairs; the first element of the pair is a set of wffs whilst the second element is a wff which can be derived from the axioms if we assume that the set of wffs in the first element are also true. We shall write $\Gamma \vdash_{\Sigma} \alpha$ to mean that α is deducible (derivable) in Σ from the axioms and the set of wffs, Γ. Provability is a particular case of deducibility; a wff is a theorem iff it is deducible from the empty set. Thus, $\vdash_{\Sigma} \alpha$ is an alternative notation for $\alpha \in TH(\Sigma)$. Further discussion about deducibility relations and their properties can be found in [Avr87, BS84].

Even if this thesis deals with provability, most of the analysis could have been given for deducibility. A deducibility preserving abstraction will preserve provability and, subject to the conditions of the following theorem, a provability preserving abstraction will preserve deducibility.

Theorem 2 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a T^*-abstraction that preserves implications (that is, $f(\alpha \rightarrow \beta) = f(\alpha) \rightarrow f(\beta)$), and the deduction theorem and modus ponens hold in both Σ_1 and Σ_2 then $f : \Sigma_1 \Rightarrow \Sigma_2$ also preserves deducibility.*

Proof: We only consider TI-abstractions. The other proofs are entirely analogous. If $\alpha_1, ...\alpha_n \vdash_{\Sigma_1} \beta$ then, from the deduction theorem in Σ_1, $\vdash_{\Sigma_1} \alpha_1 \rightarrow ...\alpha_n \rightarrow \beta$. Since the abstraction is truthful, $\vdash_{\Sigma_2} f(\alpha_1 \rightarrow ...\alpha_n \rightarrow \beta)$. But $f : \Sigma_1 \Rightarrow \Sigma_2$ is implication preserving. Thus $\vdash_{\Sigma_2} f(\alpha_1) \rightarrow ...f(\alpha_n) \rightarrow f(\beta)$. By modus ponens, $f(\alpha_1), ...f(\alpha_n) \vdash_{\Sigma_2}$

$f(\beta)$. $\Box$

As provability is a notion that is perhaps more commonly used in theorem proving than deducibility, we will restrict ourselves in the rest of this thesis to abstractions which preserve provability. However, subject to the rather weak hypotheses of the above theorem, everything true of provability preserving abstractions also holds for deducibility preserving abstractions.

2.8 Preserving Inconsistency

So far, abstractions have been classified by the relationship between provability in the ground space and provability in the abstract space. This is appropriate for proof systems where the deductive machinery of both spaces is used to generate theorems. In refutation systems, however, the deductive machinery is used to determine inconsistency. In such situations, abstractions are better classified by how inconsistency in one formal system is mapped onto inconsistency in the other formal system. Entirely dual to definition 3, we define the following inconsistency preserving abstractions:

Definition 4 (NT*-abstractions) *: An abstraction $f : \Sigma_1 \Rightarrow \Sigma_2$ is said to be a*

1. **NTC-Abstraction** *iff, for any wff α, $\alpha \in NTH(\Sigma_1)$ iff $f(\alpha) \in NTH(\Sigma_2)$;*
2. **NTD-Abstraction** *iff, for any wff α, if $f(\alpha) \in NTH(\Sigma_2)$ then $\alpha \in NTH(\Sigma_1)$;*
3. **NTI-Abstraction** *iff, for any wff α, if $\alpha \in NTH(\Sigma_1)$ then $f(\alpha) \in NTH(\Sigma_2)$.*

NTI-abstractions have also been called **falseful** abstractions [GW89a]. An abstraction is a **NT*-abstraction** iff it is a NTC-abstraction, a NTD-abstraction or a NTI-abstraction. It is a **TC*-abstraction** iff it is a TC-abstraction or a NTC-abstraction, **TD*-abstraction** iff it is a TD-abstraction or a NTD-abstraction, and **TI*-abstraction** iff it is a TI-abstraction or a NTI-abstraction. When an abstraction is known to fall in more than one of the above classes we will write all of them in the prefix. Thus a "**TC/NTC-abstraction**" is both a TC- and a NTC-abstraction. We will also write **TH*(**Σ**)** to mean NTH(Σ) or TH(Σ). Statements made involving names containing "*" should be read by substituting in all valid ways the same letter *uniformly* inside the sentence. For statements like "T*-abstraction" or "NT*-abstraction" this means substituting the letter "C", "D", or "I", whilst for statements like "TC*-abstraction", "TD*-abstraction" or "TI*-abstraction" this means substituting either the letter "N" or " ". Thus, the claim that "a TC*-abstraction is a TD*-abstraction" should be read as "a TC-abstraction is a TD-abstraction" or as "a NTC-abstraction is a NTD-abstraction". It should not be read as "a TC-abstraction is a NTD-abstraction" or "a NTC-abstraction is a TD-abstraction".

Everything that has been said about T*-abstractions holds dually for NT*-abstractions. For instance, we can generalise Theorem 1 to give:

> **Theorem 3** *: A TC*-abstraction is both a TD*-abstraction and a TI*-abstraction.*
>
> **Proof:** Immediate from Definitions 3 and 4. □

We introduced NT*-abstractions to describe mappings between refutation systems. In the next section, we will argue that NT*-abstractions play the same rôle for mappings between refutation systems as T*-abstractions play for mappings between proof systems.

2.9 Refutation Systems

The essential idea behind most refutation systems is that a wff, α is a theorem of a formal system, Σ if adding its negation to the axioms, Ω gives inconsistency.

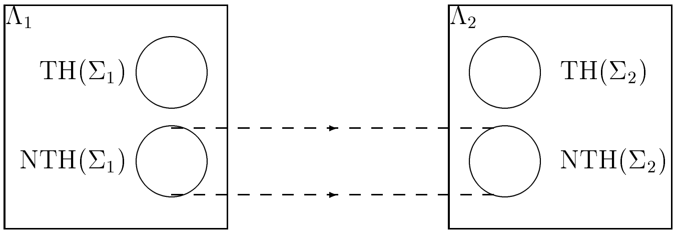


Figure 2.4: NTC-abstractions

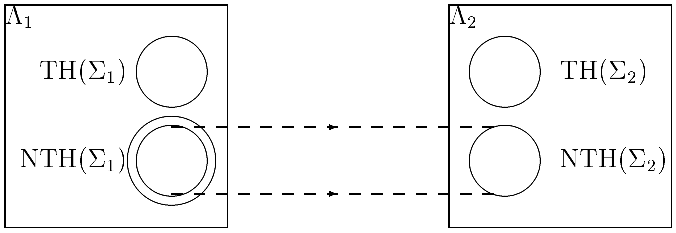


Figure 2.5: NTD-abstractions

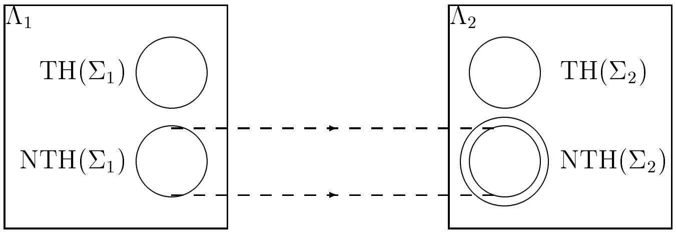


Figure 2.6: NTI-abstractions (Falseful abstractions)

In formulae,

$$\vdash_\Sigma \alpha \quad \text{iff} \quad \{\neg\alpha\} \vdash_\Sigma \bot$$

Thus, though we use a refutation system to determine inconsistency, we are really interested in provability. Our claim that NT*-abstractions play the same rôle in refutation systems as T*-abstractions in proof systems would therefore seem to deserve greater justification. If we are still interested in provability when using refutation systems then we might also expect to be still interested in T*-abstractions.

T*-abstractions in Refutation Systems

Everything depends on how the user interacts with the system. In some (if not all cases), the goal that we wish to prove is input and the system automatically negates it before adding it to the set of axioms. If this happens in the abstract space, the negation sign in front of the goal is **not** abstracted, the system effectively works on provability, and T*-abstractions are the appropriate abstractions to use. Before we can formalise this argument, we need to define a very common and useful notion, that of a formal system with negation. We will use this notion to show the assumptions under which a T*-abstraction can be used with refutation systems.

> **Definition 5 (System with negation)** *:* Σ *is* **a formal system with negation** *iff for any wff,* α
>
> *1.* α *is a wff iff* $\neg\alpha$ *is a wff;*
>
> *2.* $\alpha \in TH(\Sigma)$ *iff* $\neg\alpha \in NTH(\Sigma)$*;*
>
> *3.* $\neg\alpha \in TH(\Sigma)$ *iff* $\alpha \in NTH(\Sigma)$.

From the second condition, it follows that a system with negation is inconsistent iff it is absolutely inconsistent. The following theorem identifies the assumptions under which T*-abstraction can be used in refutation systems.

Theorem 4 *: If Σ_1 and Σ_2 are two formal systems with negation then $f : \Sigma_1 \Rightarrow \Sigma_2$ is a*

- **TC-abstraction** *iff for any α, $\neg\alpha \in NTH(\Sigma_1)$ iff $\neg f(\alpha) \in NTH(\Sigma_2)$;*
- **TD-abstraction** *iff for any α, if $\neg f(\alpha) \in NTH(\Sigma_2)$ then $\neg\alpha \in NTH(\Sigma_1)$;*
- **TI-abstraction** *iff for any α, if $\neg\alpha \in NTH(\Sigma_1)$, then $\neg f(\alpha) \in NTH(\Sigma_2)$.*

Proof: Immediate from Definitions 3 and 5. □

Thus, provided both the ground and abstract spaces are systems with negation and we negate the (abstracted) goal, we can use T*-abstractions with refutation systems. For instance, with a TI-abstraction if α is a theorem of the ground space then the negation of its abstraction, $\neg f(\alpha)$ will make a refutation system stop with success (that is, by generating $\perp$).

Claim *: T*-abstractions can be used with refutation systems provided the abstraction maps between systems with negation and the wff added to the axioms of the abstract space is the* **negation** *of the abstraction of the wff whose negation is added to the axioms of the ground space.*

In formulae, instead of adding $f(\neg\alpha)$ to the axioms of the abstract space, we must add $\neg f(\alpha)$. When the system does not negate the goal, and $f(\neg\alpha)$ is added to the axioms of the abstract space, NT*-abstractions should be used.

NT*-abstractions in Proof Systems

We have established the conditions under which T*-abstractions with refutation systems. The dual question also arises, namely can NT*-abstractions be used with proof systems? Theorem 4 holds dually. That is:

Theorem 5 *: If Σ_1 and Σ_2 are two formal systems with negation, then $f : \Sigma_1 \Rightarrow \Sigma_2$ is a*

- **NTC-abstraction** *iff for any α, $\neg\alpha \in TH(\Sigma_1)$ iff $\neg f(\alpha) \in TH(\Sigma_2)$;*
- **NTD-abstraction** *iff for any α, if $\neg f(\alpha) \in TH(\Sigma_2)$ then $\neg\alpha \in TH(\Sigma_1)$;*
- **NTI-abstraction** *iff for any α, if $\neg\alpha \in TH(\Sigma_1)$, then $\neg f(\alpha) \in TH(\Sigma_2)$.*

Proof: Immediate from Definitions 4 and 5. □

The interpretation of this theorem is entirely dual to that of Theorem 4.

Claim *: NT*-abstractions can be used in proof systems to prove a wff, $\neg\alpha$ provided we try to prove the wff, $\neg f(\alpha)$ in the abstract space.*

Abstractions for Refutation and Proof Systems

Finally, are there abstractions which can be used both with refutation systems and with proof systems, irrespective of how the goal is negated? To help answer this question, we define another very common notion, that of an abstraction which preserves negation. This notion is crucial for abstractions to be used in both refutation and proof systems.

Definition 6 (Negation preserving abstractions) *: Let Σ_1 and Σ_2 be two systems such that α is a wff iff $\neg\alpha$ is. An abstraction $f : \Sigma_1 \Rightarrow \Sigma_2$ is* **negation preserving** *iff, for any α, $f(\neg\alpha) = \neg f(\alpha)$.*

Preserving negation is a very important concept. If an abstraction preserves negation then the notions of preserving provability and of preserving inconsistency collapse together. A negation preserving abstraction can thus be used in both refutation and proof systems.

Theorem 6 *: If Σ_1 and Σ_2 are two formal systems with negation, and $f : \Sigma_1 \Rightarrow \Sigma_2$ is a negation preserving abstraction then $f : \Sigma_1 \Rightarrow \Sigma_2$ a T*-abstraction iff it is a NT*-abstraction.*

Proof: We only consider TI-abstractions and the forward direction as the other proofs are analogous. Since Σ_1 is a system with negation, if $\alpha \in NTH(\Sigma_1)$ then $\neg\alpha \in TH(\Sigma_1)$. But the abstraction is truthful. Thus $f(\neg\alpha) \in TH(\Sigma_2)$. As the abstraction is negation preserving, this implies $\neg f(\alpha) \in TH(\Sigma_2)$. From which it follows that $f(\alpha) \in NTH(\Sigma_2)$ and that $f : \Sigma_1 \Rightarrow \Sigma_2$ is a NTI-abstraction. $\Box$

Theorem 6 demonstrates that preserving negation is the property which links abstractions for refutation and proof systems, supporting the following claim:

Claim *: Negation preserving abstractions can be used in both refutation and proof systems.*

Preserving negation is not a very restrictive requirement since nearly all the abstractions used in the past are, in fact, negation preserving.

2.10 Summary

We have defined abstraction as a mapping between formal systems, illustrating this definition by an example. We have also considered the desirable properties preserved by such mappings. In particular, for proof systems we have concentrated on how such mappings preserve provability (and, for refutation systems, inconsistency). Under certain weak restrictions, like the preservation of negation, provability preserving abstractions also preserve inconsistency and vice versa. In later Chapters we will show that this framework is very general, and that the preservation of provability (and inconsistency) is a very useful and powerful way to characterise abstractions.

Chapter 3

Some Properties

We have defined abstraction as a mapping between formal systems that preserves certain desirable properties like provability or inconsistency; we now explore some consequences of this definition. In particular, we define various operations on abstractions and relations between them. We also consider the internal structure of the mapping, identifying different ways in which the parts of the ground and abstract spaces are related.

3.1 Introduction

In Chapter 2, we presented the beginnings of a **theory of abstraction**. We now explore some of the properties of this theory. Our analysis is along two different dimensions since we will consider both the internal and the external properties of abstractions. For the internal properties, we will define various relationships between the parts of a *single* abstraction. For the external properties, we will define various relationships between *different* abstractions.

We begin the Chapter by considering some of the external properties. We have defined an abstraction as a mathematical object; that is, as a pair of formal systems and a mapping function. Thus, it is natural to consider various mathematical operations, like composition, which can be applied to these objects. The main purpose of defining such operations is to build new abstractions from old ones. We also consider various mathematical relations, like a partial ordering, which are true of these objects. The main purpose of defining these relations is for comparing one abstraction with another.

We end the Chapter with a section describing some of the internal properties. Simple relationships often exist between the different parts of an abstraction. Indeed, in most abstractions proposed in the past, the mapping function and the pair of formal systems are intimately related. The aim of this last section is to define precisely this intimacy.

3.2 Some Operations on Abstractions

We begin, in fact, with an important relation. We need some way of identifying when two abstractions represent the same object; this is simply when every part of the two triples are equal.

> **Definition 7 (Equality)** *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ and $g : \Sigma_3 \Rightarrow \Sigma_4$ are abstractions then $f : \Sigma_1 \Rightarrow \Sigma_2$ is* **equal** *to $g : \Sigma_3 \Rightarrow \Sigma_4$ iff $f = g$, $\Sigma_1 = \Sigma_3$, and $\Sigma_2 = \Sigma_4$.*

Equality of abstractions is an equivalence relation, being reflexive, symmetric and transitive. If two abstractions are equal then they behave identically. In particular, they must share the same provability and inconsistency preserving properties.

One of the most useful operations we can perform on abstractions is that of composition; this allows us to combine old abstractions together to give new ones.

> **Definition 8 (Composition)** *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ and $g : \Sigma_2 \Rightarrow \Sigma_3$ are abstractions then $f \circ g : \Sigma_1 \Rightarrow \Sigma_3$ is their abstraction* **composition**.

Where there is no ambiguity, we will write "$f \circ g$" for "$f \circ g : \Sigma_1 \Rightarrow \Sigma_3$". The composition of two abstractions is itself an abstraction as the composition of two computable and total mapping functions is itself a computable and total function. The composition of two abstractions is at least as "strong" as either of the individual abstractions, since it throws away the same information as the first abstraction *and* the same information as the second. Abstraction composition can be repeatedly applied, and is associative. Note that the composition of two abstractions is only defined if the abstract space of the first abstraction is the same as the ground space of the second. This inability to compose two arbitrary abstractions can be overcome in one of two ways. We can insist that the same language and inference rules are used in the abstract space as in the ground space, and we define abstraction *schemata* that will work with any set of axioms; we provide such a generalisation in Chapter 5. Alternatively, we can introduce a new object for the composition of two incompatible abstractions, namely the **undefined abstraction**. Composing anything with the undefined abstraction gives the undefined abstraction. In both cases, abstraction composition is then defined everywhere and the set of abstractions, $\mathcal{ABS}$ with respect to composition is an associative binary algebra; that is, a **semigroup**.

> **Theorem 7** *: $\langle \mathcal{ABS}, \circ \rangle$ is a semigroup*
>
> **Proof:** A semigroup is a set with a binary function which is defined everywhere and is associative. $\mathcal{ABS}$ and the composition operator form such a structure. $\Box$

Note that $\langle \mathcal{ABS}, \circ \rangle$ is not a monoid, a semigroup with an unique (left and right) identity as abstractions with different ground spaces have different right identities, and abstractions with different abstract spaces have different left identities. Identity abstractions do exist; it is just that they are not unique.

> **Definition 9 (Identity)** *: An abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ is called the* **identity** *abstraction of Σ_1 iff $\Sigma_1 = \Sigma_2$ and the mapping function f is an identity function.*

An identity abstraction maps any formal system onto itself; it is uniquely determined by the ground space. Having defined an equality, a composition operator, and an identity abstraction, it is natural to define the inverse of an abstraction.

Definition 10 (Inverse) *: If* $f : \Sigma_1 \Rightarrow \Sigma_2$ *and* $g : \Sigma_2 \Rightarrow \Sigma_1$ *are abstractions and* $f \circ g : \Sigma_1 \Rightarrow \Sigma_1$ *is the identity abstraction of* Σ_1 *then* $g : \Sigma_2 \Rightarrow \Sigma_1$ *is called an* **inverse** *of* $f : \Sigma_1 \Rightarrow \Sigma_2$*.*

The inverse of an abstraction is, by definition, an abstraction. The inverse of a surjective abstraction, in which the abstract language is the image of the ground language, is unique.

Definition 11 (Surjectivity) *: Let* Σ_1 *and* Σ_2 *have the languages* Λ_1 *and* Λ_2 *respectively. An abstraction,* $f : \Sigma_1 \Rightarrow \Sigma_2$ *is* **surjective** *iff its mapping function is surjective. That is, iff* $\Lambda_2 = \{f(\varphi) |\ \varphi \in \Lambda_1\}$*.*

Theorem 8 (Uniqueness of inverse) *: If* $f : \Sigma_1 \Rightarrow \Sigma_2$ *is a surjective abstraction, and* $g : \Sigma_2 \Rightarrow \Sigma_1$ *and* $h : \Sigma_2 \Rightarrow \Sigma_1$ *are both inverses of* $f : \Sigma_1 \Rightarrow \Sigma_2$*, then* $g : \Sigma_2 \Rightarrow \Sigma_1$ *equals* $h : \Sigma_2 \Rightarrow \Sigma_1$*.*

Proof: By contradiction. Assume that the two abstractions are not equal. Their mapping functions, f and g must therefore not be equal. Thus, from the surjectivity of $f : \Sigma_1 \Rightarrow \Sigma_2$, there must exist $f(\varphi)$ such that $g(f(\varphi)) \neq h(f(\varphi))$. Hence $f \circ g(\varphi) \neq f \circ h(\varphi)$. But $f \circ g$ and $f \circ h$ are both identity functions. Thus $\varphi \neq \varphi$. $\square$

The inverses of an abstraction that is not surjective can only differ in the mapping of the (irrelevant) abstract formulae which are outside the image of the ground language. Of course, not all abstractions have inverses. In fact, it is only possible to invert those abstractions whose mapping functions do not throw away any information. To be more precise, an abstraction must be injective if it is to be invertible.

Definition 12 (Injectivity) *: An abstraction* $f : \Sigma_1 \Rightarrow \Sigma_2$ *is* **injective** *iff its mapping function is injective. That is, iff if* $\alpha \neq \beta$ *then* $f(\alpha) \neq f(\beta)$*.*

Theorem 9 (Existence of inverse) *: An abstraction has an inverse iff it is injective.*

Proof: ($\Rightarrow$) Consider the abstractions $f : \Sigma_1 \Rightarrow \Sigma_2$. Assume that this has an inverse, $g : \Sigma_2 \Rightarrow \Sigma_1$ but that $f : \Sigma_1 \Rightarrow \Sigma_2$ is not injective. Then there will exist α, β such that $\alpha \neq \beta$ but $f(\alpha) = f(\beta)$. Thus, $g(f(\alpha)) = g(f(\beta))$. That is $\alpha = \beta$, contradicting $\alpha \neq \beta$.

($\Leftarrow$) Assume that $f : \Sigma_1 \Rightarrow \Sigma_2$ is injective. We define a mapping function, $g : \Lambda_2 \mapsto \Lambda_1$ by $g(\alpha) = \beta$ if $\alpha = f(\beta)$. The injectivity of $f : \Sigma_1 \Rightarrow \Sigma_2$ guarantees the uniqueness of β. If α is outside the image of Λ_1 under f (*ie.* there is no β such that $\alpha = f(\beta)$), we define $g(\alpha)$ to be any arbitrary formula in Λ_1. The abstraction, $g : \Sigma_2 \Rightarrow \Sigma_1$ is an inverse of $f : \Sigma_1 \Rightarrow \Sigma_2$. $\square$

Since they cannot throw away any information from the language, injective abstractions do not in general give a simpler abstract theory and are not of much practical use.

3.3 Properties of these Operations

We have defined abstraction as a mapping between formal systems which preserves certain desirable properties like provability or inconsistency. Therefore, for such operations to be useful, we need to know how they affect the preservation of provability and inconsistency.

The inverse of an abstraction has the inverse provability or inconsistency preserving property of the original abstraction. In other words a TI*-abstraction inverts to a TD*-abstraction, and vice versa.

Theorem 10 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a surjective TI*-abstraction (TD*-abstraction) with an inverse $g : \Sigma_2 \Rightarrow \Sigma_1$, then this inverse is a TD*-abstraction (TI*-abstraction).*

Proof: We consider the case where $f : \Sigma_1 \Rightarrow \Sigma_2$ is TI. The other cases are entirely dual. If $\varphi \in TH(\Sigma_1)$ then $f(\varphi) \in TH(\Sigma_2)$. Now $g(f(\varphi)) = \varphi$. Thus, $g(f(\varphi)) \in TH(\Sigma_1)$ implies $f(\varphi) \in TH(\Sigma_2)$. Since $f : \Sigma_1 \Rightarrow \Sigma_2$ is surjective, $f(\varphi)$ ranges over the whole of the abstract language, and $g : \Sigma_2 \Rightarrow \Sigma_1$ is a TD-abstraction. □

Identity abstractions are TC*.

Theorem 11 *: If $f : \Sigma_1 \Rightarrow \Sigma_1$ is the identity abstraction of Σ_1 then it is a TC*-abstraction.*

Proof: Since $f(\varphi) = \varphi$, $f(\varphi) \in TH(\Sigma_1)$ iff $\varphi \in TH(\Sigma_1)$. Thus $f : \Sigma_1 \Rightarrow \Sigma_1$ is a TC-abstraction. And, by a dual argument, it is also a NTC-abstraction. □

The composition of two abstractions preserves provability or inconsistency in the same way as its components. For example, the composition of two TI*-abstractions is itself a TI*-abstraction.

Theorem 12 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ and $g : \Sigma_2 \Rightarrow \Sigma_3$ are TI*-abstractions (TD*-abstractions) then their composition $f \circ g : \Sigma_1 \Rightarrow \Sigma_3$ is also a TI*-abstraction (TD*-abstraction).*

Proof: We consider the case of TI-abstractions. The other cases are dual. If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TI-abstraction, and $\varphi \in TH(\Sigma_1)$ then $f(\varphi) \in TH(\Sigma_2)$. However, $g : \Sigma_2 \Rightarrow \Sigma_3$ is also TI. Thus, $g(f(\varphi)) \in TH(\Sigma_3)$. Hence, $f \circ g(\varphi) \in TH(\Sigma_3)$, and $f \circ g : \Sigma_1 \Rightarrow \Sigma_3$ is a TI-abstraction. □

Composing abstractions which preserve provability or inconsistency in different ways gives unpredictable results. For example, composing a TI-abstraction with a NTI-abstraction can give an abstraction that is neither a TI- nor a NTI-abstraction. In general, being a T*-abstraction gives no guarantee that the negation of the theorems are mapped in a useful way. Similarly, being a NT*-abstraction gives, in general, no guarantee that theorems are mapped in a useful way. In theorem 6, we demonstrated the very weak conditions necessary for T*-abstractions (NT*-abstractions) to map the negation of theorems (theorems) in a useful way. If a T*-abstraction preserves negation and is between systems with negation, then it is also a NT*-abstraction and vice versa. Under these conditions, T*-abstractions can be safely composed with NT*-abstractions. We summarise all these results about composing abstractions in the following table.

$\circ$	TD	TC	TI	NTD	NTC	NTI
TD	TD	TD	?	?	?	?
TC	TD	TC	TI	?	?	?
TI	?	TI	TI	?	?	?
NTD	?	?	?	NTD	NTD	?
NTC	?	?	?	NTD	NTC	NTI
NTI	?	?	?	?	NTI	NTI

This table describes the properties of the composition of an abstraction possessing the property given by the row heading with an abstraction possessing the property given by the column heading. Thus, the entry in the second row and third column indicates that the composition of a TC-abstraction with a TI-abstraction is another TI-abstraction. The symbol "?" is used to indicate that the provability or inconsistency preserving properties of the composition of two abstractions is not predictable. The table is symmetrical about the principal diagonal, and dual with respect to the other diagonal. This last fact is a reflection of the duality between T*- and NT*-abstractions, which we define more formally in Section 3.5. The table has complete symmetry about the horizontal and vertical axes for negation preserving abstractions between systems with negation; under such assumptions, a T*-abstraction is also a NT*-abstraction and vice versa, the table would contain just 8 question marks, and is completely determined by a single quadrant.

3.4 Some Relations between Abstractions

We have already described two very useful relations between abstractions. The first was an equality relation (definition 7); this identifies when two abstractions represent the *same* object. The second was a relation between T*-abstractions and NT*-abstractions; given certain weak conditions, a T*-abstraction is also a NT*-abstraction and vice versa (theorem 6). This Section and Sections 3.5 to 3.8, identify some further relations that exist between the different types of provability and inconsistency preserving abstractions.

In general, we deal with syntactically incomplete and consistent theories in which $\alpha \notin TH(\Sigma)$ does not imply $\neg\alpha \in TH(\Sigma)$ and vice versa. However, when both the ground and abstract spaces are syntactically complete, the different forms of abstractions coincide.

> **Theorem 13** *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is an abstraction between two syntactically complete systems with negation then $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TI-abstraction iff it is a NTD-abstraction. Dually, $f : \Sigma_1 \Rightarrow \Sigma_2$ is a NTI-abstraction iff it is a TD-abstraction.*
>
> **Proof:** We consider only TI- and NTD-abstractions and one direction as the other proofs are dual. Let $f : \Sigma_1 \Rightarrow \Sigma_2$ be a TI-abstraction. From Theorem 4, if $\neg f(\varphi) \notin NTH(\Sigma_2)$ then $\neg\varphi \notin NTH(\Sigma_1)$. But, from the syntactical completeness of Σ_1, $\neg\varphi \notin NTH(\Sigma_1)$ iff $\varphi \in NTH(\Sigma_1)$. Similarly, from the syntactical completeness of Σ_2, $\neg f(\varphi) \notin NTH(\Sigma_2)$ iff $f(\varphi) \in NTH(\Sigma_2)$. Thus $f(\varphi) \in NTH(\Sigma_2)$ implies $\varphi \in NTH(\Sigma_1)$. Hence $f : \Sigma_1 \Rightarrow \Sigma_2$ is a NTD-abstraction. $\Box$

Of course, this theorem does not hold if one of the formal systems is syntactically incomplete. For example, an NTI-abstraction in which the abstract space, Σ_2 is syntactically complete and the ground space, Σ_1 is syntactically incomplete can map the members of TH(Σ_1), and formulae which belong neither to TH(Σ_1) nor to NTH(Σ_1) onto TH(Σ_2). A TD-abstraction, by comparison, can map only the members of TH(Σ_1) into TH(Σ_2). For abstractions from syntactically incomplete systems, the type of abstraction determines where the formulae which neither belong to TH(Σ_1) nor to NTH(Σ_1) can be mapped. Analogous results hold for the other types of abstractions.

3.5 A Duality Relation

Often in our proofs we have called upon a **duality** between T*-abstractions and NT*-abstractions. Given some theorem about T*-abstractions, there has been a dual theorem about NT*-abstractions. We capture this fact with the following duality theorem.

We informally define $\mathcal{MT}_0$ as the class of meta-theoretic statements about abstractions which mention only the notions of:

> { equality, $\circ$, identity, inverse, TI, TD, TC, NTI, NTD, NTC, T*, NT*, TI*, TD*, TC*, TH, NTH, $TH*$, theorem, negation of theorem, syntactic completeness, system with negation, negation preserving, surjectivity, injectivity }

This set consists of all the notions we have introduced so far which respect the duality between T*- and NT*-abstractions. For example, the statement, s_1 that "A TI-abstraction between two syntactically complete systems with negation is NTD" is in $\mathcal{MT}_0$. However, the statement, s_2 that "A TD-abstraction cannot map into an inconsistent abstract theory" is not in $\mathcal{MT}_0$ as it mentions the notion of "inconsistency", and there is not an exact dual to the notion of inconsistency. Given a statement s in $\mathcal{MT}_0$, we define the **dual to** s as the statement formed by simultaneously applying the substitutions:

> { TI/NTI, TD/NTD, TC/NTC, NTI/TI, NTD/TD, NTC/TC, T*/NT*,

NT*/T*, TH/NTH, NTH/TH, theorem/negation of theorem, negation of theorem/theorem }

We call this set the **duality substitutions for $\mathcal{MT}_0$**. The dual to a statement in $\mathcal{MT}_0$ is itself in $\mathcal{MT}_0$. For example, the dual to s_1 is the statement that "A NTI-abstraction between two syntactically complete systems with negation is TD". The following very useful meta-theorem holds.

Meta-theorem 1 (Duality) *: If s is a statement from $\mathcal{MT}_0$ then s is a theorem iff the dual to s is a theorem.*

Proof:[Informal] The definitions of all the concepts in $\mathcal{MT}_0$ respect the duality substitution exactly. For example, the dual of the definition of a TD-abstraction is the definition of a NTD-abstraction. Thus any proof can be mapped onto a proof of the dual theorem simply by applying the duality substitutions for $\mathcal{MT}_0$ to the proof. □

For example, since s_1 is a theorem, the dual to s_1 is also a theorem. A NTI-abstraction between two syntactically complete theories with negation is indeed TD. Having proved a statement in $\mathcal{MT}_0$, we know immediately that its dual is also true. This can considerably reduce the number and the size of proofs we need find. This duality theorem is only true for statements in $\mathcal{MT}_0$. For example, s_2, the statement that "A TD-abstraction cannot map into an inconsistent abstract theory" is true but not in $\mathcal{MT}_0$. If we apply the duality substitution for $\mathcal{MT}_0$ to s_2, we get the statement that "A TI-abstraction cannot map into an inconsistent abstract theory". This is false. Indeed, Chapter 5 is devoted solely to the understanding why the duality breaks down for this statement, and how we can overcome this problem.

3.6 An Ordering Relation

Another very important relation between abstractions is that of an order. We have called one abstraction "stronger" than another without precisely defining how we might order abstractions. Such an order would help us describe the complexity of the abstract spaces. The definitions of T*- and NT*-abstractions suggest two obvious ways of ordering abstractions: T*-abstractions can be ordered by the number of theorems in the abstract spaces, and NT*-abstractions by the number of wffs whose negations are theorems. These orderings will prove very useful in Chapter 5 when we consider the problem of abstractions which map into (absolutely) inconsistent spaces; that is, when the number of theorems of the abstract space is maximal.

The orders we define are only partial orders; given an arbitrary pair of abstractions it is not always possible to put an order on them. To order two abstractions, a necessary condition is that they share the same ground space. However, this is not a sufficient condition as we cannot order all abstractions with the same ground space. For T*-abstractions, we define the following order:

Definition 13 ($\leq$) *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ and $g : \Sigma_1 \Rightarrow \Sigma_3$ are two abstractions then $f \leq g$ iff for all wffs φ, if $f(\varphi) \in TH(\Sigma_2)$ then $g(\varphi) \in TH(\Sigma_3)$. We say that g is* **stronger** *than f, or that f is* **weaker** *than g.*

g is "stronger" than f in the sense that there are more wffs, α such that $g(\alpha) \in TH(\Sigma_3)$ than wffs, β such that $f(\beta) \in TH(\Sigma_2)$. Note that we can order two abstractions even if they have completely different abstract languages. We also introduce three derived symbols:

Definition 14 ($\equiv$) *: $f \equiv g$ iff $f \leq g$ and $g \leq f$. We say that f is* **equivalent** *to g.*

Definition 15 ($<$) *: $f < g$ iff $f \leq g$ and $\neg(f \equiv g)$. We say that g is* **strictly stronger** *than f, or that f is* **strictly weaker** *than g.*

Definition 16 ($\bowtie$) *: $f \bowtie g$ iff $\neg(f \leq g)$ and $\neg(g \leq f)$. We say that f is* **incomparable** *to g.*

" $\leq$ " is a preorder, being transitive, and reflexive.

Theorem 14 (Preorder) *:*

1. $f \leq g$ *and* $g \leq h$ *implies* $f \leq h$
2. $f \leq f$

Proof: Immediate from Definitions 13 and 14. □

" $\equiv$ " is defined by the antisymmetry of " $\leq$ ". Alternatively, we could have defined " $\equiv$ " using the following identity:

Theorem 15 *: If* $f : \Sigma_1 \Rightarrow \Sigma_2$ *and* $g : \Sigma_1 \Rightarrow \Sigma_3$ *are abstractions then* $f \equiv g$ *iff* $f(\varphi) \in TH(\Sigma_2)$ *implies and is implied by* $g(\varphi) \in TH(\Sigma_3)$.

Proof: Immediate from Definitions 13 and 14. □

" $\equiv$ " itself is a normal equivalence relation, being transitive, symmetric and reflexive.

Theorem 16 (Equivalence relation) *:*

1. $f \equiv g$ *and* $g \equiv h$ *implies* $f \equiv h$
2. $f \equiv g$ *implies* $g \equiv f$
3. $f \equiv f$

Proof: Immediate from Definition 14 and Theorem 14. □

Note that equivalence is a weaker notion than equality of abstractions. Equality states that the abstractions share the same abstract spaces and map wffs identically (in effect, that the abstractions are the same object). Equivalence states that the two abstract theories are isomorphic in some sense (wffs which abstract onto theorems in one abstract theory also abstract onto theorems in the other). Equality implies equivalence, but not vice versa.

Theorem 17 : *$f = g$ implies $f \equiv g$*

Proof: Immediate from Definition 14. □

"$<$" is a strict partial order, being transitive and irreflexive.

Theorem 18 (Strict partial order) :

1. *$f < g$ and $g < h$ implies $f < h$*
2. *$\neg(f < f)$*

Proof: Immediate from Definitions 15 and 14 and Theorem 14. □

Those abstractions which we cannot order are incomparable.

Theorem 19 (Incomparability relation) : *One and only one of $f < g$, $g < f$, $f \equiv g$ and $f \bowtie g$ holds.*

Proof: Immediate from Definitions 13, 14, 15 and 16. □

We can extend "$\leq$" in the conventional way from a preorder to a weak partial order, "$\leq^*$" on the equivalence classes of $\mathcal{ABS}$ with respect to "$\equiv$"; that is, we can define a weak partial order on $\mathcal{ABS}/\equiv$ as follows:

Definition 17 ($\leq^*$) : *If $[f]$ and $[g]$ belong to the quotient set $\mathcal{ABS}/\equiv$ then $[f] \leq^* [g]$ iff $f \leq g$*

Theorem 20 (Weak partial order) :

1. *$[f] \leq^* [g]$ and $[g] \leq^* [h]$ implies $[f] \leq^* [h]$*
2. *$[f] \leq^* [g]$ and $[g] \leq^* [f]$ implies $[f] = [g]$*
3. *$[f] \leq^* [f]$*

Proof: Immediate from Definitions 13, 14 and 17. □

3.7 Properties of this Ordering

We have argued at length for abstractions which preserve provability. Thus we are interested in the relationship between this partial order and the preservation of provability. A very important property is that the provability preserving properties of an abstraction are inherited by another if we can order them.

Theorem 21 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TI-abstraction (TD-abstraction), and $g : \Sigma_1 \Rightarrow \Sigma_3$ is another abstraction with $f \leq g$ $(g \leq f)$ then g is also TI (TD).*

Proof: We give the proof for TI-abstractions; the proof for TD-abstractions is entirely dual. If $f : \Sigma_1 \Rightarrow \Sigma_2$ is TI then $\varphi \in TH(\Sigma_1)$ implies $f(\varphi) \in TH(\Sigma_2)$. As $f \leq g$, $f(\varphi) \in TH(\Sigma_2)$ implies $g(\varphi) \in TH(\Sigma_3)$. Thus $\varphi \in TH(\Sigma_1)$ implies $g(\varphi) \in TH(\Sigma_3)$. That is, $g : \Sigma_1 \Rightarrow \Sigma_3$ is TI. □

Additionally any TI-abstraction is stronger than any TD-abstraction with the same ground space.

Theorem 22 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TI-abstraction and $g : \Sigma_1 \Rightarrow \Sigma_3$ is a TD-abstraction then $g \leq f$.*

Proof: Since $g : \Sigma_1 \Rightarrow \Sigma_3$ is TD, $g(\varphi) \in TH(\Sigma_3)$ implies $\varphi \in TH(\Sigma_1)$. But, as $f : \Sigma_1 \Rightarrow \Sigma_2$ is TI, $\varphi \in TH(\Sigma_1)$ implies $f(\varphi) \in TH(\Sigma_2)$. Thus, $g(\varphi) \in TH(\Sigma_3)$ implies $f(\varphi) \in TH(\Sigma_2)$. That is, $g \leq f$. □

A simple corollary of this theorem is that $g \leq f_1 \leq f$ where f_1 is the identity abstraction of Σ_1. Another is that a TC-abstraction is equivalent to the identity abstraction of its ground space. Thus "$\leq$" generates orders with chains of TD-abstractions on the left, TC-abstractions in the middle, and chains of TI-abstractions on the right. Given a set of ordered abstractions, if one of the abstractions is TI then all stronger abstractions are also TI, and if one of the abstractions is TD then all weaker abstractions are also TD.

Composition is a very natural way to build such ordered sets of T*-abstractions as the following result holds.

Theorem 23 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is an abstraction and $g : \Sigma_2 \Rightarrow \Sigma_3$ is a TI-abstraction (TD-abstraction) then $f \leq f \circ g$ ($f \circ g \leq f$).*

Proof: As usual the proof for TD-abstractions is dual to the proof for TI-abstractions. If $f(\varphi) \in TH(\Sigma_2)$ and $g : \Sigma_2 \Rightarrow \Sigma_3$ is TI then $g(f(\varphi)) \in TH(\Sigma_3)$. Thus $f(\varphi) \in TH(\Sigma_2)$ implies $f \circ g(\varphi) \in TH(\Sigma_3)$. That is, $f \leq f \circ g$. $\square$

These results suggest that "$\leq$" is a very natural order on T*-abstractions. The quotient set $\mathcal{ABS}/\!\equiv$ together with "$\leq^*$" form a **poset**; that is, a set plus a partial ordering. As with any poset, certain subsets can have upper and lower bounds. For example the subset, $\mathcal{ABS}_\Sigma/\!\equiv$ in which all abstractions share the same ground space, Σ has an upper bound which is the equivalence class of all abstractions from Σ with absolutely inconsistent abstract spaces. $\mathcal{ABS}_\Sigma/\!\equiv$ also has a lower bound which is the equivalence class of all abstractions from Σ with abstract spaces which have no theorems. Note that $\langle \mathcal{ABS}/\!\equiv, \leq^* \rangle$ does not form a **lattice**; that is, a poset in which any two elements have a least upper bound and a greatest lower bound. However, when we restrict the set of abstractions to those with the same ground space we get a **complete lattice**; that is, a lattice in which every subset has a least upper bound and a greatest lower bound.

Theorem 24 *: $\langle \mathcal{ABS}_\Sigma/\!\equiv, \leq^* \rangle$ is a complete lattice.*

Proof: We use the fact that a poset with a greatest element in which every (non-empty) subset has a greatest lower bound is a complete lattice.

We define the abstraction, $f : \Sigma \Rightarrow \Sigma_1$ in which the abstract space, Σ_1 is absolutely inconsistent, and the mapping function is an identity function. The equivalence class of this abstraction is the greatest element of the set $\mathcal{ABS}_\Sigma/\!\equiv$. Consider any subset, A of $\mathcal{ABS}_\Sigma/\!\equiv$. The set of upper bounds, B of this subset is non-empty as it contains the equivalence class of $f : \Sigma \Rightarrow \Sigma_1$. Consider the abstraction, $g : \Sigma \Rightarrow \Sigma_2$ with an identity mapping function and with abstract space given by $\alpha \in TH(\Sigma_2)$ iff for all abstractions, $h : \Sigma \Rightarrow \Sigma_3$ in the subset B, $h(\alpha) \in TH(\Sigma_3)$. The equivalence class of this abstraction is the greatest lowest bound of B. It is also the least upper bound of A. The greatest lower bound exists by an analogous argument. Every subset of $\mathcal{ABS}_\Sigma/\!\equiv$ therefore has a least upper bound and a greatest lower bound, and $\langle \mathcal{ABS}_\Sigma/\!\equiv, \leq^* \rangle$ forms a complete lattice. □

3.8 A Dual Ordering

In the last section, we considered an ordering on the number of theorems of the abstract space. This gave an ordering suitable for comparing T*-abstractions. We can also define a dual ordering on the number of wffs whose negations are theorems of the abstract space. Such an ordering is suitable for comparing NT*-abstractions.

Definition 18 ($\preceq$) *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ and $g : \Sigma_1 \Rightarrow \Sigma_3$ are two abstractions then $f \preceq g$ iff for all wffs φ, if $f(\varphi) \in NTH(\Sigma_2)$ then $g(\varphi) \in NTH(\Sigma_3)$.*

We also define an equivalence, "$\cong$", a strict order, "$\prec$", an incomparability relation, "$\asymp$" and a weak partial order, "$\preceq^*$" identically to those relations associated with "$\leq$". The properties of $\{\preceq, \cong, \prec, \asymp \preceq^*\}$ are entirely dual to those of $\{\leq, \equiv, <, \bowtie, \leq^*\}$. Rather than consider this ordering in any more detail, we will simply extend the duality theorem (Meta-theorem 1) to include both "$\leq$" and "$\preceq$". Since the properties of "$\preceq$" are entirely dual to those of "$\leq$", everything that we have shown for "$\leq$" will hold dually for "$\preceq$".

We informally define $\mathcal{MT}_1$ as the class of meta-theoretic statements about abstractions which mention only those notions found in $\mathcal{MT}_0$ and the set:

> $\{ \leq, \equiv, <, \bowtie, \leq^* \preceq, \cong, \prec, \asymp, \preceq^*, \mathcal{ABS}, \mathcal{ABS}_\Sigma,$ preorder, strict partial order, equivalence relation, weak partial order, quotient set, equivalence class, poset, (complete) lattice, (greatest) lower bound, (least) upper bound $\}$

For example, the statement, s_3 that "$\langle \mathcal{ABS}_\Sigma / \equiv, \leq^* \rangle$ is a complete lattice " is in $\mathcal{MT}_1$. The set of sentences in $\mathcal{MT}_0$ is a subset of those in $\mathcal{MT}_1$. Given a statement s in $\mathcal{MT}_1$, we define the **dual to** s as the statement formed by simultaneously applying both the duality substitutions for $\mathcal{MT}_0$ and the substitutions:

$$\{ \leq / \preceq, \equiv / \cong, < / \prec, \bowtie / \asymp, \leq^* / \preceq^*, \preceq / \leq, \cong / \equiv, \prec / <, \asymp / \bowtie, \preceq^* / \leq^* \}$$

The dual to a statement in $\mathcal{MT}_1$ is itself in $\mathcal{MT}_1$. For example, the dual to s_3 is the statement that "$\langle \mathcal{ABS}_\Sigma / \cong, \preceq^* \rangle$ is a complete lattice". The following extended duality theorem holds:

> **Meta-theorem 2 (Extended duality)** *: If s is a statement from $\mathcal{MT}_1$ then s is a theorem iff the dual to s is a theorem.*
>
> **Proof:**[Informal] The definitions of all the concepts in $\mathcal{MT}_1$ respect the duality substitution exactly. For example, the dual of the definition of "$<$" is the definition of "$\prec$". Thus any proof can be mapped onto a proof of the dual theorem simply by applying the duality substitutions for $\mathcal{MT}_1$ to the proof. $\square$

For example, since s_3 is a theorem, the dual to s_3 is also a theorem. That is, $\langle \mathcal{ABS}_\Sigma / \cong, \preceq^* \rangle$ is indeed a complete lattice. "$\preceq$" thus inherits all the properties for orderings of NT*-abstractions we have proved of "$\leq$" for orderings of T*-abstractions.

Both these orderings also possess another duality which is orthogonal to this one. Given a true statement about the ordering of TI*-abstractions there exists a dual and true statement about the inverted ordering of TD*-abstractions. We informally define $\mathcal{MT}_2$ as the subset of $\mathcal{MT}_1$ sentences which discuss just the ordering of T*- or NT*-abstractions. For example, the statement, s_4 that "If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TI-abstraction, and $g : \Sigma_1 \Rightarrow \Sigma_1$ is the identity abstraction of Σ_1 then $g \leq f$" is in $\mathcal{MT}_2$ as both f and g are T*-abstractions. However, the statement, s_2 that "A TD-abstraction cannot map into an inconsistent abstract theory" is not in $\mathcal{MT}_2$ as it is not about either ordering. Given a statement s in $\mathcal{MT}_2$, we define the **inverted dual** to s as the statement formed by simultaneously applying the substitutions:

> { $X \leq Y / Y \leq X$, $X < Y / Y < X$, $X \preceq Y / Y \preceq X$, $X \prec Y / Y \prec X$, TI/TD, TD/TI, NTI/NTD, NTD/NTI }

We use "X" and "Y" to stand for meta-variables that can be substituted for any abstraction. "X" and "Y" can unify with different abstractions in the different substitutions. The inverted dual to a statement in $\mathcal{MT}_2$ is itself in $\mathcal{MT}_2$. For example, the inverted dual to s_4 is the statement that "If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TD-abstraction, and $g : \Sigma_1 \Rightarrow \Sigma_1$ is the identity abstraction of Σ_1 then $f \leq g$". The following duality theorem holds:

> **Meta-theorem 3 (Inverted duality)** *: If s is a statement from $\mathcal{MT}_2$ then s is a theorem iff the inverted dual to s is a theorem.*
>
> **Proof:**[Informal] The idea is to take a proof about the ordering of a T*-abstraction (NT*-abstraction), and invert the chain of reasoning within it. For example, the implication for a TI-abstraction that $\varphi \in TH(\Sigma_1)$ implies $f(\varphi) \in TH(\Sigma_2)$ can be inverted if we replace the TI-abstraction with a TD-abstraction in which $f(\varphi) \in TH(\Sigma_2)$ implies $\varphi \in TH(\Sigma_1)$. We also need to invert the ordering of abstractions. For instance, $f \leq g$ will become $g \leq f$. $\Box$

For example, since s_4 is a theorem, the inverted dual to s_4 is also a theorem. That is, if $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TD-abstraction, and $g : \Sigma_1 \Rightarrow \Sigma_1$ is the identity abstraction of Σ_1 then $f \leq g$ is indeed true. This inverted duality theorem also considerably reduces the number and the size of proofs we need to find.

Finally, we note that under very special conditions, the two orders actually coincide. For a syntactically complete theory, Σ either $\alpha \in TH(\Sigma)$ or $\alpha \in NTH(\Sigma)$; the two orders will therefore collapse together.

Theorem 25 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ and $g : \Sigma_1 \Rightarrow \Sigma_3$ are abstractions with syntactically complete abstract theories then $f \preceq g$ iff $g \leq f$.*

Proof: We only consider the forward direction as the reverse direction is entirely dual. Since $f \preceq g$, $f(\varphi) \in NTH(\Sigma_2)$ implies $g(\varphi) \in NTH(\Sigma_3)$. Thus $g(\varphi) \notin NTH(\Sigma_3)$ implies $f(\varphi) \notin NTH(\Sigma_2)$. That is, $g(\varphi) \in TH(\Sigma_3)$ implies $f(\varphi) \in TH(\Sigma_2)$. Hence, $g \leq f$. $\Box$

3.9 Hierarchies of Abstractions

So far we have concentrated on applying abstraction only once. However, the process of abstracting a problem can easily be iterated to give a hierarchy of abstractions. Theorem 23 shows that abstraction composition is a very natural way to generate hierarchies of abstractions of increasing strength. An important question is whether there is any limit to the size of the hierarchies of abstractions we can build. How many times can we abstract a problem? Can we get to an abstract space such that abstracting it again generates the same abstract space (or more generally, one with the same number of theorems)? In other words, does abstraction composition have a fixed point? And if we get there, will we know? There are two possible answers. We shall concentrate on TI-abstractions though similar observations hold for NTI-abstractions.

In the first case, the abstract space becomes inconsistent. In fact, as we increase the strength of the abstraction, we monotonically increase the number of theorems; we eventually reach an upper bound when the set of theorems coincides with the language. Since testing consistency is in general undecidable, we may not be able to recognise that we are at this fixed point. Thus, in trying to generate simpler and simpler abstract spaces, it is actually possible to generate indefinitely long chains of abstractions. Putting an upper bound on the number of abstractions we apply is one solution. A better solution perhaps is to require that there is a point after which all the abstract spaces are decidable. This topic is discussed in much more detail in Chapter 5.

In the second case, the abstract space remains consistent but abstracting it further does not increase the number of theorems. This occurs with most types of abstraction if we apply them repeatedly enough. In order to decrease complexity, an abstraction throws away some details; eventually there comes a point after which there are no more details to throw away. For example, with an abstraction that maps the names of predicates together, we cannot abstract a problem beyond a single predicate symbol. The fixed point is different for each different type of abstraction but we are, however, usually able to recognise when we are at such a fixed point.

3.10 Some Properties of Abstractions

Having concentrated on properties that exist between *different* abstractions, we now consider some properties possessed by *individual* abstractions. We have defined an abstraction as a pair of formal systems and a mapping function. Often the parts of this definition are closely related. The aim of this section is to identify different ways in which they are related. The definitions we introduce here will prove very useful in describing the examples of abstractions presented in Chapter 4.

The definitions of T*- and NT*-abstractions capture some important relations between one part of the formal systems; that is, between the theorems of the ground and abstract spaces. The properties we introduce here capture relations between other parts of the formal systems. They divide into three classes: relations between the ground and abstract languages, relations between the ground and abstract axioms, and relations between the ground and abstract inference rules.

Relations between Languages

Often an abstraction will use the same language in the abstract space as in the ground space. This can allow the same inference engine to be used in the abstract space as in the ground space, providing a great economy in implementation and permitting the use of hierarchies of abstractions. An abstraction in which the language of the abstract space is a subset of that of the ground space is called **Λ-invariant**.

> **Definition 19 (Λ-invariance)** *: If Σ_1 has language Λ_1, and Σ_2 has language Λ_2 then an abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ is called* **Λ-invariant** *iff $\Lambda_2 \subseteq \Lambda_1$.*

Λ-invariance is, in fact, an over-sufficient condition for the same inference engine to be usable in both the abstract and ground spaces. In general, we just need the language of the abstract space to share the same logical syntax as that of the ground space. For simplicity, we have taken the language to be the set of well formed formulae. More commonly a language is given by an alphabet and a set of rules. For example, a first order language can be given by an alphabet, a set of recursive rules applied to the alphabet defining the well formed terms, and a set of recursive rules applied to the alphabet and the well formed terms defining the well formed formulae. In formulae, $\Lambda = \langle \mathcal{A}, \mathcal{T}, \mathcal{W} \rangle$. The alphabet consists of a set of logical symbols (the connectives and quantifiers), a set of variables, a set of function symbols, and a set of predicate symbols. In formulae, $\mathcal{A} = \langle \mathcal{L}, \mathcal{V}, \mathcal{F}, \mathcal{P} \rangle$. Two languages will share the same logical syntax if they have the same logical symbols and the same rules for defining well formed terms and well formed formulae; they can differ in the symbols used for variables, for functions, and for predicates. Thus, we can capture the notion of an abstraction which preserves the logical syntax of the language by the following definition:

Definition 20 (Syntax-invariance) *: Let* $\Lambda_1 = \langle \mathcal{A}_1, \mathcal{T}_1, \mathcal{W}_1 \rangle$, $\Lambda_2 = \langle \mathcal{A}_2, \mathcal{T}_2, \mathcal{W}_2 \rangle$ *be two languages, with alphabets* $\mathcal{A}_1 = \langle \mathcal{L}_1, \mathcal{V}_1, \mathcal{F}_1, \mathcal{P}_1 \rangle$, $\mathcal{A}_2 = \langle \mathcal{L}_2, \mathcal{V}_2, \mathcal{F}_2, \mathcal{P}_2 \rangle$. *An abstraction,* $f : \Sigma_1 \Rightarrow \Sigma_2$ *is called* **syntax-invariant** *iff* $\mathcal{L}_2 \subseteq \mathcal{L}_1$, $\mathcal{T}_2 \subseteq \mathcal{T}_2$, *and* $\mathcal{W}_2 \subseteq \mathcal{W}_1$.

This definition is limited to theories with languages like that of first order logic which can be described by an alphabet and rules for defining the well formed terms and well formed formulae. It would not, however, be too difficult to generalise this definition to languages with a different type of logical syntax (*eg.* the languages of modal or temporal logic).

An important class of syntax-invariant abstractions is those whose mapping functions abstract only the theory; such abstractions leave the logical syntax of formulae unchanged, mapping just the atomic wffs. We call these **theory abstractions**.

Definition 21 (Theory abstraction) *: An abstraction* $f : \Sigma_1 \Rightarrow \Sigma_2$ *is a* **theory abstraction** *iff for any wffs* α *and* β,

$$f(\alpha \circ \beta) = f(\alpha) \circ f(\beta)$$

for all the logical connectives, $\circ$ *(with the analogous condition for unary and n-ary connectives) and*

$$f(\diamond x.\alpha) = \diamond x.f(\alpha)$$

for all the quantifiers, $\diamond$.

For example, a theory abstraction would map the ground wff "$\forall x.p(x) \vee \neg p(x)$" onto the abstract wff "$\forall x.f(p(x)) \vee \neg f(p(x))$". Again this definition is limited to theories with languages like that of first order logic which have logical connectives and quantifiers. It would not, however, be too difficult to generalise this definition to languages with a different type of logical syntax. Note that both syntax-invariant and theory abstractions overload the use of the logical symbols; the same connective and quantifier symbols are used in the ground and the abstract languages. This is usually a reflection of the similar meanings given to these symbols in the ground and abstract spaces. Note also that a theory abstraction can drop variables. For example, if "$f(p)$" does not mention "x" then "$\forall x.f(p)$" is equivalent to "$f(p)$". It can also drop conjuncts and disjuncts. For example, if "$f(q)$" maps onto "$\perp$" then "$f(p \vee q)$" is equivalent to "$f(p)$". The idea behind theory abstractions is that most useful abstractions abstract the theory but preserve the logic. In general, the logic is well behaved and it is the theory that needs to be simplified. Indeed, you should only change the logical structure of a wff with great care as the consistency of a logic is often very finely balanced and even the smallest change can prove catastrophic. Chapter 5 explores this delicate balance in more detail. As theory abstractions preserve the logical structure of formulae, they are often use the same inference rules in both the ground and the abstract spaces; we capture this property in Definition 23.

Relations between Axioms

Often the axioms of the abstract space are related to the axioms of the ground space. Sometimes the axioms are just mapped in the same way as the language. The following definition characterises such a situation.

> **Definition 22 (Λ/Ω-invariance)** *: Let $\Sigma_1 = \langle \Lambda_1, \Omega_1, \Delta_1 \rangle$, $\Sigma_2 = \langle \Lambda_2, \Omega_2, \Delta_2 \rangle$ be two formal systems. An abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ is* **Λ/Ω-invariant** *iff $\Omega_2 = f(\Omega_1)$.*

Λ/Ω-invariant abstractions are useful when we do not want to distinguish between wffs and axioms. They are especially common when the axioms of the ground space are not known in advance. TI-abstractions are frequently Λ/Ω-invariant since this is a simple way of guaranteeing that the abstraction of the axioms of the ground space remain (as they must) theorems of the abstract space.

Relations between Inference Rules

For reasons of simplicity and economy, an abstraction often uses the same inference rules in the abstract space as in the ground space. This is characterised by the following definition.

Definition 23 (Δ-invariance) *: Let* $\Sigma_1 = \langle \Lambda_1, \Omega_1, \Delta_1 \rangle$*, and* $\Sigma_2 = \langle \Lambda_2, \Omega_2, \Delta_2 \rangle$ *be two formal systems. An abstraction,* $f : \Sigma_1 \Rightarrow \Sigma_2$ *is* **Δ-invariant** *iff* $\Delta_2 \subseteq \Delta_1$.

If an abstraction is not Δ-invariant then we call it Δ-variant. Δ-invariant abstractions use the same inference engine in both the ground and the abstract spaces. As well as providing considerable economy in implementation, this simplifies the problem of mapping abstract proofs back onto ground proofs since the abstract proof will often resemble (parts of) the ground proof; we will return to this topic in Chapter 7. Notice that $\Delta_2 \subseteq \Delta_1$ may hold even if the languages, Λ_1 and Λ_2 are different provided that they have the same logical syntax; that is, provided the abstraction is syntax-invariant. The variables used in defining the inference rules are meta-variables and usually only commit us to a particular *logical* syntax for our object language. Δ-invariance is frequently associated with syntax-invariance as the latter guarantees that the logical syntax is preserved whilst the former guarantees that the logical semantics are in some way preserved.

Not all abstractions are Δ-invariant since we sometimes want to change the inference rules. Occasionally an abstraction will map the inference rules in the same way as the language.

Definition 24 (Λ/Δ-invariance) *: Let* $\Sigma_1 = \langle \Lambda_1, \Omega_1, \Delta_1 \rangle$ *and* $\Sigma_2 = \langle \Lambda_2, \Omega_2, \Delta_2 \rangle$ *be two formal systems. An abstraction* $f : \Sigma_1 \Rightarrow \Sigma_2$ *is* **Λ/Δ-invariant** *iff*

$$\Delta_2 = \{ \frac{f(\alpha_1), ..., f(\alpha_n)}{f(\alpha_{n+1})} \quad | \quad \frac{\alpha_1, ..., \alpha_n}{\alpha_{n+1}} \in \Delta_1 \}$$

"$f(\alpha_j)$" should be read as applying f to the wff substituted for α_j when the inference rule is applied. As stated earlier, we will ignore the difficult problem of providing a uniform and general representation for inference rules by simply adopting the notation used by Prawitz [Pra65]. Unless it is explicitly stated to the contrary, we assume that the side conditions associated with the application of an inference rule remain true in the abstract space if they are true in the ground space. Additionally, we shall assume that the same assumptions are discharged in the abstract space as in the ground space. Λ/Δ-invariant abstractions provide a direct connection between the rules applied in the abstract space and those in the ground space. This can make the problem of mapping an abstract proof back onto a ground proof easier.

An abstraction can possess several of these properties. One common combination is when both the axioms and the inference rules are mapped in the same way as the language. We call such abstractions **Σ-invariant**.

> **Definition 25 (Σ-invariance)** *: An abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ is* **Σ-invariant** *iff it is Λ/Ω-invariant and Λ/Δ-invariant.*

In [Sim88, Sim89] **veridicality**, a notion very similar to Σ-invariance for state space search, was claimed to be *fundamental* for an abstraction to make any sense at all. This seems a difficult statement to justify since we may want to allow the abstraction of the conclusion of an inference rule to be deducible in *several* steps from the abstraction of the hypotheses. Σ-invariance requires that the abstraction of the conclusion is an *immediate* consequence of the abstraction of the hypotheses.

3.11 Summary

We have explored some of the consequences of our theory of abstraction. We have considered both properties between different abstractions, and properties of an individual abstraction. For the former, we have defined various operations on abstractions, and relations between them. For the latter, we have defined various relations between the different parts of the abstraction mapping. The concepts introduced here will prove very useful in later Chapters for describing abstractions, and for understanding how and why abstraction works.

Chapter 4

Examples of Abstractions

In this Chapter, we will describe some important and, in some cases, famous examples of abstraction. Each is described formally using our theory of abstraction. In each example, we observe many of the properties defined in the last Chapter. We end by identifying several connections between what initially appeared to be very dissimilar abstractions.

4.1 Introduction

The examples of abstraction described in this Chapter are taken from many areas: planning, theorem proving, commonsense reasoning, formal methods, *etc.* In most cases, the original description of the abstraction was informal; to fit these examples into our framework we have therefore performed a reconstruction which, we hope, is faithful to the original description. We view this reconstruction as one of the most important contributions of our theory of abstraction. Indeed, our understanding and definition of abstraction has been strongly influenced by these examples.

This Chapter has many goals. First, we hope it will convince the reader that

Many of the examples in this Chapter are also described in [GW89a], [GW90a] and [GW90b].

our framework is very powerful and can capture most previous and, at first sight, unrelated work in abstraction. Second, it provides a unified view of work from many different areas which was carried out with a great variety of goals. Third, it supports our claim that work in automating reasoning can often be fruitfully characterised in terms of the provability relation. Finally, as we believe that the sample of abstractions described here is representative of abstractions as a whole, it highlights the simple properties, like truthfulness, possessed by most abstractions used in past.

The structure of this Chapter is as follows. We begin by describing some historically important examples of abstraction. The remaining examples are divided into four classes: propositional abstractions (in which the abstract space is propositional), domain abstractions (in which constants from the domain are mapped together), predicate abstractions (in which the names of predicates are mapped together) and finally some abstractions used in formal methods (*eg.* hardware and software verification). We conclude with a brief summary of the properties of each of the examples.

4.2 Historical examples

Example 1 (ABSTRIPS):

One of the very first and arguably one of the most famous AI systems to use abstraction was ABSTRIPS [Sac74]. This system operated in a STRIPS planning domain in which operators are applied to states of the world, generating new states. ABSTRIPS built abstract plans in a hierarchy of abstract spaces by ignoring certain preconditions to operators; to refine these abstract plans into ground plans, further operators might need to be applied to satisfy the abstracted preconditions.

To put this into a theorem proving context, we follow Green [Gre69] and adopt a situation calculus. We chose this formalism because it is simple and sufficiently descriptive for our purposes; we do not, however, wish to defend its weaknesses (for example, its difficulty in solving the Frame Problem). The abstraction used in ABSTRIPS can be formalised as a Λ/Ω-invariant abstraction, $f_{AB} : \Sigma_1 \Rightarrow \Sigma_2$ which maps between the situation calculi, Σ_1 and Σ_2. These calculi have first order languages, frame, operator and theory axioms and natural deduction rules of inference. Operators are wffs of the form:

$$\forall s \,.\, \bigwedge_{1 \leq i \leq n} p_i(s) \rightarrow q(f(s))$$

p_i is a precondition, s is a state of the world, f is some action, and q describes the new state of the world, $f(s)$. Preconditions are thrown away according to their **criticality**; this is a measure of how difficult the precondition is to satisfy. The mapping function at the k-th level of abstraction is defined as follows:

1. $f_{AB}(\alpha) = \alpha$ if α is an atomic formula.
2. $f_{AB}(\neg\alpha) = \neg f_{AB}(\alpha)$;
3. $f_{AB}(\alpha \wedge \beta) = f_{AB}(\alpha) \wedge f_{AB}(\beta)$;
4. $f_{AB}(\alpha \vee \beta) = f_{AB}(\alpha) \vee f_{AB}(\beta)$;
5. $f_{AB}(\forall x.\alpha) = \forall x.f_{AB}(\alpha)$;
6. $f_{AB}(\exists x.\alpha) = \exists x.f_{AB}(\alpha)$;
7. $f_{AB}(\alpha \rightarrow \beta) = f_{AB}(\alpha) \rightarrow f_{AB}(\beta)$, provided "$\alpha \rightarrow \beta$" is not an operator;
8. $f_{AB}(\bigwedge_{1 \leq i \leq n} p_i(s) \rightarrow r) = \bigwedge_{i \in crit(k)} p_i(s) \rightarrow r$, for any operator, where $i \in crit(k)$ if the criticality of p_i is greater than k.

For example, consider an operator for an agent y to climb onto an object z at location x:

$$at(z, x, s) \wedge climbable(y, z, s) \rightarrow at(z, x, climb(y, z, s))$$

This might abstract to an operator in which we check that the object z is at location x but we do not check whether the agent y can actually climb it:

$$at(z, x, s) \rightarrow at(z, x, climb(y, z, s))$$

If we view the abstracted preconditions as having been mapped onto true, $\top$ then $f_{AB} : \Sigma_1 \Rightarrow \Sigma_2$ is a theory abstraction. It is also a TI/NTI-abstraction.

Theorem 26 *: The abstraction used in ABSTRIPS, $f_{AB} : \Sigma_1 \Rightarrow \Sigma_2$ is a TI/NTI-abstraction.*

Proof: Since $f_{AB} : \Sigma_1 \Rightarrow \Sigma_2$ is a negation preserving mapping between systems with negation, we merely have to prove that it is a TI-abstraction. We do this by showing that given a proof tree Π_1 of φ, you can build a proof tree Π_2 of $f_{AB}(\varphi)$. The proof proceeds by induction on the depth of Π_1; that is the length of the longest branch.

For proofs of depth 1, f_{AB} is applied to the single wff in Π_1; this generates a valid proof in Π_2.

Assume that we have shown it for trees up to depth n. We use $f_{AB}(\Pi)$ to represent the tree in Σ_2 constructed from a tree, Π in Σ_1 of depth n or less. We show that it is true for all proof trees of depth $n + 1$ irrespective of the rule application used to construct the tree of depth $n + 1$ from tree(s) of depth n (or less). Any rule application that is not modus ponens involving an operator translates unmodified. For instance, an or introduction on φ in Π_1 becomes an or introduction on $f_{AB}(\varphi)$ in Π_2. For an operator application, the following transformation is performed:

$$\frac{\dfrac{\Pi}{\bigwedge_{1<i<n} p_i} \qquad \bigwedge_{1<i<n} p_i \to q}{q} \qquad \Longrightarrow$$

$$\frac{\dfrac{f_{AB}\left(\dfrac{\Pi}{\bigwedge_{1\leq i\leq n} p_i}\right)}{\bigwedge_{i\in crit(k)} p_i} \qquad \bigwedge_{i\in crit(k)} p_i \to q}{q}$$

By the induction hypothesis, and the fact that $\bigwedge_{i\in crit(k)} p_i$ follows from $\bigwedge_{1\leq i\leq n} p_i$ by a (possibly empty) sequence of applications of and elimination, this is a valid abstract proof. □

Note that the abstract proof we have constructed is actually *larger* than the ground proof. The purpose of abstraction is not to find such larger proofs; we hope that there are also going to be smaller proofs. These smaller proofs will not try to satisfy p_i for $i \notin crit(k)$. However, there is no guarantee that there will be a smaller proof than the one exhibited; we will always be able to devise an obtuse theory in which to prove p_i for $i \in crit(k)$ we have to prove p_i for $i \notin crit(k)$. This problem would be eliminated if ABSTRIPS had abstracted both left and righthand sides of operators. Under such circumstances, p_i for $i \notin crit(k)$ would not even appear in the abstract language.

An abstract proof should be useful in helping us build a ground proof since it provides the "skeleton" for a ground proof. We will need to add to this skeleton the proofs of preconditions which have been abstracted away. We formalise this idea in Chapter 7 when we describe how the structure of abstract proofs can be used to guide theorem proving in the ground space. Unfortunately, there is no guarantee that we can construct a ground proof which uses the same operators as an abstract proof since adding new operators may undo the effects of old ones. ABSTRIPS works because operators are usually independent; we can therefore solve the preconditions separately. Actually, ABSTRIPS is a little more clever than this; the use of criticalities and multiple levels of abstraction allow for some operator dependence to occur as it restricts the order in which preconditions have to be satisfied. For an abstraction like ABSTRIPS to be useful, the area of the region $TH(\Sigma_2) - f_{AB}(TH(\Sigma_1))$, that is those abstract theorems which don't map back to ground theorems, needs to be small in comparison to $f_{AB}(TH(\Sigma_1))$, those abstract theorems which do.

To summarise:

ABSTRIPS	
Ground space:	situation calculus
Abstract space:	situation calculus
Provability preserving:	TI/NTI
Negation preserving:	yes
Theory abstraction:	yes
Λ/Ω-invariance:	yes
Δ-invariance:	yes
Reference:	[Sac74]

Example 2 (GPS):

Another very early use of abstraction was in the planning method of GPS [NS72]. As with ABSTRIPS, this abstraction guided a state space search. After the objects and operators for a problem have been abstracted, the entire deductive machinery of GPS is used to solve the abstract problem; the abstract solution is then used to construct a plan to guide the solution of the original problem. We will just consider one of the abstractions, $f_{GPS} : \Sigma_1 \Rightarrow \Sigma_2$ suggested for propositional logic problems in [NS72]. This abstractions is Σ-invariant; that is, the language, axioms and deductive machinery of the abstract space are the images of the language, axioms and deductive machinery of the ground space with respect to the mapping function. The ground space, Σ_1 is the propositional calculus of *Principia Mathematica* [WR25]. The abstract space, Σ_2 is a formal system in which the wffs are (nested pairs of) propositional sentence letters. To construct the abstract space, the same mapping is applied to the wffs, the axioms and the premises and consequences of the inference rules of the ground space. If α, β are two formulae in the ground language then:

1. $f_{GPS}(\alpha \vee \beta) = f_{GPS}(\alpha \wedge \beta) = f_{GPS}(\alpha \rightarrow \beta) = \langle f_{GPS}(\alpha), f_{GPS}(\beta)\rangle$;
2. $f_{GPS}(\neg\alpha) = f_{GPS}(\alpha)$;
3. $f_{GPS}(\alpha) = \alpha$ if α is a propositional sentence letter.

For example, $p \vee (\neg q \rightarrow p)$ maps to $\langle p, \langle q, p\rangle\rangle$. This is a TI-abstraction:

Theorem 27 *: $f_{GPS} : \Sigma_1 \Rightarrow \Sigma_2$ is a TI-abstraction.*

Proof: By induction on the depth of the proof. We just take a proof tree Π_1 of φ and apply f_{GPS} to every wff in the tree. This constructs a proof tree of $f_{GPS}(\varphi)$. $\Box$

Note that $f_{GPS} : \Sigma_1 \Rightarrow \Sigma_2$ is not a TD-abstraction. For example, $\langle p, p \rangle \in TH(\Sigma_2)$ but $p \wedge \neg p \notin TH(\Sigma_1)$.

GPS	
Ground space:	propositional calculus, Σ_1
Abstract space:	$f_{GPS}(\Sigma_1)$
Provability preserving:	TI
Negation preserving:	no
Theory abstraction:	no
Λ/Ω-invariance:	yes
Δ-invariance:	no
Reference:	[NS72]

Example 3 (Plaisted's weak and ordinary abstractions):

Closest in spirit to our work is that of Plaisted [Pla80, Pla81]; he defines three classes of abstractions that preserve inconsistency. This work is less general than ours as Plaisted restricts his attention just to mappings between refutation systems that use resolution. Additionally, his classes of abstraction fail to capture all inconsistency preserving mappings between systems that use resolution. We would also claim that our definition of abstraction is more natural since it captures more accurately the properties for which it was defined.

Plaisted's first two classes of abstraction, weak and ordinary abstractions map a set of clauses onto a simpler set of clauses. These abstractions are Λ/Ω-invariant NTI-abstractions (remember we are mapping between refutation systems). However, not all Λ/Ω-invariant NTI-abstractions are weak or ordinary. The advantage of Plaisted's stronger definitions of weak and ordinary abstraction over our definition of NTI-abstractions is that his abstractions are *always* guaranteed to map into simpler theories. Or, to be more precise, for every theorem there is an abstract refutation proof that is no deeper than any of its ground proofs.

With ordinary abstraction, the same mapping function is used to abstract the wffs and the axioms of the ground space onto those of the abstract space. This function maps a clause in the abstract language onto a set of clauses in the abstract language subject to the following conditions:

a) $f(\perp) = \{\perp\}$;

b) if α_3 is a resolvent of α_1 and α_2 in Σ_1, and $\beta_3 \in f(\alpha_3)$ then there exist $\beta_2 \in f(\alpha_2)$ and $\beta_1 \in f(\alpha_1)$ such that a resolvent of β_1 and β_2 subsumes β_3 in Σ_2;

c) if α_1 subsumes α_2 in Σ_1, and $\beta_2 \in f(\alpha_2)$ then there exists $\beta_1 \in f(\alpha_1)$ such that β_1 subsumes β_2 in Σ_2.

Weak abstractions are defined identically to ordinary abstractions except condition **b)** is weakened to the property that if α_3 is a resolvent of α_1 and α_2 in Σ_1, and $\beta_3 \in f(\alpha_3)$ then there exist $\beta_2 \in f(\alpha_2)$ and $\beta_1 \in f(\alpha_1)$ such that β_1 subsumes β_3, **or** β_2 subsumes β_3, **or** a resolvent of β_1 and β_2 subsumes β_3 in Σ_2. Trivially, all ordinary abstractions are weak, but not all weak abstractions are ordinary. Both ordinary and weak abstractions are NTI-abstractions, but, as we said above, not all NTI-abstractions are weak or ordinary.

Theorem 28 : *Weak and ordinary abstractions are NTI.*

Proof: A simple corollary to theorem 2.4 on page 268 of [Pla80] is that, if $f : \Sigma_1 \Rightarrow \Sigma_2$ is a weak or ordinary abstraction, and $\perp$ is derivable from $\Omega \cup \{\varphi\}$ then $f(\perp)$ or something that subsumes it is derivable from $f(\Omega) \cup \{f(\varphi)\}$. As $f(\perp) = \{\perp\}$, this means that $\perp$ is derivable from $f(\Omega) \cup \{f(\varphi)\}$. That is, $\varphi \in NTH(\Sigma_1)$ implies $f(\varphi) \in NTH(\Sigma_2)$. In other words, $f : \Sigma_1 \Rightarrow \Sigma_2$ is a NTI-abstraction. $\Box$

Theorem 29 : *There exist NTI-abstractions between resolution systems that are not weak or ordinary abstractions.*

Proof: We can find NTI-abstractions that fail every one of the conditions in the definition of weak and ordinary abstractions.

Condition **a)** is failed by the NTI-abstraction for which for every φ, $f(\varphi) = \{\varphi \vee \bot\}$.

The problem with condition **b)** is that we may also need to resolve with an axiom of the theory. Consider, for instance, the abstraction defined by $f(p \vee q) = \{p \vee r\}$ and $f(\varphi) = \{\varphi\}$ otherwise. If Σ_1 contains the axioms $\neg q$, and $\neg r$ then f is NTI. In particular, $p \vee q$ resolves with $\neg p$ in Σ_1 to give q. However, no clause in the abstraction of $p \vee q$, or $\neg p$ (or their resolvent) subsumes the clause q found in the abstraction of q. We therefore fail condition **b)**.

For condition **c)**, consider the abstraction defined by $f(p \vee q) = \{r, p \vee q\}$ and $f(\varphi) = \varphi$ otherwise. Now f is NTI. However, f fails condition **c)** of the definition of weak and ordinary abstractions as p subsumes $p \vee q$ but no clause in the abstraction of p subsumes r which is in $f(p \vee q)$. $\Box$

The definition of weak and ordinary abstractions could be extended to overcome the first counter-example by replacing condition **a)** with the more general requirement that there exists $\varphi \in f(\bot)$ such that $\varphi \in NTH(\Sigma_2)$. However, this still leaves useful NTI-abstractions that fail conditions **b)** and **c)**. The purpose of condition **c)** is to guarantee that the composition of two abstractions remains an abstraction (see the proof of theorem 2.3 on page 54 of [Pla81]). This condition seems rather unnatural, especially as it is unnecessary with our definition of abstraction; not only does composing two of our abstractions give a well defined abstraction, but the composition of two TI*-abstractions (TD*-abstractions) is itself a TI*-abstraction (TD*-abstraction).

Even if we restrict ourselves to Λ/Ω-invariant abstractions between resolution systems, weak and ordinary abstractions are not as general as NTI-abstractions. For example, the ABSTRIPS abstraction described earlier in this section is a NTI-abstraction but is not a weak or an ordinary abstraction. Indeed, any purely syntactic definition of abstraction (like Plaisted's definitions of weak and ordinary abstractions) will inevitably fail to capture the complete class of NTI-abstractions as this, in general, requires performing an arbitrary amount of theorem proving.

Plaisted's weak and ordinary abstractions do have one advantage. The proof of theorem 2.4 on page 268 of [Pla80] shows how, given a proof of a theorem in the ground space, we can construct a proof of the abstract theorem; Plaisted notes that this abstract proof is no deeper than the original ground proof. Ordinary and weak abstractions are therefore guaranteed to map us into simpler abstract spaces. The cost of this guarantee is that ordinary or weak abstractions fail to capture the whole class of NTI-abstractions; this seems a high price to pay as we intuitively expect NTI-abstractions that are not also NTC-abstractions to map onto simpler spaces. Indeed there are many NTI-abstractions which are not weak or ordinary, but which build simpler abstract spaces; the abstraction used in ABSTRIPS is one such example.

We end by observing that not all weak or ordinary abstractions are negation preserving. For example, the abstraction given on page 266 of [Pla80] which changes the sign of a set of literals is not negation preserving (and is thus not a theory abstraction); if $f(\neg p) = p$ then $\neg f(p) = \neg\neg p$, and $f(\neg p) \neq \neg f(p)$. Of course, this mapping is logically equivalent to a negation preserving abstraction. We might therefore be tempted to weaken the definition of negation preservation to the property that $f(\neg p) \leftrightarrow \neg f(p)$. The disadvantage with such a change is that this would, in general, make determining the preservation of negation undecidable. It also seems rather unnecessary as every other example of abstraction Plaisted gives in this paper is negation preserving. The preservation of negation is very important as it is closely related to the (in)consistency of our abstract space. We will explore this topic in more detail in Chapter 5.

Weak/Ordinary abstractions	
Ground space:	clauses & resolution
Abstract space:	clauses & resolution
Provability preserving:	NTI
Negation preserving:	sometimes
Theory abstraction:	sometimes
Λ/Ω-invariance:	yes
Δ-invariance:	yes
Reference:	[Pla80]

Example 4 (Plaisted's generalisation abstraction):

Plaisted's third and final class of abstractions use **generalisation functions** [Pla86]. Ordinary and weak abstractions abstract the language and axioms of a theory but keep the deductive machinery the same, what we have called Δ-invariant abstractions and what Plaisted calls **input abstractions**. Abstractions using generalisation functions keep the language and axioms the same but abstract the resolution rule of inference; after every resolution, a generalisation operation is performed on the resolvent. For example, we could replace all terms of depth n or greater by new variables. We thereby construct a proof which is more general than one we could find in the ground space. This abstract proof is guaranteed to be no longer than one of the ground proofs and can be used to aid the search for a ground proof as it has a similar structure. Though the abstract proof can be shorter than the ground proof, the cost of inference in the abstract space is more expensive than that in the ground space. In general we have to perform both full-blown resolution and generalisation; this extra cost could outweigh the advantage of having a shorter proof.

An abstraction using a generalisation function maps a first order calculus using resolution onto another first order calculus with the same axioms but with a "generalised resolution" rule of inference. The identity function is used to map wffs between the two formal systems. Generalised resolution is resolution followed by application of a generalisation function, g to the resolvent; this generalisation function maps a wff, φ onto a set of (more general) wffs that have φ as instances. In other words:

$$g(\varphi) \subseteq \{\alpha \mid \varphi = \alpha\theta\}$$

Note that the generalisation function can map a wff onto the empty set, $\{\}$ which is interpreted as $\perp$. If $g(\varphi) = \{\varphi\}$, the identity generalisation, then Σ_1 will be identical to Σ_2, and the abstraction is TC/NTC. Generalisation abstractions are TI/NTI.

Theorem 30 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a generalisation abstraction for which for any φ, $g(\varphi) \neq \{\}$ then $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TI/NTI-abstraction.*

Proof: A simple corollary to theorem 1 on page 368 of [Pla86] is that if there is a proof of $\perp$ from $\Omega_1 \cup \{\varphi\}$ in Σ_1 then there is a proof of $g(\perp)$ from $\Omega_1 \cup \{\varphi\}$ in Σ_2. Since $g(\perp) = \perp$ for all generalisation functions, this means that $f : \Sigma_1 \Rightarrow \Sigma_2$ is a NTI-abstraction. As $f : \Sigma_1 \Rightarrow \Sigma_2$ is negation preserving, it is also a TI-abstraction. Theorem 1 of [Pla86] also guarantees that the abstract proof is no longer than the ground proof and has the same shape. $\Box$

As generalising a theorem can produce a non-theorem, generalisation abstractions are not usually TD*-abstractions. For example, if we have the generalisation function that replaces all terms by free variables, and the set of axioms, $\Omega_1 = \{p, \neg p \vee q(a), \neg p \vee \neg q(b)\}$ then $\perp \in TH(\Sigma_2)$ but $\perp \notin TH(\Sigma_1)$. We can come up with (extreme) examples that are TD*-abstractions. Consider, for instance an abstraction with a ground space which contains p as an axiom and for which $g(\varphi) = \{\}$ if $\varphi = \neg p$ and $\{\varphi\}$ otherwise. This abstraction is TC/NTC as the generalisation function merely replaces a wff that is false with $\{\}$, which is itself interpreted as $\perp$.

Generalisation abstractions	
Ground space:	clauses & resolution
Abstract space:	clauses & generalised resolution
Provability preserving:	TI/NTI
Negation preserving:	yes
Theory abstraction:	yes
Λ/Ω-invariance:	yes
Δ-invariance:	no
Reference:	[Pla86]

Example 5 (Gazing):

Gazing is a heuristic for controlling the unfolding of definitions and lemmas in theorem proving. It has been used in a natural deduction theorem prover [Plu87], in a connection method theorem prover [War87], and within a proof development system (as a tactic to the Oyster system [Sim89]).

At the heart of gazing is the **common currency model**; to prove a theorem in a complex theory of definitions and lemmas we need to find a common language of concepts between our hypotheses and conclusion. For example, to construct a proof of the sequent,

$$a =_{set} b \ => \ a \subseteq_{set} b$$

we need to find a common currency between the concepts of set equality, "$=_{set}$" and subset, "$\subseteq_{set}$"; we can achieve this by unfolding the definition of "$=_{set}$",

$$a =_{set} b \leftrightarrow (\forall x . x \in a \leftrightarrow x \in b)$$

and that of "$\subseteq_{set}$",

$$a \subseteq_{set} b \leftrightarrow (\forall x . x \in a \rightarrow x \in b)$$

to give a common currency of set membership, "$\in$"; having unfolded these definitions, the proof can be completed by logical inference alone. Gazing constructs a plan of definitions to unfold and lemmas to apply by considering a hierarchy of abstraction spaces: the predicate space and the function/polarity space. Although we will just consider the predicate space, a similar analysis could be made of the function/polarity space.

The abstraction, $f_{gaze} : \Sigma_1 \Rightarrow \Sigma_2$ used in the predicate space can be seen as a mapping from a first order sequent calculus onto a propositional sequent calculus. Definitions and lemmas in both ground and abstract spaces are represented by a set of directed rewrite rules; the direction on these rewrite rules ensure that concepts are only ever expanded into more primitive concepts. The languages of both spaces consists of sequents of the form $A => B$, the wffs in the set A are implicitly conjoined whilst those in B are implicitly disjoined. The mapping function abstracts the sequent $A => B$ onto the abstract sequent $f_{gaze}(A) => f_{gaze}(B)$ where:

1. $f_{gaze}(A) = \bigcup_{\alpha \in A} \ f_{gaze}(\alpha)$;

2. $f_{gaze}(\alpha \vee \beta) = f_{gaze}(\alpha \wedge \beta) = f_{gaze}(\alpha \rightarrow \beta) = f_{gaze}(\alpha \leftrightarrow \beta) = f_{gaze}(\alpha) \cup f_{gaze}(\beta)$;

3. $f_{gaze}(\forall x.\alpha) = f_{gaze}(\exists x.\alpha) = f_{gaze}(\neg \alpha) = f_{gaze}(\alpha)$;

4. $f_{gaze}(p(\underline{x})) = \{p\}$.

For example, the sequent,

$$\{a =_{set} b\} \ => \ \{a \subseteq_{set} b\}$$

abstracts onto,

$$\{=_{set}\} \ => \ \{\subseteq_{set}\}$$

The ground and abstract spaces contain a set of rewrite rules of the form $\alpha \Rightarrow \beta$; each of these represents a definition or important lemma. The rewrite rules of the abstract space can be obtained by abstracting the left and righthand side of the rewrite rules of the ground space in the same way as the sequents. For example, the rewrite rules,

$$a =_{set} b \Rightarrow (\forall x.x \in a \leftrightarrow x \in b)$$

$$a \subseteq_{set} b \Rightarrow (\forall x.x \in a \rightarrow x \in b)$$

abstract onto,

$$=_{set} \Rightarrow \in$$

$$\subseteq_{set} \Rightarrow \in$$

We can prove the abstract sequent, $\{=_{set}\} \ => \ \{\subseteq_{set}\}$ by using these two abstract rules to rewrite the abstract sequent onto the tautology,

$$\{\in\} \ => \ \{\in\}$$

This suggests applying (the unabstraction of) these rewrite rules in the ground space. Indeed, with some additional logical manipulation this is sufficient to prove the sequent,

$$\{a =_{set} b\} \ => \ \{a \subseteq_{set} b\}$$

Unfortunately, this heuristic is neither sound (it will sometime suggest applying the wrong rewrite rules) nor complete (it will not always suggest the appropriate rewrite rule). That is, $f_{gaze} : \Sigma_1 \Rightarrow \Sigma_2$ is not a TI-, nor a TD-abstraction. Since negation is not part of the abstract language, we do not need to consider whether it is a NTI- or NTD-abstraction.

Theorem 31 *:* $f_{gaze} : \Sigma_1 \Rightarrow \Sigma_2$ *is not a* T**-abstraction*

Proof: By example.

As $\{\}$ => $\{p, \neg p\}$ is provable in Σ_1 but $\{\}$ => $\{p\}$ is not provable in Σ_2, it is not a TI- or TC-abstraction.

Since $\{\exists x.p(x)\}$ => $\{\forall x.p(x)\}$ is not provable in Σ_1 but $\{p\}$ => $\{p\}$ is provable in Σ_2, it is not a TD-abstraction.

□

As a consequence, we cannot be certain when gazing will suggest the appropriate definitions to unfold, and when it will fail to identify the appropriate definitions to unfold. This use of an abstraction therefore seems rather *ad hoc*.

Gazing	
Ground space:	first order sequent calculus
Abstract space:	propositional sequent calculus
Provability preserving:	no
Negation preserving:	no
Theory abstraction:	no
Λ/Ω-invariance:	yes
Δ-invariance:	yes
Reference:	[Plu87]

4.3 Propositional abstractions

An abstraction is **propositional** iff the abstract space is propositional. Propositional abstractions are very important as their abstract spaces are decidable.

Example 6 (Connection methods):

Connection methods have been proposed in various forms as an efficient way to perform theorem proving [Kow75, Cha79]. Common to these proposals is a *connection graph* which represents possible resolutions between complementary literals. The approaches differ in how they search this graph; most, however, can be treated as propositional abstractions. Chang [Cha79], for instance, describes an approach in which the connection graph is searched for a **resolution plan**, a list of possible resolutions from which we can derive $\perp$; this plan is then executed by finding a unifier that simultaneously makes all the appropriate literals in the plan complementary. The expensive cost of unification is thus delayed until we have a complete plan. This approach can eliminate many redundancies of conventional resolution (*eg.* simple reorderings of the resolutions) and allows all the strategies developed for resolution (like linear, set of support ...) to be used. Indeed, the fact that we can use conventional resolution strategies is a trivial observation once we have described this example as the abstract space in which we construct the resolution plan merely uses a restricted form of propositional resolution.

We formalise this example as a Λ/Ω-invariant theory abstraction which maps from a first order calculus using resolution to a propositional calculus using "restricted" resolution. The abstraction preserves the logical structure of wffs and axioms, mapping the atomic formulae as follows:

$$f(p(\underline{x})) = p$$

All atomic formulae with the same predicate symbol map onto the same propositional constant.

The restriction on resolution in the abstract space is that the abstract wffs $f(\alpha \vee \beta)$ and $f(\alpha' \vee \neg\beta')$ are only allowed to resolve together in Σ_2 if β resolves with $\neg\beta'$ in Σ_1 (allowing for renaming of any common variables); this prevents us drawing up plans in which literals could never be made complementary. For example, $f(p(a))$ and $f(\neg p(b))$ are not allowed to unify in the abstract space even though they abstract to p and $\neg p$ respectively. The resolution plan may not be of any use as no consistent set of substitutions for the variables need exist.

The links in a connection graph are just a means of pre-compiling the allowed resolutions. It also allows inference in Σ_2 to be performed *independently* of Σ_1.

Consider, for example, the problem in [Cha79] where we wish to show that the following set of clauses is unsatisfiable:

$$\{\ \neg p(x) \vee p(g(x)), p(a), \neg p(g(g(a)))\ \}$$

We will give below an abstract proof and show how this maps onto a ground proof. To do this, we need to find suitable unifications for every step suggested by the abstract proof.

$$\begin{array}{cc}
\neg p \vee p & p \\ \hline
p & \neg p \vee p \\ \hline
p & \neg p \\ \hline
\perp &
\end{array}
\qquad \Longrightarrow \qquad
\begin{array}{cc}
\neg p(x) \vee p(g(x)) & p(a) \\ \hline
p(g(a)) & \neg p(y) \vee p(g(y)) \\ \hline
p(g(g(a))) & \neg p(g(g(a))) \\ \hline
\perp &
\end{array}$$

Note that the ground and abstract proofs have identical shapes. In attempting to find a ground proof, Chang does not use an abstract proof itself. Instead, he builds a resolution plan which gives, in reverse order, the literals we will attempt to resolve together in the ground proof; this involves choosing how to unabstract each wff from the abstract proof. Using Chang's notation [Cha79], one of the resolution plans that can be generated from the abstract proof is:

$$\langle p(g(y)), \neg p(g(g(a)))\rangle \quad \langle p(g(x)), \neg p(y)\rangle \quad \langle \neg p(x), p(a)\rangle$$

Each pair of wffs represents a resolution; reading from **right to left**, we get a three step plan: first, to resolve $\neg p(x)$ from $\neg p(x) \vee p(g(x))$ with $p(a)$, then to resolve the resolvent of this, $p(g(x))$ with $\neg p(y)$ from $\neg p(y) \vee p(g(y))$, and finally to resolve the resolvent of this, $p(g(y))$ with $\neg p(g(g(a)))$. The substitution, $\{a/x, g(a)/y\}$ unifies all the literals in this plan, giving a valid ground proof. Chang's resolution plans are very similar to the abstract proof plans we will introduce in Chapter 7.

Note that the abstract proof is not the shortest we can find; having deduced p, there is a redundant step where we resolve with $\neg p \vee p$ to deduce p again; this extra resolution is needed so that the unifications in the ground space can succeed. In the proof that this abstraction is truthful, we show that for every ground proof there is an abstract proof which contains the same resolutions. Abstract proofs are therefore useful as plans for ground proofs – given an abstract proof, we build a ground proof by trying to find a unifier that makes all the appropriate literals complementary.

Theorem 32 : *The abstraction used by Chang's connection method is a TI/NTI-abstraction.*

Proof: By mapping a proof Π_1 of φ in Σ_1 onto a proof Π_2 of $f(\varphi)$ in Σ_2. We simply apply f to every wff in Π_1. Note that φ might be $\perp$, and $f(\perp) = \perp$. □

The abstract proof we construct is "simpler" than the ground proof as it contains the same resolutions as the ground proof but without any unification.

A significant problem with many TI-abstractions, this one included, is that they can map a consistent theory onto an inconsistent theory. This is explored in more detail in Chapter 5. The restriction on resolution in the abstract space prevents many mappings from giving an inconsistent abstract space. For example, the axioms, $\{p(a), \neg p(b)\}$ do not map onto an inconsistent abstract space as $f(p(a))$ and $\neg f(p(b))$ are not allowed to resolve together; however, there still exists less trivial but consistent sets of axioms which give an inconsistent abstract space (*eg.* $\{p(x) \vee q(x), \neg p(a), \neg q(b)\}$).

Connection methods	
Ground space:	clauses & resolution
Abstract space:	clauses & restricted resolution
Provability preserving:	TI/NTI
Negation preserving:	yes
Theory abstraction:	yes
Λ/Ω-invariance:	yes
Δ-invariance:	no
Reference:	[Pla86]

Example 7 (Propositional decider):

A different propositional abstraction is used in [GG88] to implement a decider for a fragment of first order logic; this abstraction is used to determine whether a first order wff is provable using only the propositional connective inference rules. We define it as a Λ/Ω-invariant abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ between a first order calculus, Σ_1 with a complete deductive machinery and a propositional calculus with a complete propositional decider, Σ_2. Though this abstraction can be thought of as $\Delta-$invariant (that is, the abstract space just uses the propositional inference rules of the ground space), we will often use an efficient propositional decision procedure in the abstract space; this decision procedure can be based on a completely different notion of inference than that used in the ground space. The mapping function used to abstract wffs and the axioms is defined by:

1. $f(\alpha) = P_i$, where α is any atomic formula; occurrences of identical atomic formulae are rewritten as occurrences of the same propositional constant P_i; occurrences of different atomic formulae are rewritten differently.

2. $f(\exists x.\alpha) = P_j$ where α is any formula; occurrences of identical existentially quantified formulae or of existentially quantified formulae which differ only in the name of their bounded variables are rewritten as occurrences of the same propositional constant P_j;

3. $f(\forall y.\alpha) = P_k$ where α is any formula; occurrences of identical universally quantified formulae or of universally quantified formulae which differ only in the name of their bound variables are rewritten as occurrences of the same propositional constant P_k;

4. $f(\alpha \wedge \beta) = f(\alpha) \wedge f(\beta)$;

5. $f(\alpha \vee \beta) = f(\alpha) \vee f(\beta)$;

6. $f(\neg\alpha) = \neg f(\alpha)$;

7. $f(\alpha \rightarrow \beta) = f(\alpha) \rightarrow f(\beta)$;

8. $f(\alpha \leftrightarrow \beta) = f(\alpha) \leftrightarrow f(\beta)$.

This is a TD/NTD-abstraction.

Theorem 33 : *The abstraction used by the propositional decider, $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TD/NTD-abstraction.*

Proof: Since $f : \Sigma_1 \Rightarrow \Sigma_2$ is a negation preserving abstraction between systems with negation, we just need to show that it is a TD-abstraction. It does not affect what we can prove to assume that Σ_1 has the standard natural deduction rules of [Pra65] whilst Σ_2 just has the propositional rules. For any proof in Σ_2, a proof can be built in Σ_1 which performs the same sequence of inference rules. No complications arise from the different names of bound variables as we can always prove the equivalence of formulae which differ only in the names of the bound variables. □

Note that this is not a TI-abstraction; for example, "$\forall x.p(x) \vee \neg p(x)$" is provable in the ground space but its abstraction is not provable in the abstract space. Since it is a TD-abstraction, abstract theorems always correspond to ground theorems. This is rather unusual – most abstractions used in the past are TI-abstractions and abstract theorems may or may not correspond to ground theorems.

Propositional decider	
Ground space:	first order calculus
Abstract space:	propositional calculus
Provability preserving:	TD/NTD
Negation preserving:	yes
Theory abstraction:	no
Λ/Ω-invariance:	yes
Δ-invariance:	yes
Reference:	[GG88]

Example 8 (Variable-free abstractions):

An interesting class of propositional abstractions introduced by Plaisted [Pla80] is the class he called "ground abstractions". To avoid overloading the word "ground", we will call them **variable-free abstractions**.

A variable-free abstraction maps wffs onto some of their variable-free instances. It can be described as a Λ/Ω-invariant abstraction from a ground space of first order clauses and a complete first order inference engine (Plaisted uses resolution, but the choice is arbitrary) to an abstract space of variable free clauses and a propositional inference engine; this propositional inference engine should treat instances of atomic formulae as propositions. Although strictly speaking the abstract space is not propositional, its inference engine is. As in the previous example, this abstraction can be thought of as $\Delta-$invariant even though, for efficiency, we may use different inference rules for propositional reasoning in the abstract space. The mapping function abstracts clauses in the ground space onto (implicitly conjoined) sets of their instances. We could extend this abstraction to full first order languages if the mapping function first skolemized formulae; Chapter 15 of [Bun83] presents a longer discussion on skolemizing formulae, including those not in prenex normal form.

Consider, for example, the maximum function defined by:

$$\max(0, y) = \max(x, 0) = 0$$

$$\max(s(x), s(y)) = s(max(x, y))$$

The clause:

$$\{\max(x, y) \geq x\}$$

might be abstracted onto the clauses,

$$\{\max(0, 0) \geq 0\}\ ,\ \{\max(s(0), s(0)) \geq s(0)\}$$

The proof of these clauses could be used to guide the proof of the original clause; such a use of abstraction is intimately linked to the idea of generalisation. The theorem, $\{\max(x, y) \geq x\}$ is simply a generalisation of the theorem, $\{\max(0, 0) \geq 0\}$ and the theorem, $\{\max(s(0), s(0)) \geq s(0)\}$. This use of abstraction can also be seen as theorem proving guided by example; the proofs of $\{\max(0, 0) \geq 0\}$ and $\{\max(s(0), s(0)) \geq s(0)\}$ can be used as outlines for the base and step cases of an inductive proof of $\{\max(x, y) \geq x\}$. Interestingly, the use of examples in theorem proving was another avenue of research followed by Plaisted [Pla84]. Other work in this area can be found in [Ble83], [Gel59], [Hen75] and [Rei73]. As this example might suggest, a variable-free abstraction is a TI/NTI-abstraction.

Theorem 34 *: A variable-free abstraction is a TI/NTI-abstraction.*

Proof: A variable-free abstraction satisfies all the requirements necessary to be one of Plaisted's ordinary abstractions. Since such abstractions are NTI, a variable-free abstraction is also NTI. Alternatively, most introductory texts to automated theorem proving (*eg.* [Bun83]) will show that a conjunction of variable-free instances is unsatisfiable if φ is unsatisfiable. That is, a variable-free abstraction is a NTI-abstraction. Since it is negation preserving, it is also TI. □

Herbrand's theorem is just a special case of a variable-free abstraction where the instances range over the whole Herbrand universe [Her67]. It translates to the statement that $f : \Sigma_1 \Rightarrow \Sigma_2$ is a NTC-abstraction for these instances; this result in turn provides the theoretical justification for the resolution proof procedure.

Variable-free abstractions	
Ground space:	first order clauses and resolution
Abstract space:	variable-free clauses and resolution
Provability preserving:	TI/NTI
Negation preserving:	yes
Theory abstraction:	yes
Λ/Ω-invariance:	yes
Δ-invariance:	yes
Reference:	[Pla80]

4.4 Domain abstractions

Domain abstractions are abstractions which map the domain (that is, the constants) of the ground space.

Example 9 (Granularity):

Hobbs has suggested a theory of **granularity** in which a complex theory is abstracted onto a simpler, more coarse-grained theory with a smaller domain [Hob85]. For example, the real world of continuous time and positions could be mapped onto a (micro)world of discrete time and positions. Granularity can be formalised as a Λ/Ω-invariant theory abstraction, $f_{gran} : \Sigma_1 \Rightarrow \Sigma_2$. The ground and abstract spaces are first order calculi. Different constants in the ground space are mapped onto (not necessarily different) constants in the abstract space according to an **indistinguishability relation**, "$\sim$". This is defined by the second-order axiom:

$$\forall x, y \,.\, x \sim y \;\leftrightarrow\; \forall p \in R \,.\, p(x) \leftrightarrow p(y)$$

where R is the subset of the predicates of the ground space determined too be **relevant** to the situation at hand. Hobbs does not say much about determining which predicates are relevant, except that it is a very hard problem. Like Hobbs, we have only defined indistinguishability for unary predicates; it could, however, easily be generalised to n-ary predicates. The mapping function keeps the same logical structure of wffs but abstracts any constant onto its equivalence class. That is,

$$f_{gran}(p(a)) = p([a])$$

where a is a constant symbol and $[a]$ is the constant in the abstract language representing the equivalence class of the constant a with respect to the indistinguishability relation;

$$[x] = \{y : x \sim y\}$$

All variables are left unchanged. That is,

$$f(p(x)) = p(x)$$

The mapping extends to n-ary predicates in the obvious way. This abstraction is TI/NTI.

Theorem 35 : *A granularity abstraction is a TI/NTI-abstraction.*

Proof: We can map a proof tree Π_1 of φ onto a proof tree Π_2 of $f_{gran}(\varphi)$ merely by applying f to every wff in the proof tree. $f_{gran} : \Sigma_1 \Rightarrow \Sigma_2$ is therefore a TI-abstraction. As it is a negation preserving mapping between systems of negation, it is also a NTI-abstraction. $\Box$

Like many other TI-abstractions, this abstraction can map a consistent ground space onto an inconsistent abstract space. For example, if the constants a and b are considered "indistinguishable" and $[a]$ represents the equivalence class of a and b, then a consistent ground space with equality and the theorem that $\neg(a = b)$ maps onto an inconsistent abstract space containing the theorem, $\neg([a] = [a])$. The consistency of the abstract space is, however, guaranteed for a suitable indistinguishability relation.

Theorem 36 *: If the abstraction, $f_{gran} : \Sigma_1 \Rightarrow \Sigma_2$ defines indistinguishability with respect to all the predicates of the ground language, and Σ_1 is consistent then Σ_2 is also consistent.*

Proof: By contradiction. Assume that a consistent ground space, Σ_1 maps onto an inconsistent abstract space, Σ_2. That is, we can find a proof tree, Π_2 of $\perp$. We show how you can construct a valid proof tree, Π_1 of $\perp$ in Σ_1, contradicting the assumption that Σ_1 is consistent. For every equivalence class, $[a]$ we pick one member of that class, b; to every wff, φ in Π_2 we apply the substitutions $\{b/[a]\}$. This will generate a proof tree, Π_1 whose assumptions will either be axioms of Σ_1 or will be derivable from them using the indistinguishability relation and substitution of equivalences. If indistinguishability were not defined over **all** predicates, this last step would not always be possible. $\Box$

If indistinguishability is defined over all predicates, $f_{gran} : \Sigma_1 \Rightarrow \Sigma_2$ must preserve consistency. To avoid inconsistency, we might therefore decide that *all* predicates are relevant to the situation at hand. Unfortunately this gives a TC-abstraction. As we argued in Chapter 2, TC-abstractions are too strong since they do not throw away any information. We end with the observation that f_{gran} is just an example of one of the abstractions proposed by Plaisted [Pla80] in which function symbols (including constants or 0-ary functions) are renamed in a systematic, but not necessarily 1-to-1 way.

Granularity	
Ground space:	first order calculus
Abstract space:	first order calculus
Provability preserving:	TI/NTI
Negation preserving:	yes
Theory abstraction:	yes
Λ/Ω-invariance:	yes
Δ-invariance:	yes
Reference:	[Hob85]

Example 10 (Imielinski I):

Abstraction has been proposed by Imielinski [Imi87] as a basis for approximate methods of reasoning in first order logic. He argues that, though such approximate methods can return answers that are not always correct, their errors should be characterisable. As an example, Imielinski considers two domain abstractions; the first is a TI-abstraction (and thus provides an over-estimate of the correct answers) whilst the second is a TD-abstraction (and thus provides an under-estimate of the correct answers).

Like Hobbs' theory of granularity (see the last example), Imielinski's abstractions are based upon an equivalence relation between objects in the domain. Imielinski gives the equivalence relation two different interpretations, each of which leads to a different abstraction. One interpretation, which he calls the "second-order interpretation", views the equivalence classes as the names of objects in the abstract domain; this gives rise to a TI-abstraction in which a property holds of an abstract object iff the property holds for one or more members of the equivalence class.

We can formalise this second-order interpretation as a Λ/Ω-invariant theory abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$. Imielinski considers simple non-deductive databases in which the languages of the ground and abstract spaces are restricted to closed formulae containing only the $\exists$ quantifier, and $\wedge$, $\vee$ connectives; this is sufficient to express queries to a (non-deductive) database of positive axioms. No problems would, however, arise in moving to a full first-order language. The mapping function is the same as in the granularity abstraction. That is:

$$f(p(a)) = p([a])$$

where a is any constant symbol, $[a]$ is the equivalence class of the constant a with respect to an indistinguishability relation, $\sim$, and:

$$f(p(x)) = p(x)$$

where x is any variable. The mapping extends to n-ary predicates in the obvious way. See [Hob85] and [Imi87] for some examples of indistinguishability relations. This abstraction is truthful.

Theorem 37 *: Imielinski's first abstraction is a TI-abstraction.*

Proof: We can map a proof tree Π_1 of φ onto a proof tree Π_2 of $f(\varphi)$ merely by applying f to ever wff in the proof tree. □

This abstraction *over-estimates* answers; that is, the abstract space will suggest certain wffs are true which are, in fact, false in the ground space. Though this abstraction is not always sound, it is complete; that is, the abstract space returns all the theorems of the ground space. Since the ground and abstract languages lack negation, there is no possibility of mapping into an inconsistent abstract space. If, however, we use a closed world assumption to deduce negative information from the database, we would have to worry about the consistency of the abstract space.

Imielinski I	
Ground space:	non-deductive database
Abstract space:	non-deductive database
Provability preserving:	TI
Negation preserving:	n/a
Theory abstraction:	yes
Λ/Ω-invariance:	yes
Δ-invariance:	yes
Reference:	[Imi87]

Example 11 (Imielinski II):

Imielinski's second domain abstraction [Imi87] is a TD-abstraction. This *under-estimates* answers to the database; that is, the abstract space will suggest certain wffs are false which are, in fact, true in the ground space. Though such an abstraction is not complete, it is always sound; all the abstract theorems it returns are guaranteed to be theorems of the ground space. Imielinski derives this abstraction by giving a very different interpretation to the equivalence relation, what he calls the "first-order interpretation". Under this interpretation, properties true of an object in the ground space hold in the abstract space for *some* unidentified member of the (possibly larger) equivalence class of the object.

We can formalise this as a Λ/Ω-invariant abstraction, $g : \Sigma_1 \Rightarrow \Sigma_3$ from the same ground space, Σ_1 as in the previous example (that is, a non-deductive database whose language consists of closed formulae containing only the $\exists$ quantifier, and $\wedge, \vee$ connectives) to another first order theory whose language consists of closed disjunctive normal formulae containing only the $\exists$ quantifier, and $\wedge, \vee$ connectives. The language of the abstract space differs in two significant ways from the language of the abstract space in the previous example; first, its domain is the same as that of ground language (and is not the equivalence classes); second, it contains a new unary predicate for each equivalence class, $[a]$ true for every member, x of the equivalence class; we will use "$[a](x)$" to represent this predicate.

The mapping function first transforms all wffs of the ground language into disjunctive normal form. Then for each disjunct φ:

a) if φ contains the constant a it is replaced by an existentially quantified variable restricted to members of a's equivalence class. That is, φ is rewritten to $\exists x \ . \ [a](x) \wedge \varphi\{x/a\}$. The idea is to replace each named constant by an arbitrary member of its equivalence class. The domain of the abstract space is thus no smaller than that of the ground space; however, the properties true in the ground space of an object a are merely true in the abstract space of *some* object in the equivalence class of a ;

b) if φ contains multiple occurrences of an existentially quantified variable, each occurrence is replaced by a new existentially quantified variable. The occurrences of an existentially quantified variable can thereby represent different members of an equivalence class.

For example, $\exists x \ . \ p(a, x) \wedge p(b, x)$ is mapped to:

$$\exists x, u, w \ . \ [a](u) \wedge p(u, x) \wedge [b](w) \wedge p(w, x)$$

by step a), and then to:

$$\exists u, v, w, y \ . \ [a](u) \wedge p(u, v) \wedge [b](w) \wedge p(w, y)$$

by step b).

Since this mapping changes the logical structure of wffs, it is not, strictly speaking, a theory abstraction. However, as this change is only a normal forming, we shall be rather loose and call it a theory abstraction. As with the preservation of negation, extending the definition of theory abstraction to include any abstraction logically equivalent to a theory abstraction is undesirable as, in general, such an extension is undecidable. Imielinski suggests a slightly more complex mapping if we know the **selectivity** of the equivalence relation; that is, if we know the sizes of the equivalence classes. In such circumstances, we need not throw away quite so much information. Such a modification could, however, make reasoning much more expensive. As Imielinski himself points out, it is often computationally better to be ignorant about the selectivity of the equivalence relation. With or without his modification, this is a TD-abstraction.

Theorem 38 *: Imielinski's second abstraction, $g : \Sigma_1 \Rightarrow \Sigma_3$ is a TD-abstraction.*

Proof: This is shown in Lemma 5 on page 1001 of [Imi87]. Alternatively we can prove that the axioms of Σ_3 follow from a set of axioms logically equivalent to the axioms of Σ_1. We first transform each axiom of Σ_1 into (the equivalent) disjunctive normal form. Then for each disjunct φ, if φ contains the constant a, we replace φ by the equivalent formula, $\exists x \,.\, [a](x) \wedge \varphi\{x/a\} \wedge x = a$. If φ contains multiple occurrences of an existentially quantified variable, we replace each occurrence by a new existentially quantified variable, and add the equality condition that these new existentially quantified variables are equal. The resulting axioms are logically equivalent to those of Σ_1. The axioms of Σ_3 are just the result of dropping from these clauses all the equality conditions added for the constants and existentially quantified variables. By monotonicity, the axioms of Σ_3 follow from the axioms of Σ_1, and $g : \Sigma_1 \Rightarrow \Sigma_3$ is a TD-abstraction. $\Box$

Note that the language of the abstract space is more complex than the language of the ground space, and that the axioms of the abstract space are more complicated than the axioms of the ground space. It therefore seems doubtful that theorems will be much easier to solve in the abstract space than in the ground space. The only saving is that instead of having to show a property true for a, we merely have to find it true for some member of $[a]$ and one of these proofs could be easier to find.

Imielinski II	
Ground space:	non-deductive database
Abstract space:	non-deductive database
Provability preserving:	TD
Negation preserving:	n/a
Theory abstraction:	"yes"
Λ/Ω-invariance:	yes
Δ-invariance:	yes
Reference:	[Imi87]

4.5 Predicate abstractions

Predicate abstractions collapse several predicate symbols in the ground space onto a single predicate symbol in the abstract space.

Example 12 (Predicate mappings):

Tenenberg [Ten87] defines a **predicate mapping** as a Λ/Ω-invariant theory abstraction between resolution systems with a mapping function defined by:

$$f(p(\underline{x})) = q(\underline{x})$$

for all $p \in R_q$. R_q is the set of all the predicate symbols which map onto q; by definition, $R_q \cap R_s = \emptyset$ for $q \neq s$. Predicate mappings are TI-abstractions; since they are negation preserving, they are also NTI-abstractions. Of course, any such mapping that is 1–1 is a TC/NTC-abstraction.

> **Theorem 39** *: A predicate mapping, $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TI/NTI-abstraction.*
>
> **Proof:** By showing that, given a resolution proof Π_1 of φ, you can build a resolution proof of $f(\varphi)$. The proof proceeds by induction on the depth of Π_1. We just take the proof Π_1 and apply f to every wff in it. $\Box$

Note that predicate mappings are not, in general, TD. For example, if $f(p(x)) = f(q(x)) = r(x)$ then $r(x) \vee \neg r(x) \in TH(\Sigma_2)$ but $p(x) \vee \neg q(x) \notin TH(\Sigma_1)$. As with other TI-abstractions, predicate mappings can map a consistent ground space onto an inconsistent abstract space. For example, the consistent set of axioms, $\{p(x), \neg q(x)\}$ maps onto an inconsistent abstract space with the above mapping function.

Predicate mappings	
Ground space:	clauses & resolution
Abstract space:	clauses & resolution
Provability preserving:	TI/NTI
Negation preserving:	yes
Theory abstraction:	yes
Λ/Ω-invariance:	yes
Δ-invariance:	yes
Reference:	[Ten87]

Example 13 (Restricted predicate mappings):

To overcome the problem of inconsistent abstract spaces, Tenenberg has suggested [Ten87] the class of **restricted predicate mappings**; such abstractions are guaranteed to map a consistent ground space onto a consistent abstract space. Unfortunately, imposing the "restriction" involves an arbitrary amount of theorem proving in the ground space *and* loses truthfulness.

We define a restricted predicate mapping as a theory abstraction between first order calculi using resolution. The same mapping function as with the (unrestricted) predicate mapping is used to map wffs from the ground space onto wffs in the abstract space. The "restriction" on the mapping is that not every axiom of the ground space is kept in the abstract space; that is, the abstraction is not Λ/Ω-invariant. The idea is that we don't want to keep those axioms that introduce inconsistency; the axioms of the abstract space are given by $g(\Omega_1)$ where Ω_1 is the set of axioms of the ground space, Σ_1, and:

$$\begin{aligned} g(\Omega_1) = \{ f(\varphi) \mid \varphi \in \Omega_1 \textit{ and } (\varphi \textit{ is a positive clause or} \\ \forall\alpha \, . \, f(\alpha) = f(\varphi) \rightarrow \alpha \in TH(\Sigma_1)) \, \} \end{aligned}$$

Trivially, $g(\Omega_1) \subseteq f(\Omega_1)$. Unfortunately, determining which axioms to keep in the abstract space requires an arbitrary amount of theorem proving in the *ground* space; the purpose of this theorem proving is to guarantee that consistency is preserved.

Theorem 40 : *If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a restricted predicate mapping and Σ_1 is consistent then Σ_2 is also.*

Proof: On page 1013 of [Ten87], Tenenberg demonstrates how, given a model for the axioms of Σ_1 you can construct a model for the axioms of Σ_2. Since a set of clauses is consistent iff it is satisfiable (that is, has a model), this proves that if Σ_1 is consistent then Σ_2 is as well. $\Box$

As a simple corollary to this theorem, Tenenberg shows that for every wff provable in the abstract space, some wff in the ground space that abstracts onto it is also provable.

Theorem 41 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a restricted predicate mapping and $\beta \in TH(\Sigma_2)$ then there exists a wff α such that $f(\alpha) = \beta$ and $\alpha \in TH(\Sigma_1)$.*

This is *not* the same as being a TD-abstraction; indeed, some restricted predicate mappings are *not* TD-abstractions. Consider, for instance, the restricted predicate mapping with $f(p(x)) = f(q(x)) = r(x)$ and $p(x)$ as the only axiom of the ground space. Now, $f(q(x)) \in TH(\Sigma_2)$ but $q(x) \notin TH(\Sigma_1)$. By comparison, with a TD-abstraction, any wff in Σ_1 that maps onto an abstract theorem in Σ_2 is provable in Σ_1.

The definition of restricted predicate mappings can be strengthened slightly so that they are all TD-abstractions. We merely have to remove the condition in the definition of the $g(\Omega_1)$ that allows positive clauses regardless of whether their unabstractions are provable. That is, the axioms of the abstract space should be given by $h(\Omega_1)$ where Ω_1 is the set of axiom of the ground space Σ_1 and:

$$h(\Omega_1) = \{f(\varphi) \mid \varphi \in \Omega_1 \textit{ and } \forall\alpha \,.\, f(\alpha) = f(\varphi) \rightarrow \alpha \in TH(\Sigma_1)\}$$

We shall call this a *very restricted predicate mapping*; unlike a restricted predicate mappings, a very restricted predicate mapping is TD.

Theorem 42 *: A very restricted predicate mapping is a TD/NTD-abstraction.*

Proof: Since the mapping is negation preserving, we merely need to show that it is a TD-abstraction. Given a proof tree Π_2 in Σ_2 that ends in $f(\varphi)$ we show how you can construct a proof tree Π_1 in Σ_1 that ends in φ. Note that φ is not necessarily $\perp$. The proof proceeds by induction on the depth, n of Π_2.

For the base case, $n = 1$ and $f(\varphi)$ is an axiom of Σ_2. From the definition of $h(\Omega_1)$, φ is either an axiom of Σ_1 or $\varphi \in TH(\Sigma_1)$.

For the step case, assume we have shown it for all proof trees up to depth m. Consider a proof tree Π_2 of depth $m+1$ that ends in $f(\varphi)$ and in which $f(\varphi)$ is the resolvent of $c \vee q_1 ... \vee q_n$ and $c' \vee \neg q'_1 ... \vee \neg q'_m$. That is, $f(\varphi) = (c \vee c')\theta$ where θ is a most general unifier and for $1 \leq i \leq n$ there exists a j such that $q_i\theta \equiv q'_j\theta$. Let q_i have a predicate symbol r. Then, as they are complementary literals, q'_j must have the same predicate symbol. Construct the wffs p_i and p'_j which map to q_i and q'_j and which both have the same predicate symbol $s \in R_r$. If there is a choice of predicate symbols, s (that is, if R_r has more than one element), pick the alphabetically smallest one. Now, p_i and $\neg p'_j$ will resolve together with unifier θ since the mapping function is transparent to substitutions. Pick the wffs d and d' which map to c and c' such that $\varphi \equiv (d \vee d')\theta$. Again, if there is a choice, pick the alphabetically smallest. By the induction hypothesis, we can prove $d \vee p_1 ... \vee p_n$ and $d' \vee \neg p'_1 ... \vee \neg p'_m$. Finally, note that $d \vee p_1 ... \vee p_n$ and $d' \vee \neg p'_1 ... \vee \neg p'_m$ will successful resolve together with unifier, θ to give φ. $\Box$

A major problem with (very) restricted predicate mappings is that, they are not, in general, truthful; they therefore lose completeness. There are usually wffs which are provable in the ground space whose abstractions are not provable in the abstract space; abstract theorem proving therefore reject these wffs as theorems of the ground space. Another major problem with restricted predicate mappings is that determining which axioms to include in the abstract space is, in general, undecidable. One solution suggested in [Ten87] is to weaken the derivability condition in the definition of $g(\Omega_1)$ or $h(\Omega_1)$ (namely, that $\alpha \in TH(\Sigma_1)$) to a condition of derivability within certain resources. This conservative approach would construct an abstract space weaker than it theoretically needs to be to preserve consistency (that is, an abstract space with fewer theorems than is strictly necessary). For example, Tenenberg proposes a class of predicate abstractions, called **weak abstraction mappings** [Ten88] in which the abstraction of an axiom is included in the abstract space iff certain clauses that map onto the abstract wff are also *axioms* of the ground space; this is, of course, trivial to compute. Weak predicate mappings satisfy a property similar to TD-abstractions; if we can prove a clause in the abstract space then some (but not all) clauses that map onto this will be provable in the ground space (corollary 5.3, page 101 of [Ten88]). Such abstractions therefore also lose completeness.

Restricted predicate mappings	
Ground space:	clauses & resolution
Abstract space:	clauses & resolution
Provability preserving:	TD/NTD
Negation preserving:	yes
Theory abstraction:	yes
Λ/Ω-invariance:	yes
Δ-invariance:	yes
Reference:	[Ten87]

4.6 Formal methods

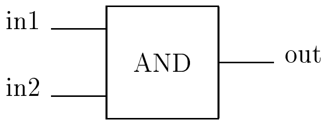
Abstractions have so far been characterised by how they *globally* preserve provability or inconsistency. Often, however, we are only interested in how abstractions preserve provability or inconsistency with respect to some subset of the language. To capture such an idea, we can introduce the notion of a mapping being a T*-abstraction (or NT*-abstraction) with respect to some subset of the language. For example, we will say that an abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ is **TI with respect to** Γ iff, for any wff $\varphi \in \Gamma$, $\varphi \in TH(\Sigma_1)$ implies $f(\varphi) \in TH(\Sigma_2)$. We will need this new notation in the next example.

Example 14 (Hardware verification):

Abstraction has been proposed as a tool for tackling the size and complexity of proofs needed for hardware verification [Mel87]. There are several competing approaches to hardware verification based upon a variety of logics: higher-order logic [Gor86], type theory [Bas89], Boyer and Moore's Computational Logic [Hun89], ... *etc.* For the purpose of this example, we will adopt the higher-order approach advocated by Mike Gordon's HOL group [CGM87].

The behaviour of a hardware device can be represented by a specification predicate, $spec(\underline{ext})$. This predicate describes the values of the external ports, $\underline{ext}$ of the device; to model time-dependent behaviour, $\underline{ext}$ can be a function of time. Although a function from inputs to outputs is usually just as adequate as a predicate, a predicate can give greater expressivity; for example, devices with unstable signals may have no functional interpretation.

Consider an AND-gate:



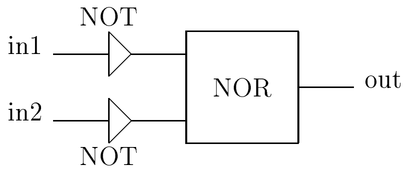
The behaviour of this gate can be described by the axiom:

$$spec(in1, in2, out) \;\leftrightarrow\; out = and(in1, in2)$$

where and is a function of type $bool \times bool \rightarrow bool$, $bool = \{True, False\}$, and:

$$\begin{aligned} and(True, True) &= True \\ and(True, False) &= False \\ and(False, True) &= False \\ and(False, False) &= False \end{aligned}$$

A specification predicate describes how we want the device to behave. The actual implementation of the device is described by an implementation predicate, $imp(\underline{int}, \underline{ext})$ where $\underline{int}$ and $\underline{ext}$ are the internal and external signals. The implementation predicate describes the components of the device, and their interconnections; often this is given by a conjunction of predicates, each specifying a different component, with shared variables for the connections. For example, the AND-gate described by $spec(in1, in2, out)$ could actually be implemented by a NOR-gate with NOT-gates on its inputs:



This can be described by the implementation axiom:

$$\begin{aligned} imp(in1, in2, int1, int2, out) \quad &\leftrightarrow \quad not(in1, int1) \ \wedge \ not(in2, int2) \ \wedge \\ & \qquad nor(int1, int2, out) \end{aligned}$$

where not and nor are two predicates describing NOT-gates and NOR-gates respectively; that is,

$$\begin{aligned} not(in, out) \quad &\leftrightarrow \quad (in = True \wedge out = False) \vee \\ & \qquad (in = False \wedge out = True) \\ nor(in1, in2, out) \quad &\leftrightarrow \quad (out = True \leftrightarrow in1 = False \wedge in2 = False) \vee \\ & \qquad (out = False \leftrightarrow in1 = True \vee in2 = True) \end{aligned}$$

The task of the hardware verifier is to prove that the implementation gives the desired behaviour. The specification of the device's behaviour ignores **irrelevant** details that concern the actual implementation; the specification predicate can therefore be seen as an abstraction of the implementation predicate that ignores how the device is actually implemented. Indeed, the behaviour of a device is often described at various levels of abstraction; by ignoring the internal structure, prohibited states, and the nature of the signals, we can give increasingly abstract descriptions of a device's behaviour. The relationship between implementation and (abstract) specification can therefore be described as an abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ from a higher order theory, Σ_1 which includes a predicate, $imp(\underline{x})$ which describes the implementation of the device to another higher order theory, Σ_2 which includes a predicate, $spec(\underline{y})$ which specifies its (abstract) behaviour.

The abstraction's mapping function links the signals which satisfy, $imp(\underline{x})$ and $spec(\underline{y})$; that is,

$$f(imp(\underline{x})) = spec(\underline{y})$$

where $\underline{y}$ is some function of $\underline{x}$; note that $\underline{x}$ does not necessarily equal $\underline{y}$. For example, $\underline{x}$ may be in terms of signal values, like $\{low, high, floating\}$ whilst $\underline{y}$ is in terms of booleans or high-level data types like n-bit words. The exact relationship between $\underline{x}$ and $\underline{y}$ will depend on the particular abstraction.

Melham describes the implementation and the (abstract) specification in the *same* theory; for the sake of clarity, we would advocate two theories as this keeps separate the two very different levels of description. Melham gives four classes of abstraction commonly used in hardware verification: structural abstraction, behavioural abstraction, data abstraction, and temporal abstraction. For each class, he gives a *correctness relation*; each of these correctness relations can be seen as a TI-abstraction with respect to some subset of the language.

Melham's first class of abstractions is the class of **structural abstractions**; these suppress information about a device's internal structure. For such abstractions, the mapping function is given by:

$$f(\exists \underline{int}.imp(\underline{int}, \underline{ext})) \;=\; f(imp(\underline{int}, \underline{ext})) \;=\; spec(\underline{ext})$$

where $\underline{int}$ are the internal signals whose value we can ignore, and $\underline{ext}$ are the external signals whose values we are interested in. This is an example of an abstraction which both renames and throws away arguments to predicates. Melham calls a structural abstraction **correct** iff it is provable that $\exists \underline{int}.imp(\underline{int}, \underline{ext}) \in TH(\Sigma_1)$ is equivalent to $spec(\underline{ext}) \in TH(\Sigma_2)$.

Theorem 43 *: The correctness relation for structural abstractions is* *the property of being a TC-abstraction with respect to the set of wffs* $\{\exists \underline{int}.imp(\underline{int}, \underline{ext})\}$.

Proof: Immediate from the definition of correctness. □

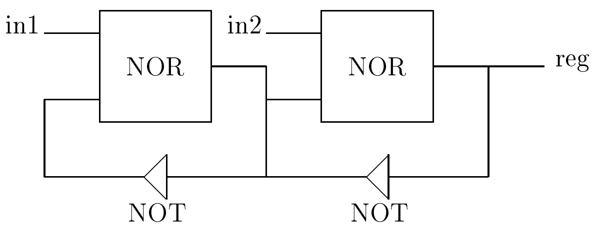
This equivalence justifies why we are allowed to ignore the internal signals.

Melham's second class of abstractions is the class of **behavioural abstractions**; these leave unspecified the behaviour of a device for illegal states. For such abstractions, the mapping function satisfies:

$$f(imp(\underline{x})) \;=\; spec(\underline{x})$$

The behaviour of the device is, however, only partially specified; that is, it is only constrained for a subset of the possible situations, the *allowed* situations. Outside this set, the specification predicate can be satisfied by signal values that would be disallowed by a more complete specification. For instance, we could define the behaviour of a device for the expected input signal values but not for prohibited or incorrect input signal values. The abstract space thus imposes weaker constraints on the signal values than the ground space.

Consider, for example, a simple register:



The implementation predicate for this circuit is given by:

$$\begin{aligned} imp(in1, in2, reg) \;\leftrightarrow\; & \exists int1, int2 \;.\; nor(in1, int1, int2) \wedge not(int2, int1) \wedge \\ & nor(in2, int2, reg) \wedge not(reg, int2) \end{aligned}$$

The inputs, $in1$ and $in2$ are usually kept $False$. If $in1$ is made $True$ then reg takes and stores the value, $True$. However, if $in2$ is made $True$ then reg takes and stores the value, $False$. Since this circuit contains feedback loops, we might decide to add *Undefined* to $\{True, False\}$ as an allowed signal and to augment the descriptions of *nor* and *not* accordingly. This will allow for the circuit to be put in an undefined, possibly oscillating, state.

The behaviour of this circuit can be described by two predicates, *store* and *stable*. The wff, $store(in1, in2, t, val)$ means that the value, val is stored in the register, reg at time t by the input signals, $in1$ and $in2$. That is:

$$\begin{aligned} store(in1, in2, t, val) \quad \leftrightarrow \quad & (in1(t) = True \wedge val = True) \vee \\ & (in2(t) = True \wedge val = False) \end{aligned}$$

The term, $stable(sig, t1, t2, val)$ means that the signal, sig has the stable value, val, from time, $t1$ to time, $t2$. That is:

$$stable(sig, t1, t2, val) \quad \leftrightarrow \quad \forall t \,.\, t1 \leq t \leq t2 \rightarrow sig(t) = val$$

An abstract specification of the register's behaviour is then:

$$\begin{aligned} spec(in1, in2, reg) \quad \leftrightarrow \quad & \forall t1, val \,.\, store(in1, in2, t1, val) \rightarrow \\ & \forall t2 \,.\, stable(in1, t1, t2, False) \wedge \\ & stable(in2, t1, t2, False) \rightarrow \\ & stable(reg, t1, t2, val) \end{aligned}$$

This specification predicate only defines the behaviour of the register for well behaved inputs; that is, one of $in1$ or $in2$ must always be $False$. It does not specify the behaviour of the device when $in1 = in2 = True$ since the hypotheses are false, and the specification is vacuously satisfied for any value of reg; that is, reg can take the values $True$, $False$ or *Undefined*. The implementation predicate, however, gives a precise value for reg in this situation, *Undefined*. For this reason, our specification of the register's behaviour can be seen as an abstraction of the description of its implementation.

Melham calls such a behavioural abstraction **correct** iff it is provable that $imp(\underline{x}) \in TH(\Sigma_1)$ implies $spec(\underline{x}) \in TH(\Sigma_2)$. Whatever signals satisfy $imp(\underline{x})$, also satisfy $spec(\underline{x})$. However, there may also be signals for which $spec(\underline{x})$ is true but $imp(\underline{x})$ is false; the implication does not therefore reverse. This is the case with the register.

> **Theorem 44** *: The correctness relation for behavioural abstractions is the property of being TI with respect to the set,* $\{imp(\underline{x})\}$.
>
> **Proof:** Immediate from the definition of correctness. □

Melham's third class of abstractions is the class of **data abstractions**; these abstractions allow the behaviour of a device to be specified in terms of different data types. For example, we might want to specify the behaviour of an 8-bit adder in terms of two 8-bit inputs and one 8-bit output instead of 16 boolean inputs and 8 boolean outputs. To formalise such abstractions, we provide a function, g that abstracts between the data types. For example, we can define a function for mapping eight boolean values onto an 8-bit word by:

$$g(x_1, ..., x_8) = \langle x_1, ..., x_8 \rangle$$

The mapping function for a data abstraction satisfies:

$$f(imp(\underline{x})) = spec(g \diamond \underline{x})$$

where $g \diamond \underline{x}$ is g applied to the signal values $\underline{x}$ in some predefined way. Consider, for example, a 25 place predicate which describes the implementation of an 8-bit adder with carry:

$$imp(x_1, ..., x_8, y_1, ..., y_8, z_1, ..., z_8, c)$$

This would abstract onto the 4 place predicate:

$$spec(\underline{x}, \underline{y}, \underline{z}, c)$$

where $\underline{x}$, $\underline{y}$, and $\underline{z}$ are 8-bit words, and c is a carry bit, Such a data abstraction is **correct** iff it is provable that $imp(\underline{x}) \in TH(\Sigma_1)$ implies $spec(g \diamond \underline{x}) \in TH(\Sigma_2)$. It is not an equivalence as the abstraction of the data types may ignore some irrelevant values (like floating signals).

Theorem 45 *: The correctness relation for data abstractions is the property of being TI with respect to the set,* $\{imp(\underline{x})\}$.

Proof: Immediate from the definition of correctness. □

Melham's final class of abstractions is the class of **temporal abstractions**; these allow the behaviour of a device to be viewed over time at different "grain sizes". For example, we might want to specify the behaviour of a device in terms of its behaviour at each (high-level) instruction cycle instead of at each (low-level) clock cycle. To formalise such abstractions, we provide a function, h that maps the discrete points of low-level time onto discrete points in high-level times. For example, for an instruction cycle of n clock cycles and points in time represented by the natural numbers, we might define

$$h(t) = t \mid n$$

where $\mid$ denotes integer division. We impose one condition on h; that is, it must be an increasing function:

$$\forall t_1, t_2 \,.\, t_1 > t_2 \rightarrow h(t_1) \geq h(t_2)$$

The mapping function is given by:

$$f(imp(\underline{x})) = spec(h \otimes \underline{x})$$

where $h \otimes \underline{x}$ is the result of applying h to the low-level time signals in $\underline{x}$. This is a granularity abstraction which renames constants (representing points in time) in a systematic way. A temporal abstraction is **correct** iff it is provable that h is increasing and $imp(\underline{x}) \in TH(\Sigma_1)$ implies $spec(h \otimes \underline{x}) \in TH(\Sigma_2)$.

Theorem 46 *: The correctness relation for temporal abstractions is the property that h is increasing and the abstraction is TI with respect to the set,* $\{imp(\underline{x})\}$.

Proof: Immediate from the definition of correctness. □

Although this analysis of the abstractions used in hardware verification has not brought great new insights, it is nevertheless useful. In particular, we note that the sort of mappings that are performed on the syntax of a representation (renaming constants, dropping arguments, ...) are exactly those operations that we also observed with other abstractions used in theorem proving, planning, and commonsense reasoning. The main conclusion we draw from this is that (the syntax of) a representation can only be abstracted in a limited number of ways. We return to this topic in Chapter 6 when we identify the very limited number of ways you can build abstractions.

Hardware verification	
Ground space:	higher order logic
Abstract space:	higher order logic
Provability preserving:	TI
Negation preserving:	sometimes
Theory abstraction:	sometimes
Λ/Ω-invariance:	sometimes
Δ-invariance:	yes
Reference:	[Mel87]

Example 15 (Software verification, and synthesis):

Abstraction has also been proposed as a technique for reducing the complexity of formal methods used in the verification, and synthesis of computer programs. Our notion of T*-abstractions seems very useful for describing these sorts of abstractions. There are several competing approaches to the verification and synthesis of computer programs based upon a variety of logics: sequent calculus [MW80], Martin Löf type theory [CAB*86, BvHHS90], set theory [Spi87], many-sorted algebra [BG80], rewrite rules [BD77], ... *etc.* For the purpose of this example, we will consider the algebraic approach used in CLEAR [BD77] in which a (functional) program is modelled by an **algebraic theory**: that is, by a set of equations.

> **Definition 26 (Algebraic theory)** *: An* **algebraic theory**, *T is a pair, $\langle S, E\rangle$ where S is the* **signature** *and E a set of equations.*

The signature defines the (sorted) language of T; it is a pair, $\langle \mathcal{S}, \mathcal{O}\rangle$ where $\mathcal{S}$ is a set of **sorts**, and $\mathcal{O}$ is a set of **operators** (constants and function names) together with their sorts. The set of equations define the axioms of the theory; $t_1 = t_2$ is an equation of the algebraic theory, T=$\langle S, E\rangle$ written $t_1 = t_2 \in \overline{E}$ iff it is in the set E, or it follows from this set using the rules of reflexivity, transitivity, symmetry of equality, and substitution. For example, consider an algebraic theory for part of Peano arithmetic. Its signature is defined by:

Peano	
Sorts:	$bool$, nat
Operators:	$True : bool$, $False : bool$
	$0 : nat$
	$s : nat \to nat$
	$\leq: nat \times nat \to bool$

The equations of this theory would include:

$$\begin{aligned} 0 \leq n &= True \\ s(m) \leq s(n) &= m \leq n \\ &etc. \end{aligned}$$

An axiomatic formal system can be used to describe such an algebraic theory; the language defines the signature, the axioms give the set of equations and the rules of inference are reflexivity, transitivity, symmetry of equality and substitution.

To tackle the problems of building the complex theories necessary to describe real computer programs, Burstall, Goguen [BG77] and Sannella [San82] have suggested various operations for combining together small and comprehensible theories to make larger and well structured theories; for example, we can combine together two theories, enrich a theory with new operators and equations, or derive a simpler, abstract theory from a more complex theory by ignoring unnecessary details. To describe such operations, they introduce the idea of a **theory morphism**, a mapping from one theory to another; this is defined in turn in terms of a **signature morphism**, a mapping from one signature to another. A theory morphism is a truthful or TI-abstraction.

> **Definition 27 (Signature morphism)** *: A* **signature morphism***, $\sigma : S_1 \Rightarrow S_2$ is a pair $\langle f, g\rangle$ where S_1 is the signature $\langle \mathcal{S}_1, \mathcal{O}_1\rangle$, S_2 is the signature $\langle \mathcal{S}_2, \mathcal{O}_2\rangle$, $f : \mathcal{S}_1 \mapsto \mathcal{S}_2$, $g : \mathcal{O}_1 \mapsto \mathcal{O}_2$, and g preserves arity and sorts.*

By preserving arity and sorts we mean that if $\omega \in \mathcal{O}_1$ is of sort:

$$s_1 \times ...s_m \;\rightarrow\; s_{m+1} \times ...s_n$$

then $g(\omega)$ must be of sort:

$$f(s_1) \times ...f(s_m) \;\rightarrow\; f(s_{m+1}) \times ...f(s_n)$$

A signature morphism specifies an arity and sort preserving mapping on the language of an algebraic theory; given a formulae of the theory, it tells us how to map this onto a formulae of another algebraic theory – essentially we just have to apply g to all the function names and constant symbols. Signature morphisms are extended to mappings of equations in the obvious way; that is, $\sigma(t_1 = t_2)$ equals $\sigma(t_1) = \sigma(t_2)$.

> **Definition 28 (Theory morphism)** *: A theory morphism, written $\sigma : T_1 \Rightarrow T_2$ is a signature morphism $\sigma : S_1 \Rightarrow S_2$ where T_1 is the theory $\langle S_1, E_1\rangle$, T_2 is the theory $\langle S_2, E_2\rangle$ and if $t_1 = t_2 \in \overline{E_1}$ then $\sigma(t_1 = t_2) \in \overline{E_2}$.*

A theory morphism, $\sigma : T_1 \Rightarrow T_2$ can be described by a Λ/Ω-invariant abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ where Σ_1 describes the algebraic theory T_1 and Σ_2 describes the algebraic theory T_2. The signature morphism, $\sigma : S_1 \Rightarrow S_2$ defines a mapping function on the language, in which $f(t_1 = t_2)$ equals $\sigma(t_1 = t_2)$.

> **Theorem 47** *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is the abstraction representing the theory morphism, $\sigma : T_1 \Rightarrow T_2$ then $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TI-abstraction.*
>
> **Proof:** As $\sigma : T_1 \Rightarrow T_2$ is a theory morphism, if $t_1 = t_2 \in \overline{E_1}$ then $\sigma(t_1 = t_2) \in \overline{E_2}$. That is, if $t_1 = t_2 \in TH(\Sigma_1)$ then $f(t_1 = t_2) \in TH(\Sigma_2)$. □

As with any TI-abstraction, we have to consider the problem of inconsistent abstract spaces. Consider a boolean theory with signature:

Bool	
Sorts:	*bool*
Operators:	$True : bool$, $False : bool$
	$\neg : bool \rightarrow bool$
	$\wedge : bool \times bool \rightarrow bool$
	$\vee : bool \times bool \rightarrow bool$

The equations of this theory will include:

$$\begin{aligned} p \vee \neg p &= True \\ p \wedge \neg p &= False \end{aligned}$$

The theory morphism that maps $\wedge$, and $\vee$ onto a new operator, $\circ$, will lead to an inconsistent abstract space as $p \circ \neg p = True$, $p \circ \neg p = False$ and thus $True = False$ will be equations of the abstract space. Note that if "$\circ$" is interpreted as the pair constructor, this mapping is very similar to the abstraction proposed for propositional logic in GPS.

This is perhaps not such a surprising result; if we use theory morphisms to build new (and sometimes) bigger theories, we will occasionally introduce inconsistency. Burstall and Goguen suggest [BG77] that programs might be viewed as theory morphisms from one theory, the specification to another theory, the machine which consists of the primitive operators and sorts of a programming language. They are thereby saved from problem of inconsistent abstract spaces as the abstract theory of the machine is given and is, we presume, consistent. Note that this is in the opposite direction to Melham's abstractions for hardware verification when we map *from* the implementation to the behavioural specification.

Theory morphisms can describe the construction of theories in a structured way. For example, if we add equations or extend the language of a theory, T_1 to give a new theory, T_2 the relationship between the two theories can be described by a theory morphism, $\sigma : T_1 \Rightarrow T_2$. Note that T_2 is, in these case, a more elaborate theory than T_1. We can also build new theories that abstract away irrelevant details, and are therefore less complex; theory morphisms can be used to describe such mappings in a very interesting way.

Consider a signature morphism from one language to a *less* abstract one. For example, consider a signature, S for a theory of the 26 lexically ordered characters:

Char	
Sorts:	$bool, char$
Operators:	$True : bool,\ False : bool$
	$A : char,\ B : char,\ \ldots,\ Z : char$
	$\preceq : char \times char \rightarrow bool$

The operator "$\preceq$" is meant to be a total ordering on the sort $char$. We can define a signature morphism from **Char** onto **Peano**, the signature of Peano arithmetic by the pair, $\langle f, g \rangle$ where:

$$\begin{array}{lll} f(bool) = bool & f(char) = nat & \\ g(True) = True & g(False) = False & g(\preceq) = \leq \\ g(A) = 0 & g(B) = s(0) & \ldots\ldots \quad etc. \end{array}$$

If we use T to represent the theory of Peano arithmetic, and $\overline{E}$ to represent the equations of T, then this signature morphism defines an abstract theory, the

quotient theory, T/σ with a signature S and equations:

$$\overline{E/\sigma} \quad = \quad \{e|\sigma(e) \in \overline{E}\}$$

For example, since $0 \leq s(0)) = True$, and $0 \leq s(s(0)) = True$ are equations of T, $A \preceq B = True$, and $A \preceq C = True$ are equations of T/σ. The relationship between the ground theory T and the abstract quotient theory T/σ can be described by the theory morphism $\sigma : T/\sigma \Rightarrow T$ which maps *from* the abstract quotient theory T/σ *to* the ground theory T. Note that this is using a TI-abstraction in the opposite way to usual. It neatly avoids the problem of inconsistent abstract spaces since if T is consistent then T/σ is guaranteed to be also. Indeed, this mapping is a TC-abstraction and therefore throws away no information.

Theorem 48 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is the abstraction representing the theory morphism $\sigma : T/\sigma \Rightarrow T$ then $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TC-abstraction.*

Proof: By the definition of the quotient theory, $t_1 = t_2 \in \overline{E/\sigma}$ iff $\sigma(t_1 = t_2) \in \overline{E}$. That is, $t_1 = t_2 \in TH(\Sigma_1)$ iff $f(t_1 = t_2) \in TH(\Sigma_2)$. $\Box$

Note that the ground space can be larger than the abstract. Indeed it can even have more theorems. The abstract quotient theory is isomorphic to a *sub*theory of the ground space. Any equation in the ground space outside the image of the abstract space's signature will not be related to an equation of the abstract space. For example, no equation of the theory of lexically ordered characters maps onto the equation $42 \preceq s(42) = True$ of Peano arithmetic (where "42" is syntactic sugar for the term $s(s(...0...))$ which has 42 applications of the operator, s applied to 0).

Theory morphisms	
Ground space:	algebraic theory
Abstract space:	algebraic theory
Provability preserving:	TI
Negation preserving:	sometimes
Theory abstraction:	sometimes
Λ/Ω-invariance:	sometimes
Δ-invariance:	yes
Reference:	[BG77]

4.7 Summary

The properties of these abstractions are summarised in fig. 4.1. We will make some general remarks about this table. We believe that this is a representative sample of abstractions and that our observations will therefore be true of abstractions in general.

First, all the examples (except gazing) preserve provability in one way or another; almost all are, in fact, TI*-abstractions. This supports our emphasis on provability and inconsistency preserving abstractions, and most especially on TI*-abstractions. Second, the majority of these examples are negation preserving and theory abstractions. Abstraction is mostly used to control combinatorial explosion in the theory not the logic. Indeed, we would argue that you should only change the logical structure of wffs with great care. A logic is, in general, very finely balanced - the slightest changes can bring the whole formal structure crashing down. Third, almost all the abstractions are Δ-invariant; that is, they use the same deductive machinery in the ground and the abstract spaces. This makes implementation very economical; it also allows us to use hierarchies of abstractions. Fourth, nearly all abstractions are Λ/Ω-invariant. This guarantees that the abstraction of any axiom of the ground space is itself a theorem of the abstract space. Finally, most of the (theory) abstractions can be characterised by whether they map the terms or the predicate names. Since theory abstractions can only map the atomic formulae, this is perhaps not so surprising.

ABSTRACTION	T*/NT*-abs	negation pres	theory abs	Δ-invar	Λ/Ω-invar	maps **P**red/ **T**erm
ABSTRIPS	TI/NTI	$\surd$	$\surd$	$\surd$	$\surd$	N
GPS	TI	$\times$	$\times$	$\times$	$\surd$	N
Weak	NTI	?	?	$\surd$	$\surd$	?
Ordinary	NTI	?	?	$\surd$	$\surd$	?
Generalisation	TI/NTI	$\surd$	$\surd$	$\times$	$\surd$	T
Gazing	neither	$\times$	$\times$	$\surd$	$\surd$	T
Connection methods	TI/NTI	$\surd$	$\surd$	$\times$	$\surd$	T
Propositional Decider	TD/NTD	$\surd$	$\times$	$\surd$	$\surd$	N
Variable-free	TI/NTI	$\surd$	$\surd$	$\surd$	$\surd$	T
Granularity	TI/NTI	$\surd$	$\surd$	$\surd$	$\surd$	T
Imielinski I	TI	n/a	$\surd$	$\surd$	$\surd$	T
Imielinski II	TD	n/a	$\surd$	$\surd$	$\surd$	T
Predicate	TI/NTI	$\surd$	$\surd$	$\surd$	$\surd$	P
Restricted predicate	TD/NTD	$\surd$	$\surd$	$\surd$	$\surd$	P
Hardware verification	TI	?	?	$\surd$	?	?
Theory morphism	TI	?	?	$\surd$	?	?

Figure 4.1: A summary of the properties of these examples

Notes:

1. In the column headings, “negation pres” stands for negation preserving, “theory abs” for theory abstraction, “invar” for invariance, and “maps **P**red / **T**erm” for maps the predicate names or the terms.

2. In the table entries, “$\surd$” means that this abstraction has this property, “$\times$” that this abstraction does not have this property, “?” that abstraction can have this property (but does not necessarily have to), “n/a” that this property is not relevant for this abstraction, “P” that this abstraction maps just the predicate names, “T’ that this abstraction maps just the terms, and “N” if neither is true.

3. Restricted predicate abstractions satisfy a property very close to that of TD-abstractions. It is only *very restricted predicate abstractions* that are actually TD.

Chapter 5

Abstraction and Inconsistency

In this Chapter we explore how abstraction affects consistency. We identify the **problem of inconsistent abstract spaces***, where abstractions map consistent ground spaces onto inconsistent abstract spaces. We demonstrate the inevitability of this problem and consider ways to avoid it.*

5.1 Introduction

We informally defined abstraction as "*a mapping between representations of a problem which preserves certain desirable properties*". Since we have represented problems with formal systems, we have concentrated on how abstractions preserve one very important property of formal systems, that of provability. Another very important property of formal systems is consistency. Indeed, we have already defined classes of abstractions which preserve inconsistency (NT*-abstractions). Our motivation, however, was to describe mappings between refutation systems; in Chapter 2, we demonstrated that inconsistency preserving abstractions between refutation systems play the same rôle as provability preserving abstractions between proof systems. We have yet to consider how provability preserving abstractions affect consistency.

The main part of this Chapter appears in [GW89c].

Various people have observed that a consistent ground space can sometimes map onto an inconsistent abstract space [Nil80, Pla80, Ten87]. This problem was first identified by Nilsson [Nil80] for ABSTRIPS abstractions. Tenenberg [Ten87] identified the same problem for predicate abstractions, misleadingly calling it the "false proof problem". Plaisted [Pla81] introduced the term **false proof** to describe abstract proofs which do not correspond to any ground proof. False proofs have nothing directly to do with the inconsistency of the abstract space; false proofs can occur even when the abstract space is consistent. To avoid further confusion, we shall use the term, "**the problem of inconsistent abstract spaces**" to describe when a consistent ground space is mapped onto an inconsistent abstract space. The aim of this Chapter is to understand what causes this problem, how prevalent it is, and how we can avoid it.

5.2 The Problem

It is perhaps not too surprising that abstraction can lead to inconsistency. Abstraction is about throwing away information. In order to build an abstract space simpler than the ground space, we need to forget some *irrelevant* details; we keep around just those details that are judged important. The problem is that these irrelevant looking details may be exactly what is preserving the theory from inconsistency; this makes these details rather *relevant*.

The problem occurs with many types of abstractions. For example, it can occur with ABSTRIPS abstractions. Nilsson demonstrates (figure 8.13 on page 353 of [Nil80]) an abstract space in which the contradictory facts $ON(A, C)$ and $ON(C, A)$ both hold. As a second example, Tenenberg (page 1012 of [Ten87]) shows how a predicate abstraction can give an inconsistent abstract space. Tenenberg works in a refutation system so his problem is not an inconsistent abstract space (which a refutation system seeks when trying to prove an abstracted goal) but that the abstraction of the axioms without the negated goal is inconsistent. In such a situation, any abstracted goal can be proven.

Is the inconsistency of the abstract space really so bad? The answer is not immediately obvious. For instance, Nilsson states (page 352 of [Nil80]) that "*... A contradictory state description may result, but this causes no problems. ...*". Tenenberg (page 39 of [Ten88]) argues to the contrary, "*... once such a situation is reached, there are no constraints on the future choice of actions. ...*". Part of the problem is that testing consistency is not, in general, decidable. We may therefore simply decide not to worry about the consistency of the abstract space. Even if the abstract space is inconsistent, the structure of an abstract proof can still be used to guide theorem proving in the ground space since many branches of the abstract proof will not call upon the inconsistency. Additionally, if we exploit only a little information from the abstract space, the chances of using inconsistent information are slight. This is true in Nilsson's example: the inconsistent abstract space is not abstracted further and extensive reasoning is not performed in the inconsistent abstract space.

From a theoretical point of view, an inconsistent abstract space is undesirable. Inconsistency is something we usually try to avoid if at all possible. More importantly, it is a symptom of a deeper problem that calls into the question the value of the abstraction. To be of practical use, we want an abstraction to collapse together objects with similar properties; an abstraction which collapses together wildly dissimilar objects is of dubious value. An inconsistent abstract space tells us that the abstraction has actually mapped together objects which have contradictory properties. This suggests that our abstraction is *fundamentally misguided*.

From a practical point of view, an inconsistent abstract space can lead to inefficiency. Abstract proofs which call upon the inconsistency of the abstract space will not help the search for ground proofs. Of course, if the proof of the inconsistency of the abstract space is very complex, this might not be a very significant problem. Additionally, even if the abstract space is consistent, we have to cope with abstract proofs which do not help the search for ground proofs. For example, with an abstraction, $f : \Sigma_1 \Rightarrow \Sigma_2$ which is TI but not TC, there will be abstract proofs will do not correspond to any ground proof (what Plaisted called "false proofs"). The inconsistency of the abstract space just enlarges the problem of false proofs since every abstract wff is a theorem; the area of $TH(\Sigma_2) - f(TH(\Sigma_1)$ grows to fill the whole of $\Lambda_2 - f(TH(\Sigma_1)$.

One final consideration is that knowing the consistency of the abstract space can help to reduce search significantly. Consider, for example, a TI-abstraction. If the abstract space is consistent then any abstract wff whose negation is provable *cannot* be a theorem of the abstract space. Since the abstraction is truthful, any ground wff which abstracts onto this formula cannot therefore be a theorem of the ground space. Thus, we can prune from the ground search space any wff which abstracts onto a formula which is not provable in the abstract space. Since theorem proving should be easier in the abstract space, this might save us a lot of effort. In an inconsistent abstract space, there are no wffs which cannot be proved; there is therefore no possibility for pruning goals from the ground search space.

Our first task in tackling the problem of inconsistent abstract spaces is to explore its prevalence. In particular, we want to see if the problem occurs with all types of provability preserving abstractions, or alternatively if it is an inevitable risk of using only certain types of abstractions.

5.3 TD*-abstractions and Inconsistency

Implicit in most work in abstraction is the assumption that the ground space is consistent. Under this assumption, TD*-abstractions always give consistent abstract spaces.

Theorem 49 *: If Σ_1 is consistent, Σ_2 is a system with negation and $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TD*-abstraction then Σ_2 is consistent.*

Proof: By contradiction. Assume that Σ_2 is inconsistent. Let Λ_2 be the language of Σ_2. Since Σ_2 is a system with negation, $TH(\Sigma_2) = \Lambda_2$. Hence, for any wff φ, $f(\varphi) \in TH(\Sigma_2)$. Since $f : \Sigma_1 \Rightarrow \Sigma_2$ is TD, it follows that for any wff φ, $\varphi \in TH(\Sigma_1)$. This contradicts the assumption that Σ_1 is consistent. The proof for NTD-abstractions is dual. $\Box$

As a TC*-abstraction is also a TD*-abstraction, a simple corollary is that a TC*-abstraction also maps a consistent ground space onto a consistent abstract space. Thus, a simple solution to the problem of inconsistent abstract spaces is to use only TD*- or TC*-abstractions. However, as we argued before, TC*-abstractions are too strong; they do not allow us to throw away enough information. TD*-abstractions, on the other hand, lose us completeness since not all the theorems of the ground space are mapped onto theorems of the abstract space; theorem proving in the abstract space will not therefore help us find all the theorems of the ground space.

5.4 TI*-abstractions and Inconsistency

Unlike TD*-abstractions, TI*-abstraction do not always preserve the consistency of the ground space. Several examples of TI*-abstractions with inconsistent abstract spaces can be found in Chapter 4. For instance, we demonstrated how both Hobbs' granularity and Tenenberg's predicate abstraction can map a consistent ground space onto an inconsistent abstract space. For the ABSTRIPS abstraction, consider a mapping which drops the second precondition of the two operators "$\alpha_1 \wedge \alpha_2 \rightarrow \alpha_3$" and "$\alpha_1 \wedge \alpha_4 \rightarrow \neg\alpha_3$", where α_1 is a theorem, and α_2 and α_4 are not both theorems. Examples of TI*-abstractions which preserve consistency can be trivially found (*eg.* the identity abstraction).

The fact that TI*-abstractions can give inconsistent abstract spaces is a major blow. Note that the real problem is not the inconsistency of the abstract space, rather its absolute inconsistency; this distinction is rarely pointed out as in most cases the two concepts collapse. When working with a fixed ground space (*eg.* set theory and first order logic) one solution would be to build abstractions which are proved *a priori* to construct a consistent abstract space. Often, however, the axioms are not fixed in advance (*eg.* in logic programming, knowledge based systems, etc.). In such situations, a solution would be to find conditions which guarantee that, whatever the axioms, the abstraction maps a consistent ground space onto a consistent abstract space. Unfortunately, this demand turns out to be unsatisfiable. Indeed, we make the following (and very damaging) claim:

> **Claim** : *For every TI*-abstraction, there exists a consistent ground space which maps onto an inconsistent abstract space.*

To justify this statement, we first need to introduce some notation for describing abstractions in which the axioms are not fixed in advance; this is the purpose of the next section.

5.5 Abstraction Schemata

In many cases, abstractions are defined so that they will work with any choice of axioms. Usually, the languages and deductive machineries of the ground and abstract space are fixed, and some means of constructing the axioms of the abstract space are provided given axioms for the ground space. Often, the axioms of the abstract space are generated by simply applying a mapping to the axioms of the ground space. For example, with Plaisted's abstractions, the languages are fixed (first order clauses), the deductive machineries are fixed (resolution), and the axioms of the abstract space are generated by applying the abstraction's mapping function to the axioms of the ground space. We can describe such a situation with an **abstraction schema**; this is a lambda abstraction (in the λ-calculus sense) which, when given a set of axioms for the ground space, returns an abstraction from the given ground space to some new abstract space.

> **Definition 29 (Abstraction schema)** : *An abstraction schema, $\mathcal{F}$ is a lambda abstraction of the form:*
>
> $$\lambda x \,.\, f : \Sigma_1(x) \Rightarrow \Sigma_2(x)$$
>
> *where*
>
> $$\Sigma_1(x) = \langle \Lambda_1, x, \Delta_1 \rangle$$
>
> $$\Sigma_2(x) = \langle \Lambda_2, \{g(\varphi) | \varphi \in x\}, \Delta_2 \rangle$$

An abstraction schema is a function, $\mathcal{F} : 2^{\Lambda_1} \mapsto \mathcal{ABS}$. It is completely specified by giving Λ_1, Λ_2, Δ_1, Δ_2, $f : \Lambda_1 \mapsto \Lambda_2$ and $g : \Lambda_1 \mapsto \Lambda_2$.

The abstraction formed by application of a set of axioms to an abstraction schema, $\mathcal{F}$ and beta-reduction is called an **instantiation** of $\mathcal{F}$. We say that an abstraction schema has property $\mathcal{P}$ iff all its instantiations have property $\mathcal{P}$. For example, an abstraction schema is truthful iff all its instantiations are truthful abstractions. Abstraction composition, and equality can be extended to abstraction schemata in the obvious way. An abstraction schema will be Λ/Ω-invariant iff $f = g$. Note that not all abstraction schemata will have $f = g$. For example, an abstraction schema which describes Tenenberg's restricted predicate abstractions uses different functions for mapping the language and the axioms of the ground space.

5.6 The Inevitability of Inconsistency

As we claimed earlier, inconsistent abstract spaces are inevitable whenever we use TI*-abstractions. To be more precise, under certain weak conditions, every TI*-abstraction schema has an instantiation which maps a consistent ground space onto an inconsistent abstract space.

Theorem 50 *: Let $\mathcal{F}$ be an abstraction schema, all of whose instantiations are negation preserving TI-abstractions. Let one instantiation be not TC, have a consistent ground space and monotonic abstract space. Then there exists an instantiation of $\mathcal{F}$, $f : \Sigma'_1 \Rightarrow \Sigma'_2$ for which Σ'_1 is consistent but Σ'_2 is inconsistent.*

Proof: Let $f : \Sigma_1 \Rightarrow \Sigma_2$ be an instantiation of $\mathcal{F}$ which is not TC. There exists a wff, φ such that $f(\varphi) \in TH(\Sigma_2)$ but $\varphi \notin TH(\Sigma_1)$. Construct Σ'_1 from Σ_1 by adding $\neg\varphi$ to the axioms of Σ_1. Consider the instantiation of $\mathcal{F}$ with this ground space, $f : \Sigma'_1 \Rightarrow \Sigma'_2$. Now Σ'_1 is consistent as $\varphi \notin TH(\Sigma_1)$. Since $\neg\varphi$ is an axiom of Σ'_1, $\neg\varphi \in TH(\Sigma'_1)$.

As all the instantiations of $\mathcal{F}$ are TI, $f(\neg\varphi) \in TH(\Sigma_2')$. Because all the instantiations of $\mathcal{F}$ are also negation preserving, $\neg f(\varphi) \in TH(\Sigma_2')$. However, $f(\varphi) \in TH(\Sigma_2')$ as Σ_2' is a monotonic extension of Σ_2. Thus a wff and its negation are both derivable in Σ_2'. In other words, Σ_2' is inconsistent. □

Most abstraction schemata used in the past (*eg.* Plaisted's) satisfy the hypotheses of this theorem; their instantiations are all negation preserving TI-abstractions, and (at least) one instantiation is not a TC-abstraction. This second condition is important as it guarantees that abstraction makes the problem solving easier. With TC-abstractions, the theorems of the abstract space are exactly the abstraction of the theorems of the ground space; the abstract space is not therefore any "simpler" than the ground space. We can give a very similar result for NTI-abstractions since, by theorem 6, any negation preserving NTI-abstraction between systems with negation is also a TI-abstraction.

Theorem 50 demonstrates *the inevitability of inconsistent abstract spaces* for a very large and common class of abstractions. In order to avoid inconsistent abstract spaces, we might therefore decide against using this class of abstractions; Tenenberg advocates such a change in [Ten87]. He proposes a class of abstractions which are not TI*-abstractions but which are guaranteed to generate consistent abstract spaces. As argued earlier, the problem with such abstractions (and all abstractions which are not TI*-abstractions) is that **completeness** is lost; there are theorems of the ground space which are not theorems of the abstract space. When the abstract space is going to be used to help find a proof in the ground space, completeness is very desirable and should only be sacrificed reluctantly.

Theorem 50 demands that *all* instantiations of an abstraction schema be TI-abstractions; this might seem rather restrictive, even though the majority of abstraction schemata used in the past satisfy this property. This demand can be weakened if we link how the wffs and axioms of the ground and abstract spaces are mapped. To be more precise, for Λ/Ω-invariant abstraction schemata we merely need to find one instantiation that is TI but not TC.

Theorem 51 *: Let $\mathcal{F}$ be a negation preserving Λ/Ω-invariant abstraction schema, which has one instantiation, $f : \Sigma_1 \Rightarrow \Sigma_2$ which is a TI-abstraction that is not TC and which has a monotonic abstract space. Then there exists an instantiation of $\mathcal{F}$, $f : \Sigma_1' \Rightarrow \Sigma_2'$ for which Σ_1' is consistent but Σ_2' is inconsistent.*

Proof: Since $f : \Sigma_1 \Rightarrow \Sigma_2$ is TI but not TC, there exists a wff, φ such that $f(\varphi) \in TH(\Sigma_2)$ but $\varphi \notin TH(\Sigma_1)$. Construct Σ_1' from Σ_1 by adding $\neg\varphi$ to the axioms of Σ_1. Consider the instantiation of $\mathcal{F}$ with this ground space, $f : \Sigma_1' \Rightarrow \Sigma_2'$. Now Σ_1' is consistent as $\varphi \notin TH(\Sigma_1)$. Since all the instantiations of $\mathcal{F}$ are Λ/Ω-invariant and $\neg\varphi$ is an axiom of Σ_1', $f(\neg\varphi)$ is an axiom of Σ_2'. That is, $f(\neg\varphi) \in TH(\Sigma_2')$. As all the instantiations are also negation preserving, $\neg f(\varphi) \in TH(\Sigma_2')$. However, $f(\varphi) \in TH(\Sigma_2')$ as Σ_2' is a monotonic extension of Σ_2. Thus a wff and its negation are both derivable in Σ_2'. In other words, Σ_2' is inconsistent. $\Box$

It was exactly this link between the mapping on the language and that on the axioms which Tenenberg sought to break in defining his class of restricted predicate abstractions. By mapping the axioms differently to the language, he was able to avoid introducing inconsistency into the abstract space. The high price he paid for this was the loss of truthfulness.

We end this section by noting that the stronger the abstraction, the greater the chance of generating an inconsistent abstract space. A stronger abstraction will add more theorems to the abstract space, increasing the chance that the negation of one of these theorems will also be provable, and thereby increasing the chance that the abstract space will be inconsistent. This confirms the intuition that the more details we throw away, the greater the risk of generating an inconsistent abstract space.

5.7 A Solution

One solution to the problem of inconsistent abstract spaces is to use an ordered hierarchy of abstractions. We defined an order, "$\leq$" on abstractions by the number of theorems in their abstract spaces. Since the number of theorems of an inconsistent abstract space is maximal, it is not surprising that this order can also be used to tackle the problem of inconsistent abstract spaces. The solution calls upon the following fact: a totally ordered set of abstractions has all those with consistent abstract spaces on the left, and those with inconsistent abstract spaces on the right. To be more precise, if an abstract space is consistent then all weaker abstractions map onto consistent abstract spaces, whilst if an abstract space is inconsistent then all stronger abstractions map onto inconsistent abstract spaces.

> **Theorem 52** *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ and $g : \Sigma_1 \Rightarrow \Sigma_3$ are two abstractions such that $g \leq f$, and Σ_2 and Σ_3 are systems with negation then if Σ_2 is consistent, Σ_3 is also consistent. Equivalently, if Σ_3 is inconsistent then Σ_2 is also inconsistent.*
>
> **Proof:** As $g \leq f$, then $g(\varphi) \in TH(\Sigma_3)$ implies $f(\varphi) \in TH(\Sigma_2)$. Assume that Σ_2 is consistent but that Σ_3 is inconsistent. There will exist a wff, α for which $f(\alpha) \notin TH(\Sigma_2)$. But $g(\alpha) \in TH(\Sigma_3)$ as all wffs are provable in an absolutely inconsistent theory. Thus, $f(\alpha) \notin TH(\Sigma_2)$ and $g(\alpha) \in TH(\Sigma_3)$. But this contradicts $g(\varphi) \in TH(\Sigma_3)$ implying $f(\varphi) \in TH(\Sigma_2)$ for all φ. Hence, if Σ_2 is consistent then Σ_3 cannot be inconsistent. And if Σ_3 is inconsistent then Σ_2 cannot be consistent.
> $\square$

This suggests a solution to the problem of inconsistent abstract spaces; we use an ordered chain of TI-abstractions, the strongest of which gives a *decidable* abstract space. If this abstract space is consistent then all the intermediate abstract spaces back to the ground space will be also. Thus we can work back through the chain of abstractions safe in the knowledge that all the intermediate (and possibly undecidable) abstract spaces are consistent. Of course, we can't escape undecidability so this trick is inevitably cautious; there will be cases where the strongest abstract space is inconsistent but the intermediate abstract spaces are consistent. The following Prolog program illustrates this solution; it constructs an ordered set of TI-abstractions that are guaranteed to give consistent abstract spaces:

```
consistent-abs(Sigma,SetOfAbs):-
        generate(Sigma,SetOfAbs),
        strongest(SetOfAbs,F),
        abstract(Sigma,F,AbsSigma),
        consistent(AbsSigma).
```

`generate(Sigma,SetOfAbs)` returns an ordered set of abstractions, `SetOfAbs` of the formal system, `Sigma`; the strongest abstraction in `SetOfAbs` should map `Sigma` into a theory in which consistency is decidable (*eg.* propositional logic);

`strongest(SetOfAbs,F)` returns the strongest abstraction, `F` in the ordered set of abstractions, `SetOfAbs`;

`abstract(Sigma,F,AbsSigma)` maps the formal system, `Sigma` onto the abstract formal system, `AbsSigma` according to the abstraction mapping, `F`;

`consistent(AbsSigma)` tests the consistency of `AbsSigma`.

Ideally, we want `generate/2` to construct the ordered sets of abstractions in an "intelligent" way. For example, by identifying the cause of inconsistency in an abstract space, we might be able to suggest ways of subtly altering the abstraction so that it does not introduce inconsistency.

We have implemented this program using an interface to Otter [McC88], a fast, state-of-the-art resolution theorem prover to test consistency of the abstract space. The complete program is listed in Appendix A. Currently, the program generates all possible predicate abstractions of a theory. It would not be difficult to make it generate other types of abstractions (*eg.* domain abstractions). A more challenging problem would be to determine which predicate symbols are worth collapsing together instead of naively considering all possibilities; this would require representing and reasoning with lots of domain and problem dependent knowledge.

5.8 Results

We have run our program on several different theories. For example, we have applied it to the theory of containers in [Ten87]. This theory was used by Tenenberg to explore the problem of inconsistent abstract spaces generated by predicate abstractions. The axioms are as follows:

$$\begin{array}{ll}
bottle(x) \rightarrow madeofglass(x) & bottle(x) \rightarrow graspable(x) \\
glass(x) \rightarrow madeofglass(x) & glass(x) \rightarrow graspable(x) \\
glass(x) \rightarrow open(x) & box(x) \rightarrow graspable(x) \\
bottle(x) \rightarrow \neg glass(x) & glass(x) \rightarrow \neg bottle(x) \\
bottle(x) \rightarrow \neg box(x) & glass(x) \rightarrow \neg box(x) \\
box(x) \rightarrow \neg glass(x) & box(x) \rightarrow \neg bottle(x) \\
bottle(x) \rightarrow milkb(x) \vee wineb(x) & open(x) \wedge graspable(x) \rightarrow pourable(x) \\
graspable(x) \rightarrow movable(x) & madeofglass(x) \rightarrow breakable(x) \\
bottle(x) \vee glass(x) \vee box(x) & open(a)
\end{array}$$

Our program was able to suggest all possible predicate abstractions of this theory that preserve consistency. For example, although *graspable* and *movable* can be safely collapsed together, collapsing *box*, *glass* and *box* together (onto a generic *container*) introduces inconsistency.

As a second example, we consider a classic challenge problem from the theorem proving literature, Schubert's steamroller [Coh85]. The axioms are as follows:

$$\begin{array}{ll}
wolf(wolfy) & wolf(x) \rightarrow animal(x) \\
fox(foxy) & fox(x) \rightarrow animal(x) \\
bird(tweety) & bird(x) \rightarrow animal(x) \\
snail(slimey) & snail(x) \rightarrow animal(x) \\
cat(crawly) & cat(x) \rightarrow animal(x) \\
grain(stalky) & grain(x) \rightarrow plant(x) \\
fox(x) \wedge wolf(y) \rightarrow smaller(x,y) & bird(x) \wedge fox(y) \rightarrow smaller(x,y) \\
snail(x) \wedge bird(y) \rightarrow smaller(x,y) & cat(x) \wedge bird(y) \rightarrow smaller(x,y) \\
\exists y.snail(x) \rightarrow plant(y) \wedge likes(x,y) & \exists y.cat(x) \rightarrow plant(y) \wedge likes(x,y) \\
bird(x) \wedge cat(y) \rightarrow likes(x,y) & wolf(x) \wedge fox(y) \rightarrow \neg likes(x,y) \\
bird(x) \wedge snail(y) \rightarrow \neg likes(x,y) & wolf(x) \wedge grain(y) \rightarrow \neg likes(x,y)
\end{array}$$

$$\begin{array}{l}
animal(x) \wedge plant(y) \wedge animal(w) \wedge smaller(w,x) \wedge plant(z) \wedge \\
\quad likes(w,z) \rightarrow likes(x,y) \vee likes(x,w)
\end{array}$$

Our program demonstrated that there is hardly any way for Schubert's steamroller to be abstracted without introducing inconsistency. For example, *plant* and *grain* can be safely abstracted together but *cat* and *snail* cannot. Although *cat*erpillars and *snail*s are animals with very similar properties, *bird*s like to eat *cat*erpillars but not *snail*s. This suggests that Schubert's steamroller contains little redundancy, and is one reason why it is so difficult to prove.

5.9 Related Work

Tenenberg [Ten87, Ten88] presents two solutions to the problem of inconsistent abstract spaces for predicate abstractions. The first solution uses restricted predicate mappings; this class of abstractions keeps in the abstract space only those axioms from the ground space that do not distinguish between predicates which are mapped together. Unfortunately determining indistinguishability is, in general, undecidable requiring an arbitrary amount of theorem proving in the ground space. Additionally, this gives an abstraction which is not truthful; since the abstract space has fewer axioms than the ground space (we don't map all of them), the abstract space has fewer theorems than the ground space and we lose truthfulness. Tenenberg's second solution [Ten88] overcomes the objection to undecidability. In this proposal, axioms are kept in the abstract space provided they can be *trivially* shown not to distinguish in the ground space between predicates which are mapped together; thus, instead of performing an arbitrary amount of theorem proving in the ground space to determine indistinguishability, we insist that it is an immediate consequence of the axioms. Whilst being decidable, this solution is again not truthful.

Tenenberg [Ten88] also proposes a solution to the problem of inconsistent abstract spaces for ABSTRIPS abstractions. This solution restricts the assignments of criticalities so that inconsistency cannot be introduced. If the ground space includes inequality, this solution is of little use as it tends to give only one criticality to all preconditions. Tenenberg proposes two modification to overcome this problem. Unfortunately the first restricts the operators we can use whilst the second is not decidable for first order theories.

5.10 Summary

We have argued at length for abstractions which preserve provability; this Chapter has considered how such abstractions affect consistency. We identified the problem of inconsistent abstract spaces, where an abstraction maps a consistent ground space onto an inconsistent abstract space. We demonstrated that this problem occurs with nearly all TI*-abstractions, but not with TD*-abstractions. We then suggested a method to avoid inconsistent abstract spaces; this method uses the partial order on abstractions introduced in Chapter 3, and is both decidable and truthful. We compared this solution with other solutions proposed in the past; all of these other solutions either introduce some undecidable test or change the abstraction so that is no longer truthful.

Chapter 6

Building Abstractions

This Chapter explores how to build abstractions automatically. We first identify the limited number of ways you can usefully abstract a new problem domain. We then concentrate on one of these ways, the abstraction used in ABSTRIPS. We suggest a method for building ABSTRIPS abstractions automatically, motivated by various desirable properties such a method should possess. We end by testing our method both empirically and theoretically.

6.1 Introduction

Although abstraction is a frequently used technique for controlling search, there has been surprisingly little work on building abstractions. In most cases, the abstraction is fixed in advance, irrespective of the domain or the particular problem to solve. The need to be able to construct abstractions tuned to new domains and problems was first recognised in ABSTRIPS [Sac74], where there was a semi-automatic method for building abstractions. For reasons of efficiency, we will not want to use the same abstraction for all domains or problems, especially if we are trying to avoid inconsistent abstract spaces. Building abstractions automatically is therefore of great interest to anyone wishing to use abstractions on real

The contents of this Chapter appear in [BGW90].

problems.

Unfortunately, very domain-specific knowledge is needed to determine what can be usefully abstracted together. Abstractions should collapse together objects in meaningful ways; for example, it is unlikely that a domain abstraction which maps the constants, "π" and "$tweety$" onto the same abstract constant is going to be of much use. Similarly, a predicate abstraction which maps the predicates, "$transcendental(x)$" and "$bird(x)$" onto the same abstract predicate will probably not be of much practical value. This does not mean, however, that we can give no general results about building abstractions automatically.

To complicate matters, we also want to be able to build abstractions that preserve provability and it is difficult to predict in advance how an abstraction will affect the rather *global* property of provability. Ideally we want to discover some *local* properties of a mapping we can test which guarantees that it preserves provability or inconsistency. Plaisted's definition of abstraction [Pla81] provides such properties for NTI-abstractions between resolution systems. However, we do not want to restrict ourselves just to clausal languages, resolution systems and NTI-abstractions.

We can separate our results into two classes: those that build provability preserving abstractions which are Δ-variant abstractions (in which we change the deductive machinery between the ground and the abstract spaces), and those that construct provability preserving abstractions which are Δ-invariant (in which the same deductive machinery is used in both the ground and the abstract spaces). Most of the results we give are for the second class. Since the same inference engine can be used in ground and abstract spaces, Δ-invariant abstractions tend to be more common. Δ-variant abstractions, on the other hand, often require a new inference engine to be built for theorem proving in the abstract space; this greatly restricts their usefulness. One exception is Σ-invariant abstractions in which the inference engine of the ground space can simply have its inference rules replaced by their abstractions.

6.2 Building Σ-invariant Abstractions

A Σ-invariant abstraction is one in which the language, axioms and inference rules are all abstracted using the same mapping function. All Σ-invariant abstractions are truthful.

Theorem 53 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a Σ-invariant abstraction then it is a TI-abstraction.*

Proof: We can map a proof tree Π_1 of φ onto a proof tree Π_2 of $f(\varphi)$ merely by applying f to every wff in the proof tree. $\Box$

This is, of course, not the only way to build truthful abstractions. Indeed, there are many truthful abstractions that are not Σ-invariant. For example, any abstraction which involves a change in the way we axiomatise the problem domain is not Σ-invariant. Σ-invariance fails to capture all truthful abstractions because the abstraction of an axiom of the ground space only needs to be a theorem of the abstract space (and not an axiom), and the abstraction of an inference rule of the ground space only needs to be a derived rule in the abstract space (and not an inference rule). To include these more powerful tests would not really help us construct truthful abstractions as they require *arbitrary* theorem proving in the abstract space. Indeed, we would make the following claim.

Claim *: Any purely syntactic definition of abstraction (like Σ-invariance or Plaisted's definition) will inevitably fail to capture the complete class of truthful abstractions.*

6.3 Building Δ-invariant Abstractions

Often (for example [Pla81]) we inherit one fixed language and inference engine for both the ground and the abstract spaces. We therefore want to build Δ-invariant abstractions. Such abstractions make it much easier to use hierarchies of abstractions. Unfortunately, there are not many general results for constructing truthful Δ-invariant abstractions since any results are very dependent on the particular set of inference rules. However, we can consider the rather general case of Δ-invariant abstractions between first order systems. Plaisted proves (theorem 2.1 in [Pla81]) some local properties that are sufficient (but not necessary) to construct an abstraction between two clausal languages so that is NTI. We can generalise this result to find some local conditions on a Δ-invariant abstraction between first order systems that makes an abstraction TI/NTI.

As we noted in Chapter 4, most abstractions used in the past are theory abstractions; that is, mappings which abstract the theory not the logic. In general, the logic is well behaved and it is the theory that needs to be simplified. Indeed, you should only change the logical structure of a wff with great care as (the consistency of) a logic is often very finely balanced; even the smallest change to the logic can prove catastrophic. There is another good reason for using theory abstractions; if the abstraction is to be truthful then it needs to preserve the meaning of the connective introduction and elimination rules. For example, since we can derive $p \wedge q$ from p and q, we need to be able to derive $f(p \wedge q)$ from $f(p)$ and $f(q)$. A theory abstraction will guarantee that deductions using the connective introduction and elimination rules remain valid deductions in the abstract space.

The majority of abstractions used in the past are also Λ/Ω-invariant; that is, mappings in which the language and the axioms are mapped identically. For an abstraction to be truthful, the abstraction of the axioms of the ground space must also be theorems of the abstract space. This is easily achieved by making the abstraction Λ/Ω-invariant. This is especially useful when the axioms of the ground space are, as is often the case, not fixed in advance.

We have therefore reduced the problem of constructing a truthful Δ-invariant abstraction to the much easier problem of deciding on a suitable mapping of atomic formulae for a Λ/Ω-invariant theory abstraction to be truthful. The question now becomes, can we come up with a syntactic test on such a mapping which guarantees that such an abstraction is truthful? As argued before, it is impossible for us to find a test that captures the whole class of truthful abstractions. However, it is possible to come up with a test that captures a very large subclass. Indeed, the test captures most of the abstractions listed in Chapter 4.

Theorem 54 *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a Λ/Ω-invariant theory abstraction, Σ_1 and Σ_2 are complete first order formal systems, and the mapping function preserves the names of all the occurrences of free and bound variables (or drops them), and preserves substitution instances (that is, $f(p[a]) = f(p[x])\{f(a)/x\}$) then $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TI/NTI-abstraction.*

Proof: Since $f : \Sigma_1 \Rightarrow \Sigma_2$ is a theory abstraction it is negation preserving. Thus it is sufficient to prove that $f : \Sigma_1 \Rightarrow \Sigma_2$ is a TI-abstraction. We show how given a proof tree, Π_1 of φ in Σ_1 we can construct a proof tree Π_2 of $f(\varphi)$ in Σ_2. The argument proceeds by induction on the depth of the proof tree Π_1.

For the base case, let $\Pi_1 = \varphi$. Now φ must be an axiom. But, as $f : \Sigma_1 \Rightarrow \Sigma_2$ is Λ/Ω-invariant, $f(\varphi)$ is also an axiom.

For the step case, assume that we can show it for all proof trees up to depth n, and prove it is true of all proof trees, Π_1' of depth $n+1$. We consider the last inference in the proof tree, Π_1'. If it is a connective introduction or elimination, a universal elimination or an existential introduction then we apply the same rule at the bottom of the proof tree we construct in Σ_2. For a reductio ad absurdum, the preservation of substitution instances guarantees that $f(\perp) = \perp$. We can therefore apply an application of reductio ad absurdum at the bottom of Σ_2. If it is a universal introduction or a existential elimination then the same rule can also be applied in Σ_2 since the conditions on applying the rule (that the assumptions do not mention the free variable being universally quantified etc) still hold as variable names are preserved.

By the substitution $\{f(a)/x\}$, we mean that the mapping function can (if wanted) abstract constants of the domain is some uniform way.

□

Actually, we can drop the requirement on preserving the name of bound variables provided we are carefully to avoid any naming clashes. If we restrict ourselves to abstraction mappings between clausal languages, theorem 54 is very similar to theorem 2.1 in [Pla81]; with Plaisted's abstractions, our extra condition on the name of variables is redundant since all wffs are already skolemised and no variable naming problems can arise. Note that preserving variable names and substitution instances does not capture all truthful theory abstractions. For example, the theory abstraction that maps $p(a)$ onto $\top$, and $p(x)$ onto $p(x)$ is truthful if $p(a)$ is a theorem of the ground space; however, this mapping does not preserve substitution instances.

So, we have found a useful syntactic condition on the mapping that guarantees truthfulness. The question now becomes what sort of mappings preserve substitution instances and preserve (or drop) variables? A systematic characterisation can be given from the recursive definition of atomic formulae:

1. abstractions where we map the **terms**. These can be further subdivided into:
   - **domain abstractions** where we map the constants together;
   - **function abstractions** where we map the function names together, or reduce their arity (by throwing away some or all of their arguments);
2. abstractions where we map the **predicates**. These can be further subdivided into:
   - **predicate abstractions** where we map the predicate names together;
   - **propositional abstractions** where we reduce their arity (by throwing away some or all of their arguments);
   - **ABSTRIPS abstractions** where we map the whole atomic formula onto $\top$ or $\perp$.

This classification describes *all* theory abstractions that can be decomposed into mappings on the individual parts of the atomic formulae. All these different types of abstractions can be found in the literature. Most, in fact, can be found in the work of Plaisted [Pla80, Pla81]. Both Hobbs [Hob85] and Imielinski [Imi87] have proposed domain abstractions. Plaisted [Pla80, Pla81] has used function abstractions on some interesting problems and Tenenberg [Ten87] has looked at predicate abstractions in some detail.

This section has identified the five major types of abstraction. Each maps a different part of the atomic formula or alternatively the whole atomic formula. This is a very important result; a major step in building abstractions automatically is to discover the different ways we can abstract a new domain. It will depend on the problem, and the choice of representation exactly what it is worthwhile mapping together. The next section considers this problem in detail for ABSTRIPS abstractions.

Once we have built up a collection of abstractions, we can use the various operations on abstractions to construct yet more abstractions. In particular, we can use the fact that the composition of two T*-abstractions (NT*-) is itself a T*-abstraction (NT*-), and that the inverse of a TI*-abstraction is a TD*-abstraction (and vice versa). Finally, to construct NT*-abstractions, we can call upon the fact that a T*-abstraction that is negation preserving is also a NT*-abstraction (and vice versa).

6.4 Building ABSTRIPS Abstractions

One very important class of theory abstractions is the class of ABSTRIPS abstractions in which we map some of the atomic formulae onto true, $\top$. ABSTRIPS [Sac73] used such abstractions to simplify its Strips-like planning domain. In this section, we will propose a method to build such abstractions *automatically* which improves upon the *semi*-automatic method used in ABSTRIPS.

The problem domain is determined by a set of operators. As in Chapter 4, we assume each has a set of preconditions, and a conclusion or effect given by an implication in a situation calculus. Note that this is not the representation used in ABSTRIPS; the situation calculus is, however, simple and sufficiently descriptive for our purposes. For example, Green [Gre69] describes an operator for a monkey moving an object z from location $x1$ to location $x2$ given an initial state of the world s with the following axiom:

$$at(z, x1, s) \wedge movable(z) \wedge empty(x2, s) \rightarrow at(z, x2, move(monkey, z, x2, s))$$

ABSTRIPS built a hierarchy of abstractions by mapping preconditions onto true, $\top$, according to their **criticality**; this is a measure of how difficult on average it is to achieve the precondition. Those preconditions which it is impossible to change are given the highest criticality. Those preconditions which it is very hard to change are given the next highest criticality. And those preconditions which it is very easy to change are given the lowest criticality. This notion of criticality is, in fact, very general and need not be restricted just to preconditions. For instance, when using a hierarchy of domain abstractions, we can define the criticality of a constant; this would be a measure of the importance of the constant in the given domain and an indicator of when it should be abstracted. As in ABSTRIPS, we want the criticality of a precondition to be independent of its arguments; that is, it should be a measure of how difficult *on average* it is to achieve a precondition. To this end, we ignore the arguments to the preconditions. Thus "$at(robot, workstation)$" is assigned the same criticality as "$at(robot, photocopier)$". If necessary, however, we could assign different criticalities to preconditions with the same predicate name.

6.5 Calculating Criticalities

We envisage an iterative process in which we assign the preconditions some default starting value and then by a series of iterations calculate their criticality. We shall represent by $C(Ops, p, n)$ the criticality of the predicate p at the n-th iteration given a set of operators Ops. Note that, unlike ABSTRIPS, we assign a criticality to every predicate, irrespective of whether it appears as a precondition to an operator or not. We will not be particularly interested in the absolute values $C(Ops, p, n)$ returns, just their relative ordering. We will motivate our choice of a criticality assignment function by identifying some desirable properties it should possess.

The first desirable property is that the assignment function should **converge** to an unique answer.

Property 1 *Convergence*

$$\lim_{n \to \infty} C(Ops, p, n) = a_p$$

Second, we would like the assignment function to be **fair**. Initially, we have no information to distinguish between the different predicates. The assignment function should not therefore discriminate between them. Every predicate should be given a uniform starting value, a_0.

Property 2 *Fairness*

$$C(Ops, p, 0) = a_0$$

Third, we want the function to be **monotonic**. Given a set of operators, if we add another operator with conclusion p then as p is easier to satisfy (we have another way of proving it) the criticality of p should go down or, at least, stay the same. And if we add a precondition, q to an operator with conclusion p then as p is more difficult to satisfy (we have to prove another precondition) the criticality of p should go up or, at least, stay the same.

Property 3 *Monotonicity*

$$C(Ops \cup \{q \to p\}, p, n) \leq C(Ops, p, n)$$

$$C(Ops \cup \{(q \wedge r) \to p\}, p, n) \geq C(Ops \cup \{r \to p\}, p, n)$$

Finally, we want the criticality assignment function to respect **impossibility**. If we have no operators with conclusion p then p will be impossible to change. Thus, it should be assigned the maximum criticality, a_{max}. We will assume here that there is a constant maximum criticality; it would be easy to generalise this to a different maximum for each iteration.

Property 4 *Impossibility*

$$C(Ops, r, m) = a_{max}$$

if Ops contains no operator with conclusion, r and $m > 0$.

There are other desirable properties, like **selectivity** (the assignment function should return a range of values), which are less precise in their definition and whose truth may depend very precisely upon the given operators. This is also, by no means, an exhaustive list of properties that a criticality assignment function should possess. For example, we might also want the function to be **inexpensive**; that is, we might want the function to be cheap to calculate. Nevertheless, we would argue that these all the properties on this list are *necessary.* This list of properties does not specify an unique criticality assignment function. For example, the uniform criticality assignment function (for which $\forall Ops, p, n \,.\, C(Ops, p, n) = a_0$) satisfies all four of the defined properties, and is very inexpensive to calculate. It is, however, not of much real use.

6.6 A Solution

In this section, we propose a criticality assignment function that satisfies all the properties defined in the last section; it also appears to work well empirically. This function is based upon interpreting $C(Ops, p, n)$ as the difficulty of finding a proof of p of up to depth n.

$C(Ops, p, n)$ is defined recursively as a function of the difficulty of finding a proof of up to depth $n-1$ plus the difficulty of finding a proof of *exactly* depth n, which we represent by $D(Ops, p, n)$. We combine these difficulties by analogy to the calculation of parallel resistance. With two resistors in parallel, the current can go through either the first resistor or the second, thereby reducing the total resistance. Similarly, we can find either a proof of up to depth $n-1$, or we can find a proof of exactly depth n. Thus the difficulty of finding a proof of up to depth n is the parallel sum of the difficulties of finding a proof of up to depth $n-1$ and of finding a proof of exactly depth n; this gives us equation (6.2). The starting values for the calculation of criticalities are given by equation (6.1); this equation guarantees the fairness of our solution.

$$C(Ops, p, 0) = a_0 \tag{6.1}$$
$$\frac{1}{C(Ops, p, n)} = \frac{1}{C(Ops, p, n-1)} + \frac{1}{D(Ops, p, n)} \tag{6.2}$$

This still leaves us to decide how to calculate $D(Ops, p, n)$, the difficulty of finding a proof of exactly depth n. This is also defined recursively. The boundary conditions are easy. $D(Ops, p, 0)$ is the difficulty of finding a proof of depth 0; this must equal $C(Ops, p, 0)$ (equation (6.3)). The other boundary condition is when no operator has p as a conclusion. Since it will be impossible to change such a predicate, we set this difficulty to ∞ (equation (6.4)); these predicates are often *type* predicates which specify fixed properties of objects in the domain like "$robot(x)$" or "$movable(y)$". Equation (6.4) ensures that the impossibility property holds.

The step equations for $D(Ops, p, n)$ are more complicated. The difficulty of finding a proof of p of exactly depth n, that is $D(Ops, p, n)$ is the parallel sum of the difficulties of finding a proof of depth n that ends with an operator with p as conclusion (equation (6.5)). $D(Ops, q \rightarrow p, n)$ represents the difficulty of finding a proof of p of depth n that ends with the application of the operator, $q \rightarrow p$. Note that q itself might be a conjunction of preconditions, $\wedge q_i$. Finally the difficulty of finding a proof of p of depth n that ends with the operator $\wedge q_i \rightarrow p$, that is $D(Ops, \wedge q_i \rightarrow p, n)$ is at least as difficult as finding a proof of depth $n-1$ of the most difficult precondition to that operator, that is $\max\{D(Ops, q_i, n-1)\}$ (equation (6.6)). Here, we have made the simplifying assumption that the difficulty is dominated by the most difficult precondition; a more thorough analysis would also consider the difficulties of proving the less difficult preconditions. However, such a calculation could be very expensive; taking the maximum should give a good lower bound.

$$\begin{aligned}
D(Ops, p, 0) &= a_0 && (6.3)\\
D(Ops, r, m) &= \infty && (6.4)\\
\frac{1}{D(Ops, p, n)} &= \sum_{q \to p \in Ops} \frac{1}{D(Ops, q \to p, n)} && (6.5)\\
D(Ops, \wedge q_i \to p, n) &= \max\{D(Ops, q_i, n-1)\} && (6.6)
\end{aligned}$$

where Ops contains no operator with r as conclusion, and $m > 0$

There are many other criticality assignment functions we could consider. The quality of the criticality assignments will always depend on the amount of computation we are prepared to perform. We propose this function as it seems a good example. It satisfies all the theoretical properties, is easy to compute, and behaves well on real problems. It is not, however, the only or indeed the optimal criticality assignment function.

6.7 Some Worked Examples

We have compared this method for calculating criticalities with the original method used in ABSTRIPS. We will consider two problems: McCarthy's famous Monkey and Bananas problem [Gre69] and the robot operators used in ABSTRIPS [Sac74].

For McCarthy's Monkey and Bananas problem, we use the operators given in [Gre69]; these operators are listed in Appendix B. The results are summarised below; note that we have divided every entry by a_0.

$C\ /a_0$	$n = 0$	1	2	3	∞
at	1.00	0.25	0.25	0.25	0.25
has	1.00	0.50	0.33	0.25	0.25
reachable	1.00	0.50	0.33	0.33	0.33
on	1.00	0.50	0.50	0.50	0.50
movable	1.00	1.00	1.00	1.00	1.00
empty	1.00	1.00	1.00	1.00	1.00
climbable	1.00	1.00	1.00	1.00	1.00

The criticality assignment function converges to an answer rapidly. Indeed, by the third iteration it reaches its final values. The relative order of criticalities is almost identical to that assigned by the original semi-automatic method used in ABSTRIPS. The only difference is that the method used in ABSTRIPS assigns "*at*" (where objects are at) a lower criticality than "*has*" (what the robot has). With our criticality assignment function, the criticality of "*at*" falls faster than that of "*has*" as we iterate n. However, they both end up with the same final criticality.

For the ABSTRIPS robot operators, we assume as in ABSTRIPS that the effects of an operator are dominated by one clause, the *primary addition*; other consequences of the operator are ignored for planning purposes. In this way, trivial side-effects to the operators are not given greater importance than they deserve. We can therefore model each operator as a set of preconditions implying just *one* conclusion. It would, however, be possible to extend our criticality assignment method to operators with multiple conclusions. The operators are listed in Appendix B. Our results are summarised in the following table.

$C\ /a_0$	$n = 0$	1	2	∞
nextto	1.00	0.20	0.20	0.20
inroom	1.00	0.33	0.33	0.33
at	1.00	0.33	0.33	0.33
status	1.00	0.33	0.33	0.33
type	1.00	1.00	1.00	1.00
connects	1.00	1.00	1.00	1.00
pushable	1.00	1.00	1.00	1.00
locinroom	1.00	1.00	1.00	1.00

Again the criticality assignment function converges rapidly to an answer, and the relative order of criticalities is almost identical to the relative order used in

ABSTRIPS. The only difference is that in ABSTRIPS the "*status*" of the door (open or closed) was assigned a lower criticality than "*inroom*" (the objects in the room). The reason for this is that, since we ignore the arguments to the predicates, there appear to be two operators for achieving "*inroom*". In fact, one is for the robot to go through a door and the other is for an object to be pushed through a door. This makes "*inroom*" appear easier to satisfy than it actually is. If we calculated the criticalities with just one operator for going through doors, then "*inroom*" would be given a higher criticality than "*status*".

6.8 Properties of this Solution

In the last section, we demonstrated that our criticality assignment function seems to behave well empirically. It also possesses all the theoretical properties defined in Section 6.5. First, it is a fair assignment function.

Theorem 55 *: $C(Ops, p, n)$ is fair.*

Proof: By equation (6.1). □

To show that it is convergent, we show that it is finitely bounded and monotonic with respect to n.

Theorem 56 *: $C(Ops, p, n) \leq C(Ops, p, n-1)$*

Proof: From equation (6.2) and the fact that $D(Ops, p, n)$ is positive, $\frac{1}{C(Ops,p,n)}$ increases monotonically with n. Thus, $C(Ops, p, n)$ decreases monotonically with n. □

Theorem 57 *: $C(Ops, p, n) \in [0, a_0]$*

Proof: By Theorem 56, $C(Ops, p, n)$ decreases monotonically with n. By equation (6.1), $C(Ops, p, n)$ starts at a_0. It is also bounded below by 0. Hence, $C(Ops, p, n) \in [0, a_0]$. □

Theorem 58 *: $C(Ops, p, n)$ is convergent.*

Proof: Any monotonic bounded sequence is convergent. □

In fact, this assignment function possesses an even stronger convergence property; if all the criticality assignments stay constant for one iteration, they will not change any further. Thus, we can stop calculating criticalities once they have remained constant for one iteration.

Theorem 59 *: If for some n, and all predicates p, $C(Ops, p, n) = C(Ops, p, n-1)$ then $C(Ops, p, m) = C(Ops, p, n)$ for $m \geq n$.*

Proof: From equation (6.2), $D(Ops, p, n) = \infty$ for all predicates, p. By induction on m, from equations (6.6) and (6.5), $D(Ops, p, m) = \infty$ for $m \geq n$. Thus by induction again on m and equation (6.2), $C(Ops, p, m) = C(Ops, p, n) = a_p$ for $m \geq n$. □

This criticality assignment function is monotonic with respect to the operators and the preconditions; adding an operator decreases the criticality of the conclusion, and adding a precondition increases the criticality of the conclusion.

Theorem 60 *: $C(Ops \cup \{q \rightarrow p\}, p, n) \leq C(Ops, p, n)$*

Proof: If $q \rightarrow p \in Ops$ then $Ops \cup \{q \rightarrow p\} = Ops$ and $C(Ops \cup \{q \rightarrow p\}, p, n) = C(Ops, p, n)$. Otherwise, by equation (6.5), $\frac{1}{D(Ops \cup \{q \rightarrow p\}, p, n)} = \sum \frac{1}{D(Ops, q \rightarrow p, n)} + \frac{1}{D(Ops \cup \{q \rightarrow p\}, q \rightarrow p, n)}$. But $\frac{1}{D(Ops, p, n)} = \sum \frac{1}{D(Ops, q \rightarrow p, n)}$ and $D(Ops \cup \{q \rightarrow p\}, q \rightarrow p, n)$ is positive. Hence, $\frac{1}{D(Ops \cup \{q \rightarrow p\}, p, n)} \geq \frac{1}{D(Ops, p, n)}$. Using induction on n and equation (6.1) for the base case and equation (6.2) for the step case, $\frac{1}{C(Ops \cup \{q \rightarrow p\}, p, n)} \geq \frac{1}{C(Ops, p, n)}$. That is, $C(Ops \cup \{q \rightarrow p\}, p, n) \leq C(Ops, p, n)$. □

Theorem 61 *:*

$$C(Ops \cup \{q \wedge r \rightarrow p\}, p, n) \geq \max\{C(Ops \cup \{q \rightarrow p\}, p, n), C(Ops \cup \{r \rightarrow p\}, p, n)\}$$

Proof: By induction on n. The base case follows trivially from (6.1). The step case uses equation (6.2) and the fact that $D(Ops \cup \{q \wedge r \rightarrow p\}, p, n) \geq \max\{D(Ops \cup \{q \rightarrow p\}, p, n), D(Ops \cup \{r \rightarrow p\}, p, n)\}$. This last fact follows from the lemma that $D(Ops \cup \{q \wedge r \rightarrow p\}, s, n) \geq \max\{D(Ops \cup \{q \rightarrow p\}, s, n),\ D(Ops \cup \{r \rightarrow p\}, s, n)\}$ where s is any predicate or operator. This lemma is itself proven using induction on n. The base case follows trivially from equation (6.3). For the step case, we do a case analysis. If s is a predicate then from equation (6.5) and the induction hypothesis, $\frac{1}{D(Ops \cup \{q \wedge r \rightarrow p\}, s, n)} \leq \min\{\frac{1}{D(Ops \cup \{q \rightarrow p\}, s, n)}, \frac{1}{D(Ops \cup \{r \rightarrow p\}, s, n)}\}$. That is $D((Ops \cup \{q \wedge r \rightarrow p\}, s, n) \geq \max\{D(Ops \cup \{q \rightarrow p\}, s, n),\ D(Ops \cup \{r \rightarrow p\}, s, n)\}$. If, on the other hand, s is an operator then from equation (6.6) and the induction hypothesis $D(Ops \cup \{q \wedge r \rightarrow p\}, s, n) \geq \max\{D(Ops \cup \{q \rightarrow p\}, s, n),\ D(Ops \cup \{r \rightarrow p\}, s, n)\}$. $\Box$

Finally, we note that this criticality assignment function respects impossibility.

Theorem 62 *: $C(Ops, p, n)$ respects impossibility.*

Proof: If Ops contains no operator with r as a conclusion then by equation (6.4), $D(Ops, r, m) = \infty$ for $m > 0$. Thus, by equation (6.2) and induction on m, $C(Ops, r, m) = C(Ops, r, 0) = a_0$. Note that from theorem 57, a_0 is the maximum criticality assigned to any predicate. $\Box$

6.9 Related Work

ABSTRIPS had a semi-automatic method for calculating criticalities [Sac74]. This method needed to be given a partial order on the preconditions which was used to determine an order in which to examine the preconditions. Those preconditions which cannot be changed by any operators are given the highest criticality (*cf.* the impossibility property). Each remaining precondition in the partial order is considered in turn; if a short plan can be found to achieve it assuming all the previous preconditions are considered true, then it is assigned a criticality equal to its rank in the partial order. If not, it is assigned a criticality greater than the highest rank in the partial order. The slight difference in the order of criticality assignments between this method and ours actually seems more a criticism of the need to supply a partial order with which to examine the preconditions; a different partial order often produces different criticality assignments.

Tenenberg [Ten88] has also explored various methods for building ABSTRIPS like abstractions. He proposes that predicates should be given the same criticality if they appear together in a *static* axiom (an axiom that does not change between situations). This effectively partitions the theory into independent subtheories. Unfortunately such a partitioning is usually not very selective. For example, a theory with inequality will usually give only one partition or criticality assignment. Tenenberg suggests a solution to this problem which constrains the use of inequality in the axioms. However, the operators of the domain must also satisfy the *usability* condition; that is, the preconditions to (the ground instantiation of) every operator must hold in some legal situation. For many operators, this will not be true. For example, an operator for stacking blocks can never stack a table onto a block; this operator does not therefore satisfy the usability condition. Tenenberg proposes a way round this problem which essentially prohibits type preconditions from being abstracted at any level of abstraction (*cf.* the impossibility property). Unfortunately, this solution is not decidable for first-order theories.

Knoblock [Kno89] has also suggested a method for calculating criticalities automatically. He uses the operators of a domain to build a directed graph; the vertices of this graph represent the literals, whilst the edges represent constraints on their criticalities. A directed edge from one literal to another indicates that the first literal must have a criticality equal to or greater than the second literal. His algorithm places directed edges between all effects of an operator, and between the effects and the preconditions of an operator. Unfortunately, these constraints usually only specify a partial order on the criticality assignments. A total order is generated by topologically sorting the graph in some *arbitrary* way. A further criticism is that this method gives no preference for literals which can be satisfied with short plans over those that require long plans. Both ABSTRIPS and our method favoured those literals which had short plans.

6.10 Summary

We have considered the problem of building abstractions automatically. We first identified the different ways you can abstract a new problem domain. Under some very weak assumptions (*eg.* we want to abstract the theory and not the logic), we showed that there are only **five** major classes of abstractions. We then focussed on one of these classes, the abstraction used in ABSTRIPS. We described various desirable properties that should be possessed by a method for building such abstractions automatically. We used this list of properties to motivate our choice of a method for building ABSTRIPS abstractions automatically. We then demonstrated the worth of this solution both theoretically and empirically. Finally, we compared our method to other methods for building ABSTRIPS abstractions.

Chapter 7

Using Abstractions

This Chapter explores how an abstract proof can be "mapped back" onto a ground proof. We introduce a notion called, **tree subsumption**, *for describing the relationship between the structure of ground and abstract proof trees. Tree subsumption is a monotonicity property on the structure of proof trees. This structural relationship suggests a general purpose procedure for mapping an abstract proof tree back onto a ground proof tree. We end by using this procedure to explore how abstraction reduces search.*

7.1 Introduction

We have argued at length for abstractions which preserve provability. However, this is only a very weak property to demand; we also want abstractions to preserve the *structure* of proofs. Abstract proofs can then be used to guide a theorem prover in finding ground proofs. In this Chapter, we will define a notion of similarity between the structure of proofs called **tree subsumption**. This is a monotonicity property: for any node in the abstract proof tree there must be a corresponding node in the ground proof tree. The idea is that the ground proof tree can be obtained by adding (possibly zero) nodes to an "unabstraction" of

The work described in this Chapter first appeared in [GW89b].

the abstract proof tree. No abstract nodes need to be thrown away. The abstract proof provides the major "islands" we need to get between in constructing a ground proof, and thereby helps to reduce search.

7.2 Trees

We begin with some notions for describing proof trees.

Definition 30 (Formulae tree) *:*

- *any wff, φ is a formulae tree;*
- *if φ is a wff and $\Pi_1, ...\Pi_n$ for $n \geq 1$ are formulae trees then*

$$\frac{\Pi_1 ... \Pi_n}{\varphi}$$

is also a formulae tree.

These are the only rules for generating formulae trees. In both rules, φ is the **root formula***, and the top most formulae are the* **leaf formulae**.

Often directed acyclic graphs are used in presentations of the theory of trees; we, however, have chosen a more graphical notation as this emphasizes the *structure* of the trees. For the sake of brevity we will write "tree" to mean "formulae tree". The tree, Π_1 is **equal** to the tree, Π_2, written $\Pi_1 = \Pi_2$ iff Π_1 and Π_2 are constructed using the same sequence of rules and formulae. A **branch** of a tree is a sequence of wffs ending at a leaf, each wff being directly above the previous; the last wff in a branch is always a leaf wff. The **length** of a branch is the number of wffs in the branch. The **depth** of a tree Π, written $|\Pi|$, is the length of the longest branch in the tree. $hd(b)$ is the first wff of a branch b, whilst $tl(b)$ is the branch formed by removing $hd(b)$ from the branch b; $tl(b)$ is undefined for branches one wff long. $hd(b)$ is the root wff of the tree if the length of b is the same as the depth of the tree. A **sub-branch** of b is either b or, provided $tl(b)$ is defined, a sub-branch of $tl(b)$. A wff α is **below** another wff β in a tree Π iff α

occurs earlier than β in some branch of Π. A branch **contains** another branch if it mentions the same wffs (and possibly more) in the same order.

> **Definition 31 (Contains)** *:* b_1 **contains** b_2 *iff there exists some sub-branch, b of b_1 for which*
>
> - $hd(b) = hd(b_2)$*, and*
> - *if* $tl(b_2)$ *is defined then* $tl(b)$ *exists and contains* $tl(b_2)$*.*

The contains relation is a weak partial order on branches being transitive, antisymmetric, and reflexive. $\mathcal{N}(\varphi, \Pi)$ is the number of occurrences of the wff φ in the proof tree Π. The **weight** of a tree Π, written $\|\Pi\|$, is the number of wffs in the tree. That is, $\|\Pi\| = \sum_{\varphi} \mathcal{N}(\varphi, \Pi)$.

The definitions so far have described any *arbitrary* formulae tree. Our interest is mainly in those formulae trees which represent proofs or, more generally, deductions. A **deduction tree** is a tree in which every wff is derived from the wffs directly above it by the valid application of an inference rule. A **proof tree** is a deduction tree in which the leaf formulae are either discharged assumptions or axioms. Π is a **proof tree in** Σ if every wff in the tree is from the language of Σ, every deduction uses an inference rule from the deductive machinery of Σ, and every undischarged leaf node is an axiom of Σ. If necessary, we can augment the description of a deduction tree by the name of the inference rule used at each step. If $f : \Sigma_1 \Rightarrow \Sigma_2$ is an abstraction and Π is a tree containing wffs from the language of Σ_1 then the **abstraction of** Π, written $f(\Pi)$, is the tree constructed by applying the mapping function of the abstraction to every wff in Π. The abstraction of a proof tree in Σ_1 need not be a proof tree in Σ_2.

7.3 Subtrees

One very important relationship between trees is the subtree relation.

> **Definition 32 (Subtree)** *:* Π_1 *is a* **subtree** *of* Π_2*, written* $\Pi_1 \sqsubseteq \Pi_2$ *iff*

- $\Pi_2 = \Pi_1$, *or*
- $$\Pi_2 = \frac{\Gamma_1 ... \Gamma_n}{\varphi}$$

  and there exists i, $1 \leq i \leq n$ *such that* $\Pi_1 \sqsubseteq \Gamma_i$.

If Π_1 is a subtree of Π_2 then we will also say that Π_2 is a **supertree** of Π_1. Note that the leaves of a subtree must also be leaves of the supertree. A subtree cannot have leaves that come from the middle of the supertree. As an example of the subtree relation, consider the following proof tree:

$$\frac{bottle(a) \quad bottle(x) \rightarrow pushable(x)}{pushable(a)}$$

This tree is a subtree of the following proof tree:

$$\frac{\dfrac{bottle(a) \quad bottle(x) \rightarrow pushable(x)}{pushable(a)} \qquad \dfrac{bottle(a) \quad bottle(x) \rightarrow liftable(x)}{liftable(a)}}{pushable(a) \ \wedge \ liftable(a)}$$

The subtree relation is reflexive; that is, any tree is a subtree of itself. As in the last example, a subtree can be smaller than the supertree. In such circumstances, it is a **strict subtree**.

> **Definition 33 (Strict subtree)** *:* Π_1 *is a* **strict subtree** *of* Π_2*, written* $\Pi_1 \sqsubset \Pi_2$ *iff* $\Pi_1 \sqsubseteq \Pi_2$ *and* $\Pi_1 \neq \Pi_2$

7.4 Properties of Subtrees

If Π_1 is a subtree of Π_2 then the depth of Π_1 must be less than or equal to that of Π_2. Similarly, the weight of Π_1 is less than or equal to that of Π_2. Additionally, formulae which occur in Π_1 must also occur at least as frequently in Π_2. The subtree relation also preserves the structure of proofs; wffs appear in the same order in Π_1 as in Π_2 and branches that appear in Π_1 also appear in Π_2. The subtree relation is therefore a monotonicity property on the depth of formulae trees, their weight, the formulae occurrences, the ordering of formulae and the branches.

Theorem 63 (Monotonicity) *: If* $\Pi_1 \sqsubseteq \Pi_2$ *then*

1. $|\Pi_1| \leq |\Pi_2|$
2. $\|\Pi_1\| \leq \|\Pi_2\|$
3. *for any wff* φ, $\mathcal{N}(\varphi, \Pi_1) \leq \mathcal{N}(\varphi, \Pi_2)$
4. *if* α *is below* β *in* Π_1 *then* α *is below* β *in* Π_2
5. *if* b *is a branch of* Π_1 *then* b *is also a branch of* Π_2

Proof: All five properties can be proven by induction on the depth of Π_1. The base and step cases follow immediately from the definitions. Note that parts 1, 2 and 4 are simple corollaries of parts 5, 3 and 5 respectively. □

The subtree relation is a weak partial order on trees, being transitive, anti-symmetric, and reflexive.

Theorem 64 (Weak partial order) *:*

1. $\Pi_1 \sqsubseteq \Pi_2$ *and* $\Pi_2 \sqsubseteq \Pi_3$ *implies* $\Pi_1 \sqsubseteq \Pi_3$
2. $\Pi_1 \sqsubseteq \Pi_2$ *and* $\Pi_2 \sqsubseteq \Pi_1$ *implies* $\Pi_1 = \Pi_2$
3. $\Pi_1 \sqsubseteq \Pi_1$

Proof: All these properties can be proven by induction on the depth of Π_1. The base and step cases follow immediately from the definitions. □

The strict subtree relation is a strict monotonicity property. That is, if $\Pi_1 \sqsubset \Pi_2$ then the depth of Π_1 is strictly less than that of Π_2, the weight of Π_1 is strictly less than that of Π_2, at least one formulae must occur less frequently in Π_1 than in Π_2, and at least one branch occurs in Π_2 that does not occur in Π_1.

Theorem 65 (Strict monotonicity) *: If* $\Pi_1 \sqsubset \Pi_2$ *then*

1. $|\Pi_1| < |\Pi_2|$
2. $\|\Pi_1\| < \|\Pi_2\|$
3. *there exists a wff,* φ *for which* $\mathcal{N}(\varphi, \Pi_1) < \mathcal{N}(\varphi, \Pi_2)$
4. *there exists a branch,* b *of* Π_2 *which is not a branch of* Π_1

Proof: From Definition 33, the root wff, α of Π_2 does not occur in Π_1. Hence $|\Pi_1| \leq 1+|\Pi_2|$, $\|\Pi_1\| \leq 1+\|\Pi_2\|$, and $\mathcal{N}(\alpha, \Pi_1) \leq 1+\mathcal{N}(\alpha, \Pi_2)$. Finally, the longest branch of Π_2 cannot be a branch of Π_1 since Π_1 has strictly less depth than Π_2. $\Box$

The strict subtree relation is a strict partial order, being transitive and irreflexive.

Theorem 66 (Strict partial order) *:*

1. $\Pi_1 \sqsubset \Pi_2$ *and* $\Pi_2 \sqsubset \Pi_3$ *implies* $\Pi_1 \sqsubset \Pi_3$
2. $\neg(\Pi_1 \sqsubset \Pi_1)$

Proof: From Definition 33 and Theorem 64, $\Pi_1 \sqsubseteq \Pi_3$. By considering $|\Pi_1|$ and $|\Pi_3|$, $\Pi_1 \neq \Pi_3$. Thus $\Pi_1 \sqsubset \Pi_3$. From Definition 33, $\neg(\Pi_1 \sqsubset \Pi_1)$ immediately follows. $\Box$

7.5 Tree Subsumption

To describe the relationship between ground and abstract proof trees, we have argued for a monotonicity property on the structure of trees. Although the subtree relation is such a monotonicity property, it is too *strong* to describe many of the relationships that exist between the structure of ground and abstract proof trees. Often, a ground proof tree can be obtained by adding wffs (or, more strictly, trees) anywhere to an "unabstraction" of the abstract proof tree. With the subtree relation, wffs (or trees) can only be added *beneath* the root formula of the subtree. We will therefore define a new monotonicity property on trees, called **tree subsumption**, that is weaker than the subtree relation but that is closely related.

The essential idea is that the abstract proof tree, Π_2 can be obtained by deleting formula occurrences from *anywhere* in the abstraction of the ground proof tree, $f(\Pi_1)$. Or, equivalently, the ground proof tree can be obtained by adding formula occurrences to a tree that maps onto the abstract proof tree. In such circumstance, we will say that Π_2 **subsumes** $f(\Pi_1)$, or in symbols $\Pi_2 \subseteq f(\Pi_1)$. Formally:

Definition 34 (Tree subsumption) *:* $\Pi_1 \subseteq \Pi_2$ *iff*

- $\Pi_1 = \varphi$ *and* φ *occurs in* Π_2*;*
-
$$\Pi_1 = \frac{\Gamma_1 ... \Gamma_n}{\varphi}$$
and φ *is the root formula of a subtree,* Γ *of* Π_2 *and for every* i*,* $1 \leq i \leq n$ *there exists a strict and distinct subtree of* Γ*,* Γ'_i *such that* $\Gamma_i \subseteq \Gamma'_i$.

By distinct, we mean that if Γ'_i is a *subtree* of Γ'_j then $i = j$. There is a close similarity between tree subsumption and the subtree relation, reflected in a similarity between their recursive definitions. There are, however, two main differences between the two relations. With tree subsumption, wffs anywhere in the subsuming tree can be skipped, and the ordering of subtrees is ignored. With the subtree relation, only wffs in the supertree beneath the subtree are skipped, and the ordering of the subtrees is fixed.

A slightly more general notion of tree subsumption is where we demand **not** that all the wffs in one tree *appear* in the other tree **but** that all the wffs in one tree *subsume* in the logical sense wffs in the other; this generalisation is needed to capture the relationship between ground and abstract proofs that occurs with Plaisted's abstractions [Pla81].

Tree subsumption seems to be a very common relationship between the structure of abstract and ground proof trees. For example, the abstract proof in Section 2.5 subsumes the abstraction of the ground proof given earlier in that section. As a second example, consider the predicate abstraction that maps both "*liftable*(x)" and "*pushable*(x)" onto "*movable*(x)" but leaves all other predicates unchanged. And consider the following ground proof:

$$\dfrac{\dfrac{bottle(a) \quad bottle(x) \rightarrow pushable(x)}{pushable(a)} \qquad \dfrac{bottle(a) \quad bottle(x) \rightarrow liftable(x)}{liftable(a)}}{pushable(a) \ \wedge \ liftable(a)}$$

The *abstraction* of this proof is subsumed by the following abstract proof:

$$\dfrac{\dfrac{bottle(a) \quad bottle(x) \rightarrow movable(x)}{movable(a)}}{movable(a) \ \wedge \ movable(a)}$$

Note that this abstract proof is *not* a subtree of the abstraction of the ground proof.

7.6 Properties of Tree Subsumption

If Π_1 subsumes Π_2 then Π_1 can be built by chopping wffs out of Π_2. The depth of Π_1 must therefore be less than or equal to that of Π_2. Similarly, the weight of Π_1 must be less than or equal to that of Π_2. Additionally, formulae which occur in Π_1 must also occur at least as frequently in Π_2. Tree subsumption also preserves the structure of proofs. Wffs appear in the same order in Π_1 as in Π_2 and branches that appear in Π_1 also appear (possibly containing extra formulae) in Π_2; that is, every branch of Π_1 is contained in a branch of Π_2. Tree subsumption is therefore a **monotonicity** property on the depth of formulae trees, their weight, the formulae occurrences, the ordering of formulae and the branches.

Theorem 67 (Monotonicity) *: If* $\Pi_1 \subseteq \Pi_2$ *then*

1. $|\Pi_1| \leq |\Pi_2|$
2. $\|\Pi_2\| \leq \|\Pi_2\|$
3. *for any wff* φ*,* $\mathcal{N}(\varphi, \Pi_1) \leq \mathcal{N}(\varphi, \Pi_2)$
4. *if* α *is below* β *in* Π_1 *then* α *is below* β *in* Π_2
5. *if* b_1 *is a branch of* Π_1 *then there exists a branch* b_2 *of* Π_2 *which contains* b_1*.*

Proof:

1. This is a corollary of the monotonicity property on branches. Alternatively we can use induction on the depth of Π_1.

   For the base case, let $\Pi_1 = \varphi$. $|\Pi_1| = 1$. By Definition 34, there exists a subtree of Π_2 with φ as root formulae. Therefore $|\Pi_2| \geq 1$. Hence $|\Pi_1| \leq |\Pi_2|$.

   For the step case, assume it is true for all trees of depth m. Consider a tree, $\Pi_1 = \frac{\Gamma_1 \ldots \Gamma_n}{\varphi}$ of depth $m+1$. At least one of the subtrees, Γ_i must be of depth m. That is, $|\Pi_1| = |\Gamma_i| + 1$. By Definition 34, there exists a strict and distinct subtree of Π_2, Γ'_i such that $\Gamma_i \subseteq \Gamma'_i$. As Γ'_i is a strict subtree of Π_2, $|\Gamma'_i| + 1 \leq |\Pi_2|$. From the induction hypothesis, $|\Gamma_i| \leq |\Gamma'_i|$. Hence, $|\Gamma_i| + 1 \leq |\Pi_2|$. Thus, $|\Pi_1| \leq |\Pi_2|$.

2. This is a corollary of the monotonicity property on formulae occurrences. Alternatively we can use induction on the depth of Π_1.

   For the base case, let $\Pi_1 = \varphi$. $\|\Pi_1\| = 1$. By Definition 34, there exists a subtree of Π_2 with φ as root formulae. Therefore $\|\Pi_2\| \geq 1$. Hence $\|\Pi_1\| \leq \|\Pi_2\|$.

   For the step case, assume it is true for all trees up to depth m. Consider a tree, $\Pi_1 = \frac{\Gamma_1 \ldots \Gamma_n}{\varphi}$ of depth $m+1$. Now $\|\Pi_1\| = 1 + \sum_{i=1}^{n} \|\Gamma_i\|$. By Definition 34, for every i, $1 \leq i \leq n$, there exists a strict and distinct subtree of Π_2, Γ'_i such that $\Gamma_i \subseteq \Gamma'_i$. As the subtrees are strict and distinct, $1 + \sum_{i=1}^{n} \|\Gamma'_i\| \leq \|\Pi_2\|$. But by the induction hypothesis, $\|\Gamma_i\| \leq \|\Gamma'_i\|$. Hence, $1 + \sum_{i=1}^{n} \|\Gamma_i\| \leq \|\Pi_2\|$. That is, $\|\Pi_1\| \leq \|\Pi_2\|$.

3. We only need consider wffs in Π_1. The proof uses induction on the depth of Π_1.

   For the base case, let $\Pi_1 = \varphi$. $\mathcal{N}(\varphi, \Pi_1) = 1$. By Definition 34, there exists a subtree of Π_2 with φ as root formulae. Thus,

$\mathcal{N}(\varphi, \Pi_2) \geq 1$. Therefore $\mathcal{N}(\varphi, \Pi_1) \leq \mathcal{N}(\varphi, \Pi_2)$.

For the step case, assume it is true for all trees up to depth m. Consider a tree, Π_1 of depth $m+1$. Consider any wff, φ in Π_1. Consider the lowest occurrence of φ. Let this be the root formulae of the subtree, $\frac{\Gamma_1,\ldots\Gamma_n}{\varphi}$. $\mathcal{N}(\varphi, \Pi_1) = 1 + \sum_{i=1}^{n} \mathcal{N}(\varphi, \Gamma_i)$. By Definition 34, there exists a subtree, Γ of Π_2 with φ as root formulae, and for which for every i, there exist strict and distinct subtrees, Γ'_i such that $\Gamma_i \subseteq \Gamma'_i$. As the subtrees are strict and distinct, $1 + \sum_{i=1}^{n} \mathcal{N}(\varphi, \Gamma'_i) \leq \mathcal{N}(\varphi, \Pi_2)$ By the induction hypothesis, $\mathcal{N}(\varphi, \Gamma_i) \leq \mathcal{N}(\varphi, \Gamma'_i)$. Thus, $1 + \sum_{i=1}^{n} \mathcal{N}(\varphi, \Gamma_i) \leq \mathcal{N}(\varphi, \Pi_2)$. That is, $\mathcal{N}(\varphi, \Pi_1) \leq \mathcal{N}(\varphi, \Pi_2)$.

4. This is a corollary of the monotonicity property on branches. Alternatively we can use induction on the depth of Π_1.

   For the base case, let $\Pi_1 = \varphi$. As there are no branches of length more than 1, there are no wffs below φ in Π_1. The theorem is therefore vacuously satisfied.

   For the step case, assume it is true for all trees up to depth m. Consider a tree, $\Pi_1 = \frac{\Gamma_1,\ldots\Gamma_n}{\varphi}$ of depth $m+1$. Consider any wffs, α and β for which α is below β in Π_1. If $\alpha \neq \varphi$ then by Definition 34 and the induction hypothesis, α is below β in some subtree of Π_2. That is, α is below β in Π_2. If $\alpha = \varphi$ then, by Definition 34, α is the root formula of a subtree of Π_2. Let β appear in the subtree, Γ_i. By Definition 34, Γ_i subsumes some strict and distinct subtree of Π_2. Thus, β will appear in this subtree of Π_2, and α is below β in Π_2.

5. Let Π'_1 be the subtree of Π_1 with least weight which has b_1 as a branch; if there is a choice, consider the leftmost. As $\Pi_1 \subseteq \Pi_2$ and Π'_1 is a subtree of Π_1, it follows that $\Pi'_1 \subseteq \Pi_2$. The proof now uses induction on the depth of Π'_1.

   For the base case, let $\Pi'_1 = \varphi$. The only branch of Π'_1 is $\langle\varphi\rangle$.

By Definition 34, there exists a subtree of Π_2 with φ as root formulae. Thus, there exists a branch of Π_2 which contains $\langle\varphi\rangle$.

For the step case, assume it is true for all trees up to depth m. Consider a tree, $\Pi'_1 = \frac{\Gamma_1,\ldots\Gamma_n}{\varphi}$ of depth $m+1$. As Π'_1 is the subtree of least weight that has b_1 as a branch, $hd(b_1)$ will be φ, and $tl(b_1)$ will be a branch of one of the subtrees, Γ_i. If it is a branch of more than one of the subtrees, we pick the leftmost. From Definition 34, there exists a strict subtree of Π_2, Γ'_i such that $\Gamma_i \subseteq \Gamma'_i$. By the induction hypothesis, there exists some branch of Γ'_i which contains $tl(b_1)$. If we extend this branch down to the root of Π_2 then, from Definition 34, φ must also appear in this branch before $tl(b_1)$. This branch therefore contains b_1.

□

Tree subsumption is a *weaker* property than the subtree relation. If Π_1 is a subtree of Π_2 then Π_1 subsumes Π_2. However, if Π_1 subsumes Π_2 then Π_1 need not be a subtree of Π_2.

Theorem 68 *: If* $\Pi_1 \sqsubseteq \Pi_2$ *then* $\Pi_1 \subseteq \Pi_2$.

Proof: By induction on the depth of Π_1. Both the base and the step cases follow trivially from Definitions 32 and 34. □

The definition of tree subsumption was constructed by weakening the subtree relation as far as is possible without losing the important property of monotonicity. Indeed, we conjecture that:

Conjecture *: Tree subsumption is as weak a relation on trees as is possible whilst still being monotonic.*

There seems to be no part of the definition of tree subsumption which can be weakened without losing monotonicity. For example, dropping the strictness requirement on subtrees loses monotonicity on the depth, whilst dropping the distinctness requirement loses monotonicity on the weight (but not the depth).

Tree subsumption is a preorder on trees being transitive, and reflexive. Unlike the subtree relation, tree subsumption is not antisymmetric; this is because tree subsumption ignores the ordering of the subtrees.

Theorem 69 (Preorder) *:*

1. $\Pi_1 \subseteq \Pi_2$ *and* $\Pi_2 \subseteq \Pi_3$ *implies* $\Pi_1 \subseteq \Pi_3$
2. $\Pi_1 \subseteq \Pi_1$

Proof:

1. By induction on the depth of Π_1.

   For the base case, let $\Pi_1 = \varphi$. From Definition 34, Π_2 (and hence Π_3) contain φ. Thus $\Pi_1 \subseteq \Pi_3$.

   For the step case, assume it is true for all trees, Π_1 up to depth m. Consider a tree, $\Pi_1 = \frac{\Gamma_1,\ldots\Gamma_n}{\varphi}$ of depth $m+1$. From Definition 34, φ is the root formula of a subtree, Γ of Π_2 and for every i, $1 \leq i \leq n$, there exists a strict and distinct subtree of Γ, Γ'_i such that $\Gamma_i \subseteq \Gamma'_i$. As $\Pi_2 \subseteq \Pi_3$ and Γ is a subtree of Π_2, $\Gamma \subseteq \Pi_3$. Thus, φ is the root formulae of a subtree, Γ' of Π_3, and for every i, $1 \leq i \leq n$, there exists a strict and distinct subtree of Γ', Γ''_i such that $\Gamma'_i \subseteq \Gamma''_i$. But Γ_i is of depth m or less. Hence, by the induction hypothesis, $\Gamma_i \subseteq \Gamma''_i$. Thus $\Pi_1 \subseteq \Pi_3$.

2. Trivially by induction on the depth of Π_1 using Definition 34.

□

7.7 Tree Isomorphism

Tree subsumption is closely related to (but weaker than) the conventional notion of the subtree relation. It is also related to (but weaker than) the conventional notion of **tree isomorphism**. Tree isomorphism is a much stronger relation than both the subtree relation and tree subsumption. Two trees are isomorphic iff they are equal up to reordering of their subtrees.

Definition 35 (Tree isomorphism) *:* $\Pi_1 \simeq \Pi_2$ *iff*

- $\Pi_1 = \Pi_2 = \varphi$, *or*
- $$\Pi_1 = \frac{\Gamma_1, ...\Gamma_n}{\varphi} \text{ and } \Pi_2 = \frac{\Gamma'_1, ...\Gamma'_n}{\varphi}$$

  and for every i, $1 \leq i \leq n$ *there exists a unique* j *such that* $\Gamma_i \simeq \Gamma'_j$.

Tree isomorphism is, however, a weaker relation than equality between trees; trees that are equal must be isomorphic, but trees that are isomorphic need not be equal.

Theorem 70 *: If* $\Pi_1 = \Pi_2$ *then* $\Pi_1 \simeq \Pi_2$.

Proof: Immediate from Definition 35. □

Sometimes an abstract proof tree will be isomorphic to the abstraction of a ground proof tree. Consider, for example the predicate abstraction of Section 7.5 which maps "$pushable(x)$" onto "$movable(x)$". Consider the following ground proof:

$$\frac{\dfrac{bottle(a) \quad bottle(x) \rightarrow pushable(x)}{pushable(a)} \qquad \neg pushable(x)}{\perp}$$

The *abstraction* of this proof tree is isomorphic to the following abstract proof tree:

$$\frac{\neg movable(x) \qquad \dfrac{bottle(x) \rightarrow movable(x) \quad bottle(a)}{movable(a)}}{\perp}$$

Note that the abstraction of the ground proof tree does not *equal* the abstract proof tree as the ordering of the subtrees is different. Note also that the abstract proof tree subsumes the abstraction of the ground proof tree. Indeed, in the next section, we will prove that tree isomorphism implies tree subsumption.

7.8 Properties of Tree Isomorphism

When two trees subsume each other, they are isomorphic. And when two tress are isomorphic, they subsume each other.

Theorem 71 *:* $\Pi_1 \simeq \Pi_2$ *iff* $\Pi_1 \subseteq \Pi_2$ *and* $\Pi_2 \subseteq \Pi_1$.

Proof:

($\Rightarrow$) By induction on the depth of Π_1. The base case is trivial.

For the step case, assume it is true for all trees, Π_1 up to depth m. Consider a tree, $\Pi_1 = \frac{\Gamma_1,\ldots\Gamma_n}{\alpha}$ of depth $m+1$. Let $\Pi_2 = \frac{\Gamma'_1\ldots\Gamma'_n}{\alpha}$. From definition 35, for every i there exists a unique j such that $\Gamma_i \simeq \Gamma'_j$. By the induction hypothesis, $\Gamma_i \subseteq \Gamma'_j$. From definition 34, $\Pi_1 \subseteq \Pi_2$. By symmetry, $\Pi_2 \subseteq \Pi_1$.

($\Leftarrow$) By induction on the depth of Π_1.

For the base case, let $\Pi_1 = \varphi$. From Definition 34, Π_2 must also be φ. Thus $\Pi_1 \simeq \Pi_2$.

For the step case, assume it is true for all trees, Π_1 up to depth m. Consider a tree, $\Pi_1 = \frac{\Gamma_1,\ldots\Gamma_n}{\alpha}$ of depth $m+1$. Let $\Pi_2 = \frac{\Gamma'_1,\ldots\Gamma'_p}{\beta}$. By Theorem 67, $|\Pi_2| \leq m+1$ and $|\Pi_2| \geq m+1$. That is, $|\Pi_2| = m+1$. Assume $\alpha \neq \beta$. From Definition 34, there exists some strict subtree, Γ_i of Π_1 such that $\Pi_2 \subseteq \Gamma_i$. Thus $|\Pi_2| \leq |\Gamma_i|$. But, $|\Gamma_i| \leq m$. Hence $|\Pi_2| \leq m$. This contradicts $|\Pi_2| = m+1$. Therefore α must equal β.

As $\Pi_1 \subseteq \Pi_2$, for every i, $1 \leq i \leq n$ there exists a strict and distinct subtree of Π_2 such that Γ_i subsumes it. Define $g(i)$ so that this is a subtree of $\Gamma'_{g(i)}$. Now $\Gamma_i \subseteq \Gamma'_{g(i)}$. By Theorem 67, $\|\Gamma_i\| \leq \|\Gamma'_{g(i)}\|$.

Define the equivalence class $[i] = \{j|g(j) = g(i)\}$. Let $\mu = \{g(i)|1 \leq i \leq n\}$, and τ be the number of equivalence classes, $[i]$ with more than one member in them. $\|\Pi_1\| = 1+\sum_{i=1}^{n}\|\Gamma_i\|$ and $\|\Pi_2\| = 1+\sum_{i=1}^{r}\|\Gamma'_i\|$. By simple arithmetic, $\|\Pi_2\| \geq 1+\sum_{i\in\mu}\|\Gamma'_i\|$. Because the subtrees are distinct, $\sum_{i\in\mu}\|\Gamma'_i\| \geq \tau+\sum_{i=1}^{n}\|\Gamma_i\|$. Thus, $\|\Pi_2\| \geq 1+\tau+\sum_{i=1}^{n}\|\Gamma_i\|$. That is, $\|\Pi_2\| \geq \tau+\|\Pi_1\|$. But, by Theorem 67, $\|\Pi_1\| = \|\Pi_2\|$. Hence, τ (the number of equivalence classes with more than one member) is zero. Therefore for every i, there exists a unique j, namely $g(i)$, such that $\Gamma_i \subseteq \Gamma'_j$. By symmetry, for every i, there also exists a unique j such that $\Gamma'_i \subseteq \Gamma_j$. By the induction hypothesis, $\Gamma_i \simeq \Gamma'_j$. Thus, by Definition 35, $\Pi_1 \simeq \Pi_2$.

□

This is a very pleasing result; it demonstrates that our definition of tree subsumption is closely related to the conventional notion of tree isomorphism. It also adds weight to the following claim.

Claim : *Tree subsumption is a very natural relationship between the structure of trees.*

Trees that are isomorphic have the same depth, the same weight, the same formulae occurrences, the same ordering of formulae and the same branches. Indeed, they can only differ in the left to right ordering of their subtrees. Tree subsumption is, we recall, a monotonicity property on the depth, the weight, the formulae occurrences, the ordering of formulae and the branches. Similarly, tree isomorphism is an equivalence on the depth, the weight, the formulae occurrences, the ordering of formulae and the branches of trees. We conjectured that tree subsumption is as weak a relation on trees as is possible whilst still being monotonic. Similarly, tree isomorphism is, we conjecture, as weak a relation on trees as is possible whilst still being an equivalence on the depth, the weight, the formulae occurrences, the ordering of formulae and the branches.

Theorem 72 : *If* $\Pi_1 \simeq \Pi_2$ *then*

1. $|\Pi_1| = |\Pi_2|$
2. $\|\Pi_1\| = \|\Pi_2\|$
3. *for any wff* φ, $\mathcal{N}(\varphi, \Pi_1) = \mathcal{N}(\varphi, \Pi_2)$
4. α *is below* β *in* Π_1 *iff* α *is below* β *in* Π_2
5. b *is a branch of* Π_1 *iff* b *is a branch of* Π_2.

Proof: By Theorem 71, $\Pi_1 \subseteq \Pi_2$ and $\Pi_2 \subseteq \Pi_1$. Therefore, by Theorem 67:

1. $|\Pi_1| \leq |\Pi_2|$ and $|\Pi_2| \leq |\Pi_1|$. Hence $|\Pi_1| = |\Pi_2|$.
2. $\|\Pi_1\| \leq \|\Pi_2\|$ and $\|\Pi_2\| \leq \|\Pi_1\|$. Hence $\|\Pi_1\| = \|\Pi_2\|$.
3. $\mathcal{N}(\varphi, \Pi_1) \leq \mathcal{N}(\varphi, \Pi_2)$ and $\mathcal{N}(\varphi, \Pi_2) \leq \mathcal{N}(\varphi, \Pi_1)$. Hence $\mathcal{N}(\varphi, \Pi_1) = \mathcal{N}(\varphi, \Pi_2)$.
4. α is below β in Π_1 implies α is below β in Π_2 Similarly, α is below β in Π_2 implies α is below β in Π_1.
5. b is a branch of Π_1 implies b is a branch of Π_2. Similarly, b is a branch of Π_2 implies b is a branch of Π_1.

$\Box$

Tree isomorphism is, in fact, an equivalence relation on trees being transitive, symmetric and reflexive.

Theorem 73 (Equivalence relation) *:*

1. $\Pi_1 \simeq \Pi_2$ *and* $\Pi_2 \simeq \Pi_3$ *implies* $\Pi_1 \simeq \Pi_3$*;*
2. $\Pi_1 \simeq \Pi_2$ *implies* $\Pi_2 \simeq \Pi_1$*;*
3. $\Pi_1 \simeq \Pi_1$.

Proof: Immediate from Theorems 71, and 69. $\Box$

7.9 Proof preserving abstractions

We can use the concept of tree subsumption to define a very general class of abstractions; abstractions in this class preserve the structure of proofs between the ground and the abstract space.

> **Definition 36 (PI-abstraction)** *: An abstraction $f : \Sigma_1 \Rightarrow \Sigma_2$ is said to be a* **PI-abstraction** *iff, for any proof, Π_1 of φ in Σ_1, there exists a proof, Π_2 of $f(\varphi)$ in Σ_2 with $\Pi_2 \subseteq f(\Pi_1)$.*

"P" stands for proof, and "I" for increasing. It is the number of proofs in the abstract space and not their weight, depth, formulae occurrences or branches that are increasing. By preserving the structure of proofs, a PI-abstraction also preserves provability; that is, it is a TI-abstraction.

> **Theorem 74** *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is a PI-abstraction then it is a TI-abstraction.*

> **Proof:** Immediate from Definitions 36 and 3. □

The reverse is not true; not all TI-abstractions are PI-abstractions. TI-abstractions in which we radically alter the deductive machinery are unlikely to be PI-abstractions. There are, of course, many other ways for two proof trees to be similar. However, for nearly all abstractions of which we are aware, tree subsumption describes the relationship that exists between ground and abstract proof trees. For example, all the substitution preserving theory abstractions identified in Chapter 6 are PI-abstractions. Additionally, Plaisted's abstractions [Pla81] are a subclass of PI-abstractions.

7.10 Mapping Back

With PI-abstractions, there is a simple relationship between the structure of ground and abstract proof trees. This relationship can be used to guide a theorem prover. The intuitive idea is that of **jumping between islands**. The abstract proof tree provides the major steps we need to get between. Finding a ground proof tree consists of filling in the gaps between the wffs in the abstract proof tree. A PI-abstraction thereby saves us time by dividing-and-conquering the search. To make this idea more concrete, we define the notion of an abstract proof plan.

Definition 37 (Abstract proof plan) *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is an abstraction, and Π_2 is a proof tree in Σ_2 then a tree Π_1 is an* **abstract proof plan** *of Π_2 iff $\Pi_2 \subseteq f(\Pi_1)$.*

Note that an abstract proof plan is not necessarily a valid proof tree since we may need to fill in gaps to make it one. Indeed it may not even correspond to a valid theorem as the mapping back can sometimes fail. A **refinement** of an abstract proof plan is any tree that is subsumed by the abstract proof plan but is not isomorphic to it. A refinement is a tree formed by inserting wffs (or, more strictly, trees) into the abstract proof plan.

Definition 38 (Refinement) *: If Π_1 is an abstract proof plan then a tree Π_2 is a* **refinement** *of Π_1 iff $\Pi_1 \subseteq \Pi_2$.*

By definition, a refinement is itself an abstract proof plan. A **minimal** abstract proof plan is an abstract proof plan whose abstraction is isomorphic to the abstract proof. Minimal abstract proof plans are unique (up to tree isomorphism) iff the mapping function is 1-to-1.

Definition 39 (Minimal abstract proof plan) *: If $f : \Sigma_1 \Rightarrow \Sigma_2$ is an abstraction, and Π_2 is a proof in Σ_2 then a tree Π_1 is a* **minimal** *abstract proof plan of Π_2 iff $\Pi_2 \simeq f(\Pi_1)$.*

To prove a ground theorem given an abstract proof, we construct minimal abstract proof plans, and then try to refine them. How we divide our time between these two tasks will depend on many factors: the theorem we are trying to prove, the abstraction we are using, the theorem prover we are using for refining abstract proof plans, *etc.* The following PROLOG program describes the general outline of a mapping back procedure.

```
mapback(Phi,Sigma,F,AbsProof,Proof):-
        plan(AbsProof,F,Plan),
        refine(Plan,Sigma,Phi,F,Proof),
        valid(Proof,Phi,Sigma).
```

`plan(AbsProof,F,Plan)` builds a minimal abstract proof plan, `Plan` from the abstract proof, `AbsProof` where `F` is the abstraction (that is, `plan`/4 builds the islands).

`refine(Plan,Sigma,Phi,F,Proof)` refines the abstract proof plan `Plan` returning the tree `Proof` (that is, `refine`/4 fills in the gaps).

`valid(Proof,Phi,Sigma)` checks whether `Proof` is a valid proof of `Phi` in `Sigma`.

This mapping back proof procedure can even be used with abstractions which are not PI-abstractions. For example, it could be used with Tenenberg's restricted predicate abstraction. Note also that this procedure allows us to use hierarchies of abstractions; we simply find the abstract proofs by another call to `mapback`/5.

We have implemented such a mapping back proof procedure. The full program is listed in Appendix A. A simple depth-first iterative deepening resolution theorem prover is used to refine the gaps between the steps in abstract proof plans. For horn clause problems, the prover uses LUSH resolution [Bun83]. For non-horn problems the input restriction is weakened to ancestor resolution; this guarantees completeness. For reasons we explain in the next section, the theorem prover uses a sorted unification algorithm. More details about the mapping back procedure are given in the next three sections.

We have tested our program with a variety of abstractions and abstract proofs. Indeed, all the examples presented in the next three sections were automatically mapped back to ground proofs using this program. Each of the examples is taken from the literature, and is chosen to illustrate a different aspect of mapping back. The fact that we can capture these very different examples of mapping back within the one uniform framework reinforces our previous claim that *tree subsumption is a very natural relationship between the structure of ground and abstract proof trees.*

7.11 An Example

This example is taken from [Ten87]. The axioms of this example describe the properties of various containers. Tenenberg proposes a predicate abstraction which collapses together objects with similar properties. For example, "$box(x)$", "$bottle(x)$" and "$glass(x)$" all map onto the generic "$container(x)$". We shall strengthen Tenenberg's abstraction so that similar properties are also collapsed together. For example, "$liftable(x)$" and "$pushable(x)$" will both map onto "$movable(x)$".

Consider the ground theorem that a bottle, a is both *liftable* and *pushable*. That is, "$liftable(a) \wedge pushable(a)$". This theorem maps onto the abstract theorem that a container, a is *movable*. Strictly speaking the goal is "$movable(a) \wedge movable(a)$". As each of the abstract conjuncts is the same, both halves of the ground proof – that is the proof that "$liftable(a)$" and the proof that "$pushable(a)$" – can be found by mapping back in different ways the following abstract proof tree.

$$\frac{container(a) \quad container(x) \rightarrow movable(x)}{movable(a)}$$

Up to tree isomorphism, this abstract proof tree gives $3 \times 3 \times 2 \times 2$ (or 36) minimal abstract proof plans. Unfortunately, many of these minimal abstract proof plans cannot be mapped back successfully to a proof of either "$liftable(a)$" or "$pushable(a)$". Therefore, rather than construct all these minimal abstract proof plans explicitly, we construct one schema which represents all of them up to tree isomorphism. This construction uses second order sorted meta-variables. These meta-variables allow us to represent any wff in the ground language that abstracts onto a given abstract wff and thereby to delay the choice in how to unabstract a wff. For example, "$X : container(a)$", where X is a second order sorted meta-variable, represents a predicate with an unknown name of sort *container* and with an argument a. That is, a predicate which abstracts onto "$container(a)$". The sorts of the meta-language are the equivalence classes of objects which are mapped together by the abstraction. Thus, the *container* sort ranges over the predicate names *box*, *bottle*, and *glass*. Our mapping back procedure constructs the following schema for the minimal abstract proof plans.

$$\frac{Y : container(a) \quad Z : container(x) \rightarrow W : movable(x)}{X : movable(a)}$$

Up to tree isomorphism, this represents all the possible minimal abstract proof plans. This schema can be refined in two different ways to give proofs of "$liftable(a)$" and of "$pushable(a)$". Putting these subproofs together with a final and introduction, we get the following ground proof tree.

$$\frac{\dfrac{bottle(a) \quad bottle(x) \rightarrow pushable(x)}{pushable(a)} \qquad \dfrac{bottle(a) \quad bottle(x) \rightarrow liftable(x)}{liftable(a)}}{pushable(a) \ \wedge \ liftable(a)}$$

7.12 Middle-out reasoning

Our mapping back procedure allows us to do a type of **middle-out reasoning** [BSH90]. Instead of starting at the axioms and applying the inference rules until we reach some goal (forward reasoning), or starting at the goal and generating subgoals until we reach the axioms (backward reasoning), we identify some key steps in the proof and fill in the gaps between them. We thereby construct the proof from the middle-out. Middle-out reasoning introduces two major problems not present in forward or backward reasoning: identifying the key steps in the proof and controlling a possibly complicated middle-out search. Abstract proofs are one solution to the first problem and meta-variables are a partial solution to the second problem. Meta-variables allow us to delay making a choice as to which of the many wffs in the ground language abstract onto a given wff in the abstract language. Inference in other branches of the ground proof tree and consultation of the database of ground axioms will allow us to make a more informed choice at a later stage.

In Chapter 6, we identified five main classes of theory abstractions. Each class will require the use of a different type of meta-variables and a different unification algorithm. We will briefly consider each class in turn.

1. mappings on predicates:

   - **predicate** abstractions will, as we have seen, require second order meta-variables with a (flat) sorted unification algorithm. For example, if "$rat(a)$" and "$irrat(a)$" both map onto "$num(a)$" then we can represent the unabstraction of "$num(a)$" by "$X(a)$" where "X" is a second order sorted meta-variable which ranges over $\{rat, irrat...\}$. The sorts of the meta-language are the equivalence classes of predicate names that abstract together.

   - **propositional** abstractions will require first order meta-variables with an unsorted unification algorithm. For example, the unabstraction of the propositional sentence letter, "num" is "$num(X)$" where "X" is a first order unsorted meta-variable.

   - **ABSTRIPS** abstractions, where we map whole predicates onto $\top$ or $\perp$, will not in general require any meta-variables as the minimal abstract proof plan is usually unique up to tree isomorphism. We usually know from the context what atomic formula mapped onto $\top$ or $\perp$.

2. mappings on the terms:

- **domain** abstractions, where we map constants together, will require first-order meta-variables with a (flat) sorted unification algorithm. For example, if $1, 3, 5, ...$ map onto *odd* then we can represent the unabstraction of $num(odd)$ by $num(X)$ where X is a first-order meta-variable which ranges over $\{1, 3, 5, ...\}$.
- **function** abstractions can either reduce the arity of functions or map their names together; the second subclass will require second-order meta-variables with a (flat) sorted unification algorithm, whilst the first will merely need first-order meta-variables with unsorted unification.

These results are summarised in the following table:

abstraction	meta-variables	unification
ABSTRIPS	none	n/a
propositional	first-order	unsorted
domain	first-order	sorted
predicate	second-order	sorted
function	first-order	unsorted
	second-order	sorted

This table might seem to suggest a "missing abstraction". That is, the class of abstraction that require second-order *unsorted* meta-variables. Actually, this class is just a special case of predicate abstractions where all the predicate names in the ground space map onto the same predicate name in the abstract space. Note that the sorts are "flat" since there is no subsort structure. This sorting of meta-variables is very important as some of our search can be compiled into the unification algorithm. Indeed, in Section 7.15, we will propose a trick whereby all search in mapping back can be moved into the unification algorithm. Although monadic second-order unification is decidable [HO80], polyadic second-order unification has recently been shown to undecidable [Gol79]. This is not a problem for our mapping back procedure since it needs only a very restricted and decidable form of second-order matching.

Some recent work on inductive theorem proving has also suggested a similar use of meta-variables for middle-out reasoning [BSH90]. In this work, certain eureka steps at the beginning of the proof (for example, the choice of induction scheme or existential witness) can be postponed till later on in the proof. We complete the middle of the proof first and then see what steps we should have made at the beginning of the proof for the middle to go through; our choice is more informed and thus more likely to succeed. To do this, meta-variables are used to "simulate" with as little commitment as possible the opening steps of the proof. For example, we can simulate an existential introduction by replacing an existentially quantified variable with a meta-variable. The meta-variables become instantiated by unification later on in such a way that the beginning of the proof is guaranteed to go through. This is very similar to the use of meta-variables for representing abstract proof plan schemata.

7.13 A Second Example

Our next example illustrates that, even if an abstraction maps an undecidable theory onto a decidable one, it may not reduce search. This failure does, however, suggest ways in which abstraction can reduce search. The example uses the **function** abstraction described in [Pla81]; this abstraction maps every term onto its top-level function symbol, leaving variables and predicate names unchanged. By deleting all the arguments to function symbols, this maps an undecidable ground space onto a decidable abstract space whose Herbrand universe is just a (finite) set of constants. This abstraction is also a PI-abstraction. The axioms for the problem are also taken from [Pla81]:

$$in(john, boy)$$
$$in(x, boy) \rightarrow in(x, human)$$
$$hp(x, m, y) \rightarrow in(sk1(n, m, z, y, x), y) \vee hp(x, t(m, n), z)$$
$$hp(x, m, y) \wedge hp(sk1(n, m, z, y, x), n, z) \rightarrow hp(x, t(m, n), z)$$
$$in(x, hand) \rightarrow hp(x, 5, fingers)$$
$$in(x, human) \rightarrow hp(x, 2, arm)$$
$$in(x, arm) \rightarrow hp(x, 1, hand)$$

The predicate and function symbols are interpreted as follows: $in(x, y)$ means x is a member of type y, $hp(x, m, y)$ means x has m parts of type y, $t(m, n)$

means m times n, and $sk1(n, m, z, y, x)$ is a skolem function representing an object, w which is a member of type y but which does not have n parts of type z. The theorem we wish to prove is the statement that John has two hands; that is, "$hp(john, t(2, 1), hand)$". This maps onto the abstract theorem, "$hp(john, t, hand)$". Plaisted gives an abstract proof of this theorem using ancestor resolution which maps back *immediately* to a ground proof (page 80 of [Pla81]). That is, one of the minimal abstract proof plans is itself a valid proof. Or equivalently, the abstraction of the ground proof is isomorphic to the abstract proof.

Although the abstract space is decidable, finding this abstract proof and mapping it back onto a ground proof is not much easier than finding the ground proof without abstraction. Indeed, the branching rate of the abstract search space is actually larger than that of the ground space; any resolution possible in the ground space is also possible in the abstract. Additionally, we would have to search in the abstract space to the same depth as in the ground space. Thus the size of the abstract search space is actually larger than the size of the ground search space. The only saving is that the cost of unification is cheaper in the abstract space than the ground space.

7.14 Reducing Search

Abstraction can, of course, considerably reduce search. It can decrease both the branching rate and the depth to which we have to search. The depth of search is reduced if we find abstract proofs that need to be refined. That is, if the minimal abstract proof plans are not valid proofs. The branching rate is reduced by choosing an abstraction that maps a ground axiom onto an abstract axiom which is logically **redundant**. An axiom is redundant if, for example, it logically follows from the other axioms, or is a **tautology** (a wff which is theorem of propositional logic). Deleting redundant axioms will, by definition, not affect what can be proved. However, the structure of proofs might not be preserved. This problem was first noticed by Plaisted [Pla81] as his mapping back strategies were incomplete if tautologies were deleted; this problem occurs with many other PI-abstractions.

We can always transform an abstract proof containing redundant axioms to an abstract proof which does not contain redundant axioms. However, this transformation can change the shape of the proof tree so radically that it cannot be used for mapping back. The pathological example is when we are trying to prove an axiom of the ground space which maps onto a redundant axiom in the abstract space. We must keep this redundant axiom in the abstract space if we are to be sure of finding the one step abstract proof than maps back to a ground proof.

One solution is keep *some* but not all of the redundant axioms. Consider, for example, a clausal language. We will say that a clause is a **simple** tautology iff it is of the form $\alpha \vee \neg\alpha$. The following theorem shows that axioms which map onto simple tautologies can be deleted without affecting the completeness of mapping back.

Definition 40 (Natural) *: A proof* Π *is a* **natural** *proof iff it contains no axioms which are simple tautologies.*

Theorem 75 *: If* Π_1 *is a LUSH resolution proof of* φ*, then there exists a natural resolution proof* Π_2 *of* φ *with* $\Pi_2 \subseteq \Pi_1$*.*

Proof: By induction on the number of resolutions against axioms that are simple tautologies. Let Π_1 have m such resolutions. We show how to construct a proof which has fewer than m such resolutions. Consider the highest such resolution in Π_1. Let this be a resolution between β and the tautological axiom $\alpha \vee \neg\alpha$ with the resolvent β'. β' will simply be some reordering and substitution instance of β. To construct a proof with fewer than m such resolutions, we perform the following transformation:

$$\dfrac{\dfrac{\dfrac{\Gamma_1}{\beta} \quad \alpha \vee \neg\alpha}{\beta'} \quad \gamma}{\Gamma_2} \quad \Rightarrow \quad \dfrac{\dfrac{\Gamma_1}{\beta} \quad \gamma}{\Gamma_2}$$

By repeatedly performing this transformation, we construct a proof tree, Π_2 which contains no resolutions against axioms that are simple tautologies. Since Π_2 is constructed from Π_1 simply by removing some of the wffs in the proof tree, $\Pi_2 \subseteq \Pi_1$. $\Box$

Note that the natural proof is smaller than the original proof. We can therefore refine an abstract proof plan corresponding to a natural abstract proof simply by adding wffs to it. Deleting simple tautologies, therefore, does not affect the completeness of mapping back. A slight variant of this solution is to show that we can remove remove most of the inferences involving tautological axioms. For instance, we can remove any resolution against a literal which appears both positively and negatively.

7.15 A Third Example

This example illustrates both reductions in the branching rate and the depth of the abstract search space. It also demonstrates the use of hierarchies of abstractions. We use the axiomatisation of the Tower of Hanoi with four disks given in [Pla80]. The predicate, "$d(x, y, z, w, s)$" represents the fact that in situation s, the smallest disk is on peg x, the second smallest disk is on peg y, the third smallest on z, and the largest on w. Plaisted proposed a sequence of propositional abstractions which discard the situation argument, s and (in order of size) the position of each of the disks. Each abstraction throws away one more argument to the predicate, "$d(x, y, z, w, s)$" than the last. These abstractions are all PI-abstractions. Some of the axioms map onto duplicates, and others map onto simple tautologies. In the ground space, there are 25 axioms. If we remove duplicates and simple tautologies, the abstract spaces have 25, 20, 13, and 7 axioms respectively.

We start in situation s_0 with all the disks on peg 1; that is, with "$d(1, 1, 1, 1, s_0)$". We want to prove that we can move all the disks onto peg 2; that is, we can get to a state, s in which "$d(2, 2, 2, 2, s)$". We will ignore the mapping back from the weakest abstract space (in which we consider all the disks but ignore the situation argument) and the ground space as this mapping back is immediate, requiring no new proof steps, and therefore rather uninteresting. The other abstract proofs halve in size with each successive abstraction. In the most abstract proof, we just move the largest disk from peg 1 to peg 2:

$$\frac{d(1) \quad d(1) \rightarrow d(2)}{d(2)}$$

To refine this proof, we need to add two resolutions, which store the second largest disk on peg 3 whilst we move the largest disk. Both the extra steps represent deductions which abstract onto loops. In general, *we refine a proof by adding in deductions which map onto loops.* The issue of loops and axioms which map onto tautologies are intimately related. The minimal abstract proof plan schema which represents all possible minimal abstract proof plans is:

$$\frac{d(X,1) \quad d(Y,1) \rightarrow d(Z,2)}{d(W,2)}$$

This can be refined to a proof in the next strongest level of abstraction:

$$\cfrac{\cfrac{\cfrac{d(1,1) \quad d(1,x) \rightarrow d(3,x)}{d(3,1)} \quad d(3,1) \rightarrow d(3,2)}{d(3,2)} \quad d(3,x) \rightarrow d(2,x)}{d(2,2)}$$

X has been unified with 1 or 3, Y and Z with 3 and and W with 3 or 2. The choices depend on the exact strategy we use to refine the minimal abstract proof plan schema.

An interesting extension of our use of meta-variables is to let meta-variables in the abstract proof plan stand for arbitrary parts of the proof. We can then represent all possible *refinements* of an abstract proof plan with just one tree. Consider, as an example, the following schema:

$$\cfrac{\cfrac{\cfrac{U}{[d(X,1)]} \quad d(Y,1) \rightarrow d(Z,2)}{V}}{[d(W,2)]}$$

where $\frac{U}{[d(X,1)]}$ is a meta-variable representing a tree that has $d(X,1)$ as root formulae, and $\frac{V}{[d(W,2)]}$ is a meta-variable representing a tree that has $d(W,2)$ as root formulae. This schema represents all possible refinements of the original minimal abstract proof plans. Refining such a schema simply becomes a search for a suitable instantiation for the meta-variables. In this case, U needs to unify with the tree:

$$\frac{d(1,1) \quad d(1,x)\rightarrow d(3,x)}{d(3,1)}$$

V with the tree:

$$\frac{d(3,2) \quad d(3,x)\rightarrow d(2,x)}{d(2,2)}$$

X with 3, Y with 3, Z with 3, and W with 2. All the work in mapping back (that is building the proof plans and refining them) can, by this trick, be shifted to the meta-variables.

The proof at the second strongest level of abstraction can itself be mapped back a level to a proof which moves just the three largest disks. Again, the refinements we add are deductions in the ground space which map onto loops in the abstract space. In mapping proofs back, we can exploit this fact by only searching for refinements that map onto loops.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{d(1,1,1) \quad d(1,x,y)\rightarrow d(2,x,y)}{d(2,1,1) \quad d(2,1,x)\rightarrow d(2,3,x)}}{d(2,3,1) \quad d(2,x,y)\rightarrow d(3,x,y)}}{d(3,3,1) \quad d(3,3,1)\rightarrow d(3,3,2)}}{d(3,3,2) \quad d(3,x,y)\rightarrow d(1,x,y)}}{d(1,3,2) \quad d(1,3,x)\rightarrow d(1,2,x)}}{d(1,2,2) \quad d(1,x,y)\rightarrow d(2,x,y)}}{d(2,2,2)}$$

Finally, we can map this abstract proof back a level to a proof of depth 16 which moves all four disks in the appropriate way.

$d(1,1,1,1)$	$d(1,x,y,z) \rightarrow d(3,x,y,z)$
$d(3,1,1,1)$	$d(3,1,x,y) \rightarrow d(3,2,x,y)$
$d(3,2,1,1)$	$d(3,x,y,z) \rightarrow d(2,x,y,z)$
$d(2,2,1,1)$	$d(2,2,1,x) \rightarrow d(2,2,3,x)$
$d(2,2,3,1)$	$d(2,x,y,z) \rightarrow d(1,x,y,z)$
$d(1,2,3,1)$	$d(1,2,x,y) \rightarrow d(1,3,x,y)$
$d(1,3,3,1)$	$d(1,x,y,z) \rightarrow d(3,x,y,z)$
$d(3,3,3,1)$	$d(3,3,3,1) \rightarrow d(3,3,3,2)$
$d(3,3,3,2)$	$d(3,x,y,z) \rightarrow d(2,x,y,z)$
$d(2,3,3,2)$	$d(2,3,x,y) \rightarrow d(2,1,x,y)$
$d(2,1,3,2)$	$d(2,x,y,z) \rightarrow d(1,x,y,z)$
$d(1,1,3,2)$	$d(1,1,3,x) \rightarrow d(1,1,2,x)$
$d(1,1,2,2)$	$d(1,x,y,z) \rightarrow d(3,x,y,z)$
$d(3,1,2,2)$	$d(3,1,x,y) \rightarrow d(3,2,x,y)$
$d(3,2,2,2)$	$d(3,x,y,z) \rightarrow d(2,x,y,z)$
$d(2,2,2,2)$	

7.16 Related Work

Closest in spirit to this work is that of Plaisted [Pla81, Pla86]. He gives various mapping back strategies for abstractions between resolution systems. Our work generalises Plaisted even if we restrict ourselves just to mappings between resolution systems. For instance, all of Plaisted's abstractions are PI-abstractions between resolution systems but not all PI-abstractions between resolution systems can be captured by Plaisted's work; the ABSTRIPS abstraction is an example of one type of abstraction that Plaisted's framework fails to capture.

Plaisted defines a **shape correspondence** between ground and abstract proof trees (page 63 of [Pla81]) which requires that the abstract proof be of the *same* depth as the ground proof. This shape correspondence, written "$\Pi_1 \rightarrow_f \Pi_2$", is related to but weaker than the notion of tree subsumption. Indeed, if we generalise tree subsumption to the relation, "$\subseteq^*$" that every wff in one tree logically subsumes a wff in the other tree, shape correspondence implies tree subsumption.

Theorem 76 : *If* $\Pi_1 \rightarrow_f \Pi_2$ *then* $\Pi_1 \subseteq^* f(\Pi_2)$ *and* $|\Pi_1| = |\Pi_2|$.

> **Proof:** The fact that $\Pi_1 \subseteq^* f(\Pi_2)$ follows immediately from the definition of shape correspondence. The fact that the depths are also equal is proven on page 61 of [Pla81]. $\Box$

Tree subsumption does not, however, imply shape correspondence; if $\Pi_1 \subseteq \Pi_2$ then $\Pi_1 \rightarrow_f \Pi_2$ may or may not hold. Even if we add the extra and necessary condition that the two trees have the same depth, tree subsumption still does not imply shape correspondence. Tree subsumption is therefore more general than Plaisted's shape correspondence. As a consequence, Plaisted can only map back abstract proofs onto ground proofs of the same depth. The ground proofs he constructs can be larger than the corresponding abstract proofs since an abstract wff can map back onto two or more wffs in the ground proof, each wff having a separate derivation. Since Plaisted must search the abstract space to the same depth as search in the ground space, his use of abstraction will suffer similar combinatorial explosion to search in ground space alone.

Plaisted also tackles the problem of mapping back for his generalisation abstractions [Pla86]. He considers two strategies for mapping back. His first (algorithm A on page 369 of [Pla86]) maps an abstract proof back onto a ground proof which has a substitution instance that is isomorphic to the abstract proof; this abstract proof may contain loops. Plaisted's second strategy (algorithm B on page 374 of [Pla86]) ignores the shape of the abstract proof altogether; it does, however, make use of the instances of the input clauses used in the abstract proof.

Tenenberg considers the problem of mapping back for his restricted predicate abstractions. With this class of abstractions, abstract proofs always map back but not necessarily to the theorem you are trying to prove (theorem 5.3, p. 97 of [Ten88]). In fact, the abstract proof trees subsume the abstraction of the ground proof trees; mapping back just consists of a bounded search through the abstract proof plans, what Tenenberg calls the "specialisations" till a ground proof is discovered. This search does not require any further theorem proving between the proof steps. The ground proof can, however, be larger than the abstract proof as several subtrees of the ground proof may map onto the same subtree of the abstract proof. Tenenberg's mapping back procedure is incomplete since not all ground theorems will have abstract proofs. Additionally, it only applies to a very specialised class of abstractions.

Finally, in [GW89d] we used a propositional abstraction to determine which definitions to unfold in a proof. However, we did not exploit most of the information in the structure of the abstract proof, only the sequence of definitions that were unfolded.

7.17 Summary

We have considered how an abstract proof can be "mapped back" onto a proof of a ground theorem. We introduced the notion of **tree subsumption**, a very general relationship between the structure of ground and abstract proof trees. Tree subsumption is a monotonicity property on the structure of proof trees. This suggested a general purpose procedure for mapping back an abstract proof tree onto a ground proof tree. We have used this procedure to explore how abstraction reduces search. The breadth of our search space is reduced by abstractions which map axioms onto duplicates and tautologies. The depth of our search space is reduced by abstractions which map deductions in ground proofs onto loops in abstract proofs.

Chapter 8

Conclusions

We summarise the main achievements of this thesis, and discuss its originality. We then identify some of its limitations and suggest areas for future research.

8.1 Achievements

The main achievement of this thesis is a general **theory of abstraction**. Although quite simple and elegant, this theory is capable of describing the properties of a large number of abstractions used in the past. Our motivation, however, is not purely descriptive; we have also explored the theory itself in some depth. We have classified the various types of abstractions, and investigated their formal properties. Additionally, we have described operations which can be performed on abstractions, and relations which compare them; these operations and relations form the beginnings of an algebra of abstractions. Finally, we have used our theory to explore how you should actually use abstraction; we have studied how to build abstractions automatically, and how to use abstraction to aid problem solving. To test our ideas in this last area, we have implemented a program for mapping abstract proofs back onto ground proofs.

8.2 Originality

The theory of abstraction we have proposed is more general than any of those proposed before. Plaisted's theory [Pla81] is arguably the most general rival theory. However, Plaisted's theory is restricted to abstractions between refutation systems that use resolution; our theory can describe abstractions between both proof and refutation systems, including some abstractions between resolution based refutation systems for which Plaisted's theory proves inadequate.

We have used our theory of abstraction as the basis for a uniform analysis of abstractions used in the past. This has identified many new connections between abstractions. For example, we have shown that Hobb's theory of granularity is merely an example of one of Plaisted's domain abstractions. We have also discovered that most abstractions fit into one of five classes; each of these classes abstracts a different part of the *syntax* of the language. This classification of abstractions into a simple taxonomy has helped us greatly in understanding what abstraction actually is.

We have also used our theory of abstraction prescriptively. For example, it naturally suggests methods for building abstractions and for mapping abstract proofs back onto ground proofs. We have implemented a program to perform such a mapping back; this is the first step towards a general purpose shell for abstract theorem proving. Results from this implementation have fed back to the theory, helping us to understand why abstraction works.

Finally, the theory itself has yielded some new and important results. For example, we have discovered the pervasive existence of inconsistent abstract spaces. It had occasionally been noticed that a particular abstract space was inconsistent. We are the first to prove that this is an inevitable consequence of using certain very common classes of abstraction.

8.3 Future Work

Although our theory is quite mature and seems able to describe everything we have demanded of it, there are many areas for future work. For example, we would like to develop further our program for mapping back, and to use it with new types of abstractions. As well as this experimental work, we can identify three other major topics for future research. The first topic concerns extensions to the theory; though the groundwork is clearly laid out, there are many areas which could be further developed and many new areas awaiting exploration. Section 8.4 identifies one of these new areas, the semantics of abstraction. The second topic for future research is the computational analysis of abstraction. Since one of the purposes of abstraction is to aid problem solving, it would be very interesting to develop a computational model of the benefits of abstraction. This model should complement our theoretical analysis of mapping back. Section 8.5 describes how we might begin such a computational analysis. The third and final topic for future research concerns how our theory relates to other areas of AI. For example, abstraction and analogy are often closely related. Does our theory have anything to say about analogy? And can we learn anything about abstraction from these related areas? Section 8.6 explores some possible answers to these questions.

As mentioned above, we will use the next three sections to speculate about three areas for future research.

8.4 The Semantics of Abstraction

Our analysis of this theory of abstraction has been almost entirely proof theoretic; this is because we want to use abstraction to help us find *proofs*. It would, however, be very interesting to consider the model theory and, thus, the semantics of abstraction. We have argued that abstractions should be meaningful, and that they should only collapse objects together that are related. Model theory should provide us with one method for describing more precisely what this means.

Actually, the whole of Chapter 5 was devoted to considering the model theoretic consequences of abstraction. As a formal system is inconsistent if it is unsatisfiable (has no models), the problem of inconsistent abstract spaces is also the problem of abstract spaces with no models. Since we have argued for consistent abstract spaces, this is perhaps not very helpful. We are more interested in analysing the relationship between the models of the ground space and those of the abstract space. We can, however, sketch how we might perform such an analysis. Tenenberg [Ten88] has provided such an analysis, but only for his very special class of restricted predicate abstractions.

For the sake of brevity, consider first order systems in which the connectives and quantifiers are given their usual meaning. An **interpretation** $\mathcal{I}$ of a wff φ without free variables can be defined in the usual way; that is, a triple $\langle \mathcal{D}, \Phi, \Psi \rangle$ where $\mathcal{D}$ is a non-empty set of objects called the **domain** of the interpretation, Φ assigns every n-ary function of φ to a function from $\mathcal{D}^n$ to $\mathcal{D}$ (in particular 0-ary functions, or constants are assigned to elements of $\mathcal{D}$), and Ψ assigns every n-ary predicate of φ to a function from $\mathcal{D}^n$ to $True$ or $False$. The truth value of a wff is determined by applying these assignments and giving the logical connectives and quantifiers their usual meaning (*eg.* $\alpha \wedge \beta$ is $True$ iff α and β are both $True$).

An interpretation is a **model** of a wff φ if it assigns φ the truth value $True$. This notion can be extended to a set of wffs (*eg.* the axioms of a theory) by renaming apart variables and considering their conjunction. A wff is **unsatisfiable** if none of its interpretations are models. We will say that $\mathcal{I}$ models a formal system Σ iff $\mathcal{I}$ is a model of the axioms of Σ.

Given an abstraction $f : \Sigma_1 \Rightarrow \Sigma_2$ and two models, $\mathcal{I}_1$ and $\mathcal{I}_2$ of Σ_1 and Σ_2, what can we say about the relationship between the models? We have the following picture:

$$\begin{tikzcd}
\Sigma_1 \arrow[r, "f : \Sigma_1 \Rightarrow \Sigma_2"] & \Sigma_2 \\
\mathcal{I}_1 \arrow[u, "\text{models}"] \arrow[r, dashed, "?"] & \mathcal{I}_2 \arrow[u, "\text{models}"']
\end{tikzcd}$$

Since the axioms of a formal system determine which interpretations are models of the formal system, we can simplify matters by restricting ourselves to Λ/Ω-invariant abstractions.

As $\mathcal{I}_1$ and $\mathcal{I}_2$ give the logical connectives and quantifiers the same meaning, our abstraction should preserve the semantics of Σ_1 and Σ_2. Thus we will further restrict ourselves to theory abstractions which abstract atomic wffs and not logical structure. In Chapter 6, we identified five major classes of theory abstractions; each class will give a different relationship between the models of its ground and abstract spaces.

Consider, for example, the class of domain abstractions. Let $f : \Sigma_1 \Rightarrow \Sigma_2$ be a domain abstraction between two theories with equality. Let "$\sim$" be the equivalence relation defining those constants which are mapped together (*ie.* $a \sim b$ iff a and b are mapped onto the same abstract constant, $[a]$). There is a simple relationship between the two models $\mathcal{I}_1 = \langle \mathcal{D}_1, \Phi_1, \Psi_1 \rangle$ and $\mathcal{I}_2 = \langle \mathcal{D}_2, \Phi_2, \Psi_2 \rangle$ of Σ_1 and Σ_2. For example, since $\mathcal{I}_1$ and $\mathcal{I}_2$ are models of the equality axioms:

$$\Phi_1(a) = \Phi_1(b) \quad \text{implies} \quad \Phi_2([a]) = \Phi_2([b])$$

This is exactly the relationship we would expect; constants which stand for identical objects should map onto abstract constants that stand for identical abstract objects. The other classes of theory abstractions will give other types of relationship between their models.

Finding the link between models is not the only way we can complete the picture. A relationship between models can induce an abstraction. We might also have the following picture:

$$\begin{tikzcd} \Sigma_1 \arrow[r, dashed, "?"] & \Sigma_2 \\ \mathcal{I}_1 \arrow[u, "\text{models}"] \arrow[r] & \mathcal{I}_2 \arrow[u, "\text{models}"'] \end{tikzcd}$$

For example, consider $\mathcal{I}_1$ as a model of the theory of lists and $\mathcal{I}_2$ as a model of the theory of sets. $\mathcal{I}_1$ gives the functions *append*, *list* and *nil*, and the predicates *member* and *equal* their usual interpretation whilst $\mathcal{I}_2$ gives the functions $\cup$, $\cap$, *set* (a unary function with $x \in set(y)$ iff $x = y$), 0 and the predicates $\in$ and $=$ their usual interpretation. We can see a close similarity between the interpretation of *append* and that of $\cup$, of *list* and that of *set*, *nil* and that of 0, *member* and that of $\in$, and *equal* and that of $=$. This naturally suggests an abstraction from the theory of lists to the theory of sets whose mapping function abstracts *member* onto $\in$, *nil* onto 0 *etc.* For example, $\exists x \,.\, member(x, list(nil))$ would abstract onto $\exists x \,.\, x \in set(0)$. This is a TI-abstraction as it introduces new theorems into the abstract space. For instance, $\cup$ is commutative and idempotent but *append* is neither. Note that this abstraction throws away details; the ordering of members of a list is important but that of elements of a set is not.

8.5 The Cost of Abstraction

One of the main purposes of abstraction is to aid problem solving. It would therefore be very interesting to consider the computational benefits of using abstraction. The procedure for mapping back described in Chapter 7 naturally suggest a simple computational model for calculating the benefits of using a hierarchy of abstractions. This model considers four factors:

1. the time to abstract the problem;

2. the time spent theorem proving in the most abstract space finding an abstract proof;

3. the time to build the abstract proof plans;

4. the time spent theorem proving whilst trying to refine these abstract proof plans.

To compute the cost of theorem proving (items 2 and 4), we will consider just three parameters: the branching rate b, the length of proof l and the time t to perform one inference. We will also assume some complete search procedure like breadth first search. Since we must explore $b + b^2 + ... + b^{l-1} = \sum_{i=1}^{l-1} b^i$ nodes, we can define a **cost function** for the time to find a proof:

$$c(b, l, t) = \sum_{i=1}^{l-1} b^i t = \frac{b}{b-1}(b^{l-1} - 1)t$$

Although we have assumed, a constant branching rate and thus a search exponential in the length of the proof, similar qualitative results would be obtained for a variable branching rate (*eg.* a super-exponential search) or indeed any other cost function provided its boundary conditions and derivatives were similarly behaved. To consider the costs of inference in the different spaces, we will index b, l, and t with the number of the level (using "0" for the ground space). For example, t_0 represents the time to perform one inference in the ground space, whilst b_2 gives the branching rate in the second level of abstraction.

We will use τ_n to represent the time to prove a theorem using n levels of abstraction. To compute this time, we introduce $\tau(m, n)$, the time to prove the theorem at the m-th level using the levels m to n. Clearly, $\tau_n = \tau(0, n)$. We can define $\tau(m, n)$ recursively on m. Because of the way we map back, the recursion "runs backwards" from $m = n$ to $m = 0$ having its base case at $\tau(n, n)$. The base case is simply the time to prove the theorem in the most abstract space; that is, $c(b_n, l_n, t_n)$. The step case gives the time to prove the theorem in the m-th space in terms of the time to abstract the wff (represented by a_m), the time to prove the theorem in the $m + 1$-th space (that is $\tau(m + 1, n)$), the time to build the abstract proof plan (that is, the time to unabstract the l_{m+1} wffs of the abstract proof, $l_{m+1}u_m$), and the time to refine this plan.

The time to refine the abstract proof plan depends on the size of each of the $(l_{m+1} - 1)$ gaps in the plan; using the method of Lagrange multipliers, we can show that $\tau(m, n)$ is minimised for any convex cost function if the abstract proof plan has steps that need an equal amount of refining. We will therefore assume that the size of each of the $(l_{m+1} - 1)$ gaps in the abstract proof plan at the m-th level of abstraction is a constant, g_m where $(g_m - 1)(l_{m+1} - 1) = l_m - 1$. The time to refine the abstract proof plan is therefore $(l_{m+1} - 1)c(b_m, g_m, t_m)$. Note that we assume a constant time to abstract and unabstract wffs (a_m and u_m respectively). We also ignore mapping failures, abstract proofs which cannot be mapped back. In this sense, our model is optimistic; it computes the best possible savings an abstraction can provide. Thus, we define $\tau(m, n)$ as follows:

$$\begin{aligned}
\tau(n, n) &= c(b_n, l_n, t_n) \\
\tau(m, n) &= a_m + \tau(m + 1, n) + l_{m+1}u_m + (l_{m+1} - 1)c(b_m, g_m, t_m)
\end{aligned}$$

We can now compare the time to prove a theorem without abstraction, τ_0 against the time with one level of abstraction, τ_1:

- $\tau_0 \;=\; c(b_0, l_0, t_0)$ as required;

- $\tau_1 \;=\; a_0 + c(b_1, l_1, t_1) + l_1u_1 + (l_1 - 1)c(b_0, g_0, t_0)$

τ_1 consists of essentially two competing cost functions: the cost of proving the abstract theorem and the cost of refining the gaps. As we increase the strength of the abstraction, l_1 decreases and the cost of proving the abstract theorem decreases; at the same time, g_0 and the cost of refining the gaps increases. The result is that τ_1 is **convex**. At $\frac{l_1}{l_0} = 1$, τ_1 is bigger than τ_0. As the strength of the abstraction is increased τ_1 decreases since the term in $\tau(b_1, l_1, t_1)$ drops off exponentially. Eventually the term in $\tau(b_0, g_0, t_0)$, which has been quietly growing exponentially, takes over and τ_1 starts to increase again. By the time $\frac{l_1}{l_0} = 0$, τ_1 is again bigger than τ_0. The exact behaviour of τ_1 is plotted in figure 8.1.

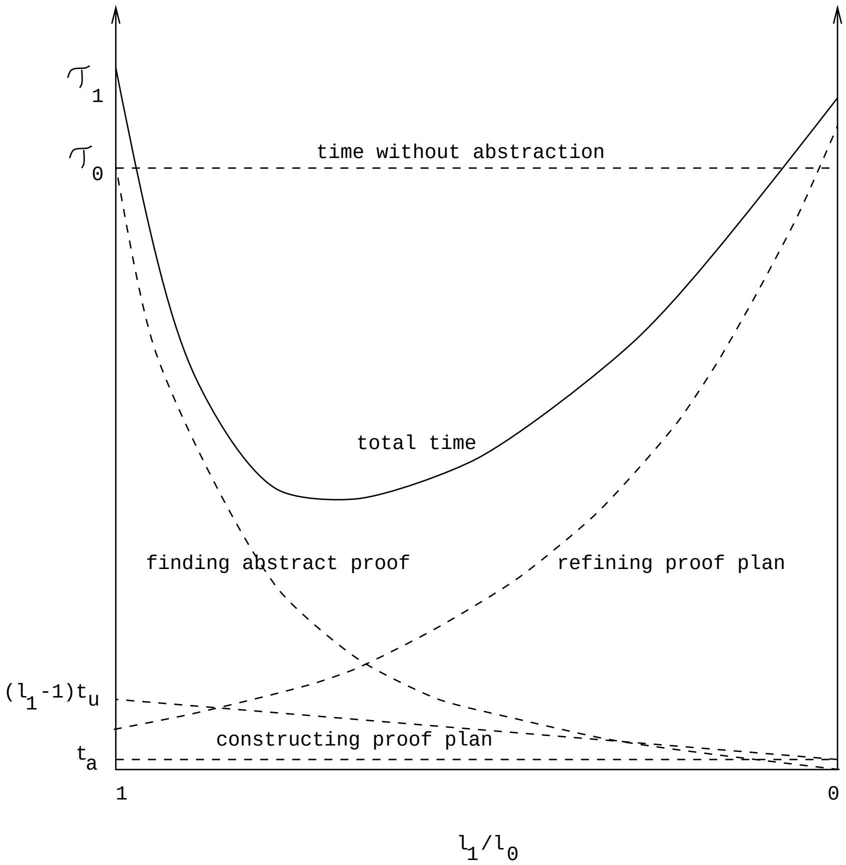


Figure 8.1: The time taken to prove a theorem using one level of abstraction, τ_1 plotted against the length of the abstract proof, l_1 with l_0 fixed.

The various partial derivatives of τ_n offer interesting information about the benefits of abstraction. In particular:

- by considering $\frac{\partial \tau_n}{\partial l_i}$, we can prove that the greatest benefits arise when the levels are evenly spaced;
- by considering $\frac{\partial \tau_n}{\partial l_n}$, we can prove there is an optimum strength for the most abstract space, neither too strong (when refining the abstract proof plan is too difficult) nor too weak (when proving the abstract theorem is too difficult);
- by considering $\frac{\partial \tau_n}{\partial n}$, we can also prove that if the spacing between abstraction levels is fixed, then there is an optimum number of abstraction levels.

Although this model may appear to be well developed mathematically, it is still *much* too early to be certain of its value; this will be determined solely by the quality of its predictions. We must therefore compare the model against empirical observation. Only then can we properly justify the simplifying assumptions we have made. Such an analysis might, for example, suggest that we cannot ignore (as we have done) mapping failures.

8.6 A Theory of Analogy

Abstraction is related to many other areas of AI, like analogy. Both abstraction and analogy are general purpose heuristics used in problem solving. Although we do not go as far as Polya in claiming that *"... Analogy pervades all our thinking ..."* (page 37 of [Pol45]), we would agree that analogy forms an important part of reasoning. Since analogy and abstraction are closely related, any sufficiently general theory of abstraction should identify the connections and the differences between the two.

There are many competing and sometimes complex approaches to the use of analogy in AI. Owen [Owe87], for example, proposes an approach in which:

> *we find a proof to a* **base** *problem which analogical matches the* **target** *problem we wish to solve; this proof is used to construct a* **plan** *to guide the search for a proof to the* **target** *problem.*

This is very similar (dare I say, "analogous"?) to our proposal for the use of abstraction:

> *we find a proof to an* **abstract** *problem which is the abstraction of the* **ground** *problem we wish to solve; this proof is used to construct a* **plan** *to guide the search for a proof to the* **ground** *problem.*

However, there are significant differences between the two approaches. Perhaps the most major difference is that the target and base representations of an analogy are more weakly connected than the ground and base representations of an abstraction. For example, "$x * y = 1$" and "$x/y = 1$" in a base representation might be considered analogous to both "$x + y = 0$" and "$x - y = 0$" in the target. Thus, an analogy connects the target and base representations by a many-to-many relation between the two languages and not, as in abstraction, by a many-to-one function; this many-to-many relation captures the concept of **analogy match** that is central to most analogy systems.

To describe an analogy, we therefore need to give (at least) the target and the base representations and the relation between them. As before, we can use formal systems as a very general method for describing the different representations. A tentative definition of analogy is therefore:

> **Definition 41 (Analogy)** *: An* **analogy***, written* $\mathrm{f} : \Sigma_1 \leadsto \Sigma_2$ *is a triple consisting of the formal systems* Σ_1*,* Σ_2 *and an* **analogy relation***,* $\mathrm{f}(\varphi_1, \varphi_2)$ *between the formulae* φ_1 *of the language of* Σ_1 *and* φ_2 *of the language of* Σ_2*.*

Strictly speaking the analogy relation is a meta-relation in some meta-theory which can name wffs from both the target and base languages, so we should perhaps use $\mathrm{f}(\lceil\varphi_1\rceil, \lceil\varphi_2\rceil)$ where $\lceil\varphi_1\rceil$, $\lceil\varphi_2\rceil$ represent the encoding in the meta-language of the base and target formulae, φ_1 and φ_2. Often the languages of

Σ_1 and Σ_2 will be identical and the analogy relation is an equivalence relation being reflexive, symmetrical, and transitive. The transitivity requirement is quite strong and perhaps the first to be dropped; although φ_1 may be analogous to φ_2, and φ_2 to φ_3, a large difference can exist between φ_1 and φ_3. For example, "$x + y = 0$" might be analogous to "$x * y = 1$", and "$x * y = 1$" to "$x^y = 1$" but we might not want "$x + y = 0$" to be analogous to "$x^y = 1$".

We can now start to see the relationship between analogy and abstraction; the class of abstractions, $\mathcal{ABS}$ is a subset of the class of analogies. And one of the major difference between analogy and abstraction is that an analogy is a *relation* between formal systems whilst an abstraction is a *mapping* between formal systems.

Theorem 77 *: An abstraction $f : \Sigma_1 \Rightarrow \Sigma_2$ is an analogy* $\mathrm{f} : \Sigma_1 \rightsquigarrow \Sigma_2$ *for which* $\mathrm{f}(\varphi_1, \varphi_2) \leftrightarrow f(\varphi_1) = \varphi_2$.

Proof: Immediate from Definitions 2 and 41. $\Box$

This notion of analogy, like our notion of abstraction, is very weak and more general than many of those previously used. In particular, we have only specified how statements in the target system are related to those in the base system. Since we want to use analogy to guide our problem solving, we will also want to know how the problems we can solve in one system are related to those we can solve in the other. As with abstraction, we could further characterise analogies by the relationships between the theorems and the proofs of the two formal systems, and use these relationships to help map target proofs back onto base proofs.

An alternative, but in many ways equivalent definition of analogy is that the target and base systems have a common abstraction. That is, we define an analogy between Σ_1 and Σ_2 by two abstractions, $g : \Sigma_1 \Rightarrow \Sigma_3$ and $h : \Sigma_2 \Rightarrow \Sigma_3$. This is equivalent to our previous definition, $\mathrm{f} : \Sigma_1 \rightsquigarrow \Sigma_2$ if:

$$\mathrm{f}(\varphi_1, \varphi_2) \;\leftrightarrow\; g(\varphi_1) = h(\varphi_2)$$

Gaines and Shaw suggest such an approach in [GS82], although they use mappings between categories and not mappings between formal systems. One possible criticism of this approach is that the system, Σ_3 that relates Σ_1 to Σ_2 may be rather artificial and not correspond to anything meaningful.

We have suggested how we might start to construct a theory of analogy, and how this theory might relate to our theory of abstraction. These are only tentative proposals since many problems still remain to be solved. Perhaps one of the largest problems is how we capture the concept of closeness of analogy; as some analogies are better than others, we want to be able to express how good a particular match is. This would go some way to overcoming the transitivity problem we identified earlier.

8.7 Summary

We have developed a general theory of abstraction. This has provided a comprehensive understanding of what abstraction is, and how it works. There are, however, many ways in which this theory could be further developed.

Bibliography

[Avr87] A. Avron. *Simple consequence relations*. LFCS Report Series 30, Laboratory for the Foundations of Computer Science, Computer Science Department, University of Edinburgh, 1987.

[Bas89] D. Basin. *Building Problem Solving Environments in Constructive Type Theory*. PhD thesis, Department of Computer Science, Cornell University, 1989. Also available as Technical Report TR 89-1063.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

[BG77] R. Burstall and J. Goguen. Putting theories together to make specifications. In *Proceedings of the 5th IJCAI*, pages 1045–1058, International Joint Conference on Artificial Intelligence, 1977.

[BG80] R. Burstall and J. Goguen. The semantics of CLEAR, a specification language. In *Proceedings of 1979 Copenhagen Winter School on Abstract Software Specification*, pages 29–332, Springer LNCS, 1980.

[BGW90] A. Bundy, F. Giunchiglia, and T. Walsh. Building abstractions. In *Proceedings of the AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions*, pages 221–232, American Association for Artificial Intelligence, 1990. Also available as DAI Research Paper No 506, Dept. of Artificial Intelligence, Edinburgh.

[Ble83] W.W. Bledsoe. Using examples to generate instantiations of set variables. In *Proceedings of the 8th IJCAI*, pages 892–901, International Joint Conference on Artificial Intelligence, 1983.

[BS84] R. Bull and K. Segerberg. Basic Modal Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, pages 1–88, D. Reidel Publishing Company, 1984.

[BSH90] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S.L.H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6, 1990. Also available from Edinburgh as DAI Research Paper 448.

[Bun83] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648, Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449.

[CAB*86] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1986.

[CGM87] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67, Elsevier Science Publishers B.V. (North-Holland), 1987.

[Cha79] C.L. Chang. Resolution plans in theorem proving. In *Proceedings of the 6th IJCAI*, pages 143–148, International Joint Conference on Artificial Intelligence, 1979.

[Coh85] A. Cohn. On the solution of Schubert's steamroller in many sorted logic. In *Proceedings of the 9th IJCAI*, International Joint Conference on Artificial Intelligence, 1985.

[Gel59] H. Gelernter. Realization of a geometry theorem-proving machine. In *Proceedings IFIP Congress*, pages 273–282, 1959.

[GG88] F. Giunchiglia and E. Giunchiglia. Building complex derived inference rules: a decider for the class of prenex universal-existential formulas. In *Proceedings of the 7th ECAI*, 1988. Extended version available as DAI Research Paper 359, Dept. of Artificial Intelligence, Edinburgh.

[Gol79] W.D. Goldfarb. The Undecidability of the Second-order Unification Problem. 1979. Unpublished manuscript.

[Gor86] M.J.C. Gordon. Why higher order logic is a good formalism for specifying and verifying hardware. In *Proceedings of 1985 Edinburgh Conference on VSLI*, North-Holland, 1986.

[Gre69] C. Green. Application of theorem proving to problem solving. In *Proceedings of the 1st IJCAI*, pages 219–239, International Joint Conference on Artificial Intelligence, 1969.

[GS82] B.R. Gaines and L.G. Shaw. Analysing analogy. In R. Trappl, L. Ricciardi, and G. Pask, editors, *Progress in Cybernetics and Systems Research Vol. IX*, pages 379–386, Hemisphere, 1982.

[GW89a] F. Giunchiglia and T. Walsh. Abstract Theorem Proving. In *Proceedings of the 11th IJCAI*, International Joint Conference on Artificial Intelligence, 1989. Also available as DAI Research Paper No 430, Dept. of Artificial Intelligence, Edinburgh.

[GW89b] F. Giunchiglia and T. Walsh. *Abstract theorem proving: mapping back*. Research Paper 460, Dept. of Artificial Intelligence, University of Edinburgh, 1989.

[GW89c] F. Giunchiglia and T. Walsh. Abstracting into inconsistent spaces (or the false proof problem). In *Proceedings of AI*IA 89*, Associazione Italiana per l'Intelligenza Artificiale, 1989. Also available as DAI Research Paper, Dept. of Artificial Intelligence, Edinburgh.

[GW89d] F. Giunchiglia and T. Walsh. Theorem Proving with Definitions. In *Proceedings of AISB 89*, Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1989. Also available as DAI Research Paper No 429, Dept. of Artificial Intelligence, Edinburgh.

[GW90a] F. Giunchiglia and T. Walsh. Abstraction in AI. *AISB Quaterly*, (73):22–26, 1990.

[GW90b] F. Giunchiglia and T. Walsh. Abstraction in automatic inference. In *Proceedings of the UK-IT 90 Conference*, pages 365–370, 1990.

[Hay79] P. Hayes. The naive physics manifesto. In D. Michie, editor, *Expert Systems in the Microelectronic Age*, Edinburgh University Press, 1979.

[Hay85] P. Hayes. The revised naive physics manifesto. In *Formal theories of the commonsense world*, Norwood Ablex Pub. Corp., 1985.

[Hen75] L.J. Henschen. Semantic resolution for horn sets. In *Proceedings of the 4th IJCAI*, International Joint Conference on Artificial Intelligence, 1975.

[Her67] J. Herbrand. Investigations in proof theory: The properties of true propositions, PhD. Thesis 1930. In J.V. Heijenoort, editor, *From Frege to Godel: A Source book in Mathematical Logic, 1879-1931*, Harvard University Press, 1967.

[HO80] G. Huet and D.C. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal languages: perspectives and open problems*, Academic Press, 1980. Presented at the conference on formal language theory, Santa Barbara, 1979. Available from SRI International as technical report CSL-111.

[Hob85] J.R. Hobbs. Granularity. In *Proceedings of the 9th IJCAI*, pages 432–435, International Joint Conference on Artificial Intelligence, 1985.

[Hun89] W.A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5:429–460, 1989.

[Imi87] T. Imielinski. Domain abstraction and limited reasoning. In *Proceedings of the 10th IJCAI*, pages 997–1003, International Joint Conference on Artificial Intelligence, 1987.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[Kno89] C.A. Knoblock. A theory of abstraction for hierarchical planning. In Paul Benjamin, editor, *Proceedings of the Workshop on Change of Representation and Inductive Bias*, Kluwer, Boston, MA, 1989.

[Kow75] R. Kowlaski. A proof procedure using connection graphs. *Journal of the Association for Computing Machinery*, 22(4):227–260, 1975.

[McC77] J. McCarthy. Epistemological problems of artificial intelligence. In *Proceedings of the 5th IJCAI*, pages 1038–1044, International Joint Conference on Artificial Intelligence, 1977.

[McC80] J. McCarthy. Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[McC88] W. W. McCune. Otter users' guide 0.91. 1988. Maths and CS. Division, Argonne National Laboratory, Argonne, Illinois.

[Mel87] T.F. Melham. *Abstraction Mechanisms for Hardware Verification.* Technical Report 106, University of Cambridge, Computer Laboratory, 1987.

[Min81] M. Minsky. A framework for representing knowledge. In J. Haugeland, editor, *MIND DESIGN. Philosophy, Psychology, Artificial Intelligence*, pages 95–128, The MIT Press, Cambridge, Mass., 1981.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2:90–121, 1980.

[Nil80] N.J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co., 1980.

[NS72] A. Newell and H.A. Simon. *Human Problem Solving*. Prentice-Hall, 1972.

[Owe87] S.G. Owen. *Finding and Using Analogies to Guide Mathematical Proof*. PhD thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1987. Examined in 1988.

[Pla80] D.A. Plaisted. Abstraction mappings in mechanical theorem proving. In *5th Conference on Automated Deduction*, pages 264–280, 5th Conference on Automated Deduction, 1980.

[Pla81] D.A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:47–108, 1981.

[Pla84] D.A. Plaisted. Using examples, case analysis, and dependency graphs in theorem proving. In *7th Conference on Automated Deduction*, pages 356–374, 7th Conference on Automated Deduction, 1984.

[Pla86] D.A. Plaisted. Abstraction using generalization functions. In *8th Conference on Automated Deduction*, pages 365–376, 8th Conference on Automated Deduction, 1986.

[Plu87] D. Plummer. *Gazing: Controlling the Use of Rewrite Rules*. PhD thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1987.

[Pol45] G. Polya. *How to Solve It*. Princeton University Press, Princeton, NJ, 1945.

[Pra65] D. Prawitz. *Natural Deduction - A proof theoretical study*. Almquist and Wiksell, 1965.

[Rei73] R. Reiter. A semantically guided deductive system for automatic therem proving. In *Proceedings of the 3rd IJCAI*, pages 41–46, International Joint Conference on Artificial Intelligence, 1973.

[Sac73] E.D. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. In *Proceedings of the 3rd IJCAI*, International Joint Conference on Artificial Intelligence, 1973.

[Sac74] E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[San82] D. Sannella. *Semantics, Implementation and Pragmatics of CLEAR, a program specification language.* PhD thesis, Department of Computer Science, Edinburgh University, 1982. Also Technical report CST-17-82.

[Sim88] A. Simpson. *Grazing: A Stand Alone Tactic for Theoretical Inference.* Master's thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1988.

[Sim89] A. Simpson. Planning the unfolding of definitions. In *Proceedings of AI*IA-89*, 1989. Also IRST research paper.

[Spi87] J.M. Spivey. *Understanding Z: a specification language and its formal semantics.* Cambridge University Press, Cambridge, England, 1987. Also PhD thesis, Oxford University, 1985.

[Ten87] J.D. Tenenberg. Preserving Consistency across Abstraction Mappings. In *Proceedings of the 10th IJCAI*, pages 1011–1014, International Joint Conference on Artificial Intelligence, 1987.

[Ten88] J.D. Tenenberg. *Abstraction in Planning.* PhD thesis, Computer Science Department, University of Rochster, 1988. Also TR 250.

[War87] K. Warren. *Implementation of a Definition Expansion Mechanism in a Connection Method Theorem Prover.* Master's thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1987.

[WR25] A.N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, Cambridge, England, 1925.

[WRD65] L. Wos, G. Robinson, and D.Carson. The automatic generation of proofs in the language of mathematics. In *Proceedings of the IFIP Congress 65*, pages 325–326, Barton Books, 1965.

Appendix A

Prolog Code

consistent-abs

```
% ---------------------------------------------------------------
%
% consistent-abs/2
%
% This Prolog program finds a hierarchy of abstractions which are
% guaranteed to have consistent abstract spaces. It has the mode
% consistent(+Sigma,-SetOfAbs). The user needs to supply the
% following two predicates:
%
%  axioms(Sigma,Axioms).
%         modes +Sigma, -Axioms    returns a list of axioms for Sigma;
%               -Sigma, +Axioms    returns a formal system with axioms
%                                  given by Axioms. The user can choose
%                                  any data structure they like for
%                                  Sigma that unifies with a variable.
%                                  A list of axioms is probably the
%                                  simplest
%  predicates(Sigma,Names).
%         modes +Sigma, -Names     returns an ordered list of the
%                                  names of predicates in Sigma
%
%----------------------------------------------------------------


% operator definitions for logical connectives

:-op( 900,xfx,[<=>]).
:-op( 800,xfx,[=>]).
```

```
:-op( 700,xfy,[&]).
:-op( 700,xfy,[v]).
:-op( 600, fx,[-]).

% consistent-abs/2 returns a hierarchy of abstractions,
% SetOfAbs of the ground space Sigma all of which have
% consistent abstract spaces.

consistent-abs(Sigma,SetOfAbs):-
        generate(Sigma,SetOfAbs),
        strongest(SetOfAbs,F),
        abstract(Sigma,F,AbsSigma),
        consistent(AbsSigma).

% generate/2 returns a hierarchy of abstractions of Sigma
% with the strongest abstraction propositional (and therefore
% decidable).

generate(Sigma,[prop(Abs),Abs]):-
        generate(Depth),
        generate(Sigma,Possibles,Depth),
        member(Abs,Possibles).

% generate/1 returns numbers of increasing size starting at 1.

generate(1).
generate(N):-
        generate(M),
        N is M+1.

% generate/3 returns the set of all possible predicate abstractions
% which divide the language of Sigma into N equivalence classes.
% A predicate abstraction is represented by a list whose elements
% are lists of equivalence classes

generate(Sigma,PredAbs,N):-
        predicates(Sigma,Names),
        setof(Abs,split(Names,N,Abs),PredAbs).

% split/3 divides a list into N ordered lists

split(List,1,[List]).
split(List,N,Ans):-
        N>1,
        M is N-1,
```

```
        split(List,M,PartialAns),
        split(PartialAns,Ans).

% split/2 divides a list into 2 ordered lists

split(Last,Next):-
        append(Front,[Split|Back],Last),
        permute(Split,Permuted),
        append([OneElement|Tail],[Another|Rest],Permuted),
        sort([OneElement,Another],[OneElement,Another]),
        append(Front,[[OneElement|Tail],[Another|Rest]|Back],Next),
        sort(Next,Next),
        sorted(Next).

% permute/2 returns all possible permutations of a list

permute([], []).
permute(List, [First|Permute]) :-
        select(First, List, Rest),
        permute(Rest, Permute).

% select/3 removes one element from a list

select(Element, [Element|Rest], Rest).
select(Element, [Head|Tail], [Head|Rest]) :-
        select(Element, Tail, Rest).

% append/2 joins two lists together

append([],List,List).
append([Head|Tail],List,[Head|Rest]):-
        append(Tail,List,Rest).

% sorted/1 checks whether a list is a sorted list of sorted lists

sorted([]).
sorted([Head|Tail]):-
        sort(Head,Head),
        sorted(Tail).

% strongest/2 returns the strongest abstraction in a hierarchy

strongest([Abstraction|_],Abstraction).

% abstract/3 maps Sigma onto AbsSigma using the abstraction F
```

```
abstract(Sigma,F,AbsSigma):-
        axioms(Sigma,Axioms),
        map(F,Axioms,AbsAxioms),
        axioms(AbsSigma,AbsAxioms).

% map/3 applies the mapping function to the axioms

map(_F,[],[]).
map(F,[Wff|Rest],[AbsWff|AbsRest]):-
        apply-abs(F,Wff,AbsWff),
        map(prop(PredAbs),Rest,AbsRest).

% apply-abs/3 applies the abstraction's mapping function to a wff

apply-abs(prop(Pred),forall(_X,P),Q):-
        apply-abs(prop(Pred),P,Q).
apply-abs(prop(Pred),exists(_X,P),Q):-
        apply-abs(prop(Pred),P,Q).
apply-abs(prop(Pred),P <=> Q,R <=> S):-
        apply-abs(prop(Pred),P,R),
        apply-abs(prop(Pred),Q,S).
apply-abs(prop(Pred),P => Q,R => S):-
        apply-abs(prop(Pred),P,R),
        apply-abs(prop(Pred),Q,S).
apply-abs(prop(Pred),P v Q,R v S):-
        apply-abs(prop(Pred),P,R),
        apply-abs(prop(Pred),Q,S).
apply-abs(prop(Pred),P & Q,R & S):-
        apply-abs(prop(Pred),P,R),
        apply-abs(prop(Pred),Q,S).
apply-abs(prop(Pred),-P,-Q):-
        apply-abs(prop(Pred),P,Q).
apply-abs(prop(Pred),P,Q):-
        literal(P),
        P=..[Name,Arg],
        scan(Pred,Name,EquivClass),
        construct(EquivClass,AbsName),
        Q=AbsName.

% scan/2 searches a list of lists returning any list that
% contains the element Name

scan([EquivClass|_],Name,EquivClass):-
        member(Name,EquivClass).
scan([Front|Rest],Name,EquivClass):-
```

```
        \+(member(Name,Front)),
        scan(Rest,Name,EquivClass).

% construct/2 is a "dirty" (alias meta-logical) predicate
% for building a name for an equivalence class. It compounds
% together the names of the individual members of the
% equivalence class.

construct(EquivClass,Name):-
        ascii_list(EquivClass,Ascii),
        name(Name,Ascii).

% ascii_list/2 returns a list of the ascii codes of all the
% names in a list

ascii_list([],[]).
ascii_list([Head|Tail],Ans):-
        name(Head,AsciiList),
        ascii_list(Tail,Rest),
        append(AsciiList,Rest,Ans).

% consistent/1 checks whether AbsSigma is consistent by calling
% Otter, a fast resolution theorem prover. It prepares a file
% temp.in for running Otter. Output is directed to a file called
% temp.out. Otter is rather verbose in its output; temp.out will
% contain the word PROOF iff a proof of [], or inconsistency
% is found.

consistent(AbsSigma):-
        prepare-file(AbsSigma),
        unix(shell('otter < temp.in >& temp.out')),
        \+unix(shell('grep PROOF temp.out > /dev/null')).

% prepare-file/1 prepares an input file, temp.in for Otter

prepare-file(AbsSigma):-
        open('temp.in',write,Stream),
        write-header(Stream),
        write-axioms(Stream,AbsSigma),
        write-footer(Stream),
        close(Stream).

% write-header/1 outputs the header of an input file for Otter.
% It selects binary resolution, with factoring and the set of
% support search strategy
```

```
write-header(Stream):-
        write(Stream,'set(binary_res).'),
        nl(Stream),
        write(Stream,'set(factor).'),
        nl(Stream),nl(Stream),
        write(Stream,'list(sos).'),
        nl(Stream).

% write-axioms/2 outputs the axioms of AbsSigma to Stream

write-axioms(Stream,AbsSigma):-
        axiom(AbsSigma,Axioms),
        write-list(Stream,Axioms).

% write-list/2 outputs a list of wffs to Stream

write-list(_,[]).
write-list(Stream,[Head|Rest]):-
        write-clause(Stream,Head),
        write-list(Stream,Rest).

% write-clause/2 outputs a clause to Stream

write-clause(Stream,P & Q):-
        write-clause(Stream,P),
        write-clause(Stream,Q).
write-clause(Stream,Clause):-
        \+(Clause = (_P & _Q)),
        write-disjunct(Stream,Clause),
        write(Stream,'.'),
        nl(Stream).

% write-disjunct/2 outputs a disjunct to Stream

write-disjunct(Stream,P v Q):-
        write-disjunct(Stream,P),
        write(Stream,' | '),
        write-disjunct(Stream,Q).
write-disjunct(Stream,Literal):-
        \+(Literal = (_P v _Q)),
        print(Stream,Literal).

% write-footer/1 outputs the footer to an input file for Otter
```

```
write-footer(Stream):-
        write(Stream,'end_of_list.'),
        nl(Stream),nl(Stream).
```

mapback

```
% ---------------------------------------------------------------
%
% mapback/5
%
% This Prolog program maps an abstract proof back onto a ground
% proof. It has the mode, mapback(+Phi,+Sigma,+F,+AbsProof,-Proof).
% The user needs to supply the following predicates:
%
% abstraction(F,Type)
%    mode +F, +Type             F is an abstraction of type Type;
%                               this can be "predicate", "domain",
%                               "operator" or "function".
% horn(Sigma)
%    mode +Sigma                Sigma is a Horn clause theory
%
% axiom(Sigma,Axiom)
%    modes +Sigma, +Axiom       Axiom is an axiom of Sigma
%          +Sigma, -Axiom
%
% abstract(+F,+Term,-AbsTerm)
%    modes +F, +Term, +AbsTerm  F is either predicate, domain or
%          +F, +Term, -AbsTerm  function. Term is a predicate name,
%                               constant name or function name.
%                               AbsTerm is the abstraction of Term.
%
% ---------------------------------------------------------------


% operator definitions for logical connectives

:-op(500,yfx,[:]).
:-op(500,xf,[-]).
:-op(300,yfx,[of]).

% mapback/5 maps an abstract proof back onto a ground proof. The
% abstract and ground proofs use LUSH resolution (with the ancestor
% restriction dropped for non-Horn clause problems). Proofs are
% represented as lists of binary resolutions and factorings. Each
```

```
% binary resolution is represented by the term, resolve(Phi,Rho).
% Each factoring is represented by the term, factor(Phi). Formulae
% are clauses represented by lists of atomic literals. Sorts to
% terms are represented by Term:Sort. This notational trick allows
% Prolog unification to do a limited form of sorted unification.
% Arguments to sorted terms are represented by structures of the
% form "Name of ArgList". For example, "p of (f of [x])" represents
% "p(f(x))". This notational trick allows Prolog unification to
% do a very limited form of higher order unification.

mapback(Phi,Sigma,F,AbsProof,Proof):-
        plan(AbsProof,F,Plan),
        refine(Plan,Sigma,Phi,F,Proof),
        valid(Proof,Phi,Sigma).

% plan/3 turns an abstract proof into a minimal abstract proof plan
% (or more accurately, a meta-level schema for all possible plans)

plan([],_F,[]).
plan([factor(AbsWff)|AbsProof],F,[factor(Wff)|Plan]):-
        unabstract(F,AbsWff,Wff),
        plan(AbsProof,F,Plan).
plan([resolve(AbsPhi,AbsRho)|AbsProof],F,[resolve(Phi,Rho)|Plan]):-
        unabstract(F,AbsPhi,Phi),
        unabstract(F,AbsRho,Rho),
        plan(AbsProof,F,Plan).

% unabstract/3 maps an abstract wff onto a meta-level term
% representing all of its possible unabstractions

unabstract(_,[],[]).
unabstract(F,[AbsLit|AbsRest],[Lit|Rest]):-
        unabstractatom(F,AbsLit,Lit),
        unabstract(F,AbsRest,Rest).

% unabstractatom/3 maps an abstract atomic literal onto a
% meta-level term representing all of its possible unabstractions

unabstractatom(F,-AbsLit,-Lit):-!,
        unabstractlit(F,AbsLit,Lit).
unabstractatom(F,AbsLit,Lit):-
        unabstractlit(F,AbsLit,Lit).

% unabstractlit/3 maps an abstract positive literal onto a
% meta-level term representing all of its possible unabstractions
```

```
unabstractlit(F,P,P of _Args):-
        abstraction(F,propositional).
unabstractlit(F,Pred,_P:Sort of Args):-
        abstraction(F,predicate),
        Pred=..[Sort|AbsArgs],
        unabstractargs(F,AbsArgs,Args).
unabstractlit(F,Pred,Name of Args):-
        \+abstraction(F,propositional),
        \+abstraction(F,predicate),
        Pred=..[Name|AbsArgs],
        unabstractargs(F,AbsArgs,Args).

% unabstractargs/3 maps a list of abstract arguments to a literal
% onto a list representing all of its possible unabstractions

unabstractargs(_F,[],[]).
unabstractargs(F,[Var|AbsArgs],[Var:_Sort|Args]):-
        abstraction(F,function),
        variable(Var),
        unabstractargs(F,AbsArgs,Args).
unabstractargs(F,[Function|AbsArgs],[_Function of Arg:Sort|Args]):-
        abstraction(F,function),
        \+variable(Function),
        Function=..[Sort|Rest],
        unabstractargs(F,Rest,Arg),
        unabstractargs(F,AbsArgs,Args).
unabstractargs(F,[Var|AbsArgs],[Var:_Sort|Args]):-
        abstraction(F,domain),
        variable(Var),
        unabstractargs(F,AbsArgs,Args).
unabstractargs(F,[Function|AbsArgs],[Name of Arg:_Sort|Args]):-
        abstraction(F,domain),
        Function=..[Name,Arg1|Rest],
        unabstractargs(F,[Arg1|Rest],Arg),
        unabstractargs(F,AbsArgs,Args).
unabstractargs(F,[Constant|AbsArgs],[_:Constant|Args]):-
        abstraction(F,domain),
        constant(Constant),
        unabstractargs(F,AbsArgs,Args).
unabstractargs(F,[Var|AbsArgs],[Var:_Sort|Args]):-
        abstraction(F,operator),
        variable(Var),
        unabstractargs(F,AbsArgs,Args).
unabstractargs(F,[Fun|AbsArgs],[Fun of _Arg:_Sort|Args]):-
```

```
        abstraction(F,operator),
        \+variable(Fun),
        unabstractargs(F,AbsArgs,Args).
unabstractargs(F,[Arg1|Rest],[Arg1|Rest]):-
        \+abstraction(F,operator),
        \+abstraction(F,domain),
        \+abstraction(F,function).

% variable/1 checks whether a term is a variable

variable(Var):-
        var(Var).

% constant/1 checks whether a term is a constant. Any 0-ary term
% which is not a variable is assumed to be a constant.

constant(Constant):-
        \+variable(Constant),
        Constant=..[_Name].

% refine/5 tries to refine an abstract proof plan into a
% ground proof

refine(Plan,Sigma,Phi,F,Proof):-
        generate(SortProof,Plan),
        sort(F,Phi,SortPhi),
        apply(Plan,Sigma,SortPhi,F,[SortPhi],SortProof),
        desort(SortProof,Proof).

% generate/2 returns a list of uninstantiated variables that
% is at least as long as its second argument

generate(List1,List2):-
        generate(List1),
        aslong(List2,List1).

% generate/1 returns a list of uninstantiated variables

generate([]).
generate([_|Rest]):-
        generate(Rest).

% aslong/2 checks whether its first argument is a longer
% list than its second argument

aslong([],_).
```

```
aslong([_|List1],[_|List2]):-
        aslong(List1,List2).

% apply/6 tries to apply an abstract proof plan. That is, it tries
% to find instantiations for the meta-level terms representing the
% unabstractions so that the binary resolutions and factorings go
% through. The 5th argument is an accumulator of derived clauses;
% this is only used for non-Horn theories when we perform ancestor
% resolution.

apply([],_Sigma,_Phi,_F,_SoFar,[]).
apply([resolve(Phi,Rho)|Plan],Sigma,Phi,F,SoFar,
      [resolve(Phi,Rho)|Proof]):-
        axiom(Sigma,UnsortRho),
        sort(F,UnsortRho,SortRho),
        Rho = SortRho,
        binary_resolve(Phi,Rho,Resolvent),
        apply(Plan,Sigma,Resolvent,F,[Resolvent|SoFar],Proof).
apply([resolve(Phi,Rho)|Plan],Sigma,Phi,F,SoFar,
      [resolve(Phi,Rho)|Proof]):-
        \+horn(Sigma),
        \+(\+member(Rho,SoFar)),
        binary_resolve(Phi,Rho,Resolvent),
        apply(Plan,Sigma,Resolvent,F,[Resolvent|SoFar],Proof).
apply([factor(Phi)|Plan],Sigma,Phi,F,SoFar,[factor(Phi)|Proof]):-
        factor(Phi,Factor),
        apply(Plan,Sigma,Factor,F,[Factor|SoFar],Proof).
apply(Plan,Sigma,Phi,F,SoFar,[resolve(Phi,Rho)|Proof]):-
        aslong(Plan,Proof),
        axiom(Sigma,UnsortedRho),
        sort(F,UnsortedRho,SortRho),
        SortRho=Rho,
        binary_resolve(Phi,Rho,Resolvent),
        apply(Plan,Sigma,Resolvent,F,[Resolvent|SoFar],Proof).
apply(Plan,Sigma,Phi,F,SoFar,[factor(Phi)|Proof]):-
        aslong(Plan,Proof),
        factor(Phi,Factor),
        apply(Plan,Sigma,Factor,F,[Factor|SoFar],Proof).
apply(Plan,Sigma,Phi,F,SoFar,[resolve(Phi,Rho)|Proof]):-
        \+horn(Sigma),
        aslong(Plan,Proof),
        member(Rho,SoFar),
        binary_resolve(Phi,Rho,Resolvent),
        apply(Plan,Sigma,Resolvent,F,[Resolvent|SoFar],Proof).
```

```
% binary_resolve/3 binary resolves two clauses together; we perform
% copy_term before doing this so that shared variables elsewhere in
% the proof are not instantiated. Note also that (in complementary/2)
% we call unify/2, a true unification algorithm with occurs check.

binary_resolve(Clause1,Clause2,Resolvent):-
        copy_term(Clause1,[Literal|Literals]),
        copy_term(Clause2,Clause),
        remove(Comp,Clause,OneOff),
        complementary(Literal,Comp),
        append(OneOff,Literals,Resolvent).

% factor/2 factors a clause;  we call copy_term before doing this so
% that shared variables elsewhere in the proof are not instantiated.
% Note also that we call unify/2, a true unification algorithm with
% occurs check.

factor(Clause,Factor):-
        copy_term(Clause,[Lit1|Factor]),
        member(Lit2,Factor),
        unify(Lit1,Lit2).

% complementary/2 checks whether two literals are complementary

complementary(Lit1,-Lit2):-
        unify(Lit1,Lit2).
complementary(-Lit1,Lit2):-
        unify(Lit1,Lit2).

% sort/3 maps a formulae onto a sorted formulae; the sorts are
% determined by the abstraction F

sort(_F,[],[]).
sort(F,[Literal|Rest],[SortLiteral|SortRest]):-
        sortatomic(F,Literal,SortLiteral),
        sort(F,Rest,SortRest).

% sortatomic/3 maps an atomic formulae onto a sorted formulae;
% the sorts are determined by the abstraction F

sortatomic(F,-Lit,-SortLit):-!,
        sortliteral(F,Lit,SortLit).
sortatomic(F,Lit,SortLit):-
        sortliteral(F,Lit,SortLit).

% sortliteral/3 maps a positive literal onto a sorted literal
```

```
sortliteral(F,Wff,Pred of Args):-
        abstraction(F,propositional),
        Wff=..[Pred|Args].
sortliteral(F,Wff,Pred:Sort of SortArgs):-
        abstraction(F,predicate),
        Wff=..[Pred|Args],
        abstract(F,Pred,Sort),
        sortargs(F,Args,SortArgs).
sortliteral(F,Wff,Pred of SortArgs):-
        \+abstraction(F,propositional),
        \+abstraction(F,predicate),
        Wff=..[Pred|Args],
        sortargs(F,Args,SortArgs).

% sortargs/3 maps a list of arguments to a literal onto a
% a list of sorted arguments

sortargs(_F,[],[]).
sortargs(F,[Var|Args],[Var:_Sort|SortArgs]):-
        abstraction(F,domain),
        variable(Var),
        sortargs(F,Args,SortArgs).
sortargs(F,[Function|Args],[Name of Sort:_|SortArgs]):-
        abstraction(F,domain),
        Function=..[Name,Arg1|Rest],
        sortargs(F,[Arg1|Rest],Sort),
        sortargs(F,Args,SortArgs).
sortargs(F,[Constant|Args],[Constant:Sort|SortArgs]):-
        abstraction(F,domain),
        constant(Constant),
        abstract(F,Constant,Sort),
        sortargs(F,Args,SortArgs).
sortargs(F,[Var|Args],[Var:_Sort|SortArgs]):-
        abstraction(F,function),
        variable(Var),
        sortargs(F,Args,SortArgs).
sortargs(F,[Function|Args],[Name of Arg:Sort|SortArgs]):-
        abstraction(F,function),
        Function=..[Name|UnsortArg],
        abstract(F,Name,Sort),
        sortargs(F,UnsortArg,Arg),
        sortargs(F,Args,SortArgs).
sortargs(F,[Var|Args],[Var:_Sort|SortArgs]):-
        abstraction(F,operator),
```

```
        variable(Var),
        sortargs(F,Args,SortArgs).
sortargs(F,[Function|Args],[Name of Arg:_Sort|SortArgs]):-
        abstraction(F,operator),
        Function=..[Name|UnsortArg],
        sortargs(F,UnsortArg,Arg),
        sortargs(F,Args,SortArgs).
sortargs(_F,Args,Args):-
        \+abstraction(F,domain),
        \+abstraction(F,function),
        \+abstraction(F,operator).

% desort/2 strips sorts off a proof

desort([],[]).
desort([resolve(SortPhi,SortRho)|SortProof],
       [resolve(Phi,Rho)|Proof]):-
        desortwff(SortPhi,Phi),
        desortwff(SortRho,Rho),
        desort(SortProof,Proof).
desort([factor(SortPhi)|SortProof],[factor(Phi)|Proof]):-
        desortwff(SortPhi,Phi),
        desort(SortProof,Proof).

% desortwff/2 strips sorts off a formulae

desortwff([],[]).
desortwff([SortAtom|SortWff],[Atom|Wff]):-
        desortatom(SortAtom,Atom),
        desortwff(SortWff,Wff).

% desortatom/2 strips sorts off an atomic formulae

desortatom(-SortAtom,-Atom) :-
        desortlit(SortAtom,Atom),!.
desortatom(SortAtom,Atom) :-
        desortlit(SortAtom,Atom).

% desortatom/2 strips sorts off a positive literal

desortlit(P:_Sort of SortX,Lit):-
        desortargs(SortX,X),
        Lit=..[P|X],!.
desortlit(P of SortX,Lit):-
        desortargs(SortX,X),
        Lit=..[P|X].
```

```
% desortargs/3  strips sorts off a list of arguments to a literal

desortargs([],[]):- !.
desortargs([Var|SortArgs],[Var|Args]):-
        variable(Var),
        desortargs(SortArgs,Args),!.
desortargs([Var:_Sort|SortArgs],[Var|Args]):-
        variable(Var),
        desortargs(SortArgs,Args),!.
desortargs([F of SortX:_Sort|SortArgs],[Function|Args]):-
        desortargs(SortX,X),
        Function=..[F|X],
        desortargs(SortArgs,Args),!.
desortargs([F of SortX|SortArgs],[Function|Args]):-
        desortargs(SortX,X),
        Function=..[F|X],
        desortargs(SortArgs,Args),!.
desortargs([Constant:_Sort|SortArgs],[Constant|Args]):-
        desortargs(SortArgs,Args),!.
desortargs([Constant|SortArgs],[Constant|Args]):-
        desortargs(SortArgs,Args).


% valid/2 checks whether a proof is valid

valid(Proof,Phi,Sigma):-
        resolve(Phi,Sigma,[Phi],Proof).

% prove/3 finds a proof using iterative deepening search. This
% is not used by mapback/5 but is usefully for finding abstract
% proofs to map back

prove(Phi,Sigma,Proof):-
        generate(Proof),
        resolve(Phi,Sigma,[Phi],Proof).

% resolve/4 finds a resolution proof. The 3rd argument is an
% accumulator used to store derived formulae for ancestor resolution.

resolve([],_Sigma,_SoFar,[]).
resolve(Phi,Sigma,SoFar,[factor(Phi)|Proof]):-
        factor(Phi,Factor),
        resolve(Factor,Sigma,[Factor|SoFar],Proof).
resolve(Phi,Sigma,SoFar,[resolve(Phi,Rho)|Proof]):-
```

```
        axiom(Sigma,Rho),
        binary_resolve(Phi,Rho,Resolvent),
        resolve(Resolvent,Sigma,[Resolvent|SoFar],Proof).
resolve(Phi,Sigma,SoFar,[resolve(Phi,Rho)|Proof]):-
        \+horn(Sigma),
        member(Rho,SoFar),
        copy_term(Rho,NewRho),
        binary_resolve(Phi,NewRho,Resolvent),
        resolve(Resolvent,Sigma,[Resolvent|SoFar],Proof).

% append/3 joins two lists together

append([],List,List).
append([Head|Tail],List,[Head|Rest]):-
        append(Tail,List,Rest).

% member/2 checks whether an element is a member of a list

member(Head,[Head|_List]).
member(Head,[_|List]):-
        member(Head,List).

% remove/3 removes an element from a list

remove(Head,[Head|Tail],Tail).
remove(Member,[Head|Tail],[Head|Rest]):-
        remove(Member,Tail,Rest).
```

Appendix B

Operators

Monkey and Bananas

$at(z, x1, s) \wedge movable(z) \wedge empty(x2, s) \rightarrow at(monkey, x2, move(monkey, z, x2, s))$
$at(z, x1, s) \wedge movable(z) \wedge empty(x2, s) \rightarrow at(z, x2, move(monkey, z, x2, s))$
$at(z, x, s) \wedge climbable(y, z, s) \rightarrow at(z, x, climb(y, z, s))$
$at(z, x, s) \wedge climbable(y, z, s) \rightarrow on(y, z, climb(y, z, s))$
$at(box, under - bananas, s) \wedge on(monkey, box, s) \rightarrow reachable(monkey, bananas, s)$
$reachable(z, x, s) \rightarrow has(z, x, reach(z, x, s))$

ABSTRIPS robot operators

$type(b, object) \wedge \exists r.inroom(b, r) \wedge inroom(robot, r) \rightarrow nextto(robot, b)$
$type(d, door) \wedge \exists r1, r2.inroom(robot, r1) \wedge connects(d, r1, r2) \rightarrow nextto(robot, d)$
$\exists r.inroom(robot, r) \wedge locinroom(x, y, r) \rightarrow at(robot, x, y)$
$type(c, object) \wedge pushable(b) \wedge nextto(robot, b) \wedge \exists r.inroom(b, r) \wedge$
$\quad inroom(c, r) \wedge inroom(robot, r) \rightarrow nextto(b, c)$
$pushable(b) \wedge type(d, door) \wedge nextto(robot, b) \wedge \exists r1, r2.inroom(robot, r1) \wedge$
$\quad inroom(b, r1) \wedge connects(d, r1, r2) \rightarrow nextto(b, d)$
$pushable(b) \wedge nextto(robot, b) \wedge \exists r.inroom(robot, r) \wedge inroom(b, r) \wedge$
$\quad locinroom(x, y, r) \rightarrow at(b, x, y)$
$type(d, door) \wedge type(r1, room) \wedge status(d, open) \wedge \exists r2.inroom(robot, r2) \wedge$
$\quad connects(d, r1, r2) \rightarrow inroom(robot, r1)$
$pushable(b) \wedge type(d, door) \wedge type(r1, room) \wedge status(d, open) \wedge$
$\quad nextto(b, d) \wedge nextto(robot, b) \wedge \exists r2.inroom(b, r2) \wedge inroom(robot, r2) \wedge$
$\quad connects(d, r1, r2) \rightarrow inroom(b, r1)$
$type(d, door) \wedge status(d, closed) \wedge nextto(robot, d) \rightarrow status(d, open)$
$type(d, door) \wedge status(d, open) \wedge nextto(robot, d) \rightarrow status(d, closed)$