Vorlesungsskript *Rechnergestütztes Beweisen*

Martin Hofmann

WS 2005/06

1 Introduction

Computer-aided theorem proving means to carry out mathematical proofs on a computer whose job it is to check steps, to perform bookkeeping tasks and to automate routine steps. Conducting a proof on a computer may be compared to and has a lot in common with implementing an informally given algorithm or model. For example, a number of details must be filled in and, more importantly, mistakes and shortcomings of the high-level model are brought to the surface.

Computer-aided theorem proving has numerous applications in program and hardware verification as well as prototype development. To a lesser, perhaps increasing, degree it is used to aid the development of genuine mathematical proofs.

1.1 Course outline

In this course, we will get to know the computer-based theorem prover PVS (`pvs.csl.sri.com`) along with its theoretical foundations and some ramifications thereof.

- Logical foundations: sequent calculus, predicate calculus, higher-order logic, set theory
- Automation of logical reasoning: resolution
- Automation of equational reasoning: rewriting and decision procedures
- Finite state verification: Modal logics and model checking
- Infinite state verification: Abstract model checking

- Type theory: Modularisation, independent checking of proof certificates, computation within proofs.

Mostly in the tutorials we will apply this knowledge to a variety of problems from

- Solving logical puzzles
- Algorithms on lists and trees
- Hardware components such as adder, counter, multiplier
- Distributed algorithms using invariants and reasoning
- Distributed algorithms using abstract model checking
- Experiments with other theorem provers.

1.2 Notions of proof

What exactly is a “proof”. When asked this question a typical mathematician would produce something like the following: A proof is a convincing, undebatable argument establishing the truth of a mathematical statement. Back to Euclid (300 B.C.) goes the following concretion of this definition: A proof is a derivation of a statement from axioms by means of logical rules.

This sound good, but it remains to say what “axioms” should be and what the “logical rules” are. In Euclid’s case (geometry) the axioms were truisms such as “for any two non-equal points there is exactly one line passing through them”. The logical rules were essentially the ones we still use today and will learn about later in the course. An example of such a rule: if “A implies B” holds and “A” holds then “B” holds, too (*modus ponens*).

Later on, more complicated concepts such as real numbers and limits were introduced which made it less clear what reasonable axioms should be. For example, even the famous 18th century mathematician Leonhard Euler struggled with the infinite series $1 - 1 + 1 - 1 + 1 - 1 + \dots$ and ended up ascribing the value $1/2$ to it on the basis of the same informal mathematical reasoning he used for his celebrated theorems.

The lack of solid logical foundations for mathematics, and in particular analysis led to an actual crisis in mathematics (*Grundlagenkrise*) which was settled early in the last century by the invention of set theory (and, following up on this, by the formalisation of real numbers, limits, integrals and so on by Weierstrass, Riemann and others.)

1.2.1 Set theory

Set theory is a formalism which allows one to *define* all other mathematical concepts and to *prove* their axioms, thus enabling a rigorous proof of their *consistency*, i.e., sensibility. For instance, we can define points as triples of real numbers (which in turn are defined as certain sequences of rational numbers (which in turn are defined as certain pairs of integer numbers (which in turn are defined as certain pairs of natural numbers (which are defined as certain sets: $0 = \emptyset$, $1 = \{0\}$, $2 = \{0, 1\}$, etc.)))) and then *prove* that through any two distinct points goes one and only one line (a line being defined, for instance, as a set of points satisfying some linear relation).

The present formulation of set theory consists of approximately nine axioms[1] among them

- two sets having the same elements are equal
- for each set we can form the set of its subsets
- there exists an infinite set
- for each set we can form the subset consisting of those elements sharing a given property
- for each set of sets A there exists a set containing exactly one element of each nonempty set in A ("axiom of choice")

Using a formalised language statements in set theory can be written as strings and recognised as such, for example the first four axioms are written as follows:

- $\forall a.\forall b.(\forall x.x \in a \iff x \in b) \Rightarrow a = b$
- $\forall a.\exists b.\forall x.x \in b \iff x \subseteq a$
  (where $x \subseteq a \overset{\text{def}}{=} \forall y.y \in x \Rightarrow y \in a$)

[1] the precise number depends on what we count as axiom and what as a logical rule.

- $\exists a.\neg\text{finite}(a)$

  (where $\text{finite}(a) \overset{\text{def}}{=} \forall b.(\emptyset \in b \wedge (\forall c.\forall x.c \in b \wedge x \in a \Rightarrow c \cup \{x\} \in b)) \Rightarrow a \in b$)

  Here $c \cup \{x\}$ is a notation for a set whose existence is asserted by two other axioms (union and singleton).

- $\forall a.\exists b.\forall x.x \in b \Leftrightarrow (x \in a \wedge \phi(x))$ (where $\phi(x)$ is an arbitrary statement involving x)

We should remark at this point that despite the formal notation the axioms of set theory necessarily remain unproved and their justification relies on philosophical and pragmatic arguments.

The *logical rules* of set theory are precisely the ones of first-order logic which we are going to learn about in more detail later in the course.

1.2.2 Proofs as formal derivations

Once we have a formal concept of axioms and rules, we can define a *proof* of a statement ϕ as a sequence of statements

$$\phi_0, \phi_1, \phi_2, \phi_3, \ldots, \phi_n = \phi$$

ending in ϕ such that each ϕ_i is either an axiom or follows from previous statements by a logical rule.

So, to check whether an alleged proof indeed is one is a matter of entirely mechanical symbol manipulation and does not require any creative skills or intelligence.

Rather than merely asserting the next formula ϕ_i one might tell by which logical rule it follows and which of the ϕ_j, $j < i$ were used as premises for that inference. In this way, one arrives at the notions of proof tree or proof-DAG. (DAG=directed acyclic graph = tree with shared nodes).

In practice, however, writing out proofs at this level of detail would be far too cumbersome and so checking whether a purported proof, say in a journal submission, indeed is one, does require considerable mathematical skill and devotion!

And while the vast majority of mathematicians agrees that any proof in mathematics can *theoretically* be formalised in set theory and hence mechanically checked, many of them believe that *in practice* such formalisation is impossible for all but the simplest toy examples. This belief might have remained unchallenged if there had not been the request for formalised proofs from informatics and the advent of sufficiently powerful machines.

1.3 Proof assistants

So, why do we need formalised proofs in informatics? Well, the correctness of software (or hardware) is nothing but a mathematical statement amenable to formalisation and mechanical checking.

Here are some examples of formal theorem proving occurring in informatics.

- Is program X correct? *very rare*
- Does method X satisfy invariant Y?
- Does variable X always hold values with property Y? E.g., Y=points to a sorted linked list, a balanced binary tree, data items consistent with store.
- Does protocol X guarantee property Y? E.g., Y=cache coherence, sequential execution, absence of deadlock.
- Does circuit X implement function Y? E.g. Y=FP multiplication, Fourier transform.
- Does algorithm X satisfy specification Y? E.g. X=garbage collector, Y=absence of interference+liveness.
- Does theorem X about programming language Y hold? E.g. X= type safety, correctness of proof rules.
- Verification of certificates (“proof-carrying code”)

While in early stages of soft- and hardware development these proofs could be carried out by hand (possibly using some notation and intermediate calculations) the size of systems has reached a state where this has become impossible in many cases.

Prompted by these requirements systems called *proof assistants* or *theorem provers* have been developed which perform not only the task of checking sizeable formalised proofs but also help with coming up with formalised proofs in the first place by bookkeeping assumptions and variables, providing tactics and decision procedures (e.g., for propositional formulas, certain fragments of arithmetic, modal and temporal logic, equational theories, etc.) and by providing libraries of definitions and already proved theorems.

- Bookkeeping of assumptions and variables
- Tactics
- Type checking
- Decision procedures
- Libraries of definitions and theorems

Figure 1: Tasks of a proof assistant

- Expressive logic
- Powerful decision procedures
- Large body of basic notions

Figure 2: Strengths of PVS

1.3.1 The PVS System

In this course we will get to know one such proof assistant in some detail, namely the PVS system developed by Owre, Rushby, and Shankar at SRI, Menlo Park, California. As any system, PVS has a number of strengths and also weaknesses. As a partial compensation for the weaknesses we will later in the course take a look at complementary systems such as SPASS, Coq, Isabelle.

PVS has a very expressive underlying logic (classical higher-order logic), it comes equipped with a number of powerful decision procedures, e.g., for linear arithmetic and equational reasoning, and it has a large body of basic notions which allow one to start a formalisation on a relatively high level.

On the other hand, PVS has recurrent soundness problems, that is, from time to time someone finds out that a weird combination of tactics use and language features allows one to prove $0 = 1$! Moreover, there is no formal representation of proofs. One reason why these problems are not trivial to fix is precisely the large body of basic notions which here turns into a disadvantage.

- Soundness problems
- No formal representation of proofs
- Large body of basic notions
- (Bad heuristics for first-order instantiation)

Figure 3: Weaknesses of PVS

1.3.2 Soundness and proof objects

A problem with a proof assistant is that its correct behaviour is hard to verify. Whether a word processor provides decent looking output can be checked at a glance (correctness for all inouts notwithstanding), similarly a video game either is fun to play with or not.

On the other hand, correct behaviour of a proof assistant is rather hard to detect. After all it's because we don't want to do the proofs by hand that we use a proof assistant in the first place. The "output" of a proof assistant doesn't consist of a nice looking document or a thrilling sequence of images.

In PVS you can have a complicated looking subgoal to prove, you type in `(grind)` and PVS responds that this proves the statement. There is no way to check this proof independently; all that's being recorded is that the tactic `(grind)` has been invoked.

PVS stands for *prototype verification system* which is explained by the following quote from the PVS Prover Guide 2.3, see `pvs.csl.sri.com`.

> *The primary purpose of PVS is to provide formal support for conceptualization and debugging in early stages of the life cycle of a hardware or software system. In these stages, both the requirements and designs are expressed in abstract terms that are not necessarily executable. We find that the best way to analyse such an abstract specification is by attempting proofs of desirable consequences of the specification.*

So, provided soundness problems occur rarely [2] they do not really compromise the usability of the system.

[2] they do sometimes, see the PVS web site

```
 |-------
[1]   FORALL (y: t, v_106: list[t]):
        (FORALL (x: t):
           occ(x, merge(null, v_106)) =
                 occ(x, null) + occ(x, v_106))
         IMPLIES
         (FORALL (x: t):
            occ(x, merge(null, cons(y, v_106))) =
             occ(x, null) + occ(x, cons(y, v_106)))
Rule? (grind)
```

lots of rewrites etc. are printed

```
This completes the proof of merge1.4.
```

Figure 4: A quick proof in PVS

Proofs as guarantee In recent years researchers have proposed a use of proofs as a certificate not unlike the cryptographic certificates such as digital signatures, etc. While the latter certify authenticity of a datum, i.e., a relationship between the datum and the sender, a formal proof certifies a property of the datum itself which is independent of the sender.

For example, a third-party provider of a component of a safety-critical system might be required to provide a formal proof of correctness.

A referee of a paper in mathematics or theoretical informatics might not be willing to verify all details of a proof but would rather run the formalised proofs of the theorems in the paper through a proof checker.

Also it has been proposed under the name *proof-carrying code* that mobile code should be equipped with independently checkable proofs of certain safety properties, e.g. memory safety, type safety, etc.

The design of systems like Coq (to some extent also Isabelle) is such that independent verification is possible. These systems generate a formal representation of a proof (a *proof object* or *proof term*) amenable to separate verification by a *proof checker* which is simple and small. Even if tactics or decision procedures contain bugs these will always show up at the checking stage so the worst that can happen is that an attempted proof has to be redone.

At present it seems that none of the systems with explicit proof objects can

Atoms: $A, B, C, D, \ldots$
Connectives:
$\phi \wedge \psi$: ϕ "and" ψ (conjunction)
$\phi \vee \psi$: ϕ "or" ψ (disjunction)
$\phi \Rightarrow \psi$: ϕ "implies" ψ (implication)
$\neg\phi$: "not" ϕ (negation)

Precedence: $\neg, \wedge, \vee, \Rightarrow$
Example: $(A \Rightarrow B) \wedge \neg A \Rightarrow \neg A$ reads $((A \Rightarrow B) \wedge (\neg A)) \Rightarrow (\neg A)$

Figure 5: Syntax of propositional formulas

compete with PVS or similar systems. However, I think that this is mainly a problem of organisation and manpower, not an inherent theoretical one. I believe that in the not too distant future we will see powerful proof assistants with (almost) bug-free proof-checkers inside.

2 Sequent calculus

In this section we will learn about the "logical rules" which underly the PVS system. In their present form they were introduced by the logician Gerhard Gentzen around 1940 as a means to analyse the proof-theoretic strength of formal arithmetic.

2.1 Formulas

We start with a set of *atoms*, aka *identifiers* or *symbols* $A, B, C, D, \ldots$. *Formulas* are built up from atoms by the *connectives* $\vee, \wedge, \Rightarrow$ (binary) and $\neg$ (unary), so $(A \Rightarrow B) \wedge \neg A \Rightarrow C$ is a formula. By convention the binding power is $\neg > \wedge > \vee > \Rightarrow$, so the above formula reads $((A \Rightarrow B) \wedge (\neg A)) \Rightarrow C$.

The *meaning* of a formula is given relative to an interpretation of the atoms as either true or false. For instance, if A is true, and B, C are both false, then our example formula will be true because then $A \Rightarrow B$ is false (the only way for an implication to be false is that its antecedent (here A) is true and its consequent (here B) is false).

Digression on semantics of implication Please notice that this so-called *classical interpretation of implication* is sometimes at odds with our intuitive understanding of implication. For instance, the sentence “if MH wears a tie during the lecture then he can turn lead into gold.” is actually true under this interpretation. It is possible to formalise the intuitive meaning of such sentence by implicitly quantifying over a set of *worlds* that describe possibilities. One could then imagine a world in which MH wears ties and also worlds in which he can turn lead into gold, but the latter would not form a superset of the former because there is no causal relationship whatsoever. On the other hand, for $\phi \Rightarrow \psi$ to be valid in this refined sense one requires that the set of worlds in which ψ holds forms a superset of the set of worlds in which ϕ holds.

Another paradox involving classical implication goes as follows: it is commonly agreed that in order to show that a recursively defined method $m()$ is correct it suffices to show that its body e is correct assuming that any recursive calls to the method already perform correctly. In other words:

$$(m() \text{ correct} \Rightarrow e \text{ correct}) \Rightarrow m() \text{ correct}$$

Were this rule valid in the sense of classical implication then any method would be correct: either it is correct in the first place or else it isn’t in which case the premise to the above rule is trivially true whereby correctness of the method follows from the rule!

Formalisation of meaning Anyway, under the aforementioned classical interpretation the formula $\phi \overset{\text{def}}{=} \neg A \Rightarrow (A \Rightarrow B)$ always comes out true, no matter what A and B actually stand for.

Likewise, $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ always holds.

Formally, a partial function η mapping atoms to $\{\mathsf{tt}, \mathsf{ff}\}$ can be extended to formulas by interpreting the connectives in the obvious way.

A formula is a *tautology* if its meaning is true regardless of the interpretation of the atoms.

A formula is *satisfiable* if it is true for *some* interpretation of the atoms. Clearly, A is satisfiable if and only if $\neg A$ is not a tautology and therefore A is a tautology if and only if $\neg A$ is unsatisfiable.

2.2 Applications of propositional logic

Many naturally occurring problems admit encodings in propositional logic in the sense that to know whether a certain formula is satisfiable or a tautology amounts

If $\eta(A) = \mathsf{tt}, \eta(B) = \eta(C) = \mathsf{ff}$ then

$$\begin{array}{rl} & \eta((A \Rightarrow B) \wedge \neg A \Rightarrow C) \\ = & \eta(A \Rightarrow B) \wedge \eta(\neg A) \Rightarrow \mathsf{ff} \\ = & (\mathsf{ff} \Rightarrow \mathsf{tt}) \wedge \mathsf{ff} \Rightarrow \mathsf{ff} = \mathsf{tt} \end{array}$$

Figure 6: Formal meaning of a formula

$$A \Rightarrow A$$

$$((A \Rightarrow B) \Rightarrow A) \Rightarrow A$$

$$\begin{array}{l} \neg((A_{11} \vee A_{12}) \wedge (A_{21} \vee A_{22}) \wedge (A_{31} \vee A_{32}) \wedge \\ \quad \neg(A_{11} \wedge A_{21}) \wedge \neg(A_{11} \wedge A_{31}) \wedge \neg(A_{21} \wedge A_{31}) \wedge \\ \quad \neg(A_{12} \wedge A_{22}) \wedge \neg(A_{12} \wedge A_{32}) \wedge \neg(A_{22} \wedge A_{32})) \end{array}$$

Figure 7: Examples of tautologies

to having a solution to the problem at hand. Examples are summarised in Figure 2.1.

It is therefore an important practical problem to determine whether a given propositional formula is a tautology or not and (equivalently) whether or not it is satisfiable.

A formula with 100 atoms admits ca. 10^{33} different valuations so checking tautologies by examining truth tables may be unfeasible.

- if Mary likes champaign then either Bob or Alice like red wine…
- Planning problems: find sequence of actions for a robot to—say—remove a certain item from a stockpile
- Behaviour of digital hardware circuits
- Combinatorial optimisation (scheduling, routing,…)

Figure 8: Applications of propositional logic

Sequents: $\Gamma \Longrightarrow \Delta$ where $\Gamma = \phi_1, \ldots, \phi_m$ and $\Delta = \psi_1, \ldots, \psi_n$ are lists of formulas.
Meaning: $\phi_1 \wedge \cdots \wedge \phi_m \Rightarrow \psi_1 \vee \cdots \vee \psi_n$
Examples: $A \Rightarrow B, C \Rightarrow D \Longrightarrow U \wedge (\neg V \wedge B)$
$A \Rightarrow B, A \Longrightarrow B$
$\Longrightarrow A, A \Rightarrow B$
$A, \neg A \Longrightarrow$
$\Longrightarrow$

Figure 9: Syntax of sequents

While no method is known to date which would be inherently better than checking truth tables there has been considerable progress in the last years at solving instances arising from practical problems (SAT solvers). These solvers vastly outperform any human logician trying to attack propositional formulas by logical reasoning! So why should we look at axioms and logical rules for propositional calculus?

The answer is that in many situations the atoms will themselves be complex formulas, typically a defined predicate applied to some variables, such as $x \in a$ or $\mathbf{sorted}(\mathbf{list}_1)$ and we want to be able to break down the validity of a formula involving these atoms into basic implications between them.

This is precisely the goal of sequent calculus which we will now describe.

2.3 Sequents

A *sequent* is an expression of the form $\Gamma \Longrightarrow \Delta$ where Γ, Δ are (possibly) empty lists of formulas.

The *meaning* of a sequent $\Gamma \Longrightarrow \Delta$ is defined as the meaning of the formula $\bigwedge \Gamma \Rightarrow \bigvee \Delta$ where $\bigwedge \Gamma$ is the conjunction (“and”, $\wedge$) of the formulas in Γ and $\bigvee \Delta$ is the disjunction (“or”, $\vee$) of the formulas in Δ.

For example, our formula $(A \Rightarrow B) \wedge \neg A \Rightarrow C$ is equivalent to the sequent $A \Rightarrow B, \neg A \Longrightarrow C$.

A *proof* in sequent calculus is a tree labelled with sequents such that the leaves are labelled with *axioms* and the label of an internal node is the conclusion of a *rule* which has the labels of its immediate descendants as premises.

A sequent $\Gamma \Longrightarrow \Delta$ is an *axiom* if Γ and Δ have a formula in common. For example, the sequent $A, B \Longrightarrow A, C$ is an axiom.

The *rules* for sequent calculus are best read backwards, i.e. “what do I need to

$$\frac{\Gamma_1, \phi, \psi, \Gamma_2 \Longrightarrow \Delta}{\Gamma_1, \psi, \phi, \Gamma_2 \Longrightarrow \Delta} \qquad (\text{P}\textsc{erm}\text{-L})$$

$$\frac{\Gamma \Longrightarrow \Delta_1, \phi, \psi, \Delta_2}{\Gamma \Longrightarrow \Delta_1, \psi, \phi, \Delta_2} \qquad (\text{P}\textsc{erm}\text{-R})$$

$$\frac{\Gamma \Longrightarrow \Delta}{\Gamma, \phi \Longrightarrow \Delta} \qquad (\text{W}\textsc{eak}\text{-L})$$

$$\frac{\Gamma \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta, \phi} \qquad (\text{W}\textsc{eak}\text{-R})$$

$$\frac{\Gamma, \phi, \phi \Longrightarrow \Delta}{\Gamma, \phi \Longrightarrow \Delta} \qquad (\text{C}\textsc{ontr}\text{-L})$$

$$\frac{\Gamma \Longrightarrow \Delta, \phi, \phi}{\Gamma \Longrightarrow \Delta, \phi} \qquad (\text{C}\textsc{ontr}\text{-R})$$

Figure 10: Structural rules

prove in order to establish a sequent $\Gamma \Longrightarrow \Delta$?". First, we have *structural rules* allowing to permute, duplicate or remove formulas. In the literature sequents are often defined as pairs of *sets* rather than lists of formulas. This makes all the structural rules except WEAK redundant. For computer-aided formal reasoning it is, however, useful to have explicit access to formulas, say by their position in a list.

For each connective there are two *logical rules*, one when the connective appears on the left, and one when it appears on the right.

To understand, e.g., $\vee$-L try to think as follows: to prove Δ under the assumption $\phi \vee \psi$ (and some other stuff Γ) we must make a case distinction as to whether ϕ or ψ holds, hence we must prove Δ under assumptions Γ, ϕ and then again under assumptions Γ, ψ.

Rule $\Rightarrow$-R is probably the easiest of these: to prove $\phi \Rightarrow \psi$ we must prove ψ under the additional assumption ϕ (if we disregard the side formulas $\Gamma, \Delta \ldots$). If we forget about Δ we can also explain $\neg$-R: to prove $\neg\phi$ we must derive a contradiction (empty Δ) from the assumption ϕ.

The $\neg$-L rule says: if $\neg\phi$ is among our assumptions then to prove anything (Δ) its enough to prove ϕ (or Δ straightaway, of course). This is known as *ex*

$$\frac{\Gamma, \phi, \psi \Longrightarrow \Delta}{\Gamma, \phi \wedge \psi \Longrightarrow \Delta} \tag{$\wedge$-L}$$

$$\frac{\Gamma \Longrightarrow \Delta, \phi \qquad \Gamma \Longrightarrow \Delta, \psi}{\Gamma \Longrightarrow \Delta, \phi \wedge \psi} \tag{$\wedge$-R}$$

$$\frac{\Gamma, \phi \Longrightarrow \Delta \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \vee \psi \Longrightarrow \Delta} \tag{$\vee$-L}$$

$$\frac{\Gamma \Longrightarrow \Delta, \phi, \psi}{\Gamma \Longrightarrow \Delta, \phi \vee \psi} \tag{$\vee$-R}$$

$$\frac{\Gamma \Longrightarrow \Delta, \phi}{\Gamma, \neg\phi \Longrightarrow \Delta} \tag{$\neg$-L}$$

$$\frac{\Gamma, \phi \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta, \neg\phi} \tag{$\neg$-R}$$

$$\frac{\Gamma \Longrightarrow \Delta, \phi \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \Rightarrow \psi \Longrightarrow \Delta} \tag{$\Rightarrow$-L}$$

$$\frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \Delta, \phi \Rightarrow \psi} \tag{$\Rightarrow$-R}$$

$$\dfrac{\dfrac{\dfrac{A \Longrightarrow B, A \quad B, A \Longrightarrow B}{A \Rightarrow B, A \Longrightarrow B}\ \Rightarrow\text{-L}}{A \Rightarrow B \Longrightarrow B, \neg A}\ \neg\text{-L}}{A \Rightarrow B, \neg B \Longrightarrow \neg A}\ \neg\text{-L}$$

Figure 11: Example proof

$$\dfrac{\dfrac{A, B \Longrightarrow A, A \wedge C \quad A, B \Longrightarrow B, A \wedge C}{A, B \Longrightarrow A \wedge B, A \wedge C} \quad \dfrac{A, C \Longrightarrow A \wedge B, A \quad A, C \Longrightarrow A \wedge B, C}{A, C \Longrightarrow A \wedge B, A \wedge C}}{A, (B \vee C) \Longrightarrow A \wedge B, A \wedge C}$$

Figure 12: Example proof

falso quodlibet.

Rule $\Rightarrow$-L, finally, says: if we want to use the assumption $\phi \Rightarrow \psi$ then we can add its conclusion (ψ) to our assumptions provided we succeed (independently) in proving its antecedent (ϕ).

One can derive the implication rules from the encoding of $\phi \Rightarrow \psi$ as $\neg\phi \vee \psi$.

2.4 Soundness and completeness

Definition: A sequent is *derivable* if there exists a proof with it as root label.

Theorem: A sequent is derivable if and only if it is a tautology.

Proof: Let us call a *proof tree* a tree whose nodes (and leaves) are labelled with sequents in such a way that whenever a node labelled S has immediate ancestors $S_1, \dots, S_n$ then there is a logical rule with $S_1, \dots, S_n$ as assumptions as S as conclusion. For example, $S_1 = A \Longrightarrow B$ and $S_2 = A \Longrightarrow C$ and $S = A \Longrightarrow B \wedge C$. Given the form of our rules we always have $n = 1$ or $n = 2$. The leaves may but do not need to be labelled with axioms. Let us write $S_1, \dots S_\ell \vdash S$ to mean that there is a proof tree whose root is labelled S and whose leaves are labelled with $S_1, \dots S_\ell$.

Notice that a *proof* of sequent S is a proof tree all whose leaves are labelled with axioms.

By induction on (depth of) proof trees one easily shows that if $S_1, \dots, S_n \vdash S$

and $S_1, \dots, S_n$ are all true under some valuation η then S, too, comes out true under η. Recall that the truth value of a sequent $S = \phi_1, \dots, \phi_m \Longrightarrow \psi_1, \dots \psi_n$ under some valuation η is defined as $\bigwedge_i \eta(\phi_i) \Rightarrow \bigvee_j \eta(\psi_j)$, i.e., S comes out false precisely if all the ϕ_i come out true and all the ψ_j come out false.

The above proves that if a sequent has a proof, i.e., a proof tree with axioms labelling its leaves, then it is tautologous, i.e., true under all valuations.

For completeness we first notice (again by induction on proof trees) that if $S_1, \dots, S_n \vdash S$ is proved by a proof tree not involving rule WEAK then S is equivalent to the conjunction of the S_i, i.e., S comes out false under some valuation η as soon as one of the S_i is falsified by η. Now, for any sequent S we can always find a proof tree not involving rule WEAK whose root is labelled S and whose leaves are labelled with sequents consisting of atoms only. This is done by successively "breaking down" all the connectives in S. If S is *not derivable* then at least one of the atomic sequents labelling the leaves of this proof tree will not be an axiom (otherwise our proof tree would be a proof!). This sequent will thus be of the form $A_1, \dots A_m \Longrightarrow B_1, \dots B_n$ where the A_i and B_j are atoms and $\{A_1, \dots, A_m\} \cap \{B_1, \dots, B_n\} = \emptyset$. Any valuation η with $\eta(A_i) = \mathtt{tt}, \eta(B_j) = \mathtt{ff}$ will falsify this sequent, hence S. So S is not a tautology. □

We remark that this also shows that if a sequent is derivable with rules CONTR and WEAK then it is provable without those rules; the "generic" proof tree obtained by breaking down the connectives must lead to a proof in this case, otherwise we would obtain a falsifying valuation.

2.4.1 Linear logic

The fact that rules WEAK and CONTR can be eliminated is due to their being built into the other rules and axioms. In *linear logic* weakening and contraction are removed and the side formulas in different premises of a rule are required to be disjoint. For instance, a linear version of $\wedge$-R would look thus:

$$\frac{\Gamma_1 \Longrightarrow \Delta_1, \phi \qquad \Gamma_2 \Longrightarrow \Delta_2, \psi}{\Gamma_1, \Gamma_2 \Longrightarrow \Delta_1, \Delta_2, \phi \wedge \psi} \qquad (\wedge\text{-R-LIN})$$

Moreover, the only axioms are $A \Longrightarrow A$ for A an atom. If we replace all rules and axioms by their linear versions and remove WEAK and CONTR (which now are no longer redundant) we obtain a (a fragment of) linear logic in which, e.g., the formula $A \Rightarrow A \wedge A$ is not provable.

2.4.2 Cut elimination

In spite of completeness it is useful to have yet another rule: the famous cut rule:

$$\frac{\Gamma_1 \Longrightarrow \Delta_1, \phi \qquad \Gamma_2, \phi \Longrightarrow \Delta_2}{\Gamma_1, \Gamma_2 \Longrightarrow \Delta_1, \Delta_2} \qquad (\textsc{Cut})$$

Usually, Γ_1 and Δ_1 are empty. In this case, CUT corresponds to invocation of a lemma: if we have proved ϕ "as a lemma" then we can add it to our list of assumptions whenever we want. The completeness entails says that lemmas can always be eliminated.

3 Introduction to PVS

As already mentioned, PVS is based on the sequent calculus. We will start by using PVS as a proof-assistant for that system.

To do that we call PVS from the (Linux!) command line with

```
pvs
```

This brings up an Emacs window entitled PVS. A possible question concerning context creation should be answered affirmatively (means: type in `yes`).

Now create a file named `sequent.pvs` containing something like

```
sequent_calculus: THEORY
BEGIN

A,B,C,D: VAR boolean
A11,A12,A21,A22,A31,A32: VAR boolean

K : PROPOSITION
    A IMPLIES B IMPLIES A

S : PROPOSITION
    (A IMPLIES B IMPLIES C) IMPLIES (A IMPLIES B)
        IMPLIES  (A  IMPLIES C)

Peirce : PROPOSITION
    ((A IMPLIES B) IMPLIES A) IMPLIES A
```

```
Contra : PROPOSITION
     (A IMPLIES B) IMPLIES NOT B IMPLIES NOT A

dist1 : PROPOSITION
    A AND (B OR C) IMPLIES (A AND B) OR (A AND C)

dist2 : PROPOSITION
    A AND B OR C AND D IMPLIES (A OR C) AND  (B OR D)

schub : PROPOSITION
    NOT (
     (A11 OR A12) AND
     (A21 OR A22) AND
     (A31 OR A32) AND
     NOT (A11 AND A21) AND
     NOT (A11 AND A31) AND
     NOT (A21 AND A31) AND
     NOT (A12 AND A22) AND
     NOT (A12 AND A32) AND
     NOT (A22 AND A32))

END sequent_calculus
```

Click on the first "proposition". This will bring up a prover window and a prompt `Rule?`. The first rule you should enter is `(skolem!)`. This will get rid of the `FORALL` quantifier which we'll talk about later and basically display the formula as a sequent with empty premise list and one conclusion named `[1]`. Unfortunately, all the atoms are decorated with `!1`. ¡ Now you can

- apply a disjunctive rule ($\wedge$-L, $\vee$-R, $\Rightarrow$-R) by entering `(flatten` x`)` where x is the number of the formula you want to apply the rule to. (x must obviously be 1 at the beginning.

- apply a conjunctive rule ($\wedge$-R, $\vee$-L, $\Rightarrow$-L) by entering `(split` x`)` where, again, x is the number of the formula.

The rules for negation are applied automatically.

Try to prove all the propositions in this way but don't waste too much time on the last one (`schub`).

Rather give up after a few steps by typing `(quit)` and redo the proof (answering no when asked whether you want to rerun the existing proof) this time typing `(prop)` after the `(skolem!)` step or `(grind)` right at the start.

You can display the tree structure of your current proof with `M-x x-show-current-proof` and of a finished proof with `M-x x-show-proof`.

You may wish to do some more ad-hoc experiments with PVS. Extensive documentation is available at the PVS homepage `pvs.csl.sri.com`.

4 Resolution

Gentzen's sequent calculus provides a decision procedure for the validity of propositional formulas: construct the proof tree as in the completeness proof and check whether all leaves are labelled with axioms.

Unfortunately, the complexity of this procedure is exponential in the size of the formula to be proved. This is due to the duplication of "goals" in rules $\vee$-L, $\wedge$-R, $\Rightarrow$-L. Worse still, a lot of work is done twice: if we break down $\Gamma \implies \Delta, \phi \wedge \psi$ into $\Gamma \implies \Delta, \phi$ and $\Gamma \implies \Delta, \psi$ then the "breaking down" of Γ and Δ must be done in each branch individually.

Of course, unless $P = NP$ we cannot expect a really efficient (polynomial) method for deciding propositional formulas; however, there are algorithms that behave quite well in practice for moderately sized formulas. One of these is the method of *resolution* invented by ROBINSON which we will take a look at for one thing for its popularity and for another because its applicability to first-order logic which we will come to shortly.

Resolution is a method for deciding *satisfiability* of a propositional formula presented as a *set of clauses*. Recall that a formula ϕ is satisfiable if $\neg\phi$ is not a tautology. A *literal* is either an atom or a negated atom, e.g., $A, \neg B, \mathrm{door\_closed}, \neg\mathrm{alarm\_on}$ are all literals. A *clause* is a set of literals whose meaning is their disjunction. Some people write a clause as an explicit disjunction using $\vee$, others use set notation. A *set of clauses* represents the conjunction of the individual clauses. Watch out for the empty clause (representing $\mathbb{ff}$) and the empty set of clauses (representing $\mathbb{tt}$).

To check whether a formula ϕ is a tautology we can represent $\neg\phi$ as a set of clauses and see whether it is not satisfiable.

- a *literal* is either an atom or a negated atom: $A, \neg B, \neg\mathrm{door\_closed}$.
- a *clause* is a set of literals understood as their *disjunction*: $\{\neg\mathrm{lift\_moves}, \mathrm{door\_closed}, \mathrm{alarm\_on}\}$
- a *set of clauses* is understood as the conjunction of the individual clauses.
- empty clause = ff, empty set of clauses = tt
- One is interested in *satisfiability* of sets of clauses.
- Validity (to be a tautology) is trivial for sets of clauses. Why?

Figure 13: Clauses

4.1 Representation of formulas as sets of clauses

We all know that any formula ϕ can be converted into conjunctive normal form, by "multyplying out" according to de Morgan's rule. The problem with this is that in general the size of the conjunctive normal form will be exponential in the size of the formula to start with. Actually, if this blow up would not occur we had a simple method for checking whether ϕ is a tautology. Just bring ϕ into conjunctive normal form and see whether each clause is a tautology.

What we can do without exponential blowup, though, is to construct a set of clauses $\mathcal{C}$ which is *satisfiable* if and only if ϕ is. To that end, we proceed as follows: first, we may assume that ϕ contains connectives $\vee, \wedge, \neg$ only and that, moreover, $\neg$ occurs in front of atoms only. One calls this the negation normal form of ϕ. Now, if ϕ happens to be a literal then there is nothing to do. If $\phi = \phi_1 \wedge \phi_2$ and $\phi_1, \phi 2$ are equi-satisfiable with $\mathcal{C}_1, \mathcal{C}_2$, respectively, then $\mathcal{C}_1 \cup \mathcal{C}_2$ is equi-satisifable with ϕ. If, finally, $\phi = \phi_1 \vee \phi_2$ then $(\mathcal{C}_1 \vee P) \cup (\mathcal{C}_2 \vee \neg P$ is equi-satisfiable with ϕ. Here P is a fresh atom and $\mathcal{C} \vee P$ means the addition of P to each clause in $\mathcal{C}$.

4.2 The method of resolution

The method of resolution decides whether a given set of clauses is satisfiable. It works as follows: given two clauses C_1 and C_2 such that C_1 contains some literal ℓ and C_2 contains its negation $\neg\ell$ (with the understanding that the negation of $\neg A$ is A) then the *rule of resolution* applied to C_1, C_2 yields the clause $C_1 \setminus \{\ell\} \cup C_2 \setminus \{\neg\ell\}$. For example, applying the rule of resolution to $\{A, \neg B, D\}$ and

Rule of resolution:

$$\frac{C_1 \cup \{\ell\} \qquad C_2 \cup \{\neg\ell\}}{C_1 \cup C_2}$$

Example: $\{\neg A, B, D\}$ and $\{\neg B, X\}$ yield $\{\neg A, D, X\}$.

Figure 14: The rule of resolution

```
INPUT: a set of clauses $\mathcal{C}$
WHILE $\emptyset \notin \mathcal{C}$ OR $\mathcal{C}$ still grows DO
    CHOOSE $C_1, C_2 \in \mathcal{C}$ S. T. $\ell \in C_1$ and $\neg\ell \in C_2$ for some literal $\ell$.
    $\mathcal{C} := \mathcal{C} \cup \{C_1 \setminus \{\ell\} \cup C_2 \setminus \{\neg\ell\}\}$
IF $\emptyset \in \mathcal{C}$ THEN OUTPUT "$\mathcal{C}$ is not satisfiable"
    ELSE OUTPUT "$\mathcal{C}$ is satisfiable"
```

Figure 15: The method of resolution

$\{A, \neg D, E\}$ yields $\{A, \neg B, E\}$. The method of resolution consists of closing up a set of clauses under the rule of resolution and seeing whether the so closed-up set ontains the empty clause or not. This is formalised in Fig.15. Note that if the initial clause set $\mathcal{C}$ is finite then the algorithm terminates since there is only a finite number of possible clauses over any given (finite) set of variables.

4.3 Correctness of resolution

If the set of clauses—after this closure—contains the empty clause then it is unsatisfiable, otherwise we can find a satisfying valuation. This is the content of the correctness theorem for resolution.

A set of clauses $\mathcal{C}$ is *closed under resolution* if the result of applying the rule of resolution to any two clauses in $\mathcal{C}$ is already contained in $\mathcal{C}$. The above method precisely computes the closure under resolution of an arbitrary set of clauses.

Let $\mathcal{C}$ be a set of clauses, A an atom. We define the clause set $\mathcal{C}[A \mapsto \mathsf{tt}]$ by removing from $\mathcal{C}$ every clause that contains the literal A and removing the literal $\neg A$ from every clause that contains it. Analogously, we define $\mathcal{C}[A \mapsto \mathsf{ff}]$. It is easily seen by case distinction that if $\mathcal{C}$ is closed under resolution so are these two sets.

If η is a valuation that satisfies $\mathcal{C}[A \mapsto v]$ then the valuation $\eta[A \mapsto v]$ which maps A to v and all other atoms according to η will satisfy $\mathcal{C}$.

Theorem: Let $\mathcal{C}$ be a possibly infinite set of clauses closed under resolution, i.e., Then $\mathcal{C}$ is satisfiable if and only if $\emptyset \notin \mathcal{C}$.

Proof: If a set of clauses $\mathcal{C}'$ has been obtained from a *satisfiable* set of clauses $\mathcal{C}$ by a single application of the rule of resolution then $\mathcal{C}'$ is satisfiable, too. One says that the rule of resolution preserves satisfiability. Thus, if we derive from $\mathcal{C}$ a set of clauses containing the empty clause then $\mathcal{C}$ must have been unsatisfiable in the first place.

For the converse, suppose that $\mathcal{C}$ does not contain the empty clause yet is closed under the rule of resolution. We explicitly construct a valuation η that will satisfy all the clauses in $\mathcal{C}$: Enumerate the atoms as $A_1, A_2, \dots$. We define the values $\eta(A_1), \eta(A_2), \dots$ in order. Let us begin with the variable A_1. Not both $\mathcal{C}[A_1 \mapsto \mathtt{tt}]$ and $\mathcal{C}[A_1 \mapsto \mathtt{ff}]$ can contain the empty clause for otherwise, $\mathcal{C}$ would contain both $\{A_1\}$ and $\{\neg A_1\}$ and hence the empty clause by resolution step. Thus, choose $\eta(A_1)$ such that $\mathcal{C}[A_1{\mapsto}\eta(A_1)]$ does not contain the empty clause. We continue in this way replacing A_1 with A_2 and $\mathcal{C}$ with $\mathcal{C}[A_1{\mapsto}\eta(A_1)]$ yielding a value $\eta(A_2)$ such that $\mathcal{C}[A_1{\mapsto}\eta(A_1)][A_2{\mapsto}\eta(A_2)]$ does not contain the empty clause.

Continuing in this way, we obtain a valuation η which satisfies all the clauses in $\mathcal{C}$. This can be seen by noticing that as the variables are considered every clause in $\mathcal{C}$ eventually disappears.

An important corollary is the compactness theorem for propositional logic:

Theorem: Let $\mathcal{C}$ be a (possibly infinite) set of clauses or propositional formulas. If every finite subset of $\mathcal{C}$ is satisfiable then the whole of $\mathcal{C}$ is satisfiable.

Proof: If contrary to the conclusion the whole of $\mathcal{C}$ is unsatisfiable then by the previous theorem it must be possible to deduce the empty clause from $\mathcal{C}$. But such a proof will only involve a finite portion of $\mathcal{C}$ which would then already be unsatisfiable contradicting the assumption.

4.4 Long resolution proofs

While for many tautologies (or rather their negations) resolution works astonishingly fast there are other ones, e.g., `schub` above for which it is rather slow. Indeed, HAKEN has shown that the resolution method has exponential worst case complexity.

Theorem (Haken): There is a constant $c > 1$ and an infinite family of (unsatisfiable) sets of clauses $P_1, P_2, \dots$ such that P_n consists of $O(n^2)$ clauses yet any derivation of $\emptyset$ from P_n will involve $O(c^n)$ many clauses.

To wit, the set of clauses P_n expresses that $n+1$ pigeons fit into n holes, e.g., $P_2 \Leftrightarrow \neg\mathtt{schub}$.

The proof of Haken's theorem is elementary but fairly long and technical. Recently, WIGDERSON has presented an important simplification. Use Google to find his paper if you are interested.

4.5 Cook's programme

Haken's theorem is quite drastic evidence for exponential time complexity of the resolution procedure. Even if Haken's theorem would not hold as stated, resolution could still fail to be a polynomial time procedure: it might be difficult to find a polynomially sized derivation of $\emptyset$ even if it exists and in the case of *satisfiable* sets of clauses the process of closing up might result in exponentially many clauses.

It is an important open complexity-theoretic question related to P=NP as to whether there is a proof system for propositional logic with the property that any tautology ϕ has a proof of size polynomial in the size of ϕ. Refuting this for concrete proof systems (as Haken has done for resolution) is a popular research activity initiated by S. COOK. As far as I know, it is presently not known whether or not each tautology has a polynomially sized proof in sequent calculus with the CUT rule.

4.6 The DPLL procedure

In practical implementations of decision procedures for propositional logic (SAT-solvers) resolution has been superseded by a surprisingly naive search procedure known as DPLL algorithm (Davis-Putnam-Loveland-Logemann).

It also operates on clause sets and is based on three interleaved steps

- Unit propagation: if the clause set contains a unit clause, i.e., one containing a single literal, then it is possible to set the value of that atom and propagate it through the other clauses. We define unit propagation formally below.

- Branching: choose an arbitrary atom A and try to satisfy first $\mathcal{C}[A{\mapsto}\mathsf{tt}]$ and $\mathcal{C}[A{\mapsto}\mathsf{ff}]$. If either turns out to be satisfiable then so is $\mathcal{C}$. Otherwise, $\mathcal{C}$ is unsatisfiable.

- Learning clauses: if during the branching it turns out that for some partial valuation η the clause set $\mathcal{C}[\eta]$ is unsatisfiable by unit propagation alone then we can identify those settings in η which lead to the empty clause and build a corresponding clause k that can be added to $\mathcal{C}$ without affecting satisfiability. If e.g., we find that setting $A{\mapsto}\mathtt{tt}, B{\mapsto}\mathtt{ff}, C{\mapsto}\mathtt{tt}$ leads to a contradiction (empty clause) then we can add the clause $\{\neg A, B, \neg C\}$ to $\mathcal{C}$. The hope is that in the future this addition will speed up unit propagation; in particular, if at a later stage we try to, again, set $A{\mapsto}\mathtt{tt}, B{\mapsto}\mathtt{ff}, C{\mapsto}\mathtt{tt}$ then we find out immeidately that this leads to a contradiction rather than having to re-perform the corresponding steps of unit propagation.

Slightly more formally DPLL can be viewed as a recursive procedure that uses a global variable containing a set of clauses $\mathcal{C}$. It is an invariant that clauses are never removed from $\mathcal{C}$ and whenever a clause k is to be added to $\mathcal{C}$ then it is a logical consequence of $\mathcal{C}$, i.e., whenever η satisfies $\mathcal{C}$ then it also satisfies k.

The procedure takes as argument a partial valuation η and *DPLL*(η) returns "satisfiable" if there exists a valuation extending η that satisifies $\mathcal{C}$; it returns "unsatisfiable", if no such valuation exists. Figure 4.6 contains pseudo code for DPLL. In a practical implementation a number of improvements are possible. Rather than using recursion one can maintain a stack of partial environments. Furthermore, it is not necessary to compute the clause set $\mathcal{C}[\eta']$ explicitly. For details, we refer to the literature.

5 First-order logic

Propositional calculus is nice, but in many applications we need a way of talking about elements, predicates, and operations. That is what first order logic is for. Figure 5 shows some examples of sentences that lend themselves to a formalisation in first-order logic. A formal system for writing down such statements is obtained by augmenting propositional logic with the *quantifiers* $\forall$ (for all) and $\exists$ (there exists). As to the range of these quantifiers one has two options which we consider in order.

Untyped first-order logic. This is the traditional and most common version of first-order logic. We let all quantifiers and variables range over one and the same implicit, a priori given, domain. In this case we must use special predicates to restrict the range of quantifiers. Figure 18 shows how the example sentences are formalised in untyped first-order logic.

```
*UNITPROP*$(\mathcal{C}, \eta) =$
  IF $\mathcal{C}[\eta]$ contains the empty clause
    let $\rho \subseteq \eta$ be the smallest sub-valuation of $\eta$ such that $\emptyset \in \mathcal{C}[\rho]$
    RETURN $(\rho,$ “unsatisfiable”$)$
  ELSE IF $\mathcal{C}[\eta]$ is empty
    RETURN $(\eta,$ “satisfiable”$)$
  ELSE IF $\mathcal{C}[\eta]$ contains a unit clause $\{A\}$
    RETURN *UNITPROP*$(\mathcal{C}, \eta[A{\mapsto}\mathsf{tt}])$
  ELSE IF $\mathcal{C}[\eta]$ contains a unit clause $\{\neg A\}$
    RETURN *UNITPROP*$(\mathcal{C}, \eta[A{\mapsto}\mathsf{ff}])$
  ELSE RETURN $(\eta,$ ”undecided”$)$

*DPLL*$(\eta) =$
  $(\eta', v) :=$ *UNITPROP*$(\mathcal{C}, \eta)$
  IF $v =$ “satisfiable”
    RETURN “satisfiable”
  ELSE IF $v =$ “unsatisfiable”
    $\mathcal{C} := \mathcal{C} \cup \{k\}$where $k$ asserts that at least one atom is valued different from $\eta'$.
    RETURN “unsatisfiable”
  ELSE
    choose an atom $A \notin \mathrm{dom}(\eta')$
    IF *DPLL*$(\eta'[A{\mapsto}\mathsf{tt}]) =$”satisfiable”
      RETURN “satisfiable”.
    ELSE
      RETURN *DPLL*$(\eta'[A{\mapsto}\mathsf{ff}])$

INPUT a set of clauses $\mathcal{X}$
$\mathcal{C} := \mathcal{X}$
OUTPUT *DPLL*$(\emptyset)$
```

Figure 16: DPLL-algorithm with clause learning

1. Every student has a matric number.
2. If a student fails to matriculate she will be expelled.
3. Every human being is a philosopher.
4. There always is one student who complains about every course.
5. There is a set with no elements.
6. For every natural number n there exists a natural number d such that $2^d \leq n$ and $n < 2^{d+1}$.
7. The knirp of each bilg is a prugl but does not bebelf any quist.

Figure 17: First-order formulas (informal)

1. $\forall x.\text{student}(x) \Rightarrow \exists n.\text{number}(x) \wedge \text{has\_matric\_no}(x, n)$
2. $\forall x.\text{student}(x) \Rightarrow \neg\text{has\_matriculated}(x) \Rightarrow \text{will\_be\_expelled}(x)$
3. $\forall x.\text{human}(x) \Rightarrow \text{philosopher}(x)$
4. $\exists x.\text{student}(x) \wedge \forall c.\text{course}(c) \Rightarrow \text{complains\_about}(x, c)$
5. $\exists x.\forall y.\neg(y \in x)$
6. $\forall n.\text{number}(n) \Rightarrow \exists d.\text{number}(d) \wedge \text{leq}(\text{power}(2, d), n) \wedge \text{lt}(n, \text{power}(2, \text{plus}(d, 1)))$
7. $\forall x.\text{bilg}(x) \Rightarrow \text{prugl}(\text{knirp}(x)) \wedge \forall y.\text{quist}(y) \Rightarrow \neg\text{bebelfs}(\text{knirp}(x), y)$

Figure 18: Formalisation in untyped first-order logic

1. $\forall x$:student.$\exists n$:number.has_matric_no(x, n)

2. $\forall x$:student $\Rightarrow \neg$has_matriculated$(x) \Rightarrow$ will_be_expelled(x)
3. $\forall x$:human.philosopher(x)
4. $\exists x$:student.$\forall c$:course.complains_about(x, c)
5. $\exists x$:set.$\forall y$:set.$\neg(y \in x)$
6. $\forall n$:number.$\exists d$:number.leq(power$(2, d), n)\wedge$
   lt(power$(2,$ plus$(d, 1)))$
7. $\forall x$:bilg.prugl(knirp$(x)) \wedge \forall y$:quist.$\neg$bebelfs(knirp$(x), y)$

Figure 19: Typed first-order logic

Typed first-order logic. Alternatively, we fix a collection of *types* and require that each quantifier is annotated with a type determining its range. Assuming that we have fixed types

human, student, number, course, set, bilg, quist

we could write the example sentences as in Figure 5. Untyped first-order logic has the advantage of being slightly simpler to formulate and it suffices for many applications, especially in mathematics. Typed first-order logic has the advantage of being more readable and, more importantly, that it allows the user to distinguish between actual properties he wants to prove and typing judgments which should follow automatically in most cases. For example, if we were to prove formula 6 above in the untyped setting then we would at some point come up with a d and would then have to prove number(d) as well as the actually interesting property of d.

In the typed setting the first one falls under typing and can often be discharged automatically.

Of course, the distinction between types and predicates is a subjective one and can be “misused”.

Typechecking can be difficult. For example, if D is the “type” consisting of quadruples of integers (x, y, z, n) such that $n > 2$ and $x^n + y^n = z^n$ then proving

$$\exists p{:}D.p = p$$

is tantamount to proving Wiles’ theorem!

If used reasonably then types can considerably simplify (formal) proofs and the appearance of statements.

5.1 Typed first-order language

A typed first-order language is specified by the following data:

1. a collection $\mathcal{T}$ of *types*
2. a collection $\mathcal{P}$ of *predicate constants*, each endowed with an *arity* $[\tau_1, \ldots, \tau_n \rightarrow \text{boolean}]$ where $\tau_1, \ldots, \tau_n \in \mathcal{T}$
3. a collection $\mathcal{F}$ of function constants each endowed with an arity $[\tau_1, \ldots, \tau_n \rightarrow \tau_{n+1}]$ where $\tau_1, \ldots, \tau_{n+1} \in \mathcal{T}$

The arity of a predicate constant is nothing but a list of types; the brackets, the arrow, and “boolean” are merely notation. We use the notation $P : [\tau_1, \ldots, \tau_n \rightarrow \text{boolean}]$ and $f : [\tau_1, \ldots, \tau_n \rightarrow \tau_{n+1}]$ to indicate the arities of predicate, resp. function constants.

Examples: For the formulas 1,. . . ,4 an appropriate language is as follows:

$$\begin{array}{rcl}
\mathcal{T} & = & \{\text{student}, \text{number}, \text{course}\} \\
\mathcal{P} & = & \{\text{has\_matric\_no} : [\text{student}, \text{number} \rightarrow \text{boolean}] \\
 & & \text{has\_matriculated} : [\text{student} \rightarrow \text{boolean}] \\
 & & \text{will\_be\_expelled} : [\text{student} \rightarrow \text{boolean}]]] \\
 & & \text{complains\_about} : [\text{student}, \text{course} \rightarrow \text{boolean}] \\
\mathcal{F} & = & \emptyset
\end{array}$$

For the formula 7 an appropriate first-order language would be

$$\begin{array}{rcl}
\mathcal{T} & = & \{\text{bilg}, \text{quist}, \text{knirp\_t}\} \\
\mathcal{P} & = & \{\text{prugl} : [\text{knirp\_t} \rightarrow \text{boolean}], \text{bebelfs} : [\text{knirp\_t}, \text{quist} \rightarrow \text{boolean}]\} \\
\mathcal{F} & = & \{\text{knirp} : [\text{bilg} \rightarrow \text{knirp\_t}]\}
\end{array}$$

Typing contexts: $K = x_1{:}\tau_1, \dots, x_n{:}\tau_n$; formally: finite function from variables to types.

Typing rules:

$$\frac{K(x) = \tau}{K \rhd_{\mathcal{L}} x : \tau} \tag{VAR}$$

$$\frac{f : [\tau_1, \dots, \tau_n \to \tau_{n+1}] \in \mathcal{F} \qquad K \rhd_{\mathcal{L}} t_1 : \tau_1, \dots, K \rhd_{\mathcal{L}} t_n : \tau_n}{K \rhd_{\mathcal{L}} f(t_1, \dots, t_n) : \tau_{n+1}} \tag{FUN}$$

Figure 20: Well formed terms

The type knirp_t arises only as the range of a function symbol not as the range of a quantifier. When translating this formula from natural language to formal language we could also have opted for a predicate constant knirp : $[\text{bilg}, \text{knirp\_t} \to \text{boolean}]$ denoting its graph. The content of the definite article "*the* knirp of…" could be rendered by another formula stating unique existence. This would require the notion of equality which we will come to later.

5.2 Syntax

In order to define properly what formulas are we have to talk about terms and formulas possibly involving variables as those occur in scopes of quantifiers. We thus assume an infinite set $\mathcal{V}$ of variables distinct from the other symbols and fix a first order language $\mathcal{L}$. A *typing context* is a finite partial function K mapping variables to types. If K is a typing context and $x \notin \mathrm{dom}(K)$ and $\tau \in \mathcal{T}$ then $K, x{:}\tau$ is the typing context K extended with $x \mapsto \tau$.

We write $K \rhd_{\mathcal{L}} t : \tau$ to mean that t is a well formed term in $\mathcal{L}$ of type τ possibly involving the variables declared and typed as given by K. This judgment is inductively defined by the following *typing rules*.

Well-formed formulas are built up from atomic formulas (predicate constants applied to appropriately typed terms) by propositional connectives and quantifiers which *bind* variables. Formally, we introduce the judgment $K \rhd_{\mathcal{L}} \phi : \text{boolean}$ to mean that ϕ is a well formed formula in $\mathcal{L}$ possibly involving the free variables declared and typed in K. This judgment is defined by the following rules:

$$\frac{P : [\tau_1, \dots, \tau_n \to \text{boolean}] \in \mathcal{P} \qquad K \rhd_{\mathcal{L}} t_1 : \tau_1, \dots, K \rhd_{\mathcal{L}} t_n : \tau_n}{K \rhd_{\mathcal{L}} P(t_1, \dots, t_n) : \text{boolean}} \tag{ATOM}$$

$$\frac{K \rhd_{\mathcal{L}} \phi : \text{boolean}}{K \rhd_{\mathcal{L}} \neg\phi : \text{boolean}} \qquad (\textsc{Neg})$$

$$\frac{K \rhd_{\mathcal{L}} \phi : \text{boolean} \qquad K \rhd_{\mathcal{L}} \psi : \text{boolean} \qquad \star \in \{\vee, \wedge, \Rightarrow\}}{K \rhd_{\mathcal{L}} \phi \star \psi : \text{boolean}} \qquad (\textsc{Conn})$$

$$\frac{K, x{:}\tau \rhd_{\mathcal{L}} \phi : \text{boolean} \qquad Q \in \{\forall, \exists\}}{K \rhd_{\mathcal{L}} Qx{:}\tau.\phi : \text{boolean}} \qquad (\textsc{Quant})$$

These rules define *abstract syntax* together with *typing* (there latter is also known and misnamed as “semantic analysis”). For concrete syntax one needs to specify precedence rules and use parentheses to disambiguate otherwise. The connectives take precedence as before; quantifiers always extend as far to the right as possible, i.e., until an unmatched closing parenthesis is encountered.

5.3 Semantics

A formula ϕ is *closed* if it contains no free variables, i.e., if $\emptyset \rhd_{\mathcal{L}} \phi$: boolean. These are the ones we are really interested in; the open formulas are introduced only as an auxiliary device for the definition of the closed ones.

The meaning of a closed first-order formula is given as a truth value relative to an interpretation of the types, the predicate constants, and the function constants. To specify the meaning of a non closed formula we also need a valuation of the variables.

Interpretation An interpretation $\mathcal{I}$ of a first-order language $(\mathcal{T}, \mathcal{P}, \mathcal{F})$ is given by

1. a set $[\![\tau]\!]_{\mathcal{I}}$ for each $\tau \in \mathcal{T}$.

2. a function $[\![P]\!]_{\mathcal{I}} : [\![\tau_1]\!]_{\mathcal{I}} \times \cdots \times [\![\tau_n]\!]_{\mathcal{I}} \rightarrow \{\mathsf{tt}, \mathsf{ff}\}$ for each $P : [\tau_1, \ldots, \tau_n \rightarrow$ boolean$]$.

3. a function $[\![f]\!]_{\mathcal{I}} : [\![\tau_1]\!]_{\mathcal{I}} \times \cdots \times [\![\tau_n]\!]_{\mathcal{I}} \rightarrow [\![\tau_{n+1}]\!]_{\mathcal{I}}$ for each function constant $f : [\tau_1, \ldots, \tau_n \rightarrow \tau_{n+1}]$.

Such an interpretation associates a truth value with every closed formula and more generally, a function mapping valuations of variables to truth values with every open formula. For pedants we give here a formal definition.

$$
\begin{array}{l}
[\![x]\!]_{\rho,\mathcal{I}} = \rho(x) \\
[\![f(t_1,\ldots,t_n)]\!]_{\rho,\mathcal{I}} = [\![f]\!]_{\mathcal{I}}([\![t_1]\!]_{\rho,\mathcal{I}},\ldots,[\![t_n]\!]_{\rho,\mathcal{I}}) \\
[\![P(t_1,\ldots,t_n)]\!]_{\rho,\mathcal{I}} = [\![P]\!]_{\mathcal{I}}([\![t_1]\!]_{\rho,\mathcal{I}},\ldots,[\![t_n]\!]_{\rho,\mathcal{I}}) \\
[\![\neg\phi]\!]_{\rho,\mathcal{I}} = \neg[\![\phi]\!]_{\rho,\mathcal{I}} \\
[\![\phi \star \psi]\!]_{\rho,\mathcal{I}} = [\![\phi]\!]_{\rho,\mathcal{I}} \star [\![\phi]\!]_{\rho,\mathcal{I}} \\
[\![\forall x{:}\tau.\phi]\!]_{\rho,\mathcal{I}} = \begin{cases} \mathtt{tt}\text{, if } [\![\phi]\!]_{\rho[x\mapsto v],\mathcal{I}} = \mathtt{tt} \text{ for all } v \in [\![\tau]\!]_{\mathcal{I}} \\ \mathtt{ff}\text{, otherwise} \end{cases} \\
[\![\exists x{:}\tau.\phi]\!]_{\rho,\mathcal{I}} = \begin{cases} \mathtt{tt}\text{, if } [\![\phi]\!]_{\rho[x\mapsto v],\mathcal{I}} = \mathtt{tt} \text{ for some } v \in [\![\tau]\!]_{\mathcal{I}} \\ \mathtt{ff}\text{, otherwise} \end{cases}
\end{array}
$$

Here ρ is a partial function on variables.

We note that if ρ is compatible with typing context K in the sense that $\rho(x) \in [\![K(x)]\!]_{\mathcal{I}}$ for all $x \in \mathrm{dom}(K)$, in particular, $\rho(x)$ is defined in this case, then $[\![t]\!]_{\rho,\mathcal{I}} \in [\![\tau]\!]_{\mathcal{I}}$ whenever $K \triangleright_{\mathcal{L}} t : \tau$ and $[\![\phi]\!]_{\rho,\mathcal{I}} \in \{\mathtt{tt}, \mathtt{ff}\}$ whenever $K \triangleright_{\mathcal{L}} t$: boolean.

A closed formula is *valid* if its meaning comes out as true under all possible interpretations of the language it is based on. Examples of such valid formulas are as follows.

- $\forall x{:}\tau.P(x) \Rightarrow \exists y{:}\tau.P(y)$ (recall that quantifiers always extend to the left as far as possible),
- $(\forall x{:}\tau_1.R(x, f(x))) \Rightarrow \forall x{:}\tau_1.\exists y{:}\tau_2.R(x, y)$ when $f : [\tau_1 \to \tau_2]$,
- $(\forall x{:}\tau_1.\forall y{:}\tau_2.P(x) \vee Q(y)) \Rightarrow (\forall x{:}\tau_1.P(x)) \vee (\forall x{:}\tau_2.Q(x))$,
- $(Q \vee \exists x{:}\tau.P(x)) \Rightarrow \exists x{:}\tau.Q \vee P(x)$ where $Q : [\to$ boolean$]$ is a constant and, moreover, we have a constant $c : [\to \tau]$,
- $\exists x{:}\tau.P(x) \Rightarrow \forall y{:}\tau.P(y)$ again in the presence of a constant $c : [\to \tau]$.

Nullary predicate and function constants are propositional, resp., “ordinary” constants. We may write Q : boolean and $c{:}\tau$ instead of $Q : [\to$ boolean$]$ and $c : [\to \tau]$ to declare them and omit empty parentheses (as done above) when using them.

5.4 First-order sequent calculus

As before we form sequents $\Gamma \Longrightarrow_{\mathcal{L}} \Delta$ from lists of *closed* formulas Γ, Δ over some language $\mathcal{L}$. The meaning of such a sequent is that the conjunction of the formulas in Γ implies the disjunction of the formulas in Δ.

Notice that the presence of constants in $\mathcal{L}$ can affect the meaning of a formula hence of a sequent even if these constants do not occur explicitly. This explains the explicit mentioning of $\mathcal{L}$.

We introduce the notation $\mathcal{L}, c{:}\tau$ for the extension of $\mathcal{L}$ with a new constant c of type τ. We keep all the rules for the propositional connectives and add four rules to deal with the quantifiers which we will now explain.

To prove an existential statement we have the rule

$$\frac{\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \phi[t/x]}{\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \exists x{:}\tau.\phi} \tag{$\exists$-R}$$

where $\emptyset \rhd_{\mathcal{L}} t : \tau$.

Here $\phi[t/x]$ denotes the substitution of *closed* term t for variable x in ϕ.

The rule says that to prove an existential statement we must come up with a witness. It corresponds to phrases like "*The desired value x is therefore given by $t\ldots$*". Next, we have the following rule to use a universally quantified statement.

$$\frac{\Gamma, \phi[t/x] \Longrightarrow_{\mathcal{L}} \Delta}{\Gamma, \forall x{:}\tau.\phi \Longrightarrow_{\mathcal{L}} \Delta} \tag{$\forall$-L}$$

where $\emptyset \rhd_{\mathcal{L}} t : \tau$.

To use a universally quantified statement we must instantiate it with some concrete term. The rule corresponds to phrases like "*We apply Lemma xxx / the above assumption to $x = t\ldots$* " or "*Applying Lemma yyy / the above assumption in this situation yields...*

One should note that these two rules preserve but do not always reflect validity, i.e., it may be that the conclusion of a rule is valid, yet the premise is not. After all, one might have chosen the wrong instantiation. Moreover, it is possible that a universal assumption must be instantiated more than once (consider e.g. an assumption asserting that some relation is transitive), so sometimes one has to keep the quantified formula for later use by prior invocation of rule CONTR.

Next, we have a rule for proving a universal statement:

$$\frac{\Gamma \Longrightarrow_{\mathcal{L},c:\tau} \Delta, \phi[c/x]}{\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \forall x{:}\tau.\phi} \tag{$\forall$-R}$$

Here $c : \tau$ is a fresh constant symbol not occurring in $\mathcal{L}$ hence in Γ, Δ, ϕ.

To prove $\forall x{:}\tau.\phi$ we must prove ϕ for a fixed but arbitrary $c : \tau$.

"Fix an arbitrary $c : \tau\ldots$ this proves $\forall x{:}\tau.\phi$

Finally, we need a rule to use an existential statement:

$$\frac{\Gamma, \phi[c/x] \Longrightarrow_{\mathcal{L},c:\tau} \Delta}{\Gamma, \exists x{:}\tau.\phi \Longrightarrow_{\mathcal{L}} \Delta} \tag{$\exists$-L}$$

To use an existential statement we introduce a fresh name for its witness. We know nothing about the witness except that it satisfies ϕ.

"Lemma xxx provides us with a c such that $\phi[c/x]$"

"Let c be the x provided by (13) above"

We notice that in the latter two rules no formulas with free variables arise as the bound variable is immediately replaced with a fresh constant. There are alternative presentations in which free variables are used for the "fixed but arbitrary constants" occurring in those rules. In a typed setting admitting empty domains of quantification this seems less appropriate as we then would have to annotate each sequent with the set of variables it depends on. Moreover, a variable is supposed to vary, whereas these constants are fixed.

Let's take a look at a couple of representative examples.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P(c) \Longrightarrow_{\mathcal{L},c:\tau} P(c)}\;\textsc{Axiom}}{P(c) \Longrightarrow_{\mathcal{L},c:\tau} \exists x{:}\tau.P(x)}\;\exists\text{-R}}{\Longrightarrow_{\mathcal{L},c:\tau} P(c) \Rightarrow \exists x{:}\tau.P(x)}\;\Rightarrow\text{-R}}{\Longrightarrow_{\mathcal{L}} \forall x{:}\tau.P(x) \Rightarrow \exists x{:}\tau.P(x)}\;\forall\text{-R}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(c_1) \vee Q(c_2) \Longrightarrow_{\mathcal{L},c_1:\tau_1,c_2:\tau_2} P(c_1), Q(c_2)}\;\textsc{Prop}}{\forall y{:}\tau_2.P(c_1) \vee Q(y) \Longrightarrow_{\mathcal{L},c_1:\tau_1,c_2:\tau_2} P(c_1), Q(c_2)}\;\forall\text{-L}}{\forall x{:}\tau_1.\forall y{:}\tau_2.P(x) \vee Q(y) \Longrightarrow_{\mathcal{L},c_1:\tau_1,c_2:\tau_2} P(c_1), Q(c_2)}\;\forall\text{-L}}{\forall x{:}\tau_1.\forall y{:}\tau_2.P(x) \vee Q(y) \Longrightarrow_{\mathcal{L}} \forall x{:}\tau_1.P(x), \forall x{:}\tau_2.Q(x)}\;\forall\text{-R}}{\forall x{:}\tau_1.\forall y{:}\tau_2.P(x) \vee Q(y) \Longrightarrow_{\mathcal{L}} (\forall x{:}\tau_1.P(x)) \vee (\forall x{:}\tau_2.Q(x))}\;\vee\text{-R}$$

5.5 Soundness and completeness of first-order sequent calculus

As before we have that a sequent is valid if and only if it is derivable in the sequent calculus, i.e., if there is a proof tree whose leaves are labelled with axioms. Unlike in the propositional case, the contraction rule CONTR is not redundant corresponding to the fact that universal premises may need to be used more than once. This thwarts a naive decision procedure for validity based on constructing a generic proof tree and, indeed, as was shown by TURING validity in first order logic is undecidable. Actually, this result is not very surprising if we consider that basically all of mathematics can be formalised in first-order logic.

Theorem: A sequent is valid if and only if it is derivable in the sequent calculus.

Proof: The "if" direction of the correctness theorem ("soundness") is proved as before by induction on derivations; we simply have to check that all the rules *preserve* validity.

For the "only if" direction ("completeness") we construct a generic proof tree as in the propositional case by breaking down connectives and if nothing else helps instantiating quantifiers $\forall$-L, $\exists$-R. We must make sure that we keep those quantified statements around using CONTR prior to instantiating. If we arrange things in such a way that eventually a quantified formula will be instantiated with every possible term we are sure to find a proof if one exists.

If no proof exists our generic proof tree contains a leaf that is not an axiom or has an infinite path.

From the infinite path we will construct a counter interpretation by taking terms (also containing the constants newly introduced along the path) to interpret the types, function constants interpreting themselves, and interpreting predicate constants according to how atomic formulas involving them occur along the path. This will ensure that the interpretation falsifies all the sequents along the path hence the root sequent which by assumption has no proof.

Let us look at this in some more detail. Firstly, to counter the information loss in the instantiating rule $\forall$-L and $\exists$-R we replace them by the following combinations with rule CONTR:

$$\frac{\Gamma, \forall x{:}\tau.\phi, \phi[t/x] \Longrightarrow_{\mathcal{L}} \Delta}{\Gamma, \forall x{:}\tau.\phi \Longrightarrow_{\mathcal{L}} \Delta} \qquad (\forall\text{-L'})$$

$$\frac{\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \exists x{:}\tau.\phi, \phi[t/x]}{\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \exists x{:}\tau.\phi} \qquad (\exists\text{-R'})$$

It is clear that there is a proof with the primed rules if and onnly if there is one in the original system.

A *generic proof tree* is a possibly infinite tree labelled with sequents which has the following properties:

1. each internal node is the conclusion of its immediate ancestors by some proof rule.

2. rule WEAK is not used,

3. rules $\forall$-L' and $\exists$-R' are used only with conclusion $\Gamma \Longrightarrow_{\mathcal{L}} \Delta$ where Γ contains atoms and universally quantified formulas only and Δ contains

atoms and existentially quantified formulas only. Otherwise we could use one of the validity-reflecting rules.

4. no internal node is labelled with an axiom, i.e., we stop once we have found an axiom

5. on every infinite path starting from $\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \exists x{:}\phi$ the formula $\exists x{:}\phi$ is instantiated with all (closed) terms of type τ in $\mathcal{L}$

6. Ditto for infinite paths starting from $\Gamma, \forall x{:}\tau \Longrightarrow_{\mathcal{L}} \Delta$

These properties basically dictate a strategy for obtaining a generic proof tree starting from any sequent. Simply apply the rules backwards with the mentioned restriction on the rules that instantiate quantifiers. When selecting instantiations make sure that every possible instantiation will be eventually chosen unless of course a path ends with an axiom leaf. Please note, that as soon as we make a language extension we must instantiate our quantified formulas with all the terms in the new language as well.

Now suppose that a sequent S has no proof. The generic proof tree constructed from S might have a finite path ending in a non axiom consisting of atomic formulas only. In this case, we can argue as in the case of propositional logic that the root sequent is unsatisfiable. Alternatively, and this is the interesting case, the generic proof tree will contain an infinite path π (starting from the root). This is “König’s Lemma”: a finitely branching tree with infinitely many nodes has an infinite path. Along this infinite path π we encounter an increasing (by constants) sequence of languages $\mathcal{L}_1 \subseteq \mathcal{L}_2 \subseteq \ldots$ whose union we call $\mathcal{L}_\infty$. So, a term in $\mathcal{L}_\infty$ will be a term of one of the $\mathcal{L}_i$.

To construct our desired counterinterpretation $\mathcal{I}$ we interpret types by

$$[\![\tau]\!]_{\mathcal{I}} = \{t \mid \emptyset \triangleright_{\mathcal{L}_\infty} t : \tau\}$$

We interpret function constants by

$$[\![f]\!]_{\mathcal{I}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$$

We interpret predicate constants by

$$[\![P]\!]_{\mathcal{I}}(t_1, \ldots, t_n) = \begin{cases} \mathsf{tt}, & \text{if } P(t_1, \ldots, t_n) \text{ is among the antecedents} \\ & \text{(left of } \Longrightarrow\text{) of a sequent in } \pi, \\ \mathsf{ff}, & \text{in all other cases.} \end{cases}$$

Now we show by induction on the size of formulas that whenever a formula ϕ appears as an antecedent of a sequent in π then $\llbracket\phi\rrbracket_{\mathcal{I}} = \mathtt{tt}$ and whenever a formula ψ appears as a succedent (to the right of the $\Longrightarrow$) of a sequent in π then $\llbracket\psi\rrbracket_{\mathcal{I}} = \mathsf{ff}$, so that in particular all the sequents along π including the root will be falsified by $\mathcal{I}$. So, $\mathcal{I}$ shows that the root is not a valid sequent.

Atomic formulas are true under $\mathcal{I}$ precisely if they appear as an antecedent. If an atomic formula appears as a succedent then—since atomic formulas never disappear along the path—it cannot also appear as an antecedent for otherwise we would have an axiom sequent on π contrary to the construction of the generic proof tree. Thus, atomic formulas appearing as succedents are falsified by $\mathcal{I}$. A formula which is not an existentially quantified succedent or a universally quantified antecedent will eventually be broken down by a validity reflecting rule into its subformulas to which the induction hypothesis applies. Consider for example a succedent of the form $\forall x{:}\tau.\phi$. At some point rule $\forall$-L will be applied, so $\phi[c/x]$ also occurs as a succedent on π. By the induction hypothesis $\llbracket\phi[c/x]\rrbracket_{\mathcal{I}} = \mathsf{ff}$, but then $\llbracket\forall x{:}\tau.\phi\rrbracket_{\mathcal{I}} = \mathsf{ff}$, too.

If, finally, we have an existentially quantified succedent, e.g., $\exists x{:}\phi$ then by the "round robin" policy used for instantiating all formulas, all the formulas $\phi[t/x]$ with $\emptyset \rhd_{\mathcal{L}_\infty} t : \tau$ will occur as succedents along π hence are falsified by $\mathcal{I}$. Since $\llbracket\tau\rrbracket_{\mathcal{I}}$ comprises precisely all those terms we conclude that $\llbracket\exists x{:}\tau.\phi\rrbracket = \mathsf{ff}$, as well. The case of a universally quantified antecedent is analogous. This completes the proof.

One should not underestimate the power of first-order logic. Even without function constants the counter interpretation may be infinite due to infinitely many newly introduced constants. Consider for example the formula

$$(\forall x, y, z{:}\tau.R(x, y)\wedge R(y, z) \Rightarrow R(x, z)) \wedge (\forall x{:}\tau.\exists y{:}\tau.R(x, y)) \Rightarrow \exists x{:}\tau.R(x, x)$$

It is not valid but holds in all finite interpretations. You may find it instructive to form the generic proof tree for this formula.

5.6 First-order logic in PVS

Language concepts are declared anywhere in a theory, but before being used

```
D, T1, T2 : TYPE+
c : T1
P : [D -> boolean]
```

```
 Q : boolean
 f : [T1->T2]
```

Here `TYPE+` stands for *nonempty* type. There is also the declaration `T:TYPE` which stands for a possibly empty type. In this case it would not be allowed to declare a constant of type `T`.

This design decision of PVS is open to debate. By declaring a constant of a type we explicitly state that it is nonempty so why say it twice?

Quantifiers are written `FORALL(x:T):` and `EXISTS(x:T):` Do not forget the colon after the parenthesis.

```
exI : THEOREM
    FORALL(x:D):P(x) IMPLIES EXISTS(x:D): P(x)
orex : THEOREM
     (Q OR EXISTS(x:D):P(x)) IMPLIES EXISTS(x:D):Q OR P(x)
depp : THEOREM
      EXISTS(x:D): FORALL(y:D): P(x) IMPLIES P(y)
gen : THEOREM
      (EXISTS(x:D):P(x)) IMPLIES FORALL (x:D):P(x)
```

The rules ∀-L and ∃-R are invoked with the command `inst` (instantiation). The rules ∀-L and ∃-R are invoked with the command `skolem` (after TH. SKOLEM).

The `inst` command takes as argument a formula number (the formula to be instantiated) and a term to instantiate with. For example, in the situation

```
{-1} P(d)
  |-------
{1}  EXISTS (x:T1): P(x)
```

the command

```
(inst 1 "d")
```

leads to

```
{-1} P(d)
  |-------
{1}  P(d)
```

which is an axiom.

The `skolem` command takes as argument a formula number and the name of a new constant. If it isn't fresh then PVS complains. For example, in the situation

```
  |-------
{1}    FORALL(x:D):P(x) IMPLIES EXISTS(x:D): P(x)
```

The command `(skolem 1 "c")` is no good because we have already used `c` for a constant above. However, `(skolem 1 "d")` succeeds and gives

```
  |-------
{1}    P(d) IMPLIES EXISTS(x:D): P(x)
```

The command `M-x show-skolem-constants` displays all the constants introduced in the course of the proof.

The commands `inst` and `skolem` allow the treatment of several variables at once. There are also the derived form `inst?` which guesses an appropriate instantiation heuristically (alas often quite badly) and `skolem!` which automatically introduces as many constants as possible (making up fresh names for them). Furthermore, `skosimp` is a combination of `skolem!` and simplification. See the PVS prover guide for details.

As an exercise try to prove all of the “theorems” below.

```
fol: THEORY
 BEGIN
 D, T1, T2 : TYPE+
 c : T1
 d : D
 P : [D -> boolean]
 Q : boolean

  allE : THEOREM
  FORALL(x:D): (FORALL(y:D): P(y)) IMPLIES P(x)



 andall : THEOREM
   (Q AND (FORALL(x:D):P(x))) IMPLIES FORALL(x:D):Q AND P(x)

 exI : THEOREM
    FORALL(x:D):P(x) IMPLIES EXISTS(x:D): P(x)

 andex : THEOREM
```

```
  (Q AND EXISTS(x:D):P(x)) IMPLIES EXISTS(x:D):Q AND P(x)

orex : THEOREM
    (Q OR EXISTS(x:D):P(x)) IMPLIES EXISTS(x:D):Q OR P(x)

depp : THEOREM
     EXISTS(x:D): FORALL(y:D): P(x) IMPLIES P(y)

doub : THEOREM
     FORALL(x,y:D) : EXISTS (z:D) : P(z) IMPLIES P(x) & P(y)

P1 : [T1->boolean]
P2 : [T2 -> boolean]

por : THEOREM
    (FORALL(x:T1,y:T2):P1(x) OR P2(y))  IMPLIES
                  (FORALL(x:T1):P1(x)) OR (FORALL(x:T2):P2(x))

R : [D,D->boolean]

per: THEOREM
      (FORALL (x,y:D):R(x, y) IMPLIES R(y, x)) AND
      (FORALL (x,y,z:D): R(x, y) AND R(y, z) IMPLIES R(x,z)) IMPLIE
      (FORALL (x:D): (EXISTS (y:D): R(x,y)) IMPLIES R(x,x))
END fol
```

6 First-order resolution

As in the propositional case the method of resolution provides a generally more efficient way to decide validity of formulas than proof search in Gentzen's sequent calculus. In the first-order case we may instantiate universally quantified variables prior to resolving so as to achieve agreement of literals. For example, we may resolve the clauses $\{P(f(x, g(y))), Q(x)\}$ which denotes $\forall x, y.P(f(x, g(y))) \vee Q(x)$ (types omitted) and $\{\neg P(f(g(z), w)), R(w)\}$ which denotes $\forall w, z.\neg P(f(g(z), w)) \vee R(w)$ to form $\{R(g(y)), Q(g(z))\}$ which denotes $\forall y, z.R(g(y)) \vee Q(g(z))$.

Notice that again a satisfying interpretation for the former two clauses will also satisfy the latter. In this example, we could also have instantiated x with

$$\begin{array}{lll} C_1 = \{P(f(x,g(y))), Q(x)\} & \text{, i.e.,} & \forall x,y.P(f(x,g(y))) \vee Q(x) \\ C_2 = \{\neg P(f(g(z),w)) \vee R(w)\} & \text{, i.e.,} & \forall w,z.\neg P(f(g(z),w)) \vee R(w)\} \\ & \text{resolve to} & \\ C_3 = \{R(g(y)) \vee Q(g(z))\} & \text{, i.e.,} & \forall y,z.R(g(y)) \vee Q(g(z)) \end{array}$$

Figure 21: Example of resolution

something like $g(f(h(c())))$ and accordingly y with $f(h(c()))$. However, in order to maximise future success it is advisable to choose the instantiation which makes the least possible commitment or, in formal terms, the *most general unifier*. While this is in practice always done, it is, for the purpose of establishing completeness, easier to allow arbitrary instantiations.

Let us look at the details. First-order resolution operates on clauses which are sets of first-order literals, i.e., negated or non-negated atomic formulas, which are understood as being universally quantified over the variables they contain. In order to avoid problems with empty types we assume that our language is such that every type contains at least one closed term.

- A first-order literal is a negated or non-negated atomic formula.
- A first-order clause is a set of first-order literals
- It denotes the disjunction of the literals universally quantified over the variables

Given a set of first-order clauses $\mathcal{C}$ we can use first-order resolution to decide whether it is satisfiable, i.e., whether there exists an interpretation which makes it true. If we can derive the empty clause from $\mathcal{C}$ by successive application of rules INST and RES then surely $\mathcal{C}$ is unsatisfiable. Conversely, if $\mathcal{C}$ is unsatisfiable then it is possible to derive the empty clause. The proof of this result is based on correctness of propositional resolution and Herbrand's theorem which asserts that a set of formulas of the form $\forall\vec{x}{:}\vec{\tau}.\phi$ with ϕ quantifier-free is satisfiable if and only if the set of its closed instantiations is propositionally satisfiable:

Theorem ("Herbrand's theorem"): Let $\mathcal{S}$ be a set of formulas of the form $\forall\vec{x}.\phi$ with ϕ quantifier-free.

Define

$$\overline{\mathcal{S}} := \{\phi[t_1/x_1,\dots,t_n/x_n)] \mid \forall x_1{:}\tau_1,\dots\forall x_n{:}\tau_n.\phi \in \mathcal{S} \text{ and } \emptyset \rhd_{\mathcal{L}} t_i{:}\tau_i\}$$

- Instantiation rule:

$$\frac{C}{C[t_1/x_1,\dots,t_n/x_n]} \qquad \text{(I}\textsc{nst}\text{)}$$

  where x_i are the variables mentioned in C and the t_i are *possibly open* terms (of the right type!).

- Resolution rule

$$\frac{C_1 \cup \{A\} \qquad C_2 \cup \{\neg A\}}{C_1 \cup C_2} \qquad \text{(R}\textsc{es}\text{)}$$

- Side condition: there is a closed term of each type.

- Aim: try to derive empty clause from initial set so as to show unsatisfiability.

Figure 22: First-order resolution

as the set of closed instantiations of formulas in $\mathcal{S}$.

There exists an interpretation $\mathcal{I}$ validating all formulas in $\mathcal{S}$ if and only if there exists a propositional valuation η of the atomic formulas (viewed as propositional atoms) validating all formulas in $\overline{\mathcal{S}}$.

Proof: Given $\mathcal{I}$ define η by $\eta(P(t_1,\dots,t_n)) = [\![P(t_1,\dots,t_n)]\!]_{\mathcal{I}}$. Given η interpret types as sets of (closed) terms, function symbols by themselves, and predicates as given by η. □

Thus to establish unsatisfiability of a set of first-order clauses it is enough to establish propositional unsatisfiability of their closed instantiations, but that's precisely what rule RES can do as shown in Section 4.

Rule INST on the other hand, allows us to generate the set of closed instantiations. Performing resolution on clauses containing (universally quantified) variables certainly does no harm, but may of course speed up success.

6.1 Most general unifiers

As already mentioned, in practice one resolves clauses by instantiating with the *most general unifier*. The most general unifier of two open terms $u(x_1,\dots,x_m)$ and $v(y_1,\dots,y_n)$ consists of two sequences of open terms $t_1,\dots,t_m$ and $s_1,\dots,s_n$ involving variables $z_1,\dots,z_k$, such that $u(t_1,\dots,t_m) = v(s_1,\dots,s_n)$ and, moreover, any instantiation making u equal to v arises from this one by instantiation,

- $u(x_1, \ldots, x_m), v(y_1, \ldots, y_n)$ two term with free variables $\vec{x}$ and $\vec{y}$.
- Most general unifier consists of $\vec{s}(\vec{z})$ and $\vec{t}(\vec{z})$ such that $u(\vec{s}(\vec{z})) = v(\vec{t}(\vec{(z)}))$ and
- whenever $u(\vec{s'}) = v(\vec{t'})$ then $s' = \vec{s}(\vec{a}), t' = \vec{t}(\vec{b})$.
- **Example:** $u = f(x, g(y)), v = f(h(y), x)$:
  $\vec{s} = [h(z)/x, x/y], \vec{t} = [z/y, g(x)/x]$
- **Note:** the most general unifier might not exist, e.g., $s = f(x), t = g(y)$.

Figure 23: Most general unifier

$$\begin{array}{ll}
\{\ \{\neg S(r), \neg A(r,x,y), A(r,y,x)\}, & (\mathbf{C1}) \\
\quad \{A(r, f(r), g(r)), S(r)\}, & (\mathbf{C2}) \\
\quad \{\neg A(r, g(r), f(r)), S(r)\}, & (\mathbf{C3}) \\
\quad \{S(s())\}, & (\mathbf{C4}) \\
\quad \{A(s(), a(), b())\}, & (\mathbf{C5}) \\
\quad \{\neg A(s(), b(), a())\}\ \} & (\mathbf{C6})
\end{array}$$

Figure 24: Example of first-order resolution

i.e., whenever $u(\vec{t'}) = v(\vec{s'})$ then $\vec{t'} = \vec{t}[\vec{a}/\vec{z}]$ and $\vec{s'} = \vec{s}[\vec{b}/\vec{z}]$ for some $\vec{a}, \vec{b}$.

For example, the most general unifier of $f(x, g(y))$ and $f(h(y), x)$ is $\vec{s} = [h(z)/x, x/y], \vec{t} = [z/y, g(x)/x]$ because we have $f(x, g(y))[h(z)/x, x/y] = f(h(z), g(x)) = f(h(y), x)[z/y, g(x)/x]$. Notice that the variable names in u, v as well as the common ones in the s, t are rather arbitrary. In particular, if the same variable happens to occur in both u and v, we can instantiate it differently in both.

The most general unifier is effectively found by comparing the terms in question in a top down fashion starting from the outermost function constant. We omit the details of the unification algorithm and also a formal proof that resolution with most general unifiers is complete. The idea is to map any proof using RES and INST to a proof using only the following combined rule.

$$\frac{C_1 \cup \{A_1\} \qquad C_2 \cup \{\neg A_2\} \qquad \vec{s}, \vec{t}\ \text{m.g.u. of}\ A_1, A_2}{C_1[\vec{s}] \cup C_2[\vec{t}]} \quad (\textsc{Res-Unif})$$

Of course it goes without saying that all the instantiations made in the course of resolution must be type correct, i.e., the resulting terms and atomic formulas must be well-formed.

At this point it is worth reiterating the point made earlier in Section 5 about types separating interesting and potentially difficult facts from uninteresting obvious facts. The number of clauses hence the search space for resolution becomes smaller the more we make use of types.

6.2 Skolemisation

We now discuss how to translate arbitrary first-order formulas into first-order clauses; somewhat surprisingly, any first-order formula is equivalent to set of first-order clauses albeit in a richer language.

In order to do that we use the following fact known as *skolemisation*, again after TH. SKOLEM.

Fact: Let $\psi := \forall x_1{:}\tau_1 \ldots \forall x_n{:}\tau_n . \exists y{:}\tau_{n+1} . \phi(\vec{x}, y)$ in some language $\mathcal{L}$ and let $\mathcal{L}'$ be the language $\mathcal{L}$ extended with a new function constant $f : [\tau_1, \ldots, \tau_n \to \tau_{n+1}]$. The formula ψ is satisfiable if and only if the formula $\forall x_1{:}\tau_1 \ldots \forall x_n{:}\tau_n . \phi(\vec{x}, f(\vec{x}))$ is satisfiable.

The function constant f is called a *Skolem function*.

Now consider an arbitrary first-order formula ϕ. Using the following tautologies

$$\begin{array}{ll}
\alpha \star (Qx{:}\tau . \beta(x)) \Leftrightarrow Qx{:}\tau . \alpha \vee \beta(x) & \star \in \{\vee, \wedge, \Rightarrow\}, Q \in \{\forall, \exists\} \\
(Qx{:}\tau . \alpha(x)) \Rightarrow \beta \Leftrightarrow \bar{Q}x{:}\tau . \alpha(x) \Rightarrow \beta & \\
(\neg Qx{:}\tau . \alpha(x)) \Leftrightarrow \bar{Q}x{:}\tau . \neg\alpha(x) &
\end{array}$$

where $\bar{\exists} = \forall, \bar{\forall} = \exists$ we can bring each formula (up to equivalence) into the form

$$Q_1 x_1{:}\tau_1 . Q_2 x_2{:}\tau_2 . \ldots . Q_n x_n{:}\tau_n . \phi_0$$

with $Q_1, \ldots, Q_n \in \{\forall, \exists\}$ and ϕ_0 quantifier-free. The latter formula is by definition in *prenex form*.

Thereafter, using the "fact" we can successively replace existential quantifiers with new function constants that take all the previous universally quantified variables as arguments so as to obtain a universally quantified boolean combination of atomic formulas which in turn is equivalent to a set of first-order clauses. Summing up, we have the following result.

Theorem: For every first-order formula ϕ one can effectively find a set of first-order clauses $\mathcal{C}$ such that ϕ is satisfiable if and only if $\mathcal{C}$ is.

Proof: Bring ϕ into prenex form (move all quantifiers to the front exchanging $\forall$ and $\exists$ when moving out of a negative position). Replace existential quantifiers by *Skolem functions* using language extension by function constants. Bring the resulting universally quantified formula into clausal form as in propositional case. □

In my view, the reason why resolution is superior to proof search in sequent calculus is that the choice of instantiations is made after looking at two clauses (and performing unification) which, when informally translated back to sequent calculus, means that the form of the side formulas Γ, Δ in a sequent, say, $\Gamma \Longrightarrow_{\mathcal{L}} \Delta, \exists x{:}\tau.\phi$ helps in finding an appropriate instantiation for x. I am not aware of a precisation of this argument in the form of a unification-based strategy for finding instantiations in sequent calculus proof search. In this context one should note that the flattening of nested quantifications using Skolemisation is crucial for the success of unification.

6.3 Some puzzles

Here are some small examples that should be brought into clausal form and proved by hand, using PVS, or automatically using SPASS.

The mislabelled boxes (from `http://www.cs.miami.edu/~tptp/`): There are three boxes a, b, and c on a table. Each box contains apples or bananas or oranges. No two boxes contain the same thing. Each box has a label that says it contains apples or says it contains bananas or says it contains oranges. No box contains what it says on its label. The label on box a says ”apples”. The label on box b says ”oranges”. The label on box c says ”bananas”. You pick up box b and it contains apples. What do the other two boxes contain?

Barber’s problem (from `http://www.cs.miami.edu/~tptp/`): There is a barbers’ club that obeys the following three conditions:

1. If any member has shaved any other member – whether himself or another – then all members have shaved him, though not necessarily at the same time.
2. Four of the members are named Guido, Lorenzo, Petrucio, and Cesare.
3. Guido has shaved Cesare. Prove Petrucio has shaved Lorenzo

Continuity of composition

$\mathcal{T} = \{\rho, \iota\}$
$\mathcal{F} = \{f : [\rho \to \rho]\}$
$\mathcal{P} = \{\in : [\rho, \iota \to \text{boolean}]\}$

$\forall x{:}\rho \forall U{:}\iota. \in (f(x), U) \Rightarrow \exists V{:}\iota. {\in}(x, V) \wedge \forall y{:}\rho. \in (y, V) \Rightarrow \in (f(y), U)$
$\Rightarrow$
$\forall x{:}\rho \forall U{:}\iota. \in (f(f(x)), U) \Rightarrow \exists V{:}\iota. {\in}(x, V) \wedge \forall y{:}\rho. \in (y, V) \Rightarrow \in (f(f(y)), U)$

6.4 Compactness of first-order logic

Theorem: Let Φ be a set of first-order formulas over some signature. If every finite subset of Φ has a model then Φ itself has a model, too.

Proof: Using skolemisation we may assume without loss of generality that Φ consists of formulas of the form
$forall\vec{x}{:}\vec{\tau}.\phi$ with ϕ quantifier-free.

Let us form the propositional theory Π consisting of closed-instantiations of the formulas ϕ as in Herbrand's theorem. If every finite subset of Φ has a model then every finite subset of Π will be satisfiable, since a finite subset of Π can only involve a finite subset of Φ. By compactness of propositional logic therefore the whole of Π is satisifable and by Herbrand's theorem Φ has a model.

The compactness theorem has a number of perhaps surprising consequences. Consider, for example, the set $\Theta_{\mathbb{N}}$ of closed formulas (over the signature $(\mathtt{nat}, +, \times, 0, 1, \geq$ $)$ that are true in the standard interpretation that interprets `nat` as the natural numbers etc. This set of formulas, the *first-order theory* of the natural numbers contains in particular all the instances of the Peano axioms but much more, e.g., those formulas that are true but not provable from the Peano axioms.

Now let us extend the signature by a special constant $c : [\to \mathtt{nat}]$ and the formulas $\phi_n := \exists y{:}\mathtt{nat}.c \geq 1 + \cdots + 1$ (n summands).

Every finite subset of this extended set has a model, namely the natural numbers with c interpreted as a large enough number. By compactness therefore the whole set has a model which is a structure validating the same first-order formulas as the natural numbers themselves, yet contains an infinitely large number—the interpretation of c.

Let us explore the structure of such a *non-standard model* of arithmetic. As we have seen, it contains a number c that is greater than any standard number. Since every number (including c has a successor there are more infinite numbers

$c+1, c+2, c+3$, etc. Since every non-zero number has a predecessor (that is a valid first-order sentence!) there must also be $c-1, c-2, c-3$, etc. So the numbers around c form a structure isomorphic to $\mathbb{Z}$. Since every number can be doubled and halved (in the floor-sense) there must be another such $\mathbb{Z}$ block above the one surrounding c and one below. In between any two distinct $\mathbb{Z}$-blocks there must be another one, etc. So any countable non-standard model has an order type isomorphic to $\mathbb{N} + \mathbb{Z}.\mathbb{Q}$.

In a similar way, we can use compactness to show consistency (with respect to first-order logic!) of infinitesimal numbers. Add to the theory of the real numbers the infinitely many axioms $0 < c < 2^{-n}$. Every finite subset is consistent so the whole set is and it thus has a model in which there is a constant c that is arbitrarily close to zero. Of course, then, say, $\pi + c$ is arbitrarily close to π, etc. We can then define a derivative as something like $f(x+c)/c$. Notice that if we prove some first-order statement from this extended theory then, again by compactness, only finitely many of the assumptions $0 < c < 2^{-n}$ will have been used, so in this case, an ordinary c will do.

7 Equality

Most mathematical statements of interest involve equality. It is in principle possible to treat equality just as a predicate constant and to assume axioms stating that equality is an equivalence relation compatible with all function and predicate constants. For practical purposes it is, however, more convenient to introduce equality as a special primitive concept.

So, in first order logic with equality atomic formulas can be formed in two ways:

- $P(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms of types $\tau_1, \ldots, \tau_n$ and $P : [\tau_1, \ldots, \tau_n \to \text{boolean}]$. That's as before.

- $t_1 = t_2$ where t_1, t_2 are terms of some common type τ.

Such an equality formula $t_1{=}t_2$ is true if and only if under the interpretation at hand the two terms t_1, t_2 have equal meaning.

The sequent calculus can be extended so as to cope with equality by adding the following rules:

$$\frac{\emptyset \rhd_{\mathcal{L}} t : \tau \text{ for some } \tau}{\Gamma \Longrightarrow_{\mathcal{L}} \Delta, t = t} \qquad (\textsc{Refl})$$

$$\dfrac{\phi[t_1/x] \Longrightarrow_{\mathcal{L}} \neg\neg\phi[t_1/x] \qquad \dfrac{\dfrac{\dfrac{\Gamma, t_1 = t_2, \phi[t_2/x] \Longrightarrow_{\mathcal{L}} \Delta}{\Gamma, t_1 = t_2, \Longrightarrow_{\mathcal{L}} \Delta, \neg\phi[t_2/x]}\,\neg\text{-R}}{\Gamma, t_1 = t_2 \Longrightarrow_{\mathcal{L}} \Delta, \neg\phi[t_1/x]}\,\textsc{Subst}\text{-R}}{\Gamma, t_1 = t_2, \neg\neg\phi[t_1/x] \Longrightarrow_{\mathcal{L}} \Delta}\,\neg\text{-L}}{\Gamma, t_1 = t_2, \phi[t_1/x] \Longrightarrow_{\mathcal{L}} \Delta}\,\textsc{Cut}$$

$$\dfrac{\dfrac{}{t_1 = t_2 \Longrightarrow_{\mathcal{L}} t_2 = t_2}\,\textsc{Refl}}{t_1 = t_2 \Longrightarrow_{\mathcal{L}} t_2 = t_1}\,\textsc{Subst}\text{-R}$$

CUTting with the conclusion gives rules SUBST-L-RL, SUBST-R-RL.

$$\dfrac{\dfrac{}{t_2 = t_3, t_1 = t_2 \Longrightarrow_{\mathcal{L}} t_2 = t_3}\,\textsc{Axiom}}{t_2 = t_3, t_1 = t_2 \Longrightarrow_{\mathcal{L}} t_1 = t_3}\,\textsc{Subst}\text{-R}$$

Figure 25: Example derivations

$$\frac{\Gamma, t_1 = t_2 \Longrightarrow_{\mathcal{L}} \Delta, \phi[t_2/x]}{\Gamma, t_1 = t_2 \Longrightarrow_{\mathcal{L}} \Delta, \phi[t_1/x]} \qquad (\textsc{Subst}\text{-R})$$

Rule REFL says that $t = t$ is vacuously true; if this is among our conclusions then we're done.

Rule SUBST-R says that if we have an equality $t_1 = t_2$ among our assumptions and we need to prove a formula $\phi[t_1/x]$ which contains t_1 as a subexpression then we can replace t_1 by t_2 and therefore prove $\phi[t_2/x]$ instead.

We immediately have the following derived rules:

$$\frac{\Gamma, t_1 = t_2, \phi[t_2/x] \Longrightarrow_{\mathcal{L}} \Delta}{\Gamma, t_1 = t_2, \phi[t_1/x] \Longrightarrow_{\mathcal{L}} \Delta} \qquad (\textsc{Subst}\text{-L})$$

$$\frac{\Gamma, t_1 = t_2 \Longrightarrow_{\mathcal{L}} \Delta, \phi[t_1/x]}{\Gamma, t_1 = t_2 \Longrightarrow_{\mathcal{L}} \Delta, \phi[t_2/x]} \qquad (\textsc{Subst}\text{-R-RL})$$

$$\frac{\Gamma, t_1 = t_2, \phi[t_1/x] \Longrightarrow_{\mathcal{L}} \Delta}{\Gamma, t_1 = t_2, \phi[t_2/x] \Longrightarrow_{\mathcal{L}} \Delta} \qquad (\textsc{Subst}\text{-L-RL})$$

Figure 7 contains a derivation of rule SUBST-L.

It is possible to show that sequent calculus with the equality rules derives all valid formulas; to that end one considers the quotient of the term model by the congruence relation generated by the equations appearing as antecedents of sequents on the infinite path in the generic proof tree.

Similarly, one can extend resolution with rules that allow one to replace equals with equals within clauses.

7.1 Equality in PVS

The rule REFL is treated like an axiom: as soon as PVS encounters an instance of reflexivity the corresponding subgoal is discarded ("This completes the proof of ...") or, if it was the last open branch of the proof, the proof is completed ("Q.E.D.").

To invoke either SUBST-R or SUBST-L we use the command

$$(\texttt{replace}\ \langle \mathit{what\_with} \rangle\ \langle \mathit{where} \rangle)$$

Here $\langle \mathit{what\_with} \rangle$ must be the (negative) number of an equation among the antecedents; $\langle \mathit{where} \rangle$ must be the number of any formula either in the antecedents (that's SUBST-L) or in the succedents (that's SUBST-R) which contains the left-hand-side of $\langle \mathit{what\_with} \rangle$.

To invoke either rule SUBST-L-RL or SUBST-R-RL we use the command

$$(\texttt{replace}\ \langle \mathit{what\_with} \rangle\ \langle \mathit{where} \rangle : \texttt{dir}\,\texttt{RL})$$

7.2 Extended example: monoids

We assume a nonempty set M with an associative operation * in infix notation.

```
M : TYPE+
* : [M,M->M]
assoc : AXIOM
   FORALL(x,y,z:M):(x*y)*z = x*(y*z)
```

Of course, all this must be placed in a .pvs file and within something like monoids : THEORY BEGIN...END.

Generalised associativity We want to prove an extended law of associativity:

```
  assoc4 : THEOREM
      FORALL(x,y,z,w:M): ((x*y)*z)*w = x*(y*(z*w))
```

We will first do it the basic way and then using some more advanced commands.

After starting the prover we introduce constants for the universally quantified variables with the command `(skolem!)`:

```
  |-------
{1}   ((x!1 * y!1) * z!1) * w!1 = x!1 * (y!1 * (z!1 * w!1))
```

where `x!1, y!1, z!1, w!1` are fresh constants of type `M`.

Recall, that in this case `(skolem!)` is equivalent to `(skolem 1 ("x!1" "y!1" "z!1" "w!1"))` which in turn is equivalent to `(skolem 1 "x!1")` followed by `(skolem 1 "y!1")` followed by `(skolem 1 "z!1")` followed by `(skolem 1 "w!1")`.

We first want to rewrite the subterm `((x!1 * y!1) * z!1)` using `assoc`. To that end we add `assoc` to our antecedents with `(lemma "assoc")`.

```
{-1}  FORALL (x, y, z: M): (x * y) * z = x * (y * z)
  |-------
[1]   ((x!1 * y!1) * z!1) * w!1 = x!1 * (y!1 * (z!1 * w!1))
```

Normally, the `lemma` command applies to something already proved, a “lemma”. In that case it corresponds to the CUT rule. With axioms it’s a bit different. We can think of them as being implicitly added to the antecedents. In that case the `lemma` command simply highlights them. It is also possible to extend the sequent calculus by real axioms.

At any rate, we will need our axiom more than once, so we start by copying it corresponding to CONTR: `copy -1`.

```
{-1}  FORALL (x, y, z: M): (x * y) * z = x * (y * z)
[-2]  FORALL (x, y, z: M): (x * y) * z = x * (y * z)
  |-------
[1]   ((x!1 * y!1) * z!1) * w!1 = x!1 * (y!1 * (z!1 * w!1))
```

The `-1` formula must now be instantiated with the `inst` command: `(inst -1 "x!1" "y!1" "z!1")`.

```
{-1}  (x!1 * y!1) * z!1 = x!1 * (y!1 * z!1)
[-2]  FORALL (x, y, z: M): (x * y) * z = x * (y * z)
  |-------
[1]   ((x!1 * y!1) * z!1) * w!1 = x!1 * (y!1 * (z!1 * w!1))
```

Now we can use an equality rule: `(replace -1 1)`:

```
[-1]  (x!1 * y!1) * z!1 = x!1 * (y!1 * z!1)
[-2]  FORALL (x, y, z: M): (x * y) * z = x * (y * z)
  |-------
{1}   (x!1 * (y!1 * z!1)) * w!1 = x!1 * (y!1 * (z!1 * w!1))
```

The parentheses around the just replaced subterm are not displayed which is irritating. Next, we must apply associativity to the whole left hand side, the middle term being this time not just a constant, but `y!1 * z!1`. This time we use a slightly more powerful command: `inst-cp` which works like `inst` but copies the formula to be instantiated beforehand. So,

```
(inst-cp -2 "x!1" "y!1*z!1" "w!1")
```

brings us to

```
-1]  (x!1 * y!1) * z!1 = x!1 * (y!1 * z!1)
[-2]  FORALL (x, y, z: M): (x * y) * z = x * (y * z)
{-3}  (x!1 * (y!1 * z!1)) * w!1 = x!1 * (y!1 * z!1 * w!1)
  |-------
[1]   x!1 * (y!1 * z!1) * w!1 = x!1 * (y!1 * (z!1 * w!1))
```

Now `(replace -3 1)` gives

```
[-1]  (x!1 * y!1) * z!1 = x!1 * (y!1 * z!1)
[-2]  FORALL (x, y, z: M): (x * y) * z = x * (y * z)
[-3]  (x!1 * (y!1 * z!1)) * w!1 = x!1 * (y!1 * z!1 * w!1)
  |-------
{1}   x!1 * ((y!1 * z!1) * w!1) = x!1 * (y!1 * (z!1 * w!1))
```

where again, I've inserted some parens. This is getting close; instantiating the remaining copy of associativity with

```
(inst -2 "y!1" "z!1" "w!1")
```

followed by `(replace -2 1)` completes the proof.

High-level proof Now let's do the same proof again with more powerful commands: After (skolem!) we get as before

```
  |-------
{1}   ((x!1 * y!1) * z!1) * w!1 = x!1 * (y!1 * (z!1 * w!1))
```

Now rather than bringing in assoc, instantiating, and then rewriting our goal with it, we can use the command rewrite-lemma (p. 65 of prover-guide.ps) which does these two steps in one go:

```
(rewrite-lemma "assoc" ("x" "x!1" "y" "y!1" "z" "z!1"))
```

results in

```
Rewriting using assoc where
   x gets x!1,
   y gets y!1,
   z gets z!1,
this simplifies to:
assoc4 :

  |-------
{1}   x!1 * (y!1 * z!1) * w!1 = x!1 * (y!1 * (z!1 * w!1))
```

The command rewrite-lemma takes as second argument a substitution which is an even length list providing the required values for all the bound variables in the lemma. After perfoming this instantiation it must become an equation with which rewriting then takes place. Admittedly, this syntax is somewhat inconsistent with the syntax of the (inst) command.

Now, we want to perform the same procedure again, but with a different substitution:

```
 (rewrite-lemma "assoc" ("x" "x!1" "y" "y!1 * z!1" "z" "w!1"))
```

Fortunately, we don't need to type in again from scratch: the keystroke M-p, that is the Alt key and the P key together, brings up the last command entered. We only need to edit the substitution. Further M-p bring up even earlier commands. If we've gone too far, we can use M-n to go back again. There's another way to ease typing: If we start to type a command like so

```
(rewri
```

and then type `M-s` it will be completed to the last command typed with the same beginning, i.e., in our case the last `rewrite-lemma` command which again can then be edited.

However we enter the command, it brings us to

```
    |-------
{1}    x!1 * (y!1 * z!1 * w!1) = x!1 * (y!1 * (z!1 * w!1))
```

at which point

```
(rewrite-lemma "assoc" ("x" "y!1" "y" "z!1" "z" "w!1"))
```

completes the job.

The `rewrite-lemma` command can also be given the `:dir RL` optional argument, so we could have worked on the right hand side instead like so:

```
    |-------
{1}    ((x!1 * y!1) * z!1) * w!1 = x!1 * (y!1 * (z!1 * w!1))

Rule? (rewrite-lemma "assoc" ("x" "y!1" "y" "z!1" "z" "w!1") :dir RL
    |-------
{1}    ((x!1 * y!1) * z!1) * w!1 = x!1 * ((y!1 * z!1) * w!1)
```

Even quicker proofs: Filling in the instantiations is tedious and can partly be automated. That's what the command `rewrite` (p.64 of `prover-guide.ps`) does for us. Unfortunately, not always successfully, which is why it's good to know the more basic commands. In the example at hand it works, however, and we can do the entire proof by issuing the following four commands:

```
(skolem!)
(rewrite "assoc")
(rewrite "assoc")
(rewrite "assoc")
```

Even quicker is the following approach: using the command (p. 89 f. of `prover-guide.ps`)

```
(auto-rewrite "assoc")
```

we tell PVS that it should consider all instances of `assoc` as automatic rewrite rules. After that command, `(grind)` completes the task.

Uniqueness of neutral elements We postulate a neutral element by adding

```
  e : M
  neutral_left : AXIOM
      FORALL(x:M):e*x=x
  neutral_right : AXIOM
      FORALL(x:M):x*e=x
```

Our goal is

```
  neutral_unique : THEOREM
      FORALL(e1:M):
        (FORALL(x:M): e1*x=x) AND
        (FORALL(x:M): x*e1=x) IMPLIES e=e1
```

Here it is useful to first get an idea of how this proof should be done informally:

If `e1` is also a neutral element then `e = e*e1`. By neutrality of `e` the right hand side equals `e1` and we're done.

In PVS after `(skolem!)` and `(flatten)` or, more compactly, `(skosimp)`, we get

```
{-1}  FORALL (x: M): e1!1 * x = x
{-2}  FORALL (x: M): x * e1!1 = x
  |-------
{1}   e = e1!1
```

We want to expand `e` as `e * e1!1` using `-2`. We instantiate. . .

```
(inst -2 "e")

{-1}  FORALL (x: M): e1!1 * x = x
{-2}  e * e1!1 = e
  |-------
{1}   e = e1!1
```

and replace

```
(replace -2 1 :dir RL)
```

bringing us to

```
[-1]  FORALL (x: M): e1!1 * x = x
[-2]  e * e1!1 = e
  |-------
{1}   e * e1!1 = e1!1
```

which is an instance of `neutral_left`. The way to convince PVS of this is

```
(use "neutral_left")
```

A more pedestrian way would be to use `lemma` and `inst`.

A slightly quicker proof After `(skosimp)` we can use `(rewrite-with-fnum -2 ("x" "e") :dir RL)` to achieve the expansion of the left hand side. The command `rewrite-with-fnum`, is like `rewrite`, so doesn't normally require a substitution (instantiation). In this case, we have to give it because otherwise the replacement is applied to the right hand side, too!.

This brings us to

```
[-1]  FORALL (x: M): e1!1 * x = x
[-2]  FORALL (x: M): x * e1!1 = x
  |-------
{1}   e * e1!1 = e1!1
```

At which point we conclude using `(use "neutral_left")`.

I couldn't find a more efficient proof of that one. Can you?

Invertible elements An element of `M` is invertible if it has an inverse:

```
  Invertible(x:M) : boolean = EXISTS(y:M): x*y=e AND y*x = e
```

We can prove that the neutral element is invertible:

```
   inv_neutral : THEOREM
     Invertible(e)
```

We see here, how, abbreviations a.k.a. definitions are introduced.

After invoking the prover the first command must be `expand "Invertible"` to open the definition. This brings us to

```
  |-------
{1}   EXISTS (y: M): e * y = e AND y * e = e
```

Now we have to come up with an alleged inverse to `e`. Surprise, it's going to be `e` itself.

```
(inst 1 "e")

  |-------
{1}   e * e = e AND e * e = e
```

Here we could also have used `(inst? 1)` which would leave it to PVS to find the correct instantiation. It sometimes does....

We conclude with `(rewrite "neutral_left")` followed by `(split)`.

The last theorem in this series is that invertibles are closed under product:

```
  |-------
{1}   FORALL (x, y: M):
        Invertible(x) AND Invertible(y) IMPLIES Invertible(x * y)
```

`(skosimp) then (expand "Invertible")` brings us to

```
{-1}  EXISTS (y: M): x!1 * y = e AND y * x!1 = e
{-2}  EXISTS (y: M): y!1 * y = e AND y * y!1 = e
  |-------
{1}   EXISTS (y: M): x!1 * y!1 * y = e AND y * (x!1 * y!1) = e
```

Notice the `then` "strategy" (p. 111 of `prover-guide.ps`). It sequences commands.

Before being able to instantiate `1`, i.e., come up with an alleged inverse to `(x!1*y!1)` we must "open" the assumptions, i.e., introduce fresh constants for the inverses of `x!1` and `y!1`, respectively, which are guaranteed by `-1` and `-2`.

I find it better to give suggestive names to these, so we do

```
(skolem -1 "xinv") then (skolem -2 "yinv") then (flatten)
```

to get

```
{-1}  x!1 * xinv = e
{-2}  xinv * x!1 = e
{-3}  y!1 * yinv = e
{-4}  yinv * y!1 = e
  |-------
[1]   EXISTS (y: M): x!1 * y!1 * y = e AND y * (x!1 * y!1) = e
```

Now, we have to think a bit as to what the inverse to `x!1*y!1` should be. Well, thinking of `*` as sequencing of "actions" it becomes clear that the inverse ought to be `yinv * xinv`. That's the "rule of sock and shoe". Therefore, `(inst 1 "yinv*xinv")` is the command of choice.

```
{-1}  x!1 * xinv = e
{-2}  xinv * x!1 = e
{-3}  y!1 * yinv = e
{-4}  yinv * y!1 = e
  |-------
[1]   x!1 * y!1 * (yinv * xinv) = e AND (yinv * xinv) * (x!1 * y!1)
```

Relying on PVS' cleverness and doing `(inst?  1)` isn't a good idea here.

Splitting ($\wedge$-R) brings us two subgoals of which we'll only treat the first here:

```
{-1}  x!1 * xinv = e
{-2}  xinv * x!1 = e
{-3}  y!1 * yinv = e
{-4}  yinv * y!1 = e
  |-------
[1]   x!1 * y!1 * (yinv * xinv) = e
```

Now we must first "rebracket" our goal to

```
x!1 * (y!1 * yinv) * xinv = e
```

While this can certainly be done by successive application of associativity, it is easier to just claim this and prove it separately. To do this, we issue the command

```
(case "x!1 * (y!1 * yinv) * xinv = e")
```

This presents us with two subgoals. One asking us to prove our goal under the extra assumption of the "claim":

```
[-1]  x!1 * (y!1 * yinv) * xinv = e
[-2]  x!1 * xinv = e
[-3]  xinv * x!1 = e
[-4]  y!1 * yinv = e
[-5]  yinv * y!1 = e
  |-------
[1]   x!1 * y!1 * (yinv * xinv) = e
```

This follows from associativity, so

```
(auto-rewrite "assoc") then (grind)
```

does the job. Next, we must prove our claim:

```
[-1]  x!1 * xinv = e
[-2]  xinv * x!1 = e
[-3]  y!1 * yinv = e
[-4]  yinv * y!1 = e
  |-------
{1}   x!1 * (y!1 * yinv) * xinv = e
[2]   x!1 * y!1 * (yinv * xinv) = e
```

The old goal is still there, we can remove it with `(delete 2)` corresponding to rule WEAK-R. The rest is a rewriting consequence of `neutral_left` and `neutral_right`, so we install these and grind.

We could have turned off associativity with the command `(stop-rewrite "assoc")`, but this wasn't even necessary here.

8 Recursive functions

Many function definitions in mathematics, programming, and more so program specification are recursive.

Even if—for the sake of efficiency—the actual program uses an iterative solution, for specification and verification a recursive definition is usually more convenient.

Examples of recursive definitions Sum in pattern-matching notation:

$$\begin{array}{l} \sum_{i=0}^{0} a_i = a_0 \\ \sum_{i=0}^{n+1} a_i = a_{n+1} + \sum_{i=0}^{n} a_i \end{array}$$

Sum in fixpoint notation:

$$\begin{array}{l} \sum_{i=0}^{n} a_i = \text{if}\, n = 0 \\ \quad \text{then}\, a_0 \\ \quad \text{else}\, a_n + \sum_{i=0}^{n-1} a_i \,\text{endif} \end{array}$$

Binary search in pattern-matching notation:

$$\begin{array}{l}
\textit{find}(a, \mathrm{null}) = \mathrm{ff} \\
\textit{find}(a, \mathrm{cons}(b, l)) = (a{=}b) \lor (a{\le}b \land \textit{find}(a, \textit{left}(l))) \lor (a{>}b \land \textit{find}(a, \textit{right}(l)))
\end{array}$$

Binary search in fixpoint notation:

$$\begin{array}{l}
\textit{find}(a, l) = \text{if}\, l = [] \\
\quad \text{then}\, \mathrm{ff} \\
\quad \text{else}\, a = \mathrm{car}(l) \lor \\
\quad\quad (a{\le}b \land \textit{find}(a, \textit{left}(\mathrm{cdr}(l)))) \lor \\
\quad\quad (a{>}b \land \textit{find}(a, \textit{right}(\mathrm{cdr}(l)))) \text{ endif}
\end{array}$$

These clauses define honest-to-goodness functions on natural numbers and lists (or arrays). As you probably know this need not always be the case. For one thing, recursively defined functions may be partial ($f(n) = f(n)$), for another, some equations may not define a function at all.

$$\begin{array}{l}
f(n) = f(n) \\
g(0) = 0 \\
g(n+2) = g(n) \\
g(1) = \min\{g(2n) \mid n \in \mathbb{N}\} \\
h(n) = 0 \\
h(n) = 1
\end{array}$$

Fixpoint form Fix $a_0, a_1, \ldots$ and let *sum* $: \mathbb{N} \to \mathbb{R}$ be the function defined by *sum*$(n) = \sum_{i=0}^{n} a_i$.

We have *sum*$(n) = F($*sum*$, n)$ where

$$\begin{array}{l}
F(f, n) = \text{if}\, n = 0 \\
\quad \text{then}\, a_0 \\
\quad \text{else}\, a_n + f(n-1) \text{ endif}
\end{array}$$

Exercise: define *fact*$(n) = n!$. Give F such that *fact*$(n) = F($*fact*$, n)$.

In PVS all functions are total and therefore, general recursive function definitions are not permitted. Rather, an explicit measure must be provided ensuring that the definition terminates.

Theorem (Well-founded recursion): Let A, B be nonempty sets, let $F : (A \to B) \to (A \to B)$ be a functional, $w : A \to \mathbb{N}$ be a function (the “measure”).

Suppose that for each $f : A \to B$ and $a \in A$ the value $F(f, a)$ depends only on those values $f(a')$ for which $w(a') < w(a)$, that is to say

$$\forall a{:}A.\forall f, g{:}A \to B.(\forall x{:}A.w(x) < w(a) \Rightarrow f(x){=}g(x)) \Rightarrow F(f, a){=}F(g, a)$$

Then there exists a uniquely determined function $f_F : A \to B$ such that

$$\forall a{:}A.f_F(a) = F(f_F, a)$$

Proof. Let b_0 be a fixed element of B.

We define $f_F(a)$ by induction on $w(a)$. Suppose $w(a) = 0$. Then $F(f, a)$ is independent of f, so we can put $f_F(a) = F(f, a)$ where f is an arbitrary function from A to B, e.g. a constant one.

Suppose that $f_F(x)$ has already been defined for all x with $w(x) < n$ and that $w(a) = n$. Then we define a function $f : A \to B$ by

$$f(x) = \begin{cases} f_F(x), \text{ if } w(x) < n \\ b_0, \text{ if } w(x) \geq n \end{cases}$$

We then put $f_F(a) \overset{\text{def}}{=} F(f, a)$.

This procedure defines $f_F(a)$ for all values a.

Next, we show that $F(f_F, a) = f_F(a)$ for all a. Well, given a fixed but arbitrary element $a \in A$ (PVS would call it $a!1$) we see that $f_F(a)$ has been defined as $F(f, a)$ where f is the function which agrees with f_F on values x with $w(x) < n$ and is b_0 elsewhere.

But we have assumed that $F(f_F, a) = F(f, a)$ in this case.

For uniqueness we argue as follows. Suppose that $F(g, a) = g(a)$ for some function $g : A \to B$. We show by induction on $w(a)$ that $f(a) = g(a)$. The details are left to the reader. □

In many examples the evaluation of $F(f, a)$ proceeds by evaluating the function f on a fixed number of arguments $a_1, \ldots, a_n$ depending only on a and having measure smaller than a, i.e., $w(a_i) < w(a)$. This was in particular the case for the *sum* and *fact* example.

Questions What would be an appropriate measure for the definition of *find*?

What is an appropriate measure for

```
$F_{\mathit{merge}}(f, l_1, l_2) =$
  if $l_1 = \text{null}$
    then $l_2$
  elsif $l_2 = \text{null}$
    then $l_1$
    else $\text{cons}(\text{car}(l_1), \text{cons}(\text{car}(l_2), f(\text{cdr}(l_1), \text{cdr}(l_2))))$
```

where $A = \text{list[nat]} \times \text{list[nat]}$ and $B = \text{list[nat]}$.

Hint: you may assume a function *length* : list$[nat] \rightarrow$ nat.

8.1 Defining functions in PVS

We have already seen the definition of a predicate, namely `Invertible`. For PVS such a predicate is nothing but a function to the type `boolean`.

Using the same syntax we can define other functions like so:

```
f(x,y:nat) : nat = (x+y)*(x-y)
```

and we can prove

```
a: THEOREM f(5,3) = 16
```

using `(grind)`. This method also does simple algebra:

```
b: THEOREM FORALL(x,y:nat):f(x,y)=x^2-y^2
```

Anyway, I'm gettting distracted from todays topic: recursive definitions. Here is how we define *sum* in PVS provided `a: [nat->real]` has been defined or declared:

```
sum(n:nat) : RECURSIVE real =
   IF n=0 THEN a(0) ELSE a(n)+sum(n-1) ENDIF
MEASURE n
```

Try to memorise the slightly awkward syntax: the keyword `RECURSIVE` goes between the colon and the result type. And don't forget the measure either. It's supposed to go down as you unfold the recursion.

8.2 TCCs

When PVS typechecks such a definition (and this takes place before you enter the prover) it attempts to show that this is the case (the measure going down, that is). If it doesn't succeed a typechecking condition (TCC) is generated which you would then have to prove interactively using the prover.

You can display the TCC with the command `M-x show-tccs`. In the example at hand the TCCs are simple enough

```
sum_TCC1: OBLIGATION FORALL (n: nat): NOT n = 0 IMPLIES n - 1 >= 0;
sum_TCC2: OBLIGATION FORALL (n: nat): NOT n = 0 IMPLIES n - 1 < n;
```

The first one comes from the use of `n-1`. The type `nat` is in fact a subtype of the integers which is a subtype of the rationals, etc. A priori the minus function returns an integer. In the situation at hand, we know that `n` is not zero, so `n-1` is in fact a natural number. PVS was able to "prove" that by itself.

The other TCC comes from the recursion. We must show that the measure of the argument of the recursive call (here `n-1`) is smaller than the measure of the current argument (here `n`). Again, PVS can prove that itself.

We can now prove (using `grind`) simple goals which follow directly from the recursive definition like

```
c: THEOREM
sum(5) = a(0) + a(1) + a(4) + a(3) + a(2) + a(5)
```

We come to more interesting goals below.

8.2.1 Higher-order functions

Function types in PVS are like any other type. We can use this feature to pass the sequence `a:[nat->nat]` as an extra argument to `sum`:

```
sum(a:[nat->real],n:nat) : RECURSIVE real =
          IF n=0 THEN a(0) ELSE a(n)+sum(a,n-1) ENDIF
       MEASURE n
```

The old definition of `sum` applies when the first argument isn't a function, this is an instance of *overloading*.

We don't even have to delete the previous definition of `sum`. PVS can tell the two apart by their types (this is known as *overloading*).

Now, we can apply `sum` to concrete functions, e.g., we might define

8.2.2 Examples

```
id(x:nat):nat = x
d : THEOREM
   sum(id,5) = 25
e : THEOREM
    sum(LAMBDA(x:nat):x*x,4) = 30
```

```
id(x:nat):nat = x
```

and then prove

```
d : THEOREM
   sum(id,5) = 25
```

If we don't want to sacrifice a name for the argument function we must use a lambda abstraction

```
e : THEOREM
    sum(LAMBDA(x:nat):x*x,4) = 30
```

Summary:

- PVS allows for definition of functions by well-founded recursion
- Such definitions generate proof obligations known as typechecking conditions (TCCs)
- TCCs also arise in conjunction with subtypes. More later.
- Within one and the same theory you can have several functions of the same name if their argument types are distinct (overloading)
- Functions can be arguments as well as results of functions. The `LAMBDA` notation allows one to construct function terms on the fly to be passed as argument to another function.
- PVS knows that a recursively defined function satisfies its defining equations

9 Proof by induction and higher-order logic

So far we have proved simple consequences of the recursive equations in which the recursive argument was a concrete value. If we want to prove more interesting universally quantified statements then we need a more powerful principle: proof by induction.

You probably have seen induction already: to prove a statement $\phi(n)$ for all natural numbers n you must prove it for 0 and then—assuming a fixed but arbitrary n'—you must prove it for $n' + 1$ under the extra assumption that it $\phi(n')$ holds.

In first-order logic:

$$\phi(0) \wedge (\forall n{:}\mathbb{N}.\phi(n) \Rightarrow \phi(n+1)) \Rightarrow \forall n{:}\mathbb{N}.\phi(n)$$

In higher-order logic:

$$\forall \phi{:}[\mathbb{N} \rightarrow \text{boolean}].\phi(0) \wedge (\forall n{:}\mathbb{N}.\phi(n) \Rightarrow \phi(n+1)) \Rightarrow \forall n{:}\mathbb{N}.\phi(n)$$

As a formula this *induction scheme* looks as follows:

$$\phi(0) \wedge (\forall n{:}\mathbb{N}.\phi(n) \Rightarrow \phi(n+1)) \Rightarrow \forall n{:}\mathbb{N}.\phi(n)$$

In first-order logic we need one such formula for every predicate ϕ. In *higher-order logic* we can quantify over ϕ just as we quantify over individuals:

$$\forall \phi{:}[\mathbb{N} \rightarrow \text{boolean}].\phi(0) \wedge (\forall n{:}\mathbb{N}.\phi(n) \Rightarrow \phi(n+1)) \Rightarrow \forall n{:}\mathbb{N}.\phi(n)$$

The proof rules for higher-order logic are essentially the same as those for first-order logic. Only the ways to form formulas are extended. A formula is just a term of type boolean and these can be formed using the connectives and quantifiers as well as by function application.

Semantically, the type boolean is interpreted as the set of truth values $\{\mathsf{tt}, \mathsf{ff}\}$; function types are interpreted as sets of all functions. Unlike pure first-order logic, higher-order logic do not admit complete proof systems. The reason is Gödel's incompleteness theorem which you may have come across in popular science books.

Higher-order logic can thus be defined as first-order logic with

- Type `bool` and types closed under $[A_1, \ldots, A_n \rightarrow B]$ ($n$ary function space)
- Unary predicate on type `bool`, i.e., every term of type `bool` can be seen / is a proposition

- $n+1$ary function symbols for application of functions to arguments.

Intended semantics: interpret `bool` as $\{\mathtt{tt}, \mathtt{ff}\}$, function spaces as sets of functions. No complete axiomatisation exists.

Approximations: λ-terms witnessing the existence of certain functions and predicates, β-equations, extensionality, comprehension axioms (or constants for quantifiers, connectives, functions), choice axioms.

Complete for non-standard models: Henkin models, toposes.

Rather than formally defining higher-order logic with its syntax and proof rules we will introduce it in PVS and get to know it by example.

Here is how the induction axiom is formulated in PVS:

```
nat_induction : LEMMA
 FORALL(p:[nat->boolean]):
     (p(0) AND (FORALL j: p(j) IMPLIES p(j+1)))
        IMPLIES (FORALL i: p(i))
```

This is proved from a slightly more general axiom (well-foundedness of $<$ on `nat`) in `lib/prelude.pvs`.

We are not so much interested in how to prove induction, but how to use it in order to prove other things.

Consider the classic $\sum_{i=0}^{n} i = n(n+1)/2$. We write $\phi(n) \equiv \sum_{i=0}^{n} i = n(n+1)/2$. We have $\phi(0) \equiv 0 = 0(0+1)/2$. True. We have $\phi(n_0+1) \equiv (n_0+1) + \sum_{i=0}^{n_0} i = (n_0+1)(n_0+2)/2$.

Using the induction hypothesis $\phi(n_0)$ this rewrites to $(n_0+1)+n_0(n_0+1)/2 = (n_0+1)(n_0+2)/2$ which is true by simple arithmetic.

Now we want to do the same thing in PVS:

We start with

```
   |-------
{1}   FORALL (n: nat): sum(LAMBDA (i: nat): i, n) = n * (n + 1) / 2
```

Detailed proof We bring in `nat_induction`

```
Rule? (lemma "nat_induction")

{-1}  FORALL (p: pred[nat]):
         (p(0) AND (FORALL j: p(j) IMPLIES p(j + 1)))
            IMPLIES (FORALL i: p(i))
```

```
  |-------
[1]   FORALL (n: nat):
         sum(LAMBDA (i: nat): i, n) = n * (n + 1) / 2
```

Now `nat_induction` works for all predicates `p`, but we need it for a particular one, so we must instantiate. But what with? We could have introduced an abbreviation for the predicate we're interested in but it's too late for that now, so we must resort to the lambda notation:

```
Rule? (inst -1 "LAMBDA(n:nat):
        sum(LAMBDA (i: nat): i, n) = n * (n + 1) / 2")

{-1}  ((LAMBDA (n: nat): sum(LAMBDA (i: nat): i, n) =
     n * (n + 1) / 2)(0) AND
        (FORALL j:
           (LAMBDA (n: nat): sum(LAMBDA (i: nat): i, n) =
         n * (n + 1) / 2)(j) IMPLIES
            (LAMBDA (n: nat): sum(LAMBDA (i: nat): i, n) =
        n * (n + 1) / 2)(j + 1)))
       IMPLIES
       (FORALL (i_49: nat): (LAMBDA (n: nat): sum(LAMBDA (i: nat): i
     n) =
          n * (n + 1) / 2)(i_49))
  |-------
[1]   FORALL (n: nat): sum(LAMBDA (i: nat): i, n) = n * (n + 1) / 2
```

That looks daunting. What has happened is that every occurrence of `p` in `-1` has been literally replaced by the instantiation we provided. But we would like a bit more than just that: we would like to see arguments to `p` such as `0` in the base case and `i+1` in the induction step to be plugged in for the lambda-bound variable `n`. This is known as beta reduction and we perform it in PVS by issuing the command `(beta)`

```
Rule? (beta)
{-1}  (sum(LAMBDA (i: nat): i, 0) = 0 * (0 + 1) / 2 AND
        (FORALL j:
           sum(LAMBDA (i: nat): i, j) = j * (j + 1) / 2 IMPLIES
            sum(LAMBDA (i: nat): i, j + 1) = (j + 1) * (j + 1 + 1) /
       IMPLIES (FORALL (i_49: nat):
          sum(LAMBDA (i: nat): i, i_49) = i_49 * (i_49 + 1) / 2)
  |-------
[1]   FORALL (n: nat): sum(LAMBDA (i: nat): i, n) = n * (n + 1) / 2
```

What we have achieved so far is that the required instance of the induction scheme is among our antecedents. We now want to use it which shouldn't be difficult as it ends exactly with what we want to prove. So we use rule ⇒-L which is of the `split` kind:

```
Rule? (split)
```

This yields two subgoals, one asking us to deduce `1` from the conclusion of `-1`—that's a propositional axiom, so PVS won't bother presenting us with it—, and another one asking us to prove the premise of `-1` (or `1` again). Since the premise of `-1` is a conjunction (of base case and induction step) we must use ∧-R which is again of the `split` kind and in fact PVS has already performed this at the last `split` command so that we don't need to enter it again. We are thus presented with two actual subgoals:

```
 |-------
{1}   sum(LAMBDA (i: nat): i, 0) = 0 * (0 + 1) / 2
[2]   FORALL (n: nat):
          sum(LAMBDA (i: nat): i, n) = n * (n + 1) / 2
```

and

```
 |-------
{1}   sum(LAMBDA (i: nat): i, 0) = 0 * (0 + 1) / 2
[2]   FORALL (n: nat):
          sum(LAMBDA (i: nat): i, n) = n * (n + 1) / 2
```

The first of these (the base case) follows by simple arithmetic: `(grind)` disposes of it.

We could also have proved this by hand using elementary properties of real arithmetic summarised in `prelude.pvs`.

The induction step is more interesting. We first delete `2` and then introduce a "fixed but arbitrary" name, say `n!0` by

```
Rule? (skolem 1 "n!0")
  |-------
{1}   sum(LAMBDA (i: nat): i, n!0) = n!0 * (n!0 + 1) / 2 IMPLIES
       sum(LAMBDA (i: nat): i, n!0 + 1) = (n!0 + 1) * (n!0 + 1 + 1)
```

This being an implication we use ⇒-L or `(flatten)` to give

```
{-1}  sum(LAMBDA (i: nat): i, n!0) = n!0 * (n!0 + 1) / 2
  |-------
{1}   sum(LAMBDA (i: nat):
         i, n!0 + 1) = (n!0 + 1) * (n!0 + 1 + 1) / 2
```

This looks like what we had expected. We may assume that what we want to show holds for a fixed but arbitrary `n!0` and from that we must show it for `n!0+1`. Let's expand the sum in the succedent:

```
Rule? (expand "sum" 1)

[-1]  sum(LAMBDA (i: nat): i, n!0) = n!0 * (n!0 + 1) / 2
  |-------
{1}   1 + sum(LAMBDA (i: nat): i, n!0) + n!0 =
              (2 + n!0 + (n!0 * n!0 + 2 * n!0)) / 2
```

Notice that with recursive definitions the `expand` command performs one recursive unfolding rather than replacing `sum` with its definition using the keyword `RECURSIVE`.

We now recognise the left-hand-side of `-1` as a subterm, it's therefore a good idea to replace it with the right hand side:

```
Rule? (replace -1 1)

[-1]  sum(LAMBDA (i: nat): i, n!0) = n!0 * (n!0 + 1) / 2
  |-------
{1}   1 + n!0 * (n!0 + 1) / 2 + n!0 =
            (2 + n!0 + (n!0 * n!0 + 2 * n!0)) / 2
```

The conclusion is an arithmetic identity so `(grind)` can establish it.

Quicker proof The first few steps were rather awkward and independent of the particular goal at hand. For this reason the command `induct` has been provided which performs them all in one go. In the situation

```
  |-------
{1}   FORALL (n: nat): sum(LAMBDA (i: nat): i, n) = n * (n + 1) / 2
```

the command `(induct "n")` produces two subgoals:

```
  |-------
{1}   sum(LAMBDA (i: nat): i, 0) = 0 * (0 + 1) / 2
```

and

```
  |-------
{1}   FORALL j:
        sum(LAMBDA (i: nat): i, j) = j * (j + 1) / 2 IMPLIES
         sum(LAMBDA (i: nat): i, j + 1) =
                (j + 1) * (j + 1 + 1) / 2
```

which we deal with as before. Actually, the second one can be proved simply with the command `(skosimp!)  then (grind)`.

An even quicker proof goes with the single command `(induct-and-simplify "n")`.

PVS is surprisingly good at doing inductive proofs almost automatically. For example, Cassini's identity

$$F_{n+2}F_n - F_{n+1}^2 = (-1)^n$$

can be proved with a single "induct-and-simplify". One should define $(-1)^n$ recursively.

The following is known as Abel's lemma and plays a role in number theory, more specificly, Dirichlet series.

Let (a_n) and (b_n) be two sequences. Put:

$$A_{m,p} = \sum_{n=m}^{n=p} a_n \text{ and } S_{m,m'} = \sum_{n=m}^{n=m'} a_n b_n$$

Then one has:

$$S_{m,m'} = A_{m,m'} b_{m'} + \sum_{n=m}^{n=m'-1} A_{m,n}(b_n - b_{n+1})$$

Here one must induct not on a single quantity, but rather the difference $m' - m$. PVS provides the command `measure-induct+` for that purpose.

10 Lists

A datatype of finite list over arbitrary type of entries is predefined.

If t is a type then list[t] is the type of finite lists with entries over t.

Semantically, elements of list[t] take the form $[x_1, x_2, \ldots, x_n]$ where the x_i are elements of type t.

The constant null denotes the empty list, the function

```
cons : [t,list[t] -> list[t]]
```

tacks an element on to the beginning of a list. For example, if l is the list $[3, 4, 1, 2]$ then $\texttt{cons}(5, l)$ is $[5, 3, 4, 1, 2]$.

The type of lists has the subtype (cons?[t]), consisting of all non-empty lists. The function car takes a nonempty list and returns its first element, i.e., we have

```
car : [(cons?[t]) -> t]
```

the function cdr takes a nonempty list and returns its "tail", i.e., the (possibly empty) list obtained by stripping off its first element.

```
cdr : [(cons?[t]) -> list[t]]
```

For example, if $l = [5, 4, 3, 2]$ then

$$\begin{aligned}\texttt{car}(l) &= 5\\ \texttt{cdr}(l) &= [4, 3, 2]\end{aligned}$$

We also have a predicate null? which tells whether a list is empty

```
null? : [list[t] -> boolean]
```

and another—less used—predicate cons? which tells whether a list is nonempty. In fact the subtype (cons?[t]) is derived from that predicate and more generally, if p:[t->boolean] then (p) is the subtype of t consisting of those elements for which p holds.

In practice, the functions car and cdr are applied to arguments of type list[t] rather than (cons?[t]).

Such application generates a typechecking condition (TCC) which either we or PVS has to prove. A typical case, when such a TCC can be proved automatically, is when car(l) or cdr(l) is used in the ELSE branch of a conditional IF null?(l).

Another typical such case is a usage like cdr(cons(x,l)) which automatically simplifies to l. Similarly, car(cons(x,l)) automatically simplifies to x.

Subtypes If `P : [t->bool]` then `(P)` is a type.

If `a:t` and `P(a)` holds then also `a:(P)`.

If we implicitly assert `a:(P)` then a proof obligation (TCC) `P(a)` arises.

10.1 Recursion on lists

There is a predefined function

```
length : [list[t] -> nat]
```

allowing us to define functions on lists by recursion (using) `length` as a measure. Here is a definition of the function which appends two lists.

```
t :TYPE

append(l1:list[t], l2:list[t]) : RECURSIVE list[t] =
IF null?(l1)
 THEN l2
 ELSE cons(car(l1),append(cdr(l1),l2))
ENDIF
MEASURE length(l1)
```

This means that the function `append` satisfies the following two defining equations:

```
append(null, l2) = l2
append(cons(x,l1), l2) = cons(x,append(l1,l2))
```

In fact, PVS provides a cases-construct allowing us to write `append` in this slightly more perspicuous form. See `prelude.pvs` or the documentation.

Here are a few more function definitions:

```
occ(x:t,l:list[t]) : RECURSIVE nat =
      IF null?(l)
         THEN 0
         ELSIF x=car(l) THEN occ(x,cdr(l))+1 ELSE occ(x,cdr(l)) ENDI
    MEASURE length(l)

filter(p:[t->boolean],l:list[t]) : RECURSIVE list[t] =
     IF null?(l)
```

```
           THEN null
           ELSIF p(car(l)) THEN cons(car(l),filter(p,cdr(l)))
            ELSE filter(p,cdr(l)) ENDIF
     MEASURE length(l)

 rev(l:list[t]) : RECURSIVE list[t] =
     IF null?(l) THEN null
       ELSE append(rev(cdr(l)),cons(car(l),null))
     ENDIF
     MEASURE length(l)

 rev1(l:list[t], acc:list[t]) : RECURSIVE list[t] =
    IF null?(l) THEN acc
      ELSE rev1(cdr(l),cons(car(l),acc))
    ENDIF
    MEASURE length(l)
```

`occ(x,l)` returns the number of occurrences of `x` in the list `l`; `filter(p,l)` returns the list consisting of those elements of `l` which satisfy the predicate `p`. `rev(l)` returns the reversal of list `l` and `rev1(l,acc)`, finally, returns the reversal of `l` followed by `acc`.

10.2 Reduce

The length function itself admits a recursive definition:

```
length(l:list[t]) : RECURSIVE nat =
 IF null?(l)
    THEN 0
    ELSE 1 + length(cdr(l))
 ENDIF
MEASURE ????
```

The trouble is that the only reasonable measure function is `length` itself. As I said, `length` is fortunately predefined (in `prelude.pvs`), but nevertheless it's worth knowing how, namely using the `reduce_nat` functional, which now really is basic. If

```
null_case : nat
cons_case : [t,nat -> nat]
```

are given terms of the indicated types then

```
reduce_nat(null_case, cons_case) : [list[t] -> nat]
```

is (as indicated) a function from lists to natural numbers, namely the function `f` defined recursively by

```
f(null) = null_case
f(cons(x,l)) = cons_case(x,f(l))
```

It is clear that whatever `null_case` and `cons_case` are this defines a unique total function so that `reduce_nat` is justified.

The length function can now be defined as

```
length : [list[t] -> nat] =
     reduce_nat(0, LAMBDA(x:t, prev:nat): prev+1)
```

More generally, we have a construct `reduce` which allows for the definition of functions on lists with result type other than `nat` by such structural recursion; however, these are more easily defined using well founded recursion with `length` as the measure, and indeed, if you find `reduce` confusing, just take `length` for granted and define all your functions using `length` or derived forms as measure.

11 Proof by list induction

Also predefined is the following induction principle for lists

```
  list_induction: AXIOM
    FORALL (p: [list -> boolean]):
      (p(null) AND
        (FORALL (cons1_var: T, cons2_var: list):
           p(cons2_var) IMPLIES p(cons(cons1_var, cons2_var))))
       IMPLIES (FORALL (list_var: list): p(list_var))
```

which states that a property `p` which

1. holds for the empty list

2. holds for an arbitrary list of the form `cons(x,l)` provided it holds for `l`

holds for all lists.

This principle can be invoked just as `nat_induction` using `lemma`, `inst`, `beta`, etc. or using the `induct` command, or, indeed, using the powerful `induct-and-simplify` command.

11.1 Associativity of append

Let's do an example proof: associativity of `append`

```
  |-------
{1}   FORALL (l1, l2, l3: list[t]):
        append(append(l1, l2), l3) = append(l1, append(l2, l3))
```

We need to do induction on one of `l1, l2, l3`.

Induction on `l1` will make a defining equation for `append` applicable which (in the `cons` case) brings even the outermost `append` into a rewritable form, so that looks promising: We invoke `(induct "l1")` and get two subgoals, the first of which is

```
  |-------
{1}   FORALL (l2, l3: list[t]):
        append(append(null, l2), l3) = append(null, append(l2, l3))
```

We could either try to induct on `l2` or `l3` here, or solve it directly which should intuitively be possible, as by virtue of the recursive equations both sides equal `append(l2,l3)`. The way to convince PVS of this is either to use `(grind)` or to introduce fresh names for the universal quantifier: `(skolem 1 ("l2!1" "l3!1"))`

```
  |-------
{1}   append(append(null, l2!1), l3!1) = append(null, append(l2!1, l
```

Now we want to replace the second and third occurrence of `append` by their definitions (as this will bring about a simplification), but not the first and fourth (as this will make things more complicated. The command `(expand "append" 1 2)` expands the second occurrence of `append` in formula 1. Using this again on occurrence 3 achieves our goal. Alternatively, we can use the command

```
(expand "append" :if-simplifies T)
```

which only expands those occurrences of append which result in a simplification (formally: those whose definition contains an if-then-else whose guard is equal to either TRUE or FALSE).

At any rate, we get

```
 |------
{1}    append(l2!1, l3!1) = append(l2!1, l3!1)
```

which is an instance of reflexivity and thus discharged.

Now, we get to see the second subgoal:

```
  |-------
{1}   FORALL (cons1_var: t, cons2_var: list[t]):
        (FORALL (l2, l3: list[t]):
           append(append(cons2_var, l2), l3) =
            append(cons2_var, append(l2, l3)))
         IMPLIES
         (FORALL (l2, l3: list[t]):
            append(append(cons(cons1_var, cons2_var), l2), l3) =
             append(cons(cons1_var, cons2_var), append(l2, l3)))
```

We introduce fresh names for the universally quantified constants:

```
(skolem 1 ("x!1" "l!1"))
```

and (flatten) giving us

```
{-1}  FORALL (l2, l3: list[t]):
        append(append(l!1, l2), l3) = append(l!1, append(l2, l3))
  |-------
{1}   FORALL (l2, l3: list[t]):
        append(append(cons(x!1, l!1), l2), l3) =
         append(cons(x!1, l!1), append(l2, l3))
```

Here -1 is the induction hypothesis. We must now introduce skolem constants for the universal quantifier as in the base case with (skolem 1 ("l2!1" "l3!1")) and we can in fact at this point isntantiate the induction hypothesis with these values: (inst -1 "l2!1" "l3!1"). This means that we cannot use the induction hypothesis with values other than these two. In general, this is risky as there are cases in which we have to use the induction hypothesis with other cleverly chosen instantiations, see Section 11.3 below. Here, however, it is safe. If in doubt, postpone instantiations as long as possible.

```
{-1}  append(append(l!1, l2!1), l3!1) = append(l!1, append(l2!1, l3!
  |-------
[1]   append(append(cons(x!1, l!1), l2!1), l3!1) =
       append(cons(x!1, l!1), append(l2!1, l3!1))
```

Now we see that we have a couple of instances of `append` which can be simplified by way of the recursive equations.

```
(expand "append" :if-simplifies T)
```

Remember that typing `(exp` followed by `M-s`, i.e., Alt s produces this command.

```
[-1]  append(append(l!1, l2!1), l3!1) = append(l!1, append(l2!1, l3!
  |-------
{1}   cons(x!1, append(append(l!1, l2!1), l3!1)) =
       cons(x!1, append(l!1, append(l2!1, l3!1)))
```

Now we discover the lhs of the induction hypothesis (-1) as a subterm and in fact the current sequent follows by mere equational reasoning. Therefore, it can be dispatched with `(grind)`. If we insist on doing it by hand we do `(replace -1 1)` leading to an instance of reflexivity.

This entire proof could also be done with the single command `(induct-and-simplify "l1")`.

11.2 Occurrences

Recall the function `occ` which counts the number of occurrences of a given element in a list. We want to prove:

```
  |-------
{1}   FORALL (x: t, l1, l2: list[t]):
        occ(x, append(l1, l2)) = occ(x, l1) + occ(x, l2)
```

Again, `(induct-and-simplify "l1")` will do the job, but for the sake of it we go for something more elementary. We start with `(induct "l1")`. As said before, there is no general rule as to whether one should just skolemise or use induction and if yes on which argument. Rule of thumb is that theorems involving recursively defined functions require proof by induction and that the induction should be on the argument which promises the most simplifications to happen.

Here this is `l1` so we do `(induct "l1")` giving us two subgoals, first the base case:

```
  |-------
{1}   FORALL (x: t, l2: list[t]):
        occ(x, append(null, l2)) = occ(x, null) + occ(x, l2)
```

which after rewriting the append-term and the second occurrence of `occ` becomes an arithmetic identity. We dispatch the whole subgoal with `(grind)` bringing us to the second subgoal, the inductive step which after skolemising and flattening looks like so:

```
{-1}  FORALL (x: t, l2: list[t]):
        occ(x, append(l!1, l2)) = occ(x, l!1) + occ(x, l2)
  |-------
{1}   FORALL (x: t, l2: list[t]):
        occ(x, append(cons(x!1, l!1), l2)) = occ(x, cons(x!1, l!1))
```

As before we introduce fresh names and instantiate our induction hypothesis with them giving us

```
{-1}  occ(x!2, append(l!1, l2!1)) = occ(x!2, l!1) + occ(x!2, l2!1)
  |-------
[1]   occ(x!2, append(cons(x!1, l!1), l2!1)) =
       occ(x!2, cons(x!1, l!1)) + occ(x!2, l2!1)
```

Again, we find a number of occurrences of recursively defined functions which now admit a simplification:

```
 (expand "occ" :if-simplifies T)
 (expand "append" :if-simplifies T)
 (expand "occ" :if-simplifies T)
```

```
[-1]  occ(x!2, append(l!1, l2!1)) = occ(x!2, l!1) + occ(x!2, l2!1)
  |-------
{1}   IF x!2 = x!1
        THEN 1 + occ(x!2, append(l!1, l2!1))
      ELSE occ(x!2, append(l!1, l2!1))
      ENDIF
       =
       IF x!2 = x!1 THEN 1 + occ(x!2, l!1) ELSE occ(x!2, l!1) ENDIF
        occ(x!2, l2!1)
```

We can now directly replace the lhs of the induction hypothesis with its rhs and end up with an instance of reflexivity thus completing the proof. For the sake of the example let's move the if-then-else constructs to the surface and proceed by case distinction: the command `(lift-if)` followed by `(split)` brings us two subgoals

```
[-1]  occ(x!2, append(l!1, l2!1)) = occ(x!2, l!1) + occ(x!2, l2!1)
  |-------
{1}   x!2 = x!1 IMPLIES
       1 + occ(x!2, append(l!1, l2!1)) = 1 + occ(x!2, l!1) + occ(x!
```

and

```
[-1]  occ(x!2, append(l!1, l2!1)) = occ(x!2, l!1) + occ(x!2, l2!1)
  |-------
{1}   NOT x!2 = x!1 IMPLIES
       occ(x!2, append(l!1, l2!1)) = occ(x!2, l!1) + occ(x!2, l2!1)
```

corresponding to the two branches of the conditional. The first subgoal is first flattened and then dispatched with `(grind)` as it is an equational consequence of the induction hypothesis. The second even becomes a propositional axiom after flattening.

11.2.1 Completeness of filtering

Of a similar kind is

```
     filter_complete : THEOREM
        FORALL(x:t, l:list[t], p:[t->boolean]):
            p(x) IMPLIES occ(x,filter(p,l)) = occ(x,l)
```

Invoking induction on `l`, grinding away the base case, skolemising and instantiating brings us to

```
{-1}  p!1(x!2) IMPLIES occ(x!2, filter(p!1, l!1)) = occ(x!2, l!1)
[-2]  p!1(x!2)
  |-------
[1]   occ(x!2, filter(p!1, cons(x!1, l!1))) = occ(x!2, cons(x!1, l!
```

Simplifying the recursive function calls gives

```
[-1]  p!1(x!2) IMPLIES occ(x!2, filter(p!1, l!1)) = occ(x!2, l!1)
[-2]  p!1(x!2)
  |-------
{1}   occ(x!2,
          IF p!1(x!1)
            THEN cons(x!1, filter(p!1, l!1))
          ELSE filter(p!1, l!1)
          ENDIF)
       = IF x!2 = x!1 THEN 1 + occ(x!2, l!1) ELSE occ(x!2, l!1) ENDI
```

`(lift-if)` followed by `(split 1)` followed by `(flatten)` gives two subgoals the first of which is

```
{-1}  p!1(x!1)
[-2]  p!1(x!2) IMPLIES occ(x!2, filter(p!1, l!1)) = occ(x!2, l!1)
[-3]  p!1(x!2)
  |-------
{1}   occ(x!2, cons(x!1, filter(p!1, l!1))) =
       IF x!2 = x!1 THEN 1 + occ(x!2, l!1) ELSE occ(x!2, l!1) ENDIF
```

This gives now the opportunity for another simplification:

```
[-1]  p!1(x!1)
[-2]  p!1(x!2) IMPLIES occ(x!2, filter(p!1, l!1)) = occ(x!2, l!1)
[-3]  p!1(x!2)
  |-------
{1}   IF x!2 = x!1
        THEN 1 + occ(x!2, filter(p!1, l!1))
      ELSE occ(x!2, filter(p!1, l!1))
      ENDIF
       = IF x!2 = x!1 THEN 1 + occ(x!2, l!1) ELSE occ(x!2, l!1) ENDI
```

and again we must lift the conditionals and split: to deal with this we need to invoke `lift-if` again and split giving us

```
{-1}  x!2 = x!1
[-2]  p!1(x!1)
[-3]  p!1(x!2) IMPLIES occ(x!2, filter(p!1, l!1)) = occ(x!2, l!1)
[-4]  p!1(x!2)
```

```
  |-------
{1}   1 + occ(x!2, filter(p!1, l!1)) = 1 + occ(x!2, l!1)
```

Now we finally are in a position to use the induction hypothesis: `(split -3)` followed by `(replace -1 1)` dispatches this branch of the proof. The other ones are dealt with similarly. Of course at the displayed point and even before we could have used `(grind)` as well.

11.3 List reversal

Here is one way to reverse a list:

```
 rev(l:list[t]) : RECURSIVE list[t] =
     IF null?(l) THEN null
       ELSE append(rev(cdr(l)),cons(car(l),null))
     ENDIF
     MEASURE length(l)
```

This is considered inefficient because the recursive implementation of the append function takes linear time hence the overall runtime is quadratic.

Better is the following tail recursive formulation of reversal:

```
 rev1(l:list[t], acc:list[t]) : RECURSIVE list[t] =
    IF null?(l) THEN acc
      ELSE rev1(cdr(l),cons(car(l),acc))
    ENDIF
    MEASURE length(l)
```

The idea is that `rev1(l,acc)` equals the reversal of `l` followed by `acc`, so that we obtain the reversal of `l` as `rev1(l,null)`. Let's prove that this is true:

```
  rev_rev1 : THEOREM
     FORALL(l,acc:list[t]):
        rev1(l,acc) = append(rev(l),acc)
```

Induction on `l`, grinding away the base case, skolemising and flattening brings us to

```
[-1]  FORALL (acc: list[t]): rev1(l!1, acc) = append(rev(l!1), acc)
  |-------
{1}   rev1(cons(x!1, l!1), acc!1) = append(rev(cons(x!1, l!1)), acc!
```

Now in this case, it is wrong to instantiate the induction hypothesis with `acc!1` and that's the reason why `induct-and-simplify` doesn't work here. We just leave the induction hypothesis and simplify our recursive functions.

```
[-1]  FORALL (acc: list[t]): rev1(l!1, acc) = append(rev(l!1), acc)
  |-------
{1}   rev1(l!1, cons(x!1, acc!1)) =
       append(append(rev(l!1), cons(x!1, null)), acc!1)
```

Now we see that the lhs is in fact an instance of the lhs of the induction hypothesis, however with `acc` set to `cons(x!1,acc!1)`. This suggests to instantiate the induction hypothesis accordingly:

```
(inst -1 "cons(x!1,acc!1)")
```

```
{-1}  rev1(l!1, cons(x!1, acc!1)) = append(rev(l!1), cons(x!1, acc!1
  |-------
[1]   rev1(l!1, cons(x!1, acc!1)) =
       append(append(rev(l!1), cons(x!1, null)), acc!1)
```

If the induction hypothesis holds for all `acc`, then in particular for the one we just gave . . .

Now we rewrite with the induction hypothesis (`replace -1 1`) and get

```
[-1]  rev1(l!1, cons(x!1, acc!1)) = append(rev(l!1), cons(x!1, acc!1
  |-------
{1}   append(rev(l!1), cons(x!1, acc!1)) =
       append(append(rev(l!1), cons(x!1, null)), acc!1)
```

This looks like an instance of associativity of append albeit slightly hidden. We could now use `(lemma "append_assoc")` etc. but it is easier to tell PVS to consider it as a rewrite rule: `(auto-rewrite "append_assoc")` after which `(grind)` can complete the proof.

11.4 Summary

To prove theorems involving recursively defined functions we usually need induction. The first decision to make is which variable to induct on. This should be the one leading to the most simplifications of recursively defined functions.

Once this decision has been made one can always try `induct-and-simplify`. If that doesn't help then we follow the following steps:

1. Invoke induction with the `induct` command
2. Skolemize and flatten
3. Simplify instances of recursively defined functions using `(expand ... :if-simplifies T)`
4. make case distinctions on conditionals using `lift-if` and `split`
5. try to massage your goal so that the induction hypothesis becomes applicable
6. when the induction hypothesis is universally quantified you must decide on the right instantiation. Often but not always it consists of the constants obtained from skolemising the current goal.

When you get stuck:

- Try to figure out what your current goal really says and what it could be a consequence of.
- Try not to get into a "symbol-pushing mode" and just blindly enter commands
- At least sketch an informal proof before you attempt a proof with PVS
- If your current goal seems true, but you can't get PVS to prove it you may try to isolate it as a separate lemma, i.e., abandon the proof, type in the lemma, prove it separately (perhaps again using induction) and then retry.

12 General datatypes

Inductive datatypes other than lists can be defined in PVS using the datatype construct. Rather than going into formalities let's look at two concrete examples:

12.1 Labelled binary trees

If T is a set then the set of binary trees $\texttt{tree}(T)$ with labels in T is inductively defined as follows:

- `leaf` is a tree,

- if *label* $\in T$ and *left, right* $\in \texttt{tree}(T)$ then $\mathsf{node}(\textit{label}, \textit{left}, \textit{right}) \in \texttt{tree}(T)$.

E.g. $\texttt{node}(3, \texttt{node}(2, \texttt{leaf}, \texttt{leaf}), \texttt{leaf}) \in \texttt{tree}(\mathbb{N})$.

One may ask in what sense this prima facie self-referential explanation is at all a valid definition.

Some people just take it for granted, others prefer to explain it in terms of set theory. For instance, we can say that a tree is a finite prefix-closed set of paths, or we can define trees by induction on their depth level. Then the only tree of level 0 would be a leaf, encoded e.g. as the empty set; a tree of level $n+1$ is either a tree of level n or a triple $(\textit{label}, \textit{left}, \textit{right})$ where *label* $\in T$ and $\textit{left}, \textit{right}$ are trees of level n. We then *define* the constructor leaf as being the empty set and node as the function which groups three things into a triple.

Since trees are inductively generated, we have the following principle of tree induction:

Let P be a property of binary T-labelled trees.

If

- $P(\texttt{leaf})$ and
- whenever $P(\textit{left})$ and $P(\textit{right})$ for some $\textit{left}, \textit{right} \in \texttt{tree}(T)$ then also $P(\texttt{node}(\textit{label}, \textit{left}, \textit{right}))$ for all *label* $\in T$

then P holds for all binary T-labelled trees.

If we take inductive definitions for granted then we must also take this principle on board; if we define trees in terms of more primitive concepts then tree induction becomes a theorem, e.g., provable by course-of-values induction on the level.

Another principle is that nodes are different from leaves:

$$\texttt{node}(\textit{label}, \textit{left}, \textit{right}) \neq \texttt{leaf}$$

- $\texttt{depth}(\texttt{leaf}) = 0$
- $\texttt{depth}(\textit{label}, \textit{leaf}, \textit{right}) = \max(\texttt{depth}(\textit{left}), \texttt{depth}(\textit{right})) + 1$

To define functions on trees we need some measure on them; the most primitive such is the depth given as on the slide. Again, we can either take the depth for granted or define it as the least level containing the tree. In that case the above equation could be proved.

Once we've got the depth we can define other functions by well-founded recursion:

- $\texttt{no\_leaves}(\texttt{leaf}) = 1$
- $\texttt{no\_leaves}(\texttt{node}(\mathit{label}, \mathit{left}, \mathit{right})) = \texttt{no\_leaves}(\mathit{left}) + \texttt{no\_leaves}(\mathit{right})$
- $\texttt{no\_nodes}(\texttt{leaf}) = 0$
- $\texttt{no\_nodes}(\texttt{node}(\mathit{label}, \mathit{left}, \mathit{right})) = \texttt{no\_nodes}(\mathit{left}) + \texttt{no\_nodes}(\mathit{right}) + 1$

This was the pattern-matching form. In order to get the fixpoint form we need (partial) destructor functions:

Define $\texttt{leaf?(t)} \iff t = \texttt{leaf}$,
Define $\texttt{node?(t)} \iff \exists \mathit{label}, \mathit{left}, \mathit{right}. t = \texttt{node}(\mathit{label}, \mathit{left}, \mathit{right})$,
Define $(\texttt{leaf?}) = \{\texttt{leaf}\}$
Define $(\texttt{node?}) = \{t \,|\, \texttt{node?}(t)\}$.
Notice: $\forall t{:}\texttt{tree}(T).\texttt{leaf?}(t) \vee \texttt{node?}(t)$
We have

$$\begin{aligned}
&\texttt{label} : (\texttt{node?}) \to T\\
&\texttt{left} : (\texttt{node?}) \to \texttt{tree}(T)\\
&\texttt{right} : (\texttt{node?}) \to \texttt{tree}(T)
\end{aligned}$$

defined by

$$\begin{aligned}
&\texttt{label}(\texttt{node}(\mathit{label}, \mathit{left}, \mathit{right})) = \mathit{label}\\
&\texttt{left}(\texttt{node}(\mathit{label}, \mathit{left}, \mathit{right})) = \mathit{left}\\
&\texttt{right}(\texttt{node}(\mathit{label}, \mathit{left}, \mathit{right})) = \mathit{right}
\end{aligned}$$

$$\texttt{no\_leaves}(t) = \begin{cases} 1, \text{ if } \texttt{leaf?}(t) \\ \texttt{no\_leaves}(\texttt{left}(t)) + \texttt{no\_leaves}(\texttt{right}(t)), \text{ o/w} \end{cases}$$

$$\texttt{no\_nodes}(t) = \begin{cases} 0, \text{ if } \texttt{leaf?}(t) \\ \texttt{no\_nodes}(\texttt{left}(t)) + \texttt{no\_nodes}(\texttt{right}(t)) + 1, \text{ o/w} \end{cases}$$

Both recursive definitions are well-founded using the depth as measure.

$\forall t{:}\texttt{tree}(T).\texttt{no\_leaves}(t) = \texttt{no\_nodes}(t) + 1$.

Proof by tree induction:

- $\texttt{no\_leaves}(\texttt{leaf}) = 1 = 0 + 1 = \texttt{no\_nodes}(\texttt{leaf}) + 1$

- $\mathtt{no\_leaves}(\mathtt{node}(\mathit{label}, \mathit{left}, \mathit{right})) =$
  $\mathtt{no\_leaves}(\mathit{left}) + \mathtt{no\_leaves}(\mathit{right}) \overset{IH}{=}$
  $\mathtt{no\_nodes}(\mathit{left}) + 1 + \mathtt{no\_nodes}(\mathit{right}) + 1 =$
  $(\mathtt{no\_nodes}(\mathit{left}) + \mathtt{no\_nodes}(\mathit{right}) + 1) + 1 =$
  $\mathtt{no\_nodes}(\mathtt{node}(\mathit{label}, \mathit{left}, \mathit{right})) + 1$

```
tree[t:TYPE]: DATATYPE
 BEGIN
    leaf : leaf?
    node(label:t,left,right:tree) : node?
 END tree
```

To introduce trees into PVS we use the above declaration. It will automatically generate a file `tree_adt.pvs` when we type check the file containing the declaration. The contents of `tree_adt.pvs` are as follows:

```
tree_adt[t: TYPE]: THEORY
 BEGIN

  tree: TYPE
  leaf?, node?: [tree -> boolean]
  leaf: (leaf?)
  node: [[t, tree, tree] -> (node?)]
  label: [(node?) -> t]
  left: [(node?) -> tree]
  right: [(node?) -> tree]

  tree_label_node: AXIOM
    FORALL (node1_var: t, node2_var: tree, node3_var: tree):
      label(node(node1_var, node2_var, node3_var)) = node1_var;

  tree_left_node: AXIOM
    FORALL (node1_var: t, node2_var: tree, node3_var: tree):
      left(node(node1_var, node2_var, node3_var)) = node2_var;

  tree_right_node: AXIOM
    FORALL (node1_var: t, node2_var: tree, node3_var: tree):
      right(node(node1_var, node2_var, node3_var)) = node3_var;
  tree_inclusive: AXIOM
    FORALL (tree_var: tree): leaf?(tree_var) OR node?(tree_var);

  tree_induction: AXIOM
    FORALL (p: [tree -> boolean]):
      (p(leaf) AND
        (FORALL (node1_var: t, node2_var: tree, node3_var: tree):
           p(node2_var) AND p(node3_var) IMPLIES
            p(node(node1_var, node2_var, node3_var))))
       IMPLIES (FORALL (tree_var: tree): p(tree_var));
```

The file contains other useful stuff such as the definition of a subtree relation, a mapping functional, as well as properties describing these. Take a look yourselves!

Unfortunately, the depth isn't defined for us, so we do it ourselves using the `reduce_nat` functional which also works for trees with different typing, though. Can you work out the typing of `reduce_nat` from the example? If not take a look at `tree_adt.pvs` where it's defined.

```
tree_depth[t:TYPE]: THEORY
BEGIN
IMPORTING tree_adt

 depth: [tree[t] -> nat] =
       reduce_nat(0, LAMBDA(x:t,l,r:nat):max(l,r)+1)

END tree_depth

IMPORTING tree_adt
```

gives us the type former `tree`[], e.g. `tree`[`nat`] is the type of `nat` labelled trees.

`leaf[t], node[t], leaf?[t], node?[t]` etc.

can usually be abbreviated by

`leaf, node, leaf?, node?` etc.

With

```
IMPORTING tree_adt, tree_depth
```

we also get the `depth` function.

```
  |-------
{1}   FORALL (t: tree[t]): no_leaves(t) = no_nodes(t) + 1

Rule? (induct-and-simplify "t")
```

Alternatively, `induct "t"`, `skosimp`, etc.

12.2 The option datatype

We saw that partial functions can be turned into total functions by defining them on a subset. Another possibility is to change the result type of the function so as to contain a special "error element" which—when taken on—flags that we are outwith the domain of the function. An example: the predecessor function on natural numbers can be defined on the set $\{n \,|\, n > 0\}$, then returning a natural number. Alternatively, we can define it on the whole of $\mathbb{N}$ and then return a value in $\mathbb{N} \cup \{\texttt{none}\}$ with the understanding that $\texttt{pred}(0) = \texttt{none}$ and $\texttt{pred}(n) = n - 1$ otherwise.

Since this situation occurs sufficiently often, it is handy to have a new type former which tacks on a special element to any other type. Since this error element might already have been present it's more convenient to also flag the other elements which is achieved with the `option` datatype. If T is a set so is `option`(T) and its members are `none` and `some(x)` when $x \in T$.

A property holds for all elements in `option`(T) provided it holds for `none` and for all elements of the form `some`(x). That's the induction principle for the type `option`. We also have the subsets $(\texttt{none}?)$, $(\texttt{some}?)$ consisting of `none` and the `some`(x), respectively.

So, everything is as before, except that this time the constructors don't take arguments from the inductively defined set. In this case, the "induction principle" is equivalent to a first-order formula (no quantification over predicates). Do you see, which one?

```
option[t:TYPE]: DATATYPE
 BEGIN
    none : none?
    some(content:t) : some?
 END option
```

Again, a file `option_adt.pvs` is created; look at it and try to understand its contents.

13 Equational theories

In this and the following section we will encounter one of the decision procedures built into PVS: *Shostak's algorithm* which is a method for combining decidable equational theories with uninterpreted function symbols. A typical application would be to derive $0 = 1$, i.e., a contradiction, from the assumptions $f(x-1)-1 = x + 1, f(y) + 1 = y - 1, y + 1 = x$. The following example shows that even without an additional theory uninterpreted function symbols can be nontrivial: derive $f(x) = x$ from $f^5(x) = f^3(x) = x$.

In PVS uninterpreted function symbols arise for example from unexpanded (recursive) definitions and arrays (to be introduced later).

Shostak's algorithm requires that the equational theory under consideration be both *solvable* and *canonisable*. We'll define these later in detail; roughly speaking a theory is solvable if systems of equations can be solved for variables occurring in them; it is canonisable if each term can be brought into a normal form so that terms are equal if and only if they have identical normal forms. Linear arithmetic is a typical example of a theory that is both solvable and canonisable.

Shostak's algorithm operates on sequents of the form $\Theta \Longrightarrow E$, where Θ is a list of equations (the assumptions) and E is an equation, the goal. Basically, the algorithm works by successively merging subterms of the goal and of the assumptions. In the first example, we start by solving $f(x - 1) - 1 = x + 1$ for $f(x - 1)$ (here $f(x - 1)$ is temporarily treated as a variable) yielding $f(x - 1) = x+2$. In view of $x = y+1$ we now merge $f(x-1)$ and $f(y)$ yielding $x+2 = y-2$. This together with $y + 1 = x$ yields the desired contradiction.

When done naively such merging may lead to nontermination, for instance, given $f(v) = v, f(u) = u - 1$ the equality $u = v$ might lead to merging first u and v, then $f(u)$ and $f(v)$, then, in view of $u = v + 1$ merging v and $v + 1$, then $v + 1$ with $v + 2$ and so on.

Before, we can present and verify the algorithm in detail we need some background on equational theories.

13.1 Equational theories

Consider a first-order language $\mathcal{L}$ with function symbols only.

(Refl) $t = t \in \mathcal{T}$

(Sym) $t_1 = t_2 \in \mathcal{T}$ whenever $t_2 = t_1 \in \mathcal{T}$

(Trans) $t_1 = t_3 \in \mathcal{T}$ whenver $t_1 = t_2 \in \mathcal{T}$ and $t_2 = t_3 \in \mathcal{T}$.

(Cong) $f(u_1, \dots, u_\ell) = f(v_1, \dots, v_\ell) \in \mathcal{T}$ whenever $u_i = v_i \in \mathcal{T}$ for $i = 1, \dots, \ell$.

(Sub) $t_1[x := u] = t_2[x := u] \in \mathcal{T}$ whenever $t_1 = t_2 \in \mathcal{T}$.

Figure 26: Rules for equality

An *equation* is an expression of the form $\Gamma \rhd_{\mathcal{L}} t_1 = t_2 : \tau$ where $\Gamma \rhd_{\mathcal{L}} t_1 : \tau$ and $\Gamma \rhd_{\mathcal{L}} t_2 : \tau$. We usually abbreviate an equation by just $t_1 = t_2$. Given an interpretation $\mathcal{I}$ of the $\mathcal{L}$ we write $\mathcal{I} \models t_1 = t_2$ if for all valuations ρ one has $[\![t_1]\!]_{\rho,\mathcal{I}} = [\![t_2]\!]_{\rho,\mathcal{I}}$. We also write $\mathcal{I}, \rho \models t_1 = t_2$ to mean $[\![t_1]\!]_{\rho,\mathcal{I}} = [\![t_2]\!]_{\rho,\mathcal{I}}$.

An *equational theory*, theory for short, is a set $\mathcal{T}$ of equations closed under the rules in Figure 26. In the sequel, it is always understood that all occurring equations are well-typed.

Traditionally, equations contain free variables and depending on the context their quantification may differ. A single equation with free variables is usually understood universally quantified. However, in a sequent, as in the informal description of Shostak's algorithm above, the free variables are understood as being universally quantified outside of the sequent, as was the case with first-order clauses. In some cases, variables are even understood existentially quantified. One just has to be attentive and read carefully.

13.2 Universal model and Birkhoff's Theorem

Clearly, every interpretation induces a theory, written Th$(\mathcal{I})$ by $E \in \text{Th}(\mathcal{I}) \iff \mathcal{I} \models E$.

Surprisingly, the converse is also true: every theory is induced by some interpretation. To wit, given $\mathcal{T}$ we construct $\mathcal{I}$ as in Figure 27

Theorem: ("Birkhoff's Theorem[3])

Let $\mathcal{T}$ be an equational theory. There exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models E$ if and only if $E \in \mathcal{T}$, i.e., Th$(\mathcal{I}) = \mathcal{T}$.

[3] This is only one part of Birkhoff's contribution to the subject. The other gives a model-theoretic characterisation of equational theories.

$\mathcal{L}_\infty$: extension of $\mathcal{L}$ with infinitely many constants of each type.
$\llbracket \tau \rrbracket_{\mathcal{I}} = \{t \mid \emptyset \rhd t : \tau\}/\sim$
where $t_1 \sim t_2 \iff \Gamma \rhd t_1 = t_2 : \tau \in \mathcal{T}$ and Γ records all the constants from $\mathcal{L}_\infty \setminus \mathcal{L}$ occurring in t_1, t_2 (viewing them as variables). This is an equivalence relation by rules (Refl), (Sym), (Trans).
Function symbols interpret themselves. This is well-defined on equivalence classes by rule (Cong).

Figure 27: The universal model of an equational theory $\mathcal{T}$

For the "if" direction assume that equation $t_1 = t_2$ is in E. Let ρ be a valuation in $\mathcal{I}$. Structural induction shows that $\llbracket t_i \rrbracket_{\mathcal{I},\rho} \equiv t_i[x_1 := \rho(x_1), x_n := \rho(x_n)]$ where the x_j are the variables occurring in t_i. Now by rule (Sub) we have $t_1[x_1 := \rho(x_1), \ldots, x_n := \rho(x_n)] = t_1[x_1 := \rho(x_1), \ldots, x_n := \rho(x_n)] \in E$, thus $\llbracket t_1 \rrbracket_{\mathcal{I},\rho} = \llbracket t_1 \rrbracket_{\mathcal{I},\rho}$.

Conversely, assume that $\mathcal{I} \models t_1 = t_2$. Let ρ be the valuation that assigns to each variable x in t_1, t_2 a unique constant c_x. The definition of $\mathcal{I}$ then shows that $t_1[x_1 := c_{x_1}, \ldots, x_n := c_{x_n}] = t_2[x_1 := c_{x_1}, \ldots, x_n := c_{x_n}] \in E$ with the c_x viewed as variables. Rule (Sub) then shows $t_1 = t_2 \in E$. □

The model $\mathcal{I}$ with $\mathcal{T} = \mathrm{Th}(\mathcal{T})$ is called the *unviersal model* of $\mathcal{T}$.

This should be contrasted with the situation in first-order logic. A theory can be defined as a set of formulas closed under the proof rules of sequent rules and not consisting of all sequents. By the completeness result for first-order logic every such theory has a model, i.e., there is an interpretation validating all the sequents of the theory. However, it is not necessarily the case that any sequent validated by this interpretation would have to be included in the theory. This is the case only for *complete* theories which have the additional property that for each formula either the formula itself or its negation is contained in the theory.

13.3 Application of universal models

The existence of universal models has the consequence that any kind of mathematical reasoning is conservative over equational reasoning. In more detail this means the following. Suppose we are given a set Γ of universally quantified equations, for example the axioms of a ring or a group and suppose that in any interpretation of Γ a certain equation E holds. Then it is possible to derive this equation E from the equations in Γ using the proof rules of equality alone.

Theorem: Let Γ be a set of equations and E be an equation. If $\mathcal{I} \models \Gamma$ implies $\mathcal{I} \models E$ for every interpretation $\mathcal{I}$ then there is a derivation of E from the equations in Γ.

To prove this we close up Γ under the rules for equality and take for $\mathcal{I}$ the universal model of the theory thus obtained. The fact that $\mathcal{I} \models E$ says that E is contained in the theory generated by Γ hence E is derivable from Γ.

A prominent example is commutativity of nilpotent rings. A ring R is nilpotent if for any $x \in R$ there exists $n \in \mathbb{N}$ such that $x^n = x$. A famous theorem due to Jacobson asserts that any nilpotent ring is commutative. This result is proved using abstract concepts such as maximal ideals.

While nilpotent rings are not axiomatisable by universally quantified equations (not even by a first order formulas) for each particular n we can consider the formula $x^n = x$, e.g., $xx = x$ or $xxx = x$. According to our general result there must therefore be a purely equational proof of $xy = yx$ from the ring axioms and $x^n = x$ for each concrete n. For $n = 2$ you have constructed such a proof in the exercises. For $n = 3$ such a proof can be given as follows.

$$xy = e_1xy + e_2xy + e_3xy = e_2y - e_3y = e_1yx + e_2yx + e_3yx = yx$$

where $e_1 = 1 - u, e_2 = u(1 - v), e_3 = uv, u = x^2, v = (1 - x)^2$.

At the time of writing one does not seem to have a general method for effectively constructing such proofs for an arbitrary n.

We should remark that this kind of conservation result also holds for first-order logic; if a formula ϕ holds in all first-order models of some first-order theory $\mathcal{T}$ then it is derivable from a finite-subset of $\mathcal{T}$ by the rules of first-order sequent calculus. However, this theorem is of less use since it is in general difficult to show that something is true in all first-order models.

Another consequence of the universal model is the following:

13.4 Convexity

Theorem: Let $\mathcal{T}$ be an equational theory and let $s_1 = t_1, \dots, s_n = t_n\}$ be a list of equations. Suppose that for all models $\mathcal{I}$ of $\mathcal{T}$ the following holds: for every valuation ρ one has $[\![s_i]\!]_{\rho,\mathcal{I}} = [\![t_i]\!]_{\rho,\mathcal{I}}$ for some i (possibly depending on ρ!). Then $\mathcal{T}$ entails $s_i = t_i$ for some i.

In other words, if $\mathcal{T}$ entails a disjunction of equations then it must already entail one of them.

Proof: We consider the universal model of $\mathcal{I}$ and the valuation given by $\rho(x) = c_x$. By assumption, $[\![s_i]\!]_{\rho,\mathcal{I}} = [\![t_i]\!]_{\rho,\mathcal{I}}$ for some i, but then $s_i = t_i \in \mathcal{T}$ by the construction of the universal model, so $\mathcal{T} \models s_i = t_i$. □

We note that this is a rather remarkable property of equationally defined theories. For example in the first-order theory of the booleans one has $x = \mathtt{tt} \vee x = \mathtt{ff}$, but none of the two disjuncts is valid on its own. In a similar vein, in the theory of fields one has $xy = 0 \Rightarrow x = 0 \vee y = 0$ but neither $xy = 0 \Rightarrow x = 0$ nor $xy = 0 \Rightarrow y = 0$.

This property is called *convexity of equational theories*. More generally, any first-order theory $\mathcal{T}$ is convex if whenever $\mathcal{T} \models \forall\vec{x}{:}\vec{\tau}.T \Rightarrow s_1{=}t_1 \vee \cdots \vee s_n{=}t_n$ where T is a conjunction of equations then there exists i such that $\mathcal{T} \models \forall\vec{x}{:}\vec{\tau}.T \Rightarrow s_i{=}t_i$. (Recall that $\mathcal{T} \models \phi$ means that all interpretations that validate $\mathcal{I}$ also validate ϕ.)

Many of the subsequent results can be generalised to arbitrary convex theories.

13.5 Decidability by normal forms

Given a theory $\mathcal{T}$ we are interested in determining whether $E \in \mathcal{T}$. If $\mathcal{T}$ is generated by a finite set of equations then the problem $E \in \mathcal{T}$ is recursively enumerable; simply enumerate derivations until (perhaps) you find one of E. It is not in general decidable, though. Well-known counterexamples are theories of certain semigroups.

Of course in computer-aided formal reasoning one is interested in decidable equational theories. One popular method to decide whether $t_1 = t_2$ in some theory is by bringing both t_1 and t_2 to some canonical form and then seeing whether the canonical forms are identical.

13.5.1 Canonisable theories

More precisely, a *canoniser* for some theory $\mathcal{T}$ is a function σ mapping terms (with variables) to terms (with variables) such that the following properties hold:

Let σ : Terms → Terms.

C1 $\sigma(t) = t \in \mathcal{T}$

C2 $t_1 = t_2 \in T$ implies $\sigma(t_1) \equiv \sigma(t_2)$

Clearly, if $\sigma(t_1) \equiv \sigma(t_2)$ then also $t_1 = t_2 \in \mathcal{T}$ by C1 and transitivity so that an algorithm for computing σ gives us a decision procedure for T.

A term t with $\sigma(t) \equiv t$ is *canonical*; since $\sigma(t)$ is always canonical (by C1 and C2) we call $\sigma(t)$ the canonical form of t.

Here are some examples of canonisable theories.

- Theory of rings (canonical forms: ordered sums of monomials (monomial=product of variables with integer coefficient)).
- Theory of linear arithmetic (unary function symbols for multiplication with scalars, normal forms=ordered sums of first-order monomials).
- Theory of pairs and projections: equations are, e.g., $\langle x, y\rangle.1 = x$, $x = \langle x.1, x.2\rangle \in \tau_1 \times \tau_2$.
- Theory of booleans and if-then-else
- Theory of cons, car, cdr.

13.5.2 Semantic canonisation

An interesting method for obtaining canonical forms and especially for proving properties C1,C2 goes via an interpretation of the theory.

Let T be an equational theory. As in the construction of the universal model we extend the language with an infinite supply of constants of every type and write Terms(τ) for the set of closed terms of type τ possibly involving these constants. We also write Const(τ) for the set of newly introduced constants of type τ.

Assume an interpretation $\mathcal{I}$ together with the following data and properties:

- for each type τ a function $q_\tau : [\![\tau]\!]_{\mathcal{I}} \rightarrow$ Terms(τ)
- for each type τ a function u_τ : Const$(\tau) \rightarrow [\![\tau]\!]_{\mathcal{I}}$

N1 $q([\![f]\!](v_1, \dots, v_n)) = f(q(v_1), \dots, q(v_n)) \in T$ for each function symbol f

N2 $q(u(c)) = c$ for each new constant c

Then $\sigma(t(x_1, \dots, x_n)) := q([\![t]\!]_\rho)[c_{x_i} := x_i]$ where $\rho(x_i) = c_{x_i}$ is a canoniser.

Property C2 follows since $\sigma(t)$ depends only on $[\![t]\!]$. Property C1 follows from N1 and N2 by structural induction on t.

13.5.3 Theory of monoids

For example, to decide the theory of monoids, i.e., T is generated by the equations $x(yz) = (xy)z$ and $ex = x$ and $xe = x$ we can interpret the single type D by the set of those functions f : Terms$(D) \to$ Terms(D) for which there exists a term t with $f(u) = tu \in T$. Putting $[\![\cdot]\!](u,v) = u \circ v$, $[\![e]\!] = \mathrm{id}$, $q(f) = f(e)$, $u(c) = \lambda t.ct$ then has all the required properties. Another example is the theory of rings where we construct an interpretation from ordered sums of monomials with appropriately defined multiplication and addition.

13.5.4 Theory of pairs and projections

Finally, let us consider the theory of pairs and projections. We start from a set $\mathcal{B}$ of base types; every base type is a type and if τ_1, τ_2 are types so is $\tau_1 \times \tau_2$.

We have mixfix function symbols $\langle -, - \rangle : [\tau_1, \tau_2 \to \tau_1 \times \tau_2]$ and postfix function symbols $.1 : [\tau_1 \times \tau_2 \to \tau_1]$ and $.2 : [\tau_1 \times \tau_2 \to \tau_2]$. The theory of pairs and projections is generated by the three equations $\langle x_1, x_2 \rangle.1 = x_1$ and $\langle x_1, x_2 \rangle.2 = x_2$ and $x = \langle x.1, x.2 \rangle$.

As an auxiliary concept we define *neutral terms* as sequences of projections applied to a constant (including the newly added ones) $c.2.2.2.1.1.2.1$.

We interpret a basic type β by

$$[\![\beta]\!] = \{\text{neutral terms of type } \beta\}, \text{ when } \beta \in \mathcal{B}$$

We interpret composite types as usual by cartesian products:

$$[\![\tau_1 \times \tau_2]\!] = [\![\tau_1]\!] \times [\![\tau_2]\!]$$

The functions q and u are then defined as follows:

$$q_\beta(t) = t \qquad u_\beta(c) = c$$

$$q_{\tau_1 \times \tau_2}((x,y)) = \langle q_{\tau_1}(x), q_{\tau_2}(y) \rangle \qquad u_{\tau_1 \times \tau_2}(c) = (u_{\tau_1}(c.1), u_{\tau_2}(c.2))$$

Here, in order for the definition to make sense, we must extend u to arbitrary neutral terms, not just the constants.

13.6 Deciding theories by term rewriting

Another approach for obtaining and verifying a canoniser is via term rewriting. Suppose we have a relation $\to$ between terms such that

$\mathcal{I} \models \{u_1 = v_1, \ldots, u_\ell = v_\ell\} \Rightarrow s = t$ means $\forall\rho.(\bigwedge_i \llbracket u_i \rrbracket_{\mathcal{I},\rho} = \llbracket v_i \rrbracket_{\mathcal{I},\rho}) \Rightarrow \llbracket s \rrbracket_{\mathcal{I},\rho} = \llbracket t \rrbracket_{\mathcal{I},\rho}$

- $x = y \Rightarrow f(x) = f(y)$
- $f^5(x) = x, f^3(x) = x \Rightarrow f(x) = x$
- $x + 1 = y + x \Rightarrow y = 1$
- $x^2y^2 = x, x^2 = xy \Rightarrow x^5 = x$

Figure 28: Implications between equations

- $\mathcal{T} \models t_1 = t_2$ if and only if $t_1(\rightarrow \cup \rightarrow^{-1})^* t_2$
- $\rightarrow$ is confluent, i.e., if $t \rightarrow^* t_1$ and $t \rightarrow^* t_2$ then $t_1 \rightarrow^* t_3$ and $t_2 \rightarrow^* t_3$ for some t_3,
- $\rightarrow$ is weakly normalising, i.e., for each t there exists t_1 such that $t \rightarrow^* t_1$ and there does not exists t_2 with $t_1 \rightarrow t_2$. Such t_1 is called a *normal form* of t

Then $\sigma(t) :=$ “some normal form of t” is a canoniser. To see this just note that by confluence normal forms are unique.

13.7 Implications between equations

Canonisation is good for deciding the question whether a given equality holds. More difficult is the question whether a (finite) set of equations implies another equation (under the same interpretation of the variables) and more generally arbitrary combinations of equalities by propositional connectives.

Let T be a set of equations and $s = t$ an equation over a common set of variables. If $\mathcal{I}$ is an interpretation let us write $\mathcal{I} \models T \Rightarrow s = t$ to mean that for all valuations ρ such that $\llbracket u \rrbracket_{\mathcal{I},\rho} = \llbracket v \rrbracket_{\mathcal{I},\rho}$ for all equations $u = v$ in T one also has $\llbracket s \rrbracket_{\mathcal{I},\rho} = \llbracket t \rrbracket_{\mathcal{I},\rho}$.

For instance, one always has things like $x = y \Rightarrow f(x) = f(y)$ or indeed $f^5(x) = x, f^3(x) = x \Rightarrow f(x) = x$. More interestingly, in a model of linear arithmetic we have $x + 1 = y + x \Rightarrow y = 1$ and in a commutative ring we have $x^2y^2 = x, x^2 = xy \Rightarrow x^4 = x$.

Let t be a monoid term involving the variables a, b, c, d, f. For given term t it is undecidable whether

$$\begin{array}{ll} ac = ca, ad = da, bc = cb, bd = db, fca = cf, fdb = df, & \\ cdca = cdcaf, caaa = aaa, daaa = aaa & \Rightarrow t = aaa \end{array}$$

Figure 29: Undecidable word problem

We write $\mathcal{T} \models T \Rightarrow E$ to mean that $\mathcal{I} \models T \Rightarrow E$ for all models $\mathcal{I}$ of $\mathcal{T}$.

To see that implication between equations is substantially more difficult consider the theory of monoids for which we have canonisation and which is thus decidable. Implication between monoid equations on the other hand is in general undecidable, see Figure 13.7.

13.8 Universal models and implications

We can strengthen the property of universal models to implications as follows:

Corollary (to Birkhoff's Theorem): Let T be a finite set of equations (possibly involving variables) and $\mathcal{T}$ an equational theory. There exists an interpretation $\mathcal{I}$ of $\mathcal{T}$ and a valuation ρ of the variables in T such that for any equation E one has $\mathcal{I} \models T \Rightarrow E$ if and only if $\mathcal{I}, \rho \models E$.

To prove this, we view the variables in T as constants and construct the universal model of $\mathcal{T}$ augmented with the equations (now between closed terms) in T. Now, ρ is obtained by assigning to each variable the meaning of the corresponding constant in this model. □

It turns out that convexity is equivalent to the existence of universal models in this extended sense.

Theorem: An arbitrary first-order theory $\mathcal{T}$ (set of first-order formulas closed under entailment) is convex if and only if for any finite set of equations T there exists a model $\mathcal{I}$ of $\mathcal{T}$ and a valuation ρ of the variables in T such that for any equation E one has $\mathcal{T} \models \forall\vec{x}. \bigwedge T \Rightarrow E$ if and only if $\mathcal{I}, \rho \models E$.

The "only if" direction is easy: if $\mathcal{T} \models \forall\vec{x}. \bigwedge T \Rightarrow \bigvee_i s_i{=}t_i$ then there must be an i such that $\mathcal{I}, \rho \models s_i{=}t_i$ and hence $\mathcal{T} \models \forall\vec{x}. \bigwedge T \Rightarrow s_i{=}t_i$.

For the converse, we consider the variables in T as constants and form the following set of closed formulas over the thus extended language:

$$M = \mathcal{T} \cup T \cup \{\neg s{=}t \mid \mathcal{T} \not\models \forall\vec{x}. \bigwedge T \Rightarrow s{=}t\}$$

We claim that any finite subset of M and hence (by compactness) M itself has a model. Such a finite subset can only involve finitely many of the disequations $\neg s{=}t$ where $\mathcal{T} \not\models \forall\vec{x}. \bigwedge T \Rightarrow s{=}t$. Enumerate these as $\neg s_i{=}t_i$ for $i = 1, \dots n$.

If every model $\mathcal{I}$ of $\mathcal{T}$ satisfies $\forall\vec{x}. \bigwedge T \Rightarrow \bigvee_{i=1}^{n} s_i{=}t_i$ then by convexity, there exists i with $\mathcal{T} \models \forall\vec{x}. \bigwedge T \Rightarrow s_i{=}t_i$ contradicting the definition of the set M. So, there must exist a model $\mathcal{I}$ and a valuation ρ such that $\mathcal{I}, \rho \models T$, and $\mathcal{I}, \rho \models \neg s_i{=}t_i$ for $i = 1 \dots n$. But this means precisely that the chosen finite subset is consistent. Thus, by compactness, we have a model $\mathcal{I}$ and a valuation ρ satisfying all of M. In particular, $\mathcal{I}$ itself is a model of $\mathcal{T}$. We claim that $\mathcal{I}, \rho$ has the required properties: indeed, if $\mathcal{T} \models \forall\vec{x}. \bigwedge T \Rightarrow s{=}t$ then, since $\mathcal{I}, \rho \models \bigwedge T$, we also have $\mathcal{I}, \rho \models E$. If, on the other hand, $\mathcal{I}, \rho \models E$ then we must have $\mathcal{T} \models \forall\vec{x} T \Rightarrow E$ for otherwise the negation $\neg E$ would have been contained in M and thus validated by $\mathcal{I}, \rho$. □

13.9 Solvable Theories

There is an important class of theories for which implication between equations is decidable; the solvable theories.

By definition a theory $\mathcal{T}$, not necessarily equational, is *solvable* if there is an algorithm *Solve* that for an equation $E \equiv u = v$ returns either $\perp$ or produces a set of equations *Solve*(E) such that $\perp$ is returned if and only if $\mathcal{T} \models \mathcal{T} \Rightarrow x{=}y$ for fresh variables x, y, hence $\mathcal{T} \models \mathcal{T} \Rightarrow E'$ for all equations E', and otherwise

- *Solve*(E) is of the form $\{x_1 = t_1, \dots, x_n = t_n\}$ where $x_1, \dots, x_n$ are among the variables of E
- none of the x_i occurs in any of the t_j;
- for each valuation ρ the following are eqivalent:
  - $[\![u]\!]_{\mathcal{I},\rho} = [\![v]\!]_{\mathcal{I},\rho}$
  - there is a valuation ρ' extending ρ (to the variables in *Solve*(E)) such that $\rho'(x) = [\![t]\!]_{\mathcal{I},\rho'}$ for each $x = t$ in *Solve*(E)

In other words, the formula $\forall\vec{x}.u = v \Leftrightarrow \exists\vec{y}. \bigwedge \mathit{Solve}(u = v)$ is valid in all models of $\mathcal{T}$.

Notice that $\mathit{Solve}(E)$ may contain more variables than E; these play the role of parameters. For example, in a suitable language a solution for $T = \{x^2 + y^2 = 1\}$ could be $\{x = \cos(\phi), y = \sin(\phi)\}$.

More elementarily, in the theory of linear arithmetic over $\mathbb{Q}$ a solution of $\{2x+3y + z = 1\}$ would be $\{x = \frac{1}{2}(a - b), y = \frac{1}{3}b, z = 1 - a\}$.

Examples of solvable theories are as follows:

- Linear arithmetic:

$$\mathit{Solve}(2x + 3y + z = 1) = \{x = \frac{1}{2}(a - b), y = \frac{1}{3}b, z = 1 - a\}$$

- Pairs and projections:

$$\mathit{Solve}(x.1 = y.2) = \{x = \langle a, b\rangle, y = \langle c, a\rangle\}$$

- Booleans with IF-THEN-ELSE

$$\begin{aligned}&\mathit{Solve}(\text{IF}\,x\text{THEN}\,y\text{ELSE}\,z = x) = \\ &\quad\{x = a, y = \text{IF}\,a\,\text{THEN}\,a\text{ELSE}\,b, z = \text{IF}\,a\,\text{THEN}\,c\text{ELSE}\,a\}\end{aligned}$$

- List with CONS, CAR, CDR. (Shostak, rather hairy)

13.9.1 Solving and implications

Theorem: Let ϕ be any first-order formula (over $\mathcal{L}$), $u = v$ an equation, and $\mathcal{I}$ a model of some solvable theory $\mathcal{T}$. One has $\mathcal{I} \models \forall\vec{x}.u = v \Rightarrow \phi$ if and only if either $\mathit{Solve}(u = v) = \bot$ or $\mathcal{I} \models \forall\vec{x}, \vec{y}.\phi[x_1 := t_1, \ldots, x_n := t_n]$ when $\mathit{Solve}(u = v)$ is $\{x_1 = t_1, \ldots, x_n = t_n\}$ and the y_i are the newly introduced variables.

Application: Decide $T \Rightarrow E$ by successively solving the equations in T and plugging in.

Examples of solvable theories are linear arithmetic and the theory of pairs and projections. In the latter case a solution of $x.1 = y.2$ for example would be $x = \langle a, b\rangle, y = \langle c, a\rangle$. In the theory of commutative rings implication between equations is also decidable (using Gröbner bases) but this theory is clearly not solvable. For instance, the equation $x^2 + y^2 = 1$ has no solution yet is satisfiable in the model $\mathbb{R}$.

Proof of Theorem. "$\Rightarrow$": Assume $\mathcal{I} \models u = v \Rightarrow \phi$ and *Solve*$(u = v) = \{x_1 = t_1, \dots, x_n = t_n\}$. Suppose furthermore that we are given a valuation ρ_0 of the variables $\vec{x}, \vec{y}$ contained in $\phi[x_1 := t_1, \dots, x_n := t_n]$. We let ρ be the valuation defined by $\rho(x_i) = [\![t_i]\!]_{\mathcal{I},\rho_0}$ and $\rho(x) = \rho_0(x)$ otherwise.

Clearly, ρ validates all the equations $x_i = t_i$ (here we need that the x_i do not occur in the t_i, thus $[\![t_i]\!]_{\mathcal{I},\rho} = [\![t_i]\!]_{\mathcal{I},\rho_0}$.)

Therefore, by the assumptions on *Solve*$(-)$, the valuation ρ validates $u = v$. By assumption, it then satisfies ϕ, but by definition of ρ this means that ρ_0 validates $\phi[x_1 := t_1, \dots, x_n := t_n]$.

"$\Leftarrow$": If *Solve*$(u = v) = \bot$ we know by assumption that no model can satisfy $u = v$, so $u = v \Rightarrow \phi$ is trivially true in any model. So assume that *Solve*$(u = v) \equiv \{x_1 = t_1, \dots, x_n = t_n\}$. After appropriate renaming we may assume that the variables contained in ϕ but not in $u = v$ are different from those newly introduced in *Solve*$(u = v)$. Suppose that $\mathcal{I} \models \phi[x_1 := t_1, \dots, x_n := t_n]$ and that we are given a valuation ρ validating $u = v$. We can thus find an extension ρ' of ρ validating the equations in *Solve*$(u = v)$. By assumption ρ' (as indeed any valuation) validates $\phi[x_1 := t_1, \dots, x_n := t_n]$ hence ϕ since $\rho'(x_i) = [\![t_i]\!]_{\mathcal{I},\rho'}$.

□

13.10 Boolean combinations of equations

We can use the preceding results to decide validity of arbitrary propositional combinations of equations over some canonisable and solvable equational theory: Given a propositional combination ϕ of equations, in other words a universally quantified formula of first-order logic with equality, we first transform it into a conjunction of clauses. Clearly, a conjunction of clauses is valid if each individual clause is valid (contrast this with unsatisfiability which is the standard application of clauses); so, it suffices to produce a method for deciding validity of clauses over equations.

To decide validity of a clause C we proceed as follows. If C has positive literals only then it is a disjunction of equations and by convexity we know that it is valid if and only if one of its literals is valid. So we can use our canoniser to check each equation in turn for validity.

If C has at least one negative literal then it is of the form $u = v \Rightarrow C'$ where C' has one negative literal less. If *Solve*$(u = v) = \bot$ then clearly our clause is trivially valid; if *Solve*$(u = v) = \{x_i = t_i \mid i = 1, \dots, n\}$ then C is equivalent to

$C'[x_1 := t_1, \ldots, x_n := t_n]$ which still has one negative literal less than C so can be decided recursively.

13.11 Uninterpreted function symbols

Let $\mathcal{L}$ be a language, $\mathcal{T}$ and equational theory and $\mathcal{I}$ be a model of $\mathcal{T}$. Let $\mathcal{L}^+$ be the extension of $\mathcal{L}$ with some function symbols which will be referred to as uninterpreted function symbols.

If $T \Rightarrow E$ is an implication between equations over $\mathcal{L}^+$, i.e., possibly involving the uninterpreted function symbols we write $\mathcal{I} \models T \Rightarrow E$ to mean that $\mathcal{I}^+ \models T \Rightarrow E$ for all interpretations $\mathcal{I}^+$ of $\mathcal{L}^+$ that agree with $\mathcal{I}$ on $\mathcal{L}$. Examples for such interpreted function symbols could be functional parameters to a procedure, e.g., an array input to a sorting procedure. Sometimes, uninterpreted function symbols do actually have some intended interpretation, which, however, one does not want to expose to the decision procedure for equality.

Now deciding implication between equations with uninterpreted function symbols or indeed any propositional combination of such equations can be reduced to the function-free case using the following method due to Ackermann (the inventor of the eponymous function).

13.12 Ackermann's method

If ϕ is a first-order formula over $\mathcal{L}^+$ choose a variable x_t for each subterm t of ϕ that starts with an uninterpreted function symbol. Define a formula ϕ^* over $\mathcal{L}$ by replacing each such subterm with its associated variable. Then ϕ is equivalent to $\phi^* \wedge$ CONG where CONG is the conjunction of all formulas of the form $\vec{a}^* = \vec{b}^* \Rightarrow x_{f(\vec{a})} = x_{f(\vec{b})}$. We can then decide validity of that latter formula using the abovedescribed procedure. A drawback is that the size of formula CONG is quadratic (in the worst case) in the size of ϕ possibly resulting in an inefficiency.

Example: We want to decide $f(x-1)-1 = x+1, f(y)+1 = y-1, y+1 = x \Rightarrow 0 = 1$.

The variables for the subterms are: $a := f(x-1), b := f(y)$

The congruence formula is CONG : $x - 1 = y \Rightarrow a = b$.

The original goal is thus equivalent to: $($CONG $\wedge\ a - 1 = x + 1 \wedge b + 1 =$

$y - 1 \wedge y + 1 = x) \Rightarrow 0 = 1$. In clausal form (cf. $\Rightarrow$-L) this becomes

$$a - 1 = x + 1 \wedge b + 1 = y - 1 \wedge y + 1 = x \Rightarrow x - 1 = y \vee 0 = 1$$
$$a - 1 = x + 1 \wedge b + 1 = y - 1 \wedge y + 1 = x \wedge a = b \Rightarrow 0 = 1 \vee y = x - 1$$

Solving the common premises yields $a = p + 2, b = p - 3, y = p - 1, x = p$, thus $x - 1 = y$ becomes $p - 1 = p - 1$ which is true so the first clause is valid.

Likewise, the second clause becomes $p + 2 = p - 3 \Rightarrow 0 = 1$ which is valid since *Solve*$(p + 2 = p - 3) = \perp$.

In this example the method works fine since there are comparatively few subterms involving uninterpreted function symbols.

On the other hand, consider the example $f^5(x) = x, f^3(x) = x \Rightarrow f(x) = x$. We introduce variables for the subterms as follows: $a := f^5(x), b := f^4(x), c := f^3(x), d := f^2(x), e := f(x)$.

The formula CONG now is

$$\begin{aligned}
&b = c \Rightarrow a = b \wedge b = d \Rightarrow a = c \wedge b = e \Rightarrow a = d \wedge b = x \Rightarrow a = e \wedge \\
&c = d \Rightarrow b = c \wedge c = e \Rightarrow b = d \wedge c = x \Rightarrow b = e \wedge \\
&d = e \Rightarrow c = d \wedge d = x \Rightarrow c = e \wedge \\
&e = x \Rightarrow d = e
\end{aligned}$$

Completing the example is left as an (unpleasant) exercise.

13.13 A version of the Nelson-Oppen procedure

We present in this section a more efficient method for deciding equations with undetermined function symbols described by Harald Ganzinger (*Shostak light*, in Proc. LICS 2003, IEEE Press, 2003) and based on an earlier method due to Nelson and Oppen which we will not describe. We call this method $\mathcal{NO}$.

As before, we let $\mathcal{L}$ be a language, $\mathcal{T}$ be a *convex* theory, e.g. an equational theory, over $\mathcal{L}$ and let $\mathcal{L}^+$ be the extension of $\mathcal{L}$ with uninterpreted function symbols.

A *sequent* is of the form $\Gamma \Longrightarrow E$ where Γ and $\{E\}$ are sets of equations over $\mathcal{L}^+$.

The meaning of such a sequent is defined as before, in particular we write $\mathcal{T} \models \Gamma \Longrightarrow E$ to mean that $\Gamma \Longrightarrow E$ is true in all extensions of models of $\mathcal{T}$. We assume that we can decide whether $\mathcal{T} \models \Gamma \Longrightarrow E$ in the case where Γ, E do not contain any of the uninterpreted function symbols using a solver for $\mathcal{T}$ or indeed any other method. Our goal is to extend this to a procedure for deciding validity of arbitrary sequents in $\mathcal{T}$.

$$\frac{\mathcal{T} \models \bar{\Gamma} \Longrightarrow E}{\Gamma \Longrightarrow E} \qquad \text{(NO-AXIOM)}$$

$$\frac{\Gamma, u = v, f(\vec{s}) = u \Longrightarrow E \qquad \mathcal{T} \models \bar{\Gamma} \Longrightarrow s_i = t_i \text{ for } i = 1 \ldots n.}{\Gamma, f(\vec{s}) = u, f(\vec{t}) = v \Longrightarrow E}$$

(NO-COMPOSE)

Figure 30: Inference rules of $\mathcal{NO}$

A *configuration* is a sequent $\Gamma \Longrightarrow E$, where E is over $\mathcal{L}$ and where Γ contains equations over $\mathcal{L}$ and, additionally, equations of the form $f(\vec{s}) = u$ where, again $\vec{s}, u$ are from $\mathcal{L}$, but f is one of the undefined function symbols.

Exercise: Show that for any sequent one can find an equivalent configuration.

If Γ is as in the above definition then we write $\bar{\Gamma}$ for the set of equations from $\mathcal{L}$ contained in Γ, i.e., the function definitions are removed.

$\mathcal{NO}$ consists of one axiom and one inference rule for configurations which are such that a sequent is valid iff it is derivable. The inference rules are given in Figure 13.13

13.14 Soundness and completeness of $\mathcal{NO}$

For the sake of simplicity, we assume that $\mathcal{T}$ is an equational theory for which universal models are available. It is possible to extend the argument to the more general case of arbitrary convex theories.

Clearly, $\mathcal{NO}$ derives valid sequents only.

For the converse we study the backwards applicability of $\mathcal{NO}$-rules to valid configurations. We show the following:

1. Every sequence of backwards applications of $\mathcal{NO}$-rules must terminate.

2. If an $\mathcal{NO}$-rule derives $\Gamma \Longrightarrow E$ from $\Gamma_0 \Longrightarrow E_0$ then $\Gamma \Longrightarrow E$ is valid if and only if $\Gamma_0 \Longrightarrow E_0$ is valid. $\mathcal{NO}$preserves and reflects validity.

3. If $\Gamma \Longrightarrow E$ is a valid configuration then it is the conclusion of an $\mathcal{NO}$-rule.

This shows that for any valid sequent a proof can be obtained by backwards application of $\mathcal{NO}$-rules in any fashion.

(1) follows from the fact that the number of definitions decreases upon invocation of a rule. (2) is immediate by equational reasoning. For (3) we argue as follows: Consider the universal model $\mathcal{I}$ and valuation ρ such that $\mathcal{T} \models \bar{\Gamma} \Rightarrow E$ iff $\mathcal{I}, \rho \models u{=}v$ for all equations $u{=}v$. If $\mathcal{I}, \rho \models E$ then $\mathcal{T} \models \bar{\Gamma} \Rightarrow E$, so our configuration is an instance of NO-AX.

Otherwise, if $\mathcal{I}, \rho \not\models E$ then, in view of the assumption that $\Gamma \Longrightarrow E$ is valid, it must be impossible to interpret the function symbols over $\mathcal{I}$ in such a way that all of Γ becomes true. This in turn can arise only if Γ contains two function definitions $f(\vec{s}) = u, f(\vec{t}) = v$ such that $\mathcal{I}, \rho \models s_i{=}t_i$ for $i = 1, \ldots, n$ but $\mathcal{I}, \rho \not\models u{=}v$. But then the side condition to NO-COMPOSE applies as required.

Note that by Section 13.8 this also proves the correctness of $\mathcal{NO}$for arbitrary convex theories.

13.15 Shostak Light

We now describe a faster algorithm for deciding implications $T \Rightarrow E$ involving uninterpreted function symbols. As already mentioned, the purpose of Shostak's algorithm is to decide implications between equations with uninterpreted function symbols. In the sequel let $\mathcal{T}$ be a canonisable and solvable theory over some language $\mathcal{L}$. Again following Ganzinger we present Shostak's algorithm as a deductive system $\mathcal{SL}$, called "Shostak Light" by Ganzinger. Shostak Light operates on configurations of the form $S \mid \Gamma \Longrightarrow E$ where $S \cup \Gamma \Longrightarrow E$ is an $\mathcal{NO}$-configuration and S contains solutions, i.e., equations of the form $x{=}t$ where x does not occur in any right-hand side of a solution. Such a configuration is by definition valid if $S \cup \Gamma \Longrightarrow E$ is valid. Figure 13.15 contains the inference rules of $\mathcal{SL}$. In rule SL-REDUCE $-[u/x]$ denotes the operation of replacing some or all occurrences of the term u by the variable x.

Again, it is clear that all rules reflect and preserve validity, so the proof system is sound and backwards application of proof rules does not lead into dead ends.

In order to ensure termination it is crucial to restrict backwards application of SL-SOLVE to the case where the equation $s = t$ to be solved does not contain any of the variables that solutions are being provided for in S. In other words, prior to solving an equation we should reduce it as much as we can. Now, SL-COMPOSE reduces the number of definitions and this number is not increased by any other rule. Rule SL-SOLVE reduces the number of equations in $\bar{\Gamma}$ and this number can only be increased by SL-COMPOSE. Rule REDUCE reduces the number of occurrences of variables for which solutions are available. The last point uses the restriction on the applicability of rule SL-SOLVE.

$$\frac{\mathit{Solve}(s{=}t) = \bot}{S \mid \Gamma, s{=}t \Longrightarrow E} \qquad (\text{SL-AXIOM-L})$$

$$\frac{\sigma(s) \equiv \sigma(t)}{S \mid \Gamma \Longrightarrow s{=}t} \qquad (\text{SL-AXIOM-R})$$

$$\frac{S \mid \Gamma, u = v, f(\vec{s}) = u \Longrightarrow E \qquad \sigma(s_i) \equiv \sigma(t_i) \text{ for } i = 1 \ldots n.}{S \mid \Gamma, f(\vec{s}) = u, f(\vec{t}) = v \Longrightarrow E} \qquad (\text{SL-COMPOSE})$$

$$\frac{S, x{=}u \mid \Gamma[u/x] \Longrightarrow E[u/x]}{S, x{=}u \mid \Gamma \Longrightarrow E} \qquad (\text{SL-REDUCE})$$

$$\frac{S, \mathit{Solve}(s{=}t) \mid \Gamma \Longrightarrow E \qquad \mathit{Solve}(s{=}t) \neq \bot}{S \mid \Gamma, s{=}t \Longrightarrow E} \qquad (\text{SL-SOLVE})$$

Figure 31: Inference rules of Shostak Light

Finally, we need to show that valid configurations are conclusions of $\mathcal{SL}$-rules. But, if $S \mid \Gamma \Longrightarrow E$ is valid then it is the conclusion of an $\mathcal{NO}$-rule! But then, if neither SL-SOLVE nor SL-REDUCE are applicable the equational side condition of the $\mathcal{NO}$-rule is completely reduced so that the corresponding $\mathcal{SL}$-rule applies.

14 Case study: n-bit ripple-carry adder

In this and the following section we look at two case studies that are part of the PVS documentation. They demonstrate many of the features that we have seen up to now and also give some idea of how to model real-world systems in PVS.

The first case study is about an n-bit ripple adder.

A bitvector of size n is a sequence of n bits. The bitvector $\mathbf{b} = (b_{n-1}, \ldots, b_0)$ denotes the integer $\sum_{i=0}^{n-1} b_i 2^i$. Here and in the sequel we use the implicit conversion from booleans to numbers that maps FALSE to 0 and TRUE to 1.

An n-bit adder takes two bitvectors $\mathbf{b}$ and $\mathbf{c}$ as well as a single bit cin (“carry in”) and computes a bitvector $\mathbf{s}$—the sum—and a single bit $cout$ (“carry out”) characterised as follows:

$$\sum_{i=0}^{n-1} s_i 2^i + cout 2^n = \sum_{i=0}^{n-1} b_i 2^i + \sum_{i=0}^{n-1} c_i 2^i + cin$$

14.1 Full adder

For $n = 1$ such a device is known as a "full adder"; it can be realised by the following boolean circuitry where $\oplus$ is *exclusive or* ("xor").

$$\begin{aligned} s &= b \oplus c \oplus cin \\ cout &= (b \wedge c) \vee (b \wedge cin) \vee (c \wedge cin) \end{aligned}$$

Characteristic property:

$$s + 2cout = b + c + cin$$

Let us define the full adder in PVS:

```
full_adder: THEORY
BEGIN
  IMPORTING bitvectors@bit

  FA(x, y, cin): [# carry, sum: bit #] =
    (# carry := (x AND y) OR (x AND cin) OR (y AND cin),
         sum := (x XOR y) XOR cin #)

  FA_char: LEMMA
      sum(FA(x, y, cin)) =
        x + y + cin - 2 * carry(FA(x, y, cin))
END full_adder
```

This definition contains a few new concepts. The `IMPORTING` clause

```
  IMPORTING bitvectors@bit
```

imports the theory `bit` residing in the PVS context `bitvectors`, see Fig. 32. Here `below(2)` is the subtype of `nat` consisting of just 0,1. More generally, `below(n)` is the subtype consisting of $0, \ldots n-1$. This is defined in the prelude. They form an example of *dependent types*, i.e., families of types depending on values. The `CONVERSION` keyword specifies that the conversion function `b2n` can be left implicit, i.e., if a bit is used where a natural number is expected then the conversion is automatically inserted.

```
bit: THEORY
BEGIN

  bit  : TYPE = bool

  b: VAR bit

  nbit : TYPE = below(2)  %  could be {n: nat | n = 0 OR n = 1}
  b2n(b:bool)     : nbit = IF b THEN 1 ELSE 0 ENDIF
  n2b(nb: nbit)   : bool = (nb = 1)

  CONVERSION b2n
END bit
```

Figure 32: The theory of bits

```
  upto(i):   NONEMPTY_TYPE = {s: nat | s <= i} CONTAINING i
  below(i):  TYPE = {s: nat | s < i}  % may be empty
```

Figure 33: Below and upto—simple dependent types

Records: The type

```
[#carry: bit, sum: bit #]
```

is a record type with two fields: `carry` and `sum` both of type `bit`. Elements of the record type are formed using the syntax

```
(#carry := ..., sum := ... #)
```

If `e:[#carry: bit, sum: bit#]` then `e‘carry` and `e‘sum` are its components. Alternative syntax is `carry(e)` and `sum(e)`. We can (functionally) update `e` using

```
e WITH [‘carry := ...]
```

This denotes the record with `carry` field equal to `...` and the rest as in `e`.

Proof of `FA_char` The characteristic property of the full adder can be proved by distinguishing eight cases (the possible values of the three input variables) and arithmetic simplification. Therefore, `(grind)` can do it. Of course, we can manually do the case distinction by invoking `bit_cases`.

14.2 Formalisation of the ripple adder

We formulate a theory `ripple_adder` parametrised by the size `N`:

```
ripple_adder[N: posnat] : THEORY

BEGIN

  IMPORTING full_adder, bitvectors@bv[N], bitvectors@bv_nat
```

We import the theories `bv[N]` defining a type of bitvectors of size `N` and a conversion function from that type to the natural numbers.

```
bv[N: nat]: THEORY
BEGIN
  IMPORTING bit
  bvec : TYPE = [below(N) -> bit]
  ^(bv: bvec, (i: below(N))): bit = bv(i)
END bv
```

```
bv_nat[N: nat]: THEORY
BEGIN
   IMPORTING bv, exp2
  bv2nat_rec(n: upto(N), bv:bvec[N]): RECURSIVE nat =
      IF n = 0 THEN 0
      ELSE exp2(n-1) * bv^(n-1) + bv2nat_rec(n - 1, bv)
      ENDIF
    MEASURE n
  bv2nat(bv:bvec[N]): below(exp2(N)) = bv2nat_rec(N, bv)
```

This defines a bitvector of size N as a function from below(N). If b is a bitvector and i:below(N) then we can use the notation b^i to access its i-th component.

The conversion to natural numbers is defined recursively.

We continue with the theory ripple_adder:

```
  nth_cin(j:below(N), bv1, bv2:bvec[N]): RECURSIVE bit =
      IF j = 0 THEN FALSE
      ELSE carry(FA(bv1(j-1), bv2(j-1),
                            nth_cin(j-1, bv1, bv2)))
      ENDIF MEASURE j

  adder(bv1, bv2): [# carry: bit, sum :bvec[N] #] =
    (# carry := nth_cin(N, bv1, bv2),
         sum := (LAMBDA (n:below(N)): sum(FA(bv1(n), bv2(n),
                          nth_cin(n, bv1,bv2)))) #)
```

The function nth_cin computes the carry input to the j-th full adder.

Typechecking these definitions generates a few typechecking conditions (TCCs) due to the use the subtype below (all except nth_cin_TCC3) and recursive definition:

```
nth_cin_TCC1: OBLIGATION
  FORALL (j: upto[N]): NOT j = 0 IMPLIES
            j - 1 >= 0 AND j - 1 < N;
nth_cin_TCC2: OBLIGATION
  FORALL (j: upto[N]): NOT j = 0 IMPLIES
           j - 1 >= 0 AND j - 1 <= N;
nth_cin_TCC3: OBLIGATION
```

```
  FORALL (j: upto[N]): NOT j = 0 IMPLIES j - 1 < j;
adder_TCC1: OBLIGATION N <= N;
adder_TCC2: OBLIGATION FORALL (n: below(N)): n <= N;
```

These are all trivial and can be proved with `M-x tcp`. We'll nevertheless prove the first one by hand:

```
 |-------
{1}   FORALL (j: upto[N]): NOT j = 0 IMPLIES j - 1 >= 0 AND j - 1 <
```

Now `(skosimp)` introduces a Skolem constant `j!1` of type `upto(n)`. We can use `M-x show-skolem-constants` to display it. With the command `(typepred "j!1")` (see prover guide p. 96) we can add the type constraint for `j!1` to our current sequent:

```
{-1}  j!1 <= N
  |-------
[1]   j!1 = 0
[2]   j!1 - 1 >= 0 AND j!1 - 1 < N
```

Implicitly, since `j!1` is a natural number we also know that `j!1 >= 0`. Therefore, we have now an arithmetic truth which we can discharge with `(grind)`. I would have thought that the constraint `j!1>=0` will also be made explicit by `typepred`. I'll try to find out why it isn't.

14.3 Specification of the ripple adder

Our goal is to prove the following.

```
  adder_correct: THEOREM
    FORALL(bv1, bv2:bvec[N]):
    bv2nat(sum(adder(bv1, bv2))) =
      bv2nat(bv1) + bv2nat(bv2) -
            carry(adder(bv1, bv2)) * exp2(N)
```

We want to prove this by induction on `N`, but `N` is fixed. So, we introduce a new parameter `j` allowing us to reformulate this goal for initial segments of the adder:

```
adder_invariant: LEMMA
    FORALL (j:upto(N), bv1, bv2:bvec[N]):
    bv2nat_rec(j,sum(adder(bv1, bv2)))
     = bv2nat_rec(j, bv1) +
    bv2nat_rec(j, bv2)
        - nth_cin(j, bv1, bv2) * exp2(j)
```

The following command proves the lemma:

```
(induct-and-simplify "j" :theories "full_adder" :exclude "FA")
```

The `:theories` directive says that the lemma `FA_char` may be used; the `:exclude` directive stipulates that the definition of `FA` should not be expanded. Without the directive `induct-and-simplify` will eagerly expand `FA` and then not see anymore where to apply `FA_char`.

Now the main result is a direct consequence.

15 Case study: Pipelined microprocessor

Our aim is to verify the pipelined microprocessor described on p. 77 of the PVS "WIFT tutorial". It executes instructions one by one; each instruction consists of three steps:

1. write the contents of registers named `src1` and `src2` into the operand registers `opreg1` and `opreg2`.

2. perform the ALU operation corresponding to the opcode (remembered in `opcoded` and store the result in the write-back register `wbreg`.

3. Update the destination register (whose address has been remembered in `dstndd`) with the value in `wbreg`.

These steps are carried out simultaneously for consecutive operations.

Therefore, we must bypass the register file in case an operation requires a result of its predecessor or pre-predecessor.

We begin by defining (discrete) time and signals:

```
time : THEORY
BEGIN
 time: TYPE = nat
END time

signal[val: TYPE+]: THEORY
BEGIN
 IMPORTING time
 signal: TYPE = [time->val]
END signal
```

The input wires of the processor are modelled as appropriately typed signals.

We model the state of the system as a recursively defined behaviour function from time to a record type consisting of appropriate values for all the data. The operation of the ALU is modelled as an uninterpreted function. Also the precise nature of data and addresses is left unspecified.

```
pipe_rec[addr, data, opcodes: NONEMPTY_TYPE]: THEORY

BEGIN

  IMPORTING time, signal
     opcode: signal[opcodes]
     src1, src2: signal[addr]
     dstn: signal[addr]

     stall: signal[bool]

aluop: [opcodes, data, data -> data]
     state : TYPE+ =
         [# dstnd : addr,
            dstndd : addr,
            stalld : bool,
            stalldd : bool,
            wbreg : data,
            regfile : [addr->data],
            opreg1 : data,
            opreg2 : data,
            opcoded : opcodes
```

```
          #]
```

Initially, we have arbitrary values assumed as constants. Thereafter, the state at time t is a function of the state at time $t - 1$.

```
     beh(t) : RECURSIVE state =
              IF t=0 THEN
              (#
               dstnd := someaddr,
               dstndd := someaddr,
               stalld := somebool,
               stalldd := somebool,
               wbreg := somedata,
               regfile := somefile,
               opreg1 := somedata,
               opreg2 :=somedata,
               opcoded := someopcode
              #)
              ELSE ...

LET dstnd  = dstn(t-1) IN
LET dstndd = beh(t-1)‘dstnd IN
LET stalld = stall(t-1) IN
LET stalldd = beh(t-1)‘stalld(t-1) IN
LET opcoded = opcode(t-1) IN
LET aluout =
 aluop(beh(t-1)‘opcoded, beh(t-1)‘opreg1, beh(t-1)‘opreg2) IN
LET wbreg = aluout IN

LET opreg1  =
    IF src1(t-1) = beh(t-1)‘dstnd & NOT beh(t-1)‘stalld
          THEN aluout
    ELSIF src1(t-1) = beh(t-1)‘dstndd & NOT beh(t-1)‘stalldd
          THEN beh(t-1)‘wbreg
    ELSE beh(t-1)‘regfile(src1(t-1)) ENDIF IN
LET opreg2 : data =
    IF src2(t-1) = beh(t-1)‘dstnd & NOT beh(t-1)‘stalld
          THEN aluout
    ELSIF src2(t-1) = beh(t-1)‘dstndd & NOT beh(t-1)‘stalldd
```

```
          THEN beh(t-1)‘wbreg
    ELSE beh(t-1)‘regfile(src2(t-1)) ENDIF IN

LET regfile =
    IF beh(t-1)‘stalldd THEN beh(t-1)‘regfile
         ELSE beh(t-1)‘regfile WITH
              [(beh(t-1)‘dstndd) := beh(t-1)‘wbreg]
    ENDIF
IN  (#
       dstnd := dstnd,dstndd :=dstndd,stalld := stalld,
       stalldd := stalldd,wbreg := wbreg,regfile:=regfile,
       opreg1 := opreg1,opreg2 := opreg2,opcoded := opcoded
      #)
ENDIF
MEASURE t
```

The correctness theorem asserts that as long as the stall line was not set the behaviour of the pipelined processor agrees with a purely sequential one.

Questions:

1. Why do we examine the source registers at time $t + 2$ instead of t?

2. What should we prove for the case when stall has been asserted?

```
    correctness: THEOREM
     FORALL t:
        NOT(stall(t))
         IMPLIES
            beh(t+3)‘regfile(dstn(t)) =
            aluop(opcode(t), beh(t+2)‘regfile(src1(t)),
                            beh(t+2)‘regfile(src2(t)))
END pipe_rec
```

The correctness property is entirely local and does not depend upon the validity of system invariants to be maintained. Therefore, if at all true the theorem should follow by propositional reasoning after expanding the definitions. The following sequence of commands achieves this effect.

```
% (INSTALL-REWRITES :DEFS T :THEORIES "pipe_rec")
% (SKOSIMP)
% (REPEAT (ASSERT))
% (REPEAT (LIFT-IF :UPDATES? NIL))
% (ASSERT))
```

16 Model checking and PVS

Suppose we are given a formula ϕ and want to know whether it holds under some particular interpretation $\mathcal{I}$. If it so happens that all the sets $[\![\tau]\!]_{\mathcal{I}}$ are finite then this question is in principle decidable automatically simply by trying out all the possibilities. Determining validity of a formula in such a finite interpretation is referred to as *model checking*. While most formulas of importance in mathematics or informatics are understood over infinite domains of interpretation (such as the natural numbers, lists, trees, etc.) there are a number of interesting problems that can be formulated as validity in finite interpretations.

Typical examples are properties of finite-state systems such as machine control mechanisms, communication protocols, or some algorithms in distributed systems:

- Does the lift not move when the door is open?
- Does protocol X never deadlock?
- Does algorithm Y guarantee mutual exclusion?
- Does the 8-queens problem have a solution?

To fix the ideas let us consider a simple binary semaphore as running example. We have the following finite set of states (in PVS notation):

```
section : TYPE = {idle, wait, critical}
semaphore : TYPE = {occupied,vacant}
state : TYPE = [# pc1, pc2:section, sem:semaphore#]
```

The behaviour of the system is modelled as a binary relation on global states, the *transition relation*.

```
N(s1,s2:state) : bool =
   pc1(s1)=idle AND s2 = s1 WITH [pc1 := wait]
OR pc2(s1)=idle AND s2 = s1 WITH [pc2 := wait]
OR pc1(s1)=wait AND sem(s1)=vacant AND s2 = s1 WITH
      [pc1:=critical,sem:=occupied]
OR pc2(s1)=wait AND sem(s1)=vacant AND s2 = s1 WITH
      [pc2:=critical,sem:=occupied]
OR pc1(s1)=critical AND s2 = s1 WITH
      [pc1:=idle,sem:=vacant]
OR pc2(s1)=critical AND s2 = s1 WITH
      [pc2:=idle,sem:=vacant]
```

Recall that `bool` and `boolean` are identical. Here are some first-order properties of this system, not all of them true:

$$\forall s_1{:}\texttt{state}\exists s_2{:}\texttt{state}.\texttt{N}(s_1, s_2)$$
$$\forall s{:}\texttt{state}\neg(\texttt{pc1}(s) = \texttt{critical} \wedge \texttt{pc2}(s) = \texttt{critical})$$
$$\forall s_1{:}\texttt{state}.\texttt{sem}(s_1) = \texttt{vacant} \Rightarrow \exists s_2{:}\texttt{state}.\texttt{N}(s_1, s_2)$$
$$\forall s_1{:}\texttt{state}.\texttt{pc1}(s_1) = \texttt{idle} \Rightarrow \exists s_2{:}\texttt{state}.\texttt{N}(s_1, s_2)$$

To decide validity of a first-order formula we simply implement the definition of validity as a recursive procedure. That is to say, we decide atomic formulas using truth tables, this is exponential in the size of the formula, but independent of the size of the interpretation; and we decide quantified formulas by trying out all possible instantiations of the quantified variables. This requires n recursive calls where n is the size of the domain quantified over. Summing up, to decide validity of a formula ϕ in an interpretation whose domains have size less or equal to some $n \in \mathbb{N}$ requires time $O(n^d)$ where d is the nesting depth of quantifiers in the formula ϕ.

Proposition Deciding validity of a formula ϕ over an interpretation of size n requires time $O(n^d)$ where n is the size of the interpretation and d is the quantifier nesting depth of ϕ.

Unfortunately, in this context first-order formulas are not particularly interesting since these only speak about local properties of the transition graph (graph of `N` in the example). Properties involving entire runs, i.e., global aspects cannot be expressed in first-order logic.

We will not make precise the distinction between "local" and "global" here[4], but rather consider the following examples:

[4] It has been done in the form of *Gaifman's theorem*

```
init(s) : bool = pc1(s)=idle & pc2(s)=idle & sem(s)=vacant
safe(s) : bool = NOT(pc1(s)=critical & pc2(s)=critical)
```

1. If `init`(s_0) then `safe`(s) for all s reachable from s_0 via `N`.

2. If `init`(s_0) then for each s_1 reachable from s_0 there exists s_2 reachable from s_1 with `sem`$(s_2) =$ `vacant`.

3. Starting from s_0 with `init`(s_0) there exists an infinite path along which eventually always `pc1` $=$ `wait`.

All these statements can be formalised in second-order logic, i.e., with quantification over types $[\tau_1, \ldots, \tau_n \rightarrow \text{bool}]$ where the τ_i are base types, e.g., here `state`. Indeed, in second-order logic one can define transitive closure and thus properties 1,2 above.

```
        Nstar(s1,s2):bool = FORALL (P:[state->bool]):P(s1)&
           (FORALL s3,s4:P(s3) & N(s3,s4) => P(s4)) => P(s2)

        safe(s) : bool = NOT(pc1(s)=critical AND pc2(s)=critical)
        Prop1:bool = FORALL (s0,s):init(s0) & Nstar(s0,s) => safe(s
...
```

We'll come to property 3 later. The domains of quantification in second-order logic are still finite, but the size of, e.g., $[D \rightarrow \text{bool}]$ is 2^n when n is the size of D. In our example, $n = 18$, so $2^n = 256K$; with three processes we would have $n = 52$ and thus $2^n \simeq 10^{16}$ which is already prohibitively large.

Fortunately, there exist fragments of second-order logic for which decidability can still be determined in polynomial (in n) time, yet which are strong enough to express all the above examples and many more interesting properties of finite-state sytems. We will look at two such fragments: the temporal logic CTL and the μ-calculus.

16.1 Computational Tree Logic (CTL)

Let `state` be a finite set and let`N : [state,state->bool]` be a relation. Furthermore, let `P:[state->bool]` be a property of states. We define

```
AX(N,P) = LAMBDA s:FORALL s':N(s,s')=>P(s')
EX(N,P) = LAMBDA s:EXISTS s':N(s,s')&P(s')
```

```
AG(N,P) = LAMBDA s:"P(s’) for all s’ reachable from s"
EF(N,P) = LAMBDA s:"P(s’) for some s’ reachable from s"
AF(N,P) = LAMBDA s:"for all infinite paths s=s(0),s(1),...
                    where N(s(i),s(i+1)) there exists i
                    such that P(s(i))"
EG(N,P) = LAMBDA s:"there is an infinite paths s=s(0),s(1),...
                    with N(s(i),s(i+1)) such that P(s(i)) for all i"
```

We have the following identities:

```
(1) EX(N,P) = NOT AX(N,NOT P)
(2) AG(N,P) = P & AX(N,AG(N,P))
(3) AF(N,P) = P OR AX(N,AF(N,P))
(4) EG(N,P) = P & EX(N,EG(N,P))
(5) EF(N,P) = P OR EX(N,EF(N,P))
(6) EF(N,P) = NOT AG(N,NOT P)
(7) EG(N,P) = NOT AF(N,NOT P)
```

Proposition: `AG(N,P)` is the largest solution of (2), `AF(N,P)` is the smallest solution of (3), `EG(N,P)` is the largest solution of (4), `EF(N,P)` is the smallest solution of (5).

Proof: It is clear that all the equations hold. For the universal property assume for example that

```
Q <= P & AX(N,Q)
```

where <= denotes set inclusion. We should prove `Q(s) => AG(N,P)(s)` for all `s`. Assume `Q(s)` and let `s=s(0),s(1),...` be an infinite path starting from `s` where `N(s(i),s(i+1)` for all `i`.

We have `Q(i)` for all `i` and thus `P(i)` for all `i`.

Now suppose

```
Q <= P & EX(N,Q)
```

We should prove `Q(s) =>  EG(N,P)(s)` for all `s`. Assume `Q(s)`; we get `P(s)` and a successor `s(1)` satisfying `Q(s(1))` continuing in this way, we form the desired infinite path.

Finally suppose

```
 P OR EX(N,Q) <= Q
```

We should prove `EF(N,P) <= Q`. Suppose `EF(N,P)(s)`, i.e., let `s=s(0),s(1),\dots,s(` be a path such that `N(s(i),s(i+1)))` and `P(s(n))`. We clearly have `Q((n))`, hence `EX(N,Q)(s(n-1))`, hence `Q(s(n-1))`, hence `Q(s)`.

16.2 Examples of CTL formulas

1. If init(s_0) then safe(s) for all s reachable from s_0 via N.

   ```
   prop1 : bool = FORALL s:init(s) => AG(N,safe)(s)
   ```

2. If init(s_0) then for each s_1 reachable from s_0 there exists s_2 reachable from s_1 with sem$(s_2) =$ vacant.

   ```
   prop2 : bool = init <= AG(N,EF(N,LAMBDA s:vacant?(sem(s))))
   ```

3. Starting from s_0 with init(s_0) there exists an infinite path along which eventually always pc1 $=$ wait.

   ```
   prop3 : bool = init <= EF(N,EG(N,LAMBDA s:wait?(pc1(s))))
   ```

We remark that CTL has two more temporal operators:

```
AU(N,P,Q) = LAMBDA s:"For all N-paths s=s(0),s(1),..., starting from s
                      there exists i0 such that P(s(i))
                      for all i<i0 and Q(s(i0))"
EU(N,P,Q) = LAMBDA s:"For some N-path s=s(0),s(1),..., starting from s
                      there exists i0 such that P(s(i))
                      for all i<i0 and Q(s(i0))"
```

You may find it instructive to state and prove a fixpoint characterisation for these operators.

16.3 μ-calculus

The μ-calculus provides a formal notation for least and greatest fixpoints of monotonic operators.

In μ-calculus we can for example define

```
AF(N,P) = mu(LAMBDA Q : P OR AX(N,Q))
EG(N,P) = nu(LAMBDA Q : P &  EX(N,Q))
```

In general, if `Phi : [pred[state]->pred[state]]` is monotone then `mu(Phi)` denotes the least fixpoint of `Phi` and `nu(Phi)` denotes the largest fixpoint of `Phi`.

If, as in the examples, `state` is finite then the least fixpoint of Φ can be computed as the union of the sets $\emptyset, \Phi(\emptyset), \Phi(\Phi(\emptyset)), \ldots$ which must reach a fixpoint after a finite number of steps.

Likewise the largest fixpoint is obtained as the intersection of $S, \Phi(S), \Phi(\Phi(S)), \ldots$ where S is the set of all states. 2 In general, fixpoints can be defined by the formulas

$$\mu(\Phi) = \bigcap_{(\Phi(P) \subseteq P)} P$$

$$\nu(\Phi) = \bigcup_{P \subseteq \Phi(P)} P$$

which, unlike the iteration, are not suitable for algorithmic purposes.

On the other hand, it is clear that the iterative characterisation of fixpoints lends itself to a polynomial time algorithm which uses a data structure S for sets of states, initialises it as the empty set (in the case of the least fixpoint and updates it by $S \leftarrow \Phi(S)$ until it stabilises.

In practice, more efficient algorithms for model-checking μ-calculus and CTL are available. Among them is *symbolic model checking* (McMillan '92) which is based on a representation of states as valuations of boolean variables and sets of states as boolean formulas over these variables. In this way, often a much more concise representation may be achieved. In this course, we will not concern ourselves with these issues and assume model-checking as a black-box provided to us via the PVS command `model-check`.

Some further properties of least and greatest fixpoints are the following which are all provable with PVS:

```
s, s1, s2: VAR T

 p, p1, p2: VAR pred[T] % pred[T] = [T->boolean]

 pp : VAR [pred[T]->pred[T]]

 <=(p1,p2): bool = FORALL s: p1(s) IMPLIES p2(s)

 monotonic?(pp): bool =
     FORALL p1, p2: p1 <= p2 IMPLIES pp(p1) <= pp(p2)


 mu(pp) : pred[T] = LAMBDA s: FORALL p:(pp(p)<=p) => p(s)
```

```
  nu(pp) : pred[T] = LAMBDA s: EXISTS p:(p<=pp(p)) & p(s)


 least_mu : PROPOSITION
      FORALL pp: FORALL p:pp(p)<=p => mu(pp)<=p

  pre_fixpoint_mu:PROPOSITION
        FORALL pp: monotonic?(pp) => pp(mu(pp)) <= mu(pp)

  post_fixpoint_mu:PROPOSITION
        FORALL pp: monotonic?(pp) => mu(pp)<=pp(mu(pp))
  greatest_nu : PROPOSITION
      FORALL pp: FORALL p:p<=pp(p) => p<=nu(pp)

  post_fixpoint_nu:PROPOSITION
        FORALL pp: monotonic?(pp) => nu(pp)<=pp(nu(pp))

  pre_fixpoint_nu:PROPOSITION
        FORALL pp: monotonic?(pp) => pp(nu(pp)) <= nu(pp)

  mu_nu:PROPOSITION
       FORALL pp:FORALL s:mu(pp)(s) <=>
            NOT(nu(LAMBDA p:LAMBDA s1:NOT(pp(LAMBDA s:NOT(p(s)))(s1)))(s))

  nu_mu:PROPOSITION
       FORALL pp:FORALL s:nu(pp)(s) <=>
            NOT(mu(LAMBDA p:LAMBDA s1:NOT(pp(LAMBDA s:NOT(p(s)))(s1)))(s))
```

Property least_mu is direct from the definition. Here is a proof of pre_fixpoint_mu. Assume that pp X<=X. By least_mu we have mu(pp)<=X, hence pp(mu(pp))<=pp(X) by assumption and then pp(mu(pp))<=X by assumption on X. Since this holds for fixed but arbitrary X we obtain pre_fixpoint_mu.

Applying monotonicity to pre_fixpoint_mu together with least_mu then gives post_fixpoint_mu. The proofs of the analogous properties of nu and mu_nu are left as an exercise.

16.4 Fairness

Liveness properties refer to the recurrent presence of desirable events.

For example, to express that at any time it is possible to reach a state in which `pc1` is critical we may define:

```
live : THEOREM
 init(s) IMPLIES
    AG(N,EF(N,LAMBDA s:critical?(pc1(s))))(s)
```

This can be proved with `(model-check)`.

Often liveness only holds relative to fairness assumptions. Our running example is too trivial to provide interesting instances of this phenomenon so we merely present the solution:

If `N` is a relation, `P` and `Ff` are properties then

```
fairAF(N,P)(Ff)(s)
```

asserts that `P` will eventually hold on all those paths along which `Ff` obtains infinitely often. In other words, the unfair paths which eventually get stuck with `NOT Ff` are exempt from the universal quantification over paths implicit in the `AF` operator.

The operator `fairAF` is not definable in CTL but admits an encoding in the μ-calculus as follows, see also `prelude.pvs`:

```
fairAF(N,S)(Ff) = mu(LAMBDA X:nu (LAMBDA Y:S OR
                       (Ff AND AX(N,X)) OR (NOT Ff AND AX(N,Y))))
```

The proof that this does represent `fairAF` is not trivial.

There are more general versions of fairness also definable in μ-calculus:

- fair paths must validate several different fairness predicates infinitely often not just one. "Justice requirements"
- fair paths must validate one or more *compassion requirements* where a compassion requirement comprises two predicates (P, Q) and is validated if whenever P holds continuously then Q will eventually obtain.

16.5 Model-checking and abstraction

Many real systems have infinite state space or a finite state space which is too large for model checking.

Abstraction: use theorem proving to show that

$$\boxed{\text{finite state system}} \models \phi^* \quad \Longrightarrow \quad \boxed{\text{infinite state system}} \models \phi$$

for some cleverly chosen formula ϕ^*

Use `sem:int` and replace

- `sem=vacant` with `sem>0`
- `sem:=occupied` with `sem:=sem(s1)-1`
- `sem:=vacant` with `sem:=sem(s1)+1`
- In `init(s)`: `sem(s)=vacant` with `sem(s)=1`

The state space is now infinite, but we know that all semaphore values ≤ 0 can be identified and likewise all semaphore values > 1.

We thus define the abstract state space as follows:

```
abs_state: TYPE =
    [# pc1, pc2: location, semleq0: bool, semgt1: bool #]
abstract(s:state):abs_state =
 (# pc1:=pc1(s),pc2:=pc2(s),semleq0:=sem(s)<=0,semgt1:=sem(s)>1 #)
```

and the abstracted predicates:

```
abs_init(abs_s) : bool = pc1(abs_s)=idle AND pc2(abs_s)=idle
   AND NOT semgt1(abs_s) AND NOT semleq0(abs_s)
abs_N(abs_s1,abs_s2) : bool = %somewhat lengthy
abs_safety(abs_s) : bool = NOT (critical?(pc1(abs_s)) AND
critical?(pc2(abs_s)))
```

We can now prove the following soundness properties:

```
        abs_N_ok : LEMMA
        N(s1,s2) => abs_N(abs(s1),abs(s2))

        abs_init_ok : LEMMA
        init(s) => abs_init(abs(s))

        abs_safety_ok : LEMMA
        abs_safety(abs(s)) => safety(s)

abs_safe : LEMMA
init(abs_s) => AG(abs_N,abs_safety)(abs_s)
```

The first three are proved with `(grind)` the fourth with `(model-check)`.

The desired result follows from a tautology of temporal logic, which we will explain in more detail below.

16.6 Automating abstraction

This theorem

```
safe : THEOREM
         init(s) IMPLIES
            AG(N,safety)(s)
```

can be proved with the single command:

```
(ABSTRACT-AND-MC state abs_state ((semleq0 "lambda s:s‘sem<=0")
                                   (semgt1 "lambda s:s‘sem>1")))
```

In the sequel, we will explain how `abstract-and-mc` works.

We assume that we are given a concrete state space Q_c, an abstract state space Q_a, and an abstraction function $f : Q_c \to Q_a$.

Given $I_c \subseteq Q_c, N_c \subseteq Q_c \times Q_c, S_c \subseteq Q_c$ we want to choose $I_a \subseteq Q_a, N_a \subseteq Q_a \times Q_a, S_a \subseteq Q_a$ such that

- If $\phi_a \equiv \forall x{\in}Q_a.I_a(x) \Rightarrow \texttt{AG}(N_a, S_a)(x)$ then $\phi_c \equiv \forall x{\in}Q_c.I_c(x) \Rightarrow \texttt{AG}(N_c, S_c)(x)$,
- Define I_a, N_a, S_a such that $\{x \mid I_a(x) \Rightarrow \texttt{AG}(N_a, S_a)\}$ is as big “as possible” under the constraint $\phi_a \Rightarrow \phi_c$.

What do we mean by “as possible?” A trivial but useless choice consists of putting

- put $I_a(x) \Leftrightarrow \mathtt{ff}$ if $\forall x{\in}Q_c.I_c(x) \Rightarrow \texttt{AG}(N_c, P_c)(x)$ holds or not,
- put $I_a(x) \Leftrightarrow \mathtt{tt}$, $N_a(x, y) = \mathtt{tt}$, $S_a(x) = \mathtt{ff}$ if not $\forall x{\in}Q_c.I_c(x) \Rightarrow \texttt{AG}(N_c, S_c)(x)$

Now $\phi_a \Rightarrow \phi_c$ but to find ϕ_a we must know to begin with whether or not ϕ_c holds. We thus seek to approximate this ideal situation:

The idea is that the implication $\phi_a \subseteq \phi_c$ should hold *because*

- $I_c(x) \Rightarrow I_a(f(x))$,
- $N_c(x, y) \Rightarrow N_a(f(x), f(y))$,
- $S_a(f(x)) \Rightarrow S_c(x)$.

Moreover, I_a, N_a should be as small as possible, P_a as big as possible with these properties.

Let's see why $\phi_a \Rightarrow \phi_c$ in this case. Assume ϕ_a, i.e., for all $y \in Q_a$ reachable with N_a from some $x \in I_a$ one has $S_a(y)$. Now, towards proving ϕ_c assume $x' \in Q_c$ with $I_c(x')$ and $y' \in Q_c$ reachable from x' with N_c via a path $x' = x'_0, \dots, x'_n = y'$ where $N_c(x'_i, x'_{i+1})$ for $i = 0, \dots, n-1$. Putting $x_i = f(x'_i)$ we have $I_a(x_0)$ and $N_a(x_i, x_{i+1})$ for $i = 0, \dots, n-1$. So, by ϕ_a we have $S_a(x_n)$, i.e., $S_a(f(y'))$, hence $S_c(f(y'))$.

16.6.1 Best approximation

We can reduce the task of finding I_a, N_a, S_a to the following general question: Given a concrete predicate $P \subseteq Q_c$ find $P_{\text{abs}} \subseteq Q_a$ as small as possible such that

$$\forall s{\in}Q_c.P(s) \Rightarrow P_{\text{abs}}(f(s))$$

Indeed, if we can solve this general question then we can put $I_a = (I_c)_{\text{abs}}$, we can put $N_a = (N_c)_{\text{abs}}$ with Q_c replaced by $Q_c \times Q_c$ and Q_a replaced by $Q_a \times Q_a$ and f replaced by $\lambda(s_1, s_2).(f(s_1), f(s_2))$. Finally, we can put $P_a = \neg((\neg P_c)_{\text{abs}})$.

The general question has the *formal* solution

$$P^*_{\text{abs}} := \{\sigma{\in}Q_a \mid \exists s{\in}Q_c.f(s) = \sigma \wedge P(s)\}$$

we have indeed

$$\forall s{\in}Q_c.P(s) \Rightarrow P^*_{\text{abs}}(f(s))$$

by existential introduction and given any $P_{\text{abs}} \subseteq Q_a$ satisfying

$$\forall s{\in}Q_c.P(s) \Rightarrow P_{\text{abs}}(f(s))$$

then we also have $P^*_{\text{abs}} \subseteq P_{\text{abs}}$ by existential elimination. So, the formal solution P^*_{abs} is indeed the best possible.

Unfortunately, the formal solution is not of much use because it contains a quantification over the infinite domain Q_c thus not amenable to automatic verification.

We will now study an important special case of abstraction where best approximations can be automatically computed or at least approximated reasonably well.

16.6.2 Predicate abstraction

Predicate abstraction refers to the case where the abstract state space Q_a is equal to the set of valuations of a finite set of boolean variables. In other words, if the abstraction function is given by a finite set of predicates on the concrete state space.

Let B be a set of predicates on Q_c and

Put $Q_a \overset{\text{def}}{=} 2^B$

and define $f : Q_c \to Q_a$ as $f(x) = \{b{\in}B \mid b(x)\}$.

A predicate $S \subseteq Q_a$ amounts to a boolean combination of the B, i.e. can be written as a *conjunction of clauses* over the B.

We have

$$P^* = \bigwedge_{\substack{\forall s.P(s) \to C(f(s)) \\ C \text{ a clause in the } B}} C$$

We can now approximate P^* by

$$\bigwedge_{\substack{\text{PVS} \vdash \forall s.P(s) \to C(f(s)) \\ C \text{ a clause in the } B}} C$$

In other words, for each clause C over B we let PVS check whether $P(s) \Rightarrow\subseteq C(f(s))$. We then let $\exists_f^{\text{approx}}(P)$ be the conjunction of those clauses for which this is the case. Of course, if C_1 is a sub-clause of C_2 (fewer literals) then $C_1 \Rightarrow C_2$ and thus if $P(s) \Rightarrow C_1(f(s))$ then all the more $P(s) \Rightarrow C_1(f(s))$. So we start with small clauses and do not consider super-clauses of clauses that have already made it.

We remark that best approximations as described above are useful only for simple safety properties. For liveness properties, let alone ones involving fairness, more complicated abstraction mechanisms are necessary.

17 Type theory and Coq

The aim of this part of the course is to introduce you to another view of logic and and proofs which among other things shows how one can represent proofs in such a way that they can be independently verified. The type-theoretic view of logic also provides a beautiful interaction between data (often programs) and proofs and can sometimes suggest new ways of structuring proofs. It also forms the basis of modern explanations of theories and modules. Finally, proofs in type theory have a rather direct interpretation as programs which means that they can sometimes be directly executed or that programs can be extracted from them. This has applications to program development.

17.1 BHK interpretation

The main idea behind type theory is that proofs are to be seen as first-class mathematical objects and that a formula is dubbed valid if it admits a proof. What such a proof is can be stated informally in the following way known as *Brouwer-Heyting-Kolmogorov* interpretation:

- A proof of $\phi \wedge \psi$ is a pair consisting of a proof of ϕ and a proof of ψ
- A proof of $\phi \vee \psi$ is either a proof of ϕ or a proof of ψ
- A proof of $\phi \Rightarrow \psi$ is a procedure which transforms an arbitrary proof of ϕ into a proof of ψ.
- A proof of $\exists x{:}\tau.\phi(x)$ is a pair consisting of a witness $t : \tau$ and a proof of $\phi(t)$.
- A proof of $\forall x{:}\tau.\phi(x)$ is a procedure which transforms an arbitrary element $t : \tau$ into a proof of $\phi(t)$.

Examples of BHK proofs *Proof of* $A \wedge B \Rightarrow A$*:* Let p be a proof of $A \wedge B$. By definition it is a pair $p = \langle p_1, p_2 \rangle$ where p_1 is a proof of A and p_2 is a proof of B. The desired proof is the procedure which turns p into p_1.

Proof of $(A \Rightarrow B) \wedge A \Rightarrow B$. Let p be a proof of $(A \Rightarrow B) \wedge A$. Its first component applied to the second gives a proof of B.

Proof of $\forall x{:}\tau.(\phi(x) \Rightarrow \exists x{:}\tau.\phi(x))$. Let $t{:}\tau$. We must construct a proof of $\phi(t) \Rightarrow \exists x{:}\tau.\phi(x)$. To that end assume a proof p of $\phi(t)$. The pair $\langle t, p \rangle$ then is a proof of $\exists x{:}\tau.\phi(x)$.

Intuitively appealing though it may be the BHK interpretation leaves some questions:

- What is a “procedure”?
- Can we decide whether a proof really is one?

- What are the ranges τ of quantifiers?
- What about higher-order logic?

To provide answers to these questions and to generalise BHK interpretation is the goal of type theory.

17.2 Mimimal logic and lambda calculus

Minimal logic is the fragment of propositional logic consisting of the formulas built up from atomic formulas by implication only.

Formulas:

- propositional variables $A, B, C, \dots$ are formulas
- if ϕ, ψ are formulas so is $\phi \Rightarrow \psi$.

Contexts: are finite functions Γ from proof variables to formulas, e.g.,

$$x{:}A \Rightarrow B, y{:}A, z{:}B$$

we view a context as a set of labelled assumptions.

We write $\Gamma \vdash t : \phi$ to mean that t is a proof of ϕ under the assumptions recorded in context Γ.

- if $\Gamma(x) = \phi$ then $\Gamma \vdash x : \phi$.
- if $\Gamma \vdash t : \phi \Rightarrow \psi$ and $\Gamma \vdash u : \phi$ then $\Gamma \vdash tu : \psi$.
- if $\Gamma, x{:}\phi \vdash t : \psi$ then $\Gamma \vdash \lambda x{:}\phi.t : \phi \Rightarrow \psi$.

$$x{:}A, y{:}B \vdash x : A$$

so

$$x{:}A \vdash \lambda y{:}B.x : B \Rightarrow A$$

so

$$\vdash \lambda x{:}A.\lambda y{:}B.x : A \Rightarrow (B \Rightarrow A)$$

I.e., $\lambda x{:}A.\lambda y{:}B.x$ is a proof of $A \Rightarrow (B \Rightarrow A)$.

Proofs under assumptions are needed as auxiliary device.

Example

$$f{:}A \Rightarrow B \Rightarrow C, g{:}A \Rightarrow B, x{:}A \vdash fx : B \Rightarrow C$$
$$f{:}A \Rightarrow B \Rightarrow C, g{:}A \Rightarrow B, x{:}A \vdash gx : B$$

so

$$f{:}A \Rightarrow B \Rightarrow C, g{:}A \Rightarrow B, x{:}A \vdash fx(gx) : C$$

so

$$\vdash \lambda f{:}A \Rightarrow B \Rightarrow C.\lambda g{:}A \Rightarrow B.\lambda x{:}A.fx(gx) : C$$

Real type theorists don't justify these proofs. According to them they speak for themselves. This is not as bad as it may sound. If we interpret propositional calculus in set theory then we have merely transported the problem of explaining truth of propositional formulas to the much more difficult problem of explaining truth of statements in set-theory.

We have to make philosophical, unprovable assumptions anyway, so why make them more complicated than required by the current application.

Nevertheless, it is the case that every formula which admits a proof is valid in the sense of truth tables; the converse is, however, not true:

To wit, the following tautology of propositional logic is not provable in type theory Peirce formula:

$$\phi := ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$$

Attempting to prove it leads to failure

$$\lambda f{:}(A \Rightarrow B) \Rightarrow A.f(\lambda x{:}A.????)$$

Those of you who have seen the simply-typed lambda calculus will notice that a proof in this sense of some formula ϕ is nothing but a lambda term of type ϕ when we view propositional variables as basic types and implication as function type.

Every proposition in minimal logic can be read as a type. A proof of a proposition is a λ-term inhabiting its type. This is called the propositions-as-types paradigm.

17.3 Propositions-as-types and predicate logic

In order to extend the propositions-as-types paradigm to predicates we represent predicates as type-valued functions which are known as *dependent types*.

If D is a type and $P : D \rightarrow$ *type* is a predicate what should a proof of the universal quantification "for all $x : D$, $P(x)$" be? It is a procedure which associates with each d in D a proof of $P(d)$, i.e., an element of $P(d)$. Thus, the type of proofs of the universal statement is the type of functions that map $d : D$ to elements of $P(d)$. The type of those functions is denoted $\Pi d{:}D.P(d)$ and is called a Π-type or a dependent product.

This type former also appears in the context of programming with dependent types; suppose for example that $\mathit{Vec}(n)$ is the type of lists of length n with entries of some fixed type A, say. Then an initialisation function *init* that produces a list with n identical entries would have type $A \to \Pi n{:}\mathit{nat}.\mathit{Vec}(n)$.

What about existential quantification? A proof of an existential statement "there exists $d : D$ such that $P(d)$" should consist of a witness $d : D$ and a proof of $P(d)$, i.e., an element of $P(d)$. Thus, a proof of the existential statement is a pair (d, p) where $d : D$ and $p : P(d)$. The type of these pairs where the type of second component depends on the first component is written $\Sigma d{:}D.P(d)$ and called a Σ-type or dependent sum.

The Σ-type comes with two projections:

$$\mathit{proj1} : (\Sigma d{:}D.P(d)) \to D$$

and

$$\mathit{proj2} : \Pi x : \Sigma d{:}D.P(d)).P(\mathit{proj1}(x))$$

Notice that the second projection can only be typed as a dependent product, not as an ordinary function space. It is common to write the projections as postfix $.1$ and $.2$.

We have the *definitional equalities* $(d, p).1 = d$ and $(d, p).2 = p$.

Definitional equalities are equalities that do not need to be justified by proofs; they hold "by definition". Of course, for this to make sense definitional equality should be decidable.

Π- and Σ-types as well as definitional equality are cornerstones of *Martin-Löf tape theory* invented in the 70s by Per Martin-Löf as a foundation of constructive mathematics. There are good textbooks on Martin-Löf type theory; here we content ourselves by showing how the axiom of choice admits a proof in this formalism. The type corresponding to the axiom of choice is the following:

$$(\Pi a{:}A.\Sigma b{:}B.R(a, b)) \to \Sigma f{:}A \to B.\Pi a{:}A.R(a, f(a))$$

The following is an inhabitant of this type:

$$\lambda H.(\lambda a.\ (H\ a).1,\ \lambda a.\ (Ha).2)$$