# SOFTWARE VERIFICATION TOOLS

# PART II

**PETER H. ROOSEN-RUNGE**

**DEPARTMENT OF COMPUTER SCIENCE**
**YORK UNIVERSITY**
**TORONTO, CANADA**

# 4. SIMPLIFYING EXPRESSIONS AND PROPOSITIONS

The ability to check that a proposition is a tautology doesn't by itself provide sufficient logical power to verify interesting properties of computational expressions or pieces of code, as evidenced by the example 0 = 0 from **3.2**, which reminds us that even "self-evident" mathematical truths aren't necessarily tautologies, if no meaning has been given to the symbols. What's missing is a way to introduce such meanings into the proof process. The purpose of this chapter is to introduce a technique which can be easily implemented computationally to do this. It involves *rewriting* complex expressions as mathematically equivalent but simpler ones—an approach which you may find quite natural, as it mimics to some extent the approach high-school students are taught for solving certain kinds of problems in algebra and arithmetic. This sort of simplification process has many applications in verification which we will explore in the following chapters. For the moment, we focus on what simplification does and how it is implemented.

To see how rewriting can be used to make a proof, consider the example

not(x < 0 and x > 0).

If we know from the definition of >, that x > 0 can be rewritten as not x < 0 and not x = 0, then we can rewrite the entire proposition as the equivalent not (x< 0 and not x < 0 and not x = 0 ) , which is easily seen to be a tautology.

An arithmetic example: the proposition x = x + (x - x) can be simplified through several steps to true, by first replacing (x-x) to 0, then rewriting x+0 as the equivalent term x, and finally, rewriting x = x as true.

An example combining logic and arithmetic simplification: the proposition

x > 0 implies a + a = a*2

can be simplified to

x > 0 implies a*2 = a*2, and then to x > 0 implies true

which is valid since any proposition of the form *T* implies true is a tautology.

Simplifying propositions in this way greatly extends the usefulness of the wang tautology checker since it allows us to apply known theorems in logic and arithmetic to establish a much wider class of true propositions. Instead of having all non-logical terms treated as Boolean variables by the tautology checker, terms can be converted to truth values or eliminated by simplifying the input prior to checking it.

The process of simplification is implemented computationally by specifying a set of **simplification rules** (also called *rewrite* rules) which are applied by a rule-interpreter to replace or rewrite any term matching the pattern on the left-hand side of one of the rules by a simpler but mathematically or logically equivalent term given on the right-hand

side of the rule. This process is repeated over and over again on the newly generated terms until no rule applies[1]. For the example above, the rules might be:

```
X + X  ->>  X*2
X = X  ->>  true.
```

(We will use the operator ->> to separate the left and right-hand sides of a simplification rule to avoid confusion with other arrows such as -> , --> and >>.)

Simplification rules which refer to a particular area of mathematics can be grouped into separate files, analogous to the division of a program into modules. We will call such files of related simplification rules *theory files*. Using just the rules in files which are relevant to a specific problem domain speeds up the matching process, since irrelevant rules are omitted.

### *4.1 The Structure of Simplification Rules*

Here are some rules from a theory file `arithmetic.simp`:

```
X * 0 ->> 0.
X * 1 ->> X.
M * N ->> P if P is M*N.

X + 0 ->> X.
X - X ->> 0.
X + Y = 0 ->> X = -Y.
(X + Y) * N ->> X*N + P if P is Y*N.

X**0 ->> 1.
X**1 ->> X.
0**X ->> 0.

X*Y*X ->> Y*X**2.
X**N * Y * X ->> X**(N+1)*Y.
```

As you can see, there are two sorts of simplification rules:

• Rules of the form *left ->> right* assert that any occurrence of the pattern on the left can be replaced by the pattern on the right.

• Rules of the form *left ->> right* **:-** *condition* state that the simplification is performed only if the condition can be determined to be true.[2]

Terms beginning with upper-case letters are *rule variables*. In the left side of a simplification rule, these variables are matched to terms in the expression to be simplified. If the entire left side can be matched to a term or subterm, the variables are

[1] [Sperschneider, Ch. 13] provides an introduction to the formal theory of simplification and rewriting. See also [Dahl, Section 2.2.6, p.30] and [Eisinger, 1989]. In [Wos, 1992], simplification rules are called *demodulators*. [Bouma. 1989] reviews Prolog implementations of rewrite rules.

[2] Conditional rewrite rules are an implementation of conditional equations, which are discussed in [Thatcher, 1982.]

given the values determined by the match, and these values are used in the expression on the right to construct the replacement for the term or subterm being simplified. (Note that the expressions being simplified should **not** contain rule variables; in the expressions to which the rules are to be applied, such as x + 0, symbols representing mathematical variables begin with lower-case letters.)

### *Conditions*

The condition in a conditional rule must evaluate to either true or false, usually by evaluating the values of the variables appearing in the condition.

Conditional rules are useful for evaluating numeric quantities. In the example above,

M * N ->> P :- P assigned M*N.

the conditional rule uses the special operator **assigned**. This operator takes a variable as a left argument and an expression as a right argument and assigns the variable the value of the expression if it can be evaluated numerically. If not, the rule fails to apply. Thus, the rule

M * N ->> P :- P assigned M * N.

will simplify -2 * 5 to -10, but will leave x * 7 alone.

In the rule

X > Y ->> true :- number(X), number(Y), X > Y.

the term matching the pattern X > Y is rewritten as true if X and Y are numbers and X > Y. Similarly, the comparison operations <>, >, =, <=, and => are used to compare numerical values in a condition.

Another use of conditional rules is to incorporate type information into a rule. In the following example, the variable i is asserted to be of type int (integer), and a rule is asserted which is applicable to X <= 0 only if X is of type int.

```
|: assert(int(i)).
|: assert(( X <= 0 ->> X < 1  :- int(X) )).
|: i <= 0.

i<=0 ->>
   i<1

|: x <= 0.

x<=0 ->>
   x<=0
```

Syntax note: asserting a conditional rule requires wrapping it in an extra set of parentheses.

### *What makes a valid simplification rule?*

Unconditional simplification rules must satisfy a simple property: if the ->> is replaced by = , the resulting equation must be true in the intended mathematical theory. Thus,

X - 0 ->> X.

is a valid simplification rule if X is a number and - is the usual minus operation, because the equation

X - 0 = X

is true in ordinary arithmetic. Similarly, a conditional rule

*Left* ->> *Right* :- *Condition*.

is valid if the proposition

*Condition* implies *Left* = *Right*

is valid.

A significant problem which must be taken into account when rules are added to a theory file, or asserted at the terminal, is that the rules must not combine to form an infinite loop, in which some rules undo the effects of others to produce the original term again[3].

Suppose, for example, we added the rule

Y * X ->> X *Y

so that a rule such as X * 1 ->> X could be used to simplify 1 * X. Unfortunately the new rule is no help as it rewrites a product into a form to which it can then reapply, so the rewrite process will never terminate. The next section shows how we can handle the problem of commutative operators like * without running the risk of non-terminating rules.

### *4.2 Canonical ordering of subterms*

Terms have no intrinsic meaning; as a result, in the absence of any rules to give a mathematical meaning to operators and functions, terms are only equal if they are *identical as terms* — a*b is not interpreted as equal to b*a. But to do ordinary arithmetic, we will want to simplify equations between mathematically equivalent expressions to true.

[3] See [Manna, Chapter 6] for a discussion of conditions which guarantee termination of rewrite rules. In the research literature on term-rewriting systems, terminating sets of rules are sometimes called *Noetherian*.

Consider a term such as x + a*b = b*a+ x. In order to simplify this equation to true, we need a way of putting subterms in expressions with commutative operators into a standard order, so that their identity with other subterms can be detected, thus allowing further simplification through a rule such as

X = X ->> true.

which tests for the identity of the terms on the left and right sides of an equation. We accomplish this by means of a standardized or *canonical ordering* of subterms, which rearranges subterms involving commutative operators into a fixed order[4]. Thus,

x + a*b

is rearranged to

a*b + x,

and then to

b*a + x.

The algorithm in the `simplify` program uses a built-in ordering of terms to re-order any expression of the form X op Y where op is commutative, so that the left subterm is 'greater' in the term ordering than the right. This choice of reordering has the small advantage that expressions such as

x + x + x = (x + x) + x

are not reordered into a form such as x + (x + x) which requires parentheses. A slight disadvantage is that equalities such as x = y * 2 get turned around to become

y * 2 = x.

### *Associativity*

Reordering expressions with commutative operators is not sufficient to match all equivalent expressions. For example, it won't enable us to simplify

(y + x) + 1 = (x + 1) + y

because each sum is already in canonical order and will not be simplified further.

In order to be able to properly simplify terms involving arithmetic operators, we need to pay attention also to the effect of associativity.

At the purely syntactic level, if parentheses can be omitted, then a binary operator is either left- or right-associative, that is, either

a ∘ b ∘ c = (a ∘ b) ∘ c,

[4] Simplifying both sides of an equation to the same canonical form is sometimes called 'normalization'. For an example of a verification system based on normalization, see [Srivas, 1990.]

or

$$a \circ b \circ c = a \circ (b \circ c).$$

This is just a feature of the parser which constructs terms from strings — it has no semantic significance. The choice of left- or right- associativity is arbitrary; in the verification tools discussed here, binary operators are all declared to be either *left-associative*, or *non-associative* (parentheses cannot be omitted).

If a binary operator is also *associative* (this is now a semantic, not a syntactic concept), then

$$a \circ b \circ c = (a \circ b) \circ c = a \circ (b \circ c).$$

As part of the reduction of a term to canonical form, right associations are rewritten as left-associations so that parentheses are removed when left-associative terms are printed. Thus

```
c + (b + a)
```

gets rewritten as

```
(c + b) + a
```

which prints as

```
c + b + a.
```

In addition, terms composed entirely of subterms with the same functor which is both commutative and associative (such as + and *) are arranged into a sequence of terms in descending order, so

`(x + 1)+ y` is rearranged into `y + x + 1`,

and

`(y + x) + 1 = (x + 1)+ y`

simplifies to

`y + x + 1 = y + x + 1`

which can then be simplified to `true` using the rule `X = X ->> true`.

> As well as rearranging commutative and associative operations, the transformation into canonical form reverses terms of the form `X > Y` and `X >= Y` into `Y < X` and `Y <= X`.
>
> Equalities of the form `X = -Y` are also rearranged into canonical form, so that, for example, `x = -y` can be recognized as being the same (for the purpose of simplification) as `y = -x`.

When designing simplification rules, we need to keep in mind that reordering is done *before* the left-hand side of a simplification rule is matched to the expression. For example, a rule of the form

```
X > X ->> false
```

would never be applicable, since expressions of the form X > X are automatically rewritten as X < X prior to simplification.

### *Computing Additional Rules for Commutative and Associative Patterns*

Suppose we have a simplification rule of the form

X ∘ Y --> Z .

If the operation ∘ is commutative, then Y ∘ X also simplifies to Z.

If ∘ is associative, then W ∘ X ∘ Y = (W ∘ X) ∘ Y = W ∘ (X ∘ Y) which simplifies to W ∘ Z.

To handle these additional cases, the `simplify` program generates the required additional simplification rules automatically. For example, if we assert the rule

```
1*A  ->> A.
```

`simplify` generates the additional rules

```
A*1 ->> A.
A*1*B ->> A*B.
A*B*1 ->> A*B.
```

as demonstrated in the following output:

```
% simplify

Version 1.4.6, August 21, 2000.
|: assert(1*A ->> A).

|: x*1.
x*1 ->>
   x

|: p*1*q.
p*1*q ->>
   q*p

|: a*b*1.
a*b*1 ->>
   b*a
```

### *Cancellation*

For associative and commutative operators which have an inverse, further simplification can be accomplished through *cancellation*. For example, in the term

(a + b + c) - a = ((a + b) + c) - a

the commutativity and associativity of + imply that

((a + b) + c) - a = ((b + c) + a) + (- a) = ((b + c) + (a + (-a)) = b + c.

This is not easily handled by rewrite rules since the term to be cancelled can be embedded arbitrarily deep in a larger term, so no finite set of simple patterns can recognize the range of cases in which cancellation is possible. But by adding cancellation to the preprocessing before simplification rules are applied, expressions such as a + b + c - a - c and a - c + b - a + c simplify to b.

Similarly, an expression such as a * b * c / a / c is automatically simplified to b by cancellation.

To allow cancellation of terms involving unary minus in the same way as reciprocals, expressions of the form -X are represented in canonical form as 0-X. This allows -X + X to cancel to 0, in the same way as 1/X*X cancels to 1.

## *4.3 Testing simplification rules*

The program `simplify` lets you test simplification rules on a list of terms which are input either from the terminal or from a data file. Rules are either entered interactively in response to the prompt by typing

`assert(`*`Rule`*`).`[5]

or are loaded from a theory file by entering the term

`theory('`*file specification*`').`

A theory file whose name is of the form *`name`*`.simp` can be loaded by entering the term `theory(`*`name`*`)`— the file extension `simp` is assumed if none is given. The program looks for the file first in your current directory and if not found, it uses a path specified in an initialization file in your home directory. (The form of the initialization file depends on the version of SVT used.)

As many theory files as desired can be loaded.

Rules entered using the theory or assert predicates are automatically checked to ensure that terms matching the right-hand pattern do not contain sub-terms which can also match the left-hand side, as in the following example:

[5] A syntactic note: if *Rule* is a conditional rule, it must be enclosed in an extra set of parentheses.

```
|:assert(X-Y ->> -(Y-X)).

!! $VAR(0)- $VAR(1)->> - ($VAR(1)- $VAR(0)) is circular.
```

(As in the `tautology` program, pattern variables are renamed internally.)

The following example illustrates the interactive use of `simplify`.

```
|: a/1.

a/1 ->>
   a/1    (since no simplification rules are loaded, the input is not simplified.)

|: assert(X/1 ->> X).
|: a/1.

a/1 ->>
   a
|: a - (a/1).

a-a/1 ->>
   a-a
|: assert((X - X ->> 0)).
|: a - a/1.

a-a/1 ->>
   0
```

To load data from a file, use the command:

`%simplify <`*data file*

The theory files to be included should be specified within the data file by terms of the form `theory('`*file specification* `').` In the following example, the file `testdata` includes a directive to load a file named `arithmetic.simp`, and `simplify` locates one in a course directory, which contains appropriate rules to simplify the input expression.

```
% cat testdata
theory(arithmetic).
3*a + 5*a + a.
% simplify <testdata
% loading file /cs/fac/bin/simplify
% simplify loaded in module user, 0.020 sec 78,768 bytes
Version 1.4.7, September 5, 2000.
Loading /cs/dept/course/2000-01/F/3341/arithmetic.simp

3*a+5*a+a ->>
   a*9
```

If the rules being tested contain a loop, it may be necessary to interrupt the program with a control-C. At this point you are in the Prolog interpreter's debugger; to exit, type 'a' (for 'abort'), and then at the `|-?` prompt, type 'halt.' as illustrated in the following example:

```
% simplify < loopy.data
^C
Prolog interruption (h for help)? a
[ Execution aborted ]

| ?- halt.
%
```

**Exercise 4.1:** Create a file `equality.simp` with rules for equalities and inequalities, and a file `logic.simp` with rules for logic operators which will simplify a + b = b+ a to true, a < b < a to false, and a+ b = b+ a implies a < b < a to false. Demonstrate how to use `simplify` to do the simplifications.

**Exercise 4.2**: Use the program `simplify` in interactive mode to enter a rule which can simplify expressions of the form x**a/x**b to x**(a - b) and use it to simplify w**(3+b)/w**(b+1) to w**2.

**Exercise 4.3:** Without the additional simplification rules automatically generated for commutative and associative operators, the rule

```
0 * X ->> 0.
```

would fail to simplify the expressions x*y*0 and x*0*y to 0. Why?

### *4.4 Tracing the rewriting process*

The interaction among simplification rules, together with the rearrangement produced by canonical forms, the effect of cancellation, and the generation of commutative and associative rules can make it difficult to see why a specific simplification occurs or fails to occur. The `simplify` program has a trace feature which shows the effect of each simplification step.

To turn on the trace, include the term

```
traceOn.
```

in the data. To turn it off, include the term `traceOff`. (The preprocessing into canonical form and cancellation steps are not separately traced.)

Here is an example:

```
% simplify
Version 1.4.2SWI, January 6, 2007
|:theory(arithmetic).
Loading /cs/dept/course/2006-07/W/3341/arithmetic.simp
|:traceOn.

        true
traceOn ->>
   true
|:2*q+q*((a+8-a)-7) + q.

        2*q+q* (8-7)+q
        2*q+q*1+q
        2*q+q+q
        2*q+q*2
        q*4
2*q+q* (a+8-a-7)+q ->>
   q*4
```

### *4.5 Proving equalities*

We can directly apply simplification to one common type of proof problem: proving an equality by reducing both sides to an identity. To illustrate, we use `simplify` to establish some arithmetic equalities, using appropriate `arithmetic.simp` and `equality.simp` theory files. (We omit presenting the rules themselves, since they are all rather obvious mathematical facts.)

The data file is

```
theory(arithmetic).
theory(equality).
traceOn.

(2 + b) * ( b - 2) = b*b - 4.
(x - (a - 1)) * 3 = 3*(x-a)+ 3.
z**(k mod 2) * (z*z)**(k div 2) =  z **k.
```

and the output from `simplify` is:

```
indigo 331 % simplify < testdata
Version 1.4.2SWI, January 6, 2007
|:Loading /cs/dept/course/2006-07/W/3341/arithmetic.simp
Loading /cs/dept/course/2006-07/W/3341/equality.simp

        true
traceOn ->>
   true

        b**2-2**2=b*b-4
        b**2-4=b*b-4
        b*b-4=b*b-4
        true
(2+b)* (b-2)=b*b-4 ->>
   true

        (x-a+1)*3=3* (x-a)+3
        (x-a)*3+3=3* (x-a)+3
        true
(x- (a-1))*3=3* (x-a)+3 ->>
   true

        z** (2* (k div 2))*z** (k mod 2)=z**k
        z** (2* (k div 2)+ (k mod 2))=z**k
        z**k=z**k
        true
(z*z)** (k div 2)*z** (k mod 2)=z**k ->>
   true
```

# 5. APPLYING SIMPLIFICATION

## *5.1 Computing with ADTs*

Simplification rules can be usefully applied to abstract data types (ADTs). The fit between simplification and proving properties of ADTs is a natural one, since ADTs are specified by axioms in the form of equations which specify how various functions apply to instances of the datatype; and as we have seen, equations can often be proved fairly directly by an appropriate set of simplification rules.

The constraints established by the axioms determine the ADT; any object which satisfies the equations counts as an instance of that data type. As an example, the abstract data type stack satisfies the following equations[6]:

```
pop(push(X, S)) = S
top(push(X, S)) = X
```

In these equations

- S denotes a stack,
- X is a data value that can be pushed on or popped off the stack,
- top is a function that returns the top value of a stack,
- push returns a stack with a new element on the top, and
- pop returns a stack with the top element removed.

A specific stack can be represented by an expression which constructs it; for example, the stack containing a single element a is constructed by pushing a onto the empty stack nil; so it is represented by push(a, nil).

Additional functions which are useful in programming stack calculations can be defined in terms of the basic ones. For example:

```
dup(S) =  push(top(S), S)
swap(S)= push(top(pop(S)), push(top(S), pop(pop(S))) )
over(S)  = push(top(pop(S)), S)

plus(S) = push(top(pop(S)) +  top(S), pop(pop(S)) )
times(S) = push(top(pop(S))  *  top(S),  pop(pop(S)))
minus(S) = push(top(pop(S)) -  top(S),  pop(pop(S)))
divide(S) = push(top(pop(S)) / top(S),  pop(pop(S)))
```

In order to use the `simplify` tool to verify stack calculations, we create a file `stack.simp` in which these equations are converted to simplification rules by replacing the '=' with '->>''. The rules are used to replace terms matching the left-hand side of an equation by the corresponding term on the right.[7]

---

[6] See, for example, the discussion in Meyer, B., *Object-Oriented Software Construction*, Prentice-Hall: 1988, pp. 55-58.

[7] The equations for the stack operations describe a very rudimentary functional programming language . Indeed, simplification rules provide a simple mechanism for adding a form of

Here is a simple example which verifies that we can construct an expression which doubles the value on the top of a stack, by composing plus with dup and push:

```
% simplify
% loading file /cs/fac/bin/simplify
% simplify loaded in module user, 0.010 sec 26,224 bytes
Version 1.3.3, April 20, 1997
|:
|: theory('stack.simp').
|: plus(dup(push(a, nil))).

plus(dup(push(a,nil))) ->>
   push(a+a,nil)
|:
```

In this example, the value `nil` is interpreted as the empty stack, so push(x, nil) represents the stack consisting of the value x. The simplification verifies that the stack expression plus(dup(push(a, nil))) returns a stack containing the single element a + a.

For a more complex example, consider the following sequence of stack operations (the stack itself is not explicitly mentioned):

```
push a
push b
over
dup
times
minus
divide
```

To verify that these operations compute a / b - a*a, we convert the operations to an equivalent stack expression, and simplify it:

```
|: divide(minus(times(dup(over(push(b, push(a, nil))))))).

divide(minus(times(dup(over(push(b, push(a, nil))))))) ->>
      push(a/ (b-a*a),nil)
```

---

functional programming to Prolog, which does not otherwise support it. For a more comprehensive implementation of functional programming in Prolog using conditional rewrite rules, see Michael Newton, “A Combined Logical and Functional Programming Language”, California Institute of Technology, CS Technical report 5172:TR:1985.

For an extended discussion of stack expressions and term evaluation, see [Wand, 1980, Ch. 3.]

Simplification can also be used to diagnose some errors in a stack calculation:

```
|: over(minus(dup(push(t, nil)))).

over(minus(dup(push(t, nil)))) ->>
   push(top(nil), push(0, nil))
```

The fact that the expression for the resulting stack fails to simplify any further is due to the erroneous attempt to compute top(nil).

We can summarize the verification of ADT expressions as follows:

| ***evaluate an ADT expression*** |
|---|
| context: ADT defined by axioms in the form of identities |
| method: input ADT axioms and definitions to the `simplify` tool in the form of simplification rules; input an expression to be checked and see if it simplifies to the expected result. |
| background: mathematical simplification rules. The correctness of the evaluation is relative to the assumed correctness of the simplification rules and the correctness of the ADT axioms and definitions. |

**Exercise 5.1** Add rules to the theory file `stack.simp` described above, so that the erroneous stack operations top(nil) or pop(nil) simplify to a special value error. All operations on the error value should also yield error.

**Exercise 5.2** The programming language Forth[8] is stack-based. References to constants and variables implicitly push the values onto the stack, and operations are applied to values popped from the top of the stack. The arithmetic operations plus, minus, times, divide are written as the usual arithmetic operators +, -, *, /. Stack operations like DUP and OVER are used to avoid repeated memory references to variables, since stack operations are assumed to be faster.

As an example, the expression 1 + a/(b - a * a) would be computed in Forth by

```
1 A B OVER DUP * - / +.
```

Suppose the following Forth expression is intended to compute w - (x*x - 10*v)*x. Translate the Forth into a stack expression, and use a stack theory file based on the equations on p. 15 to verify that it leaves right thing on the resulting stack. If it doesn't, modify the Forth code to make it correct:

```
W X DUP *  10 V  * - OVER * -
```

(Remember to make the variables and operation names *lower-case* in the stack-expression.)

### *Translating Forth to stack expressions*

Translating Forth to stack expressions, as asked for in exercise 5.2, is a rather tedious process - it's exactly the sort of chore which should done by computer. Fortunately, we have a "secret weapon" ready to hand, in the form of simplification which can be easily adapted to the requirements of this problem!

Here's a `forth.simp` file that interprets a list as a Forth expression and translates it into a stack expression:

```
forth(S) ->> forth1(R) :- reverse(S, R).
forth1([dup | Rest]) ->> dup(forth1(Rest)).
forth1([over | Rest]) ->> over(forth1(Rest)).
forth1(['+' | Rest]) ->> plus(forth1(Rest)).
forth1(['-' | Rest]) ->> minus(forth1(Rest)).
forth1(['*' | Rest]) ->> times(forth1(Rest)).
forth1(['/' | Rest]) ->> divide(forth1(Rest)).
forth1([X | Rest]) ->> push(X, forth1(Rest)).
forth1([]) ->> nil.
```

[8] See Brodie, L., *Starting Forth,* Prentice-Hall: 1981. Forth has been used by vendors such as Sun and Apple to program Fcode device drivers (http://www.firmworks.com/www/plug-in.htm) and has served as the programming language for developing Open Firmware packages. See "Fundamentals of Open Firmware", Technical Note 1061, Apple Computer: 1996, http://developer.apple.com/technotes/tn/tn1061.html

A modern and more practical stack-language is *Factor*. See http://factorcode.org/index.fhtml.

Since Forth is a postfix language, evaluation of Forth expression works right to left, but lists are much easier to manipulate left-to-right, so the first rule reverses the input list, using a predicate `reverse` defined elsewhere.

The input routine for `simplify` requires elements of a list to be separated by commas. In addition, we need to down-case the variable names for the simplification rules; so the Forth expression

```
1 A B OVER DUP * - / +
```

gets entered as `[1, a, b, over, dup, *, -, /, +]`.

And here's what `simplify` does with it:

```
|: theory(stack).
|: theory(forth).
|: forth([1, a, b, over, dup, '*', '-', '/', '+']).

forth([1,a,b,over,dup,*,-,/,+]) ->>
   push(a/ (b-a*a)+1,nil)
```

**Exercise 5.3:** Java programs are typically compiled to a sequence of bytecodes which are interpreted by a Java Virtual Machine (JVM). The JVM is stack-based. Among its stack operations are the following, as described in [Ingram, 1997]:

```
"pop=87  Stack: ..., VAL -> ...
 pop2=88  Stack: ..., VAL1, VAL2 -> ...


 dup=89     Stack: ..., V -> ..., V, V
 dup2=92    Stack: ..., V1, V2 -> ..., V1, V2, V1, V2
 dup_x1=90  Stack: ..., V1, V2 -> ..., V2, V1, V2
 dup2_x1=93 Stack: ..., V1, V2, V3 -> ..., V2, V3, V1, V2, V3
 dup_x2=91  Stack: ..., V1, V2, V3 -> ..., V3, V1, V2, V3
 dup2_x2=94 Stack: ..., V1, V2, V3, V4 -> ..., V3, V4, V1, V2, V3, V4

 swap=95    Stack: ..., V1, V2 -> ..., V2, V1

 iadd=96  Stack: ..., INT1, INT2 -> ..., INT1+INT2

 isub=100 Stack: ..., INT1, INT2 -> ..., INT1-INT2

 imul=104  Stack: ..., INT1, INT2 -> ..., INT1*INT2"
```

Suppose the two top values of the stack are a, b. Construct a bytecode sequence of operations from the above list which computes $b^4 - a$, and verify that the sequence is correct. (The bytecodes are the integers associated with each stack operation. For example, the bytecode for pop is 87.) Try to make your sequence as short as possible.

**Exercise 5.4** *Queues* can be represented by terms whose functor is a left-associative operator ; we will use '--'. To add an element X at the end of a queue Q, we just write Q -- X. More formally:

add(X, Q) = Q -- X.

Unlike stacks which have elements added and removed at the same end, in a queue, items are removed from the *front*, so, with the aid of the conditional function[9] if, we define a function drop which returns a queue with its first element removed (if it is non-empty), and a function first which returns the first element of a non-empty queue.

drop(Y -- X) = if(empty(Y), nil, drop(Y) -- X)
first(Y -- X) = if(empty(Y), X, first(Y))

(a) What is Y's datatype in these axioms? Can X be a queue? How many elements does a queue of the form first(Y) -- X have?

(b) Construct a theory file `queue.simp` for queues with simplification rules for if, empty, drop, and first. For example,

empty(nil) ->> true.
empty(Y -- X) ->> false.

(c) Use `queue.simp` to verify that

first(nil -- a -- b) is a.

(d) Verify that the value of first(drop(nil -- a -- b) -- c) is b.

### *5.2 The `prover` program*

Simplification together with a tautology checker such as `wang` provides a basis for putting together a 'grow-your-own' theorem prover, which is easy to construct but is limited to just whatever mathematical knowledge you can supply in the form of simplification rules.[10]

The program `prover` is a theorem prover of this limited sort. It reads input terms in the same way as `simplify` does, and checks if the result of the simplification is a tautology, using Wang's algorithm. As a convenience, `prover` assumes that the theory files `arithmetic.simp`. `equality.simp`, and `logic.simp` will be needed and loads them automatically. It looks for these files in the current working directory; if there are no files with these names, it expects a Prolog initialization file `prolog.ini` in your home directory to specify a path to the simplification files.

Additional rules can be loaded using `theory('`*file specification*`')` or `assert((`*rule*`))`, in the same way as in `simplify`.

[9] if(B, X, Y) = X , if B; else Y.

[10] For a discussion of the role of simplification rules in a complete program verification system, see [Waldinger, 1974].

The `prover` program includes a further feature which allows it to verify propositions containing equalities. For example,

x=0 implies x<7

is shown to be a tautology by using the equality to replace the variable in the consequent term, yielding x=0 implies 0<7, which the `equality.simp` module will simplify to true.

Notice that not all equalities can be used to replace variables by values; in the false implication

not x<7 implies x=0,

it is obvious that we cannot replace the first x by 0, since that will change the truth value of the proposition. Instead of being false as the original proposition was, the implication

not 0<7 implies x=0

is trivially true.

So when is it legitimate to use an equality to eliminate an occurrence of a variable in a proposition? One case in which it works is if the equality occurs in the form:

(V = Expression) and P implies Q.

where V represents a variable and Expression does not contain V.

In this case, we can prove (V = Expression) and P implies Q. if we can prove the proposition that results from replacing occurrences of V by Expression in P and Q. To express this formally, we introduce some notation to describe substitutions, which will also be useful later:

*e*[*r* / *x*] = the expression resulting from substituting *r* for each occurrence of the variable *x* in the expression *e* (where *x* does not occur in *r*.)[11]

Now

(V = E) implies (X iff X[E/V]),

so if we can prove that P[E/V] implies Q[E/V] , it follows that

(V = E) and P implies Q.

(You can use `wang` to show that from

(V = E implies (P iff P[E/V])), and
(V = E implies (Q. iff Q[E/V])) and
(V = E and P[E/V]) implies Q[E/V] ),

one can conclude that (V = E and P implies Q).)

[11] This operation has the technical name β-*reduction*. See, for example, [Kubiak, p. 45].

This suggests where the checking for and application of equalities should take place—namely, at that point in the tautology checking where Wang's algorithm has reduced the sequent entirely to non-logical terms (see Section 3.5). If any of the terms on the left side of the sequent are equalities of the form V = Expression (where Expression does not contain V), then V is replaced by Expression in the other terms in the sequent. Each term in the resulting equivalent sequent is simplified and the result is checked to see if it is a tautology.

The techniques used in the prover program are not sufficiently powerful to deal with the complexities of mathematically interesting theorems, but as it happens, the logic of software verification is in many cases much less demanding than that required for interesting mathematics. Verification typically requires proving a series of small results involving many tedious details but no deep mathematical ideas. This makes them quite suitable for 'mechanical' proof using the ad-hoc techniques of simplification rules, together with variable replacement through equalities, followed by tautology checking.

### *Where do rules come from?*

The verification tools discussed in subsequent chapters rely on simplification rules supplied by the user to prove propositions about computations. If the rules are inadequate, the proof attempt fails and the proposition which was not reducible to a tautology is displayed. But the failure of the proof attempt does not mean that what we were trying to prove *cannot* be proved. We could always force a proof—just create a simplification rule to reduce the proposition to true! But of course this is "cheating"; verification must be based on rules which have a genuine justification, not just wishful thinking that what we are trying to verify must be the case.

So what should count as a legitimate justification for a simplification rule? We can distinguish three kinds of sources for rules:

**1.** *General mathematical theories* which describe the basic functions and operations used in the computation, such as arithmetic and logical operations. Arithmetic rules are usually derived from equalities between complex expressions and simple ones, e. g. X + 0 ->> 0. Rules involving relational or logical operators transform one proposition into another proposition:

0 <= x and 0 < x ->> 0 <= x,

or simplify a proposition to true,:

x < 0 or x = 0 or x > 0 ->> true.

These are usually based on "facts" we have learned about arithmetic relations or on theorems and lemmas which we can find proved in textbooks.

**2.** *Special theories* about abstract datatypes used in a computation, such as stacks and arrays. These rules are based on the axioms for a datatype in the form of identities, or on theorems which can be proved about the datatype, e. g.

push(top(S), pop(S)) ->> S (for non-nil S).

**3.** *Definitions*

e. g., first(Y -- X) = if(empty(Y), X, first(Y)).

In this case, the rule does not act as a simplification since the right-side is usually more complex than the left. The definition is used to expand a term in the hope that other rules will be able to simplify the expansion.

In each case, the rule is derived from concepts which we or someone else have *constructed*. The test for whether a rule is "correct" is not "is it true?" but "does it reflect the intended meaning of the concepts which it uses? does it say what we want it to say?"

### *Examples*

The following examples show prover in action on some very simple propositions entered from the keyboard (and echoed in the output):

```
% prover
. . .
Version 1.6.2, September 5, 2000.
Loading /cs/dept/course/2000-01/F/3341/arithmetic.simp
Loading /cs/dept/course/2000-01/F/3341/equality.simp
Loading /cs/dept/course/2000-01/F/3341/logic.simp

|: x**(a-a) = 1.
x** (a-a)=1

* Valid.

|: x*(a-a) = 0.
x* (a-a)=0

* Valid.

|: x=3 and x=y+3 implies y=0.
x=3 and x=y+3 implies y=0

* Valid.

|: traceOn.
traceOn

        true
* Valid.

|: x < x+1.
x<x+1

        0<1
        true
* Valid.
```

By loading an appropriate theory file, we can prove simple properties of functions on datatypes. As an example, the function dup is intended to have the properties that

top(dup(s)) = top(pop(dup(s))) and top(dup(s)) = top(s).

This can be verified directly by `prover`, using an appropriate theory file for the stack functions:

```
|: top(dup(s))=top(pop(dup(s))) and top(dup(s))=top(s).
top(dup(s))=top(pop(dup(s)))and top(dup(s))=top(s)

   top(pop(dup(s)))=top(dup(s))and
top(push(top(s),s))=top(s)
        top(push(top(s),s))=top(s)and
top(pop(dup(s)))=top(push(top(s),s))
        top(push(top(s),s))=top(pop(dup(s)))and
top(s)=top(s)
        top(push(top(s),s))=top(pop(dup(s)))and true
        top(push(top(s),s))=top(pop(dup(s)))
        top(push(top(s),s))=top(pop(push(top(s),s)))
        top(push(top(s),s))=top(s)
        top(s)=top(s)
        true
* Valid.
```

Using the `prover` tool gives us another verification scenario for ADTs:

| ***verify an intended property of an ADT*** |
|---|
| context: ADT defined by axioms and function definitions, and an intended property represented by a proposition involving the ADT functions, equalities, and additional mathematical functions and relations. |
| method: input ADT axioms and definitions to the `prover` tool, together with appropriate mathematical theories; input the proposition describing the intended property and show that it is valid. |
| background: prover's default mathematical simplification rules, or user-defined theories. |

**Exercise 5.5:** Create simplification rules based on the following mathematical 'facts' which will allow you to use `prover` to prove that 0 <= sqrt(x) <= x, for x a non-negative integer.

X <= sqrt(Y) iff X*X <= Y
sqrt(X) <= Y iff X <= Y*Y

Some additional rules may be needed as well. (Hint: use `traceOn` to see what effect your rules have on the simplification of the input.)

**Exercise 5.6:** If the functions which define an abstract datatype give exactly the same results on two objects, then the objects are equal, as far as the theory of that datatype is concerned. Use `prover` and the theory file `stack.simp` from Section 5.1 to show that if s is not nil, then

push(top(s), pop(s)) = s.

**Exercise 5.7:** The stack function over(s) is intended to have the property that its top element is equal to the second element below the top, and is equal to the first element below the top of s. Construct a proposition which expresses this property and verify it using `prover` and a stack theory file which contains just the basic axioms for push, top, and pop, and the definition of over.

### *5.3 Verification through induction*

As a variation on prover, we can construct a program which uses simplification and tautology checking to implement a much more powerful proof technique—*mathematical induction*—in a form which is directly applicable to the verification of certain kinds of functional programs[12]. Functional programs are written in functional languages, such as Lisp, whose notation is directly translatable into the specification of a mathematical function which gives the meaning or denotation[13] of the program. Thus, verification of a functional program involves proving an equality between the function computed by the program and an *intended function*[14].

Suppose we wish to verify the Lisp program:

```
(DEFUN product (n) (IF (zerop n) 0 (+ (product (1- n)) m)))
```

with respect to the specification that:

product(n) = n * m .

The Lisp function is easily converted into term notation: in general, (*F X*) represents a function call *F(X)*; `zerop` is a Boolean function that checks if its argument is 0 and `1-`

[12] The ideas in this section are found in [McCarthy, 1962] — perhaps one of the earliest papers on proving properties of programs. It is reprinted in [Colburn, 1993].
[13] See [Mueller, 1988, Ch. 1] for a discussion of the connection between denotational and functional programming languages.
[14] The use of intended functions in specification and verification is discussed in [Stavely, 1999].

subtracts 1 from its argument. So we can write an equivalent term expression for product(n) as

if( n=0, 0, product(n - 1) + m).

That is, if the argument is 0, then the value of the function is 0, otherwise the value is product(n - 1) + m.

How could we prove that, given this definition, the specification holds for all integers n ≥ 0?

The principle of *mathematical induction* provides the needed inference rule[15]—it states that if a proposition P(n) is true for integer n = 0, and in addition, that

(n > 0 and P(n - 1)) implies P(n)

then P(n) is true for all n ≥ 0.

In this example, we can take P(n) to be the proposition:

product(n) = n*m .

So to verify the definition of product with respect to the specification, we need to establish the two propositions required by the principle of mathematical induction:

*(i)* product(0) = 0 * m .
*(ii)* product(n - 1) = (n - 1) * m implies product(n) = n * m.

These are both easily proved using the definition of the product function given by either the Lisp code or the term expression. By definition, product(0) = 0, so *(i)* is satisfied, and if n > 0, then

product(n -1) = (n - 1) * m implies product(n - 1) + m = (n - 1) * m + m = n * m

which establishes *(ii)*.

The general pattern of the verification is as follows:

• identify the variable *V* in the if-condition and the function name *F*;

• show that if 0 is substituted for *V* in the specification, then the result equals the second argument of the if-expression (this corresponds to the induction *basis*).

• substitute *V* - 1 for *V* in the specification, and then substitute the result for any occurrences of the term *F(V - 1)* in the recursive definition of the function (this corresponds to the *inductive hypothesis*). Show that the result is equal to the value required by the specification. This corresponds to the *induction step*[16].

More formally, to verify

---

[15] See [Kubiak, 1991, Ch. 9, and 10] for a more extended discussion of mathematical induction in the context of verification.

[16] Compare [Backhouse, 1986, p. 118] where it is called the *inductive* ste*p*. The terminology used here corresponds to [Hehner, 1984, p. 46].

*F(V )* = if( *V* = 0, $V_0$, *Expression*) : *Goal* ,

where *Goal* is a functional expression, we need to prove the basis step:

*Goal* [ 0 / *V* ] = $V_0$

(here we reuse the notation for substitutions we introduced in Section 5.2.)

To prove the induction step, we need to prove that if *V* > 0, then

*Expression* [ *Goal* [*V* - 1 / *V* ]/ *F(V - 1)* ] = *Goal* .

This is the process used by the program `induction`. It accepts as input a term of the form

*F(V )* = `if(` *V* = 0, $V_0$, *Expression*) : *Goal* .

outputs the basis as an identity , and reports whether it can prove it; it then outputs the induction step as an identity, and reports whether that can be proved. Here is an example:

```
% induction
% loading file /cs/fac/bin/induction
% induction loaded in module user, 0.040 sec 65,324 bytes
Version 2.6.1, September 1, 2000
|: assert((A-B)*C ->> A*C - B*C).
|: f(n) = if(n=0, 0, f(n-1) + m) : n*m.
f(n)=if(n=0,0,f(n-1)+m):n*m

Base case: 0*m=0 is valid.

Induction step: n*m = (n-1)*m+m is valid.
```

In this case, the default `arithmetic.simp` was augmented with a small bit of algebra in order to simplify (n - 1) * m + m ->> n * m - 1 * m + m ->> n * m.

As we can see, the `induction` tool does not include the hypothesis that n > 0 in the induction step; for simplicity, it assumes that the hypothesis can be omitted and attempts to prove the more general case for an arbitrary integer n using unconditional simplification rules.

*For each of these exercises, try verifying using just the default theory files, and then add additional rules as needed, either by using* `assert` *or by creating your own theory files.*

**Exercise 5.8:** Use the program `induction` to verify that:

(a) if
sumN(n) = if(n=0, 0, sumN(n-1) + n)
then
sumN(n) = n*(n+1)/2.

(b)
f(i) = if(i=0, 0, f(i-1) + 2*i - 1) : i**2

**Exercise 5.9**: Verify that the Lisp program

```
(DEFUN t (n) (IF (zerop n) 1 (+ (t (1- n))
   (t (1- n)))))
```

computes $2^n$.

**Exercise 5.10:** (based on [Kubiak, p. 60].) Convert the following code into an equivalent definition in the form

h(n) = <term>:<goal>

which can be input to `induction` to verify that the function computed by the code is just the constant function 0.

```
int h(int n) {
     if (n == 0)  return(0);
     else if (n % 2 == 0)  return n*h(n-1);
     else return h(n-1);
}
```

Note: represent the operator % by a functional term in the input to `induction`.

**Exercise 5.11:** Show that the following code computes the predicate odd(i) which is true if i is odd, else false.

```
boolean oddf( int i) {
  return (i==0) ? false : ((i==1) ? true :
      ! oddf(i-1));
}
```

### *5.4 Generalized induction*

The pattern of recursion in which *F(V)* is defined in terms of an expression involving *F(V - 1)* for *V* > 0 is the form of recursion most directly linked to mathematical induction. But recursive definitions which do not fit this pattern can still be handled in a similar way, by extending the pattern of mathematical induction. For

example, the induction proof process in Section 5.3 won't work for a recursive definition such as

odd(n) = if(n = 0, 0, if(n = 1,1, odd(n - 2))) : n mod 2

in which the recursive expression does not contain *F(V - 1)* . But a variant of the verification process which uses a strengthened inductive hypothesis will do the job. The stronger form, which we will call *generalized induction*, reads as follows:

> if (0 ≤ m < n implies P(m)) implies P(n), then P(n) is true for all integer n ≥ 0.

In other words, if we can show P(n) under the assumption that P(m) holds for **all** m in [0, n), then we can infer P(n).

This version sometimes makes it easier to establish the induction step, since we get to assume more in the hypothesis (P(m) holds for **all** m in [0, n) rather than just for n-1), but it isn't really more powerful in a logical sense; it can be shown to be equivalent to the simple version involving n - 1[17].

To apply generalized induction to the verification of the example of odd(n), that is, to prove that

odd(n) computes n mod 2,

we need to assemble an implication with a number of hypotheses:

*(i)* (0 ≤ m and m < n implies odd(m) = (m mod 2)) implies
(0 ≤ n-2 and n-2 < n implies odd(n-2) = (n-2 mod 2))

(the instance is implied by the general case)

*(ii)* odd(n-2) = (n-2 mod 2) implies odd(n-2) = (n mod 2)

(mathematical fact about mod)

*(iii)* (n = 0 implies odd(n) = 0) and (n > 0 implies ((n = 1 and odd(n) = 1) and (n > 1 and odd(n)) = odd(n - 2))

(expansion of the 'if' expression in the definition of odd)

*(iv)* (n=0 and odd(n) = 0 implies odd(n) = (n mod 2)) and (n=1 and odd(n) = 1 implies odd(n) = (n mod 2))

(mathematical facts about mod)

*(v)* not n=1 and odd(n) = odd(n-2) and odd(n-2) = (n mod 2) implies odd(n) = (n mod 2)

(fact about equality)

*(vi)* 0 < n and not n=1 implies 0 ≤ n-2 and n-2 < n

[17]See [Kubiak, pp. 95-96].

(mathematical fact about integers)

*(vii)* $0 \le n$ implies $n=0$ or $0 < n$

(mathematical fact about <)

We can now use `prover` to show that the proposition

*(i) - (vii)* and $0 \le n$ and (($0 \le m$ and $m < n$ implies odd(m) = (m mod 2)) implies odd(n) = (n mod 2))

is a tautology, from which we infer, by generalized induction, that

odd(n) = (n mod 2), for $n \ge 0$.

Luckily, verification using generalized induction can made much easier than the above proof would suggest. We can simplify the process in much the same way as we did for ordinary induction, by abstracting out the common steps and incorporating them into the `induction` tool. Now, instead of using the induction principle to convert the induction step into the substitutions

*Expression [ Goal [V - 1 / V ] / F(V - 1) ] = Goal .*

the induction step becomes:

*Expression [Goal [ g(V) / V ] / F(g(V))]= Goal.*

where *g(V)* is the argument of F in the recursive expression.

In the previous example, *g(V)* = n-2, and the equality which must be proved to establish the induction step is

if(n=1,1,(n-2) mod 2)) = n mod 2

which is true for $0 < n$.

There is an additional complication in generalized induction—the substitution of the goal for the recursive call is only justified by the generalized induction principle for values which lie in [0, n). We took this for granted before, since if $n > 0$, n-1 is obviously in [0, n), but in generalized induction, it constitutes a separate proof obligation. So, to complete the verification of odd(n), we note that if $n > 0$ and is not 1, then $0 \le n - 2 < n$.

The following output shows what the `induction` tool does with the previous example (the simplification rules required to complete the verification have been omitted, so you can see what needs to be proved):

```
odd(n)=if(n=0,0,if(n=1,1,odd(n-2))):n mod 2

Base case: 0 mod 2=0 is valid.
0<n and not n=1 implies 0<=n-2 and n-2<n is valid.

Induction step: n mod 2=if(n=1,1,(n-2)mod 2) is valid.
```

To allow induction to be applied to a wider range of recursive definitions, we generalize somewhat further, by allowing the function being defined to have more than one argument (induction requires that the first be the recursion variable), and more than one recursive call in the recursive expression. An example is the function power(n, r), which uses a halving strategy to reduce the number of multiplications involved in computing r to the power n.

```
power(n,r)=if(n=0,1,if(even(n),power(n/2,r*r),
      r*power(n-1,r))):r**n

Base case: r**0=1 is valid.

0<n and even(n)implies 0<=n/2 and n/2<n is valid.

Induction step:if(even(n),r** (n/2)* r** (n/2),
      r** (n-1)*r)=r**n is valid.
```

The verification scenario for verifying a recursive function using generalized induction is given by:

| ***verify the definition of a recursive function*** *with non-negative integer argument* |
|---|
| context: a recursive definition is specified for a function $F(n, \ldots)$, for $n \geq 0$, along with a goal expression $G$. The definition specifies the value $V_0$ of the function for $n = 0$, and a recursive expression $E$ for the value of the function for $n > 0$, in terms of values of $F$ for smaller arguments. |
| method: input the term $F(n, \ldots) = \text{if}(n = 0, V_0, E) : G$ to the induction tool, along with appropriate simplification rules. |
| proof obligations:<br>(i) $G [ 0 / n ] = V_0$<br>(ii) $E[ G [A , \ldots / n, \ldots ]/ F(A, \ldots) ] = G$ , assuming $n > 0$<br>(iii) For every occurrence of $F$ in $E$ with first argument A (other than $n - 1$), prove that (in the context in which it occurs) $0 \leq A < n$. |
| background: mathematical simplification rules supplied as theory files or asserted interactively. The verification is relative to the assumed correctness of the simplification rules. |

In this scenario, we have explicitly identified the propositions which must be proved as *proof-obligations*, which we try to have the tool *discharge*, through appropriate simplification rules.

**Exercise 5.12:** Use the `induction` tool to verify that

e(n) = if(n = 0, 1, if(n = 1, 0, e(n-2)))

computes even(n) for n ≥ 0, where even(n) = 1 if n is even and 0 if it is odd.

**Exercise 5.13:** Translate the following code into the form required by the `induction` tool, and see if you can verify that it computes the constant function 1. What goes wrong? Can it be fixed? (We will encounter this problem again in section 9.8.)

```
int collatz(int n) {
   if (n == 0) return 1;
   if (n == 1) return 1;
   if (even(n)) return collatz(n/2);
   else return collatz(3*n + 1);
   }
```

### *5.5 Verification through transformations*

In most programming languages, the implementation of a recursive call requires the allocation of dynamic memory for the arguments of the call and the value returned by the call, so that the computation specified in the previous invocation can be completed. Thus, in the sumN function of exercise 5.8, each recursive call is not completed until the addition sumN(n-1) + n is completed; so space must be allocated and retained for the pending result of the addition, as well as for the argument, which cannot be "popped off" the call stack until the computation at level n is completed.

In general, to evaluate a function defined by the simple recursion of the previous section on an argument n - 1 with m additional arguments, will require n recursive calls and space for n intermediate values. This contrasts with an iterative algorithm, in which partial results and values of variables *replace* the previous values at each iteration, so no additional memory has to be allocated.

A more complex form of recursion called **tail recursion** imitates iteration by using an additional variable to pass along to the next recursive step a partial result from which a new (partial) result is computed. When the terminating condition is reached, the partial result is returned as the desired value.

As an example, the sumN function can be defined using a helper function with an extra argument:

sumN(n) = helper(n, 0)
helper(n, result) = if(n=0, result, helper(n - 1, result + n))

The recursion in helper is an example of tail recursion, also called 'last action' recursion, because in evaluating the recursive expression, no computation follows the recursion. In the implementation of this form of recursion, the value returned at each step can simply replace the value computed at the previous step, and the input arguments can be replaced by the new values—they will not be needed for any subsequent computation.

This produces a more efficient implementation, both in the amount of memory which must be allocated and the time spent in stack operations on the call stack.

A theorem of [Kubiak, 1991, p. 118] gives us a general set of conditions under which we can convert simple recursion into tail recursion. For verification, we want to reverse the process—by transforming a tail recursive function into the simple recursive form used by the `induction` tool, we can extend the scope of the tool to included more complex function definitions.

The transformation works as follows:

We start with a tail-recursive function with 2 arguments, serving as a helper function for a 1-argument function *F*:

*F(V ) = F(V, Initial)*.
*F(V, Result)* = if(*Exit*, *Result*, *F(reduce(V), build(Result, d(V))))*.

(We use the same name for the helper function as for the main function, distinguishing it from the main function by its greater arity.)

The pieces are reassembled to produce a simple recursive definition:

*F(V)* = if(*Exit*, *Initial, build(d(V),F(reduce(V))))*.

But we need some conditions to guarantee that the new definition agrees with the original one. We will use the following:

• *Initial* must be an identity for the operation *build*, i. e, *build(Initial, X) = X.* This allows the value for the non-recursive branch in the simple recursive definition to be the same as the initial partial result in the tail-recursive definition (for example, 0 if the result is being built up by addition.)

• *build* must be associative.

In our previous example,

sumN(n, result) = if(n=0, result, sumN(n-1, result + n))

so

*Exit* = (n=0),
*Initial* = 0,
*reduce*(n) = n - 1,
*d*(n) = n

and

*build*(x, y) = x + y,

yielding a simple recursive definition

sumN(n) = if(n=0, 0, sumN(n-1) + n).

Since + is associative and 0 + X = X, the new definition is equivalent to the tail-recursive version.

The transformation from tail to simple recursion requires as inputs:

- the definition of the tail recursive function,
- the intended goal function,
- an initial value.

These are packaged as input to the `induction` tool in the form:

`tail(`*Definition*`:` *Goal*`,` *Initial*`)`

As an example:

```
|: tail(if(n=0,s,sum(n-1,s+n)):n*(n+1)/2,0).
tail(if(n=0,s,sum(n-1,s+n)):n* (n+1)/2,0)

Assume + is associative.
0+X=X is valid.
X+0=X is valid.

Verifying: sum(n)=if(n=0,0,n+sum(n-1)):n* (n+1)/2
Base case:
 Cannot prove 0/2=0.

Induction step:
 Cannot prove (n**2+n)/2= (n-1)*n/2+n.
```

(To complete the verification, use the simplification rules from Exercise 5.8.)

The construction of the simple recursive expression from the tail-recursive has one rather subtle aspect—the analysis of the expression which computes the partial result as a function of two arguments *Result* and *d(V)*. The role of the function *d* here is to allow us to treat the tail-recursive expression as a function of *V* even if *V* does not appear in it. For example, suppose

f(n, r) = if(n=0, r, f(n -1, r+a)).

Take *d*(n) = a, and define *build*(x, y) = x + y. Then *build* meets the conditions, and we can construct an equivalent simple recursive expression using

*build*(*d*(n), r) = a + r.

To see the role which the associativity and identity conditions play in the transformation, let us 'unroll' a tail-recursive and a simple recursive definition for a few steps and compare:

Let f(n) = f(n, e) where f(n, r) = if(n=0, r, f(n-1, r ∘ n)), so
*Initial* = e, *reduce*(n) = n-1, *d*(n) = n, and *build*(x, y) = r ∘ n.

Now unroll the computation of f(3) = f(3, e):

f(3, e) = f(2, (e ∘ 3)) = f(1, (e ∘ 3) ∘ 2) = f(0, ((e ∘ 3) ∘ 2) ∘ 1) = ((e ∘ 3) ∘ 2) ∘ 1.

The transformation into simple recursion yields

f(n) = if(n=0, e, n ∘ f(n-1)),

and unrolling f(3) gives:

$$f(3) = 3 \circ f(2) = 3 \circ (2 \circ f(1)) = 3 \circ (2 \circ (1 \circ f(0))) = 3 \circ (2 \circ (1 \circ e)) .$$

So if e is an identity for $\circ$, and $\circ$ is associative, then the two expressions are equivalent.

We sum up the discussion in the form of a verification scenario for tail-recursive functions:

| ***verify the definition of a tail-recursive function with non-negative integer argument*** |
|---|
| context: a recursive definition is specified for a function *F(V, result,. . )*, for $V \geq 0$, along with a goal expression *G*. The function to be verified is assumed to be defined by *F(V, . . ) = F(V, Initial,. . )*.<br><br>The value of *F(V, result, . . )* is specified to be *result* if $V = 0$, otherwise<br>*F(V, Result) = F(reduce(V), build(Result, d(V))*). |
| method: input the term `tail(`*Definition*`:` *Goal*`,` *Initial*`)` to the `induction` tool, along with appropriate simplification rules to discharge the proof obligations |
| proof obligations:<br>(i) *build* is associative (this is assumed to be known on the basis of general mathematical knowledge or a separate proof)<br>(ii) *build(Initial, X) = X* and *build(X, Initial) = X*<br><br>(iii) The expression *F(V)* = if(*Exit*, *Initial, build(d(V)*, *F(reduce(V)*))) can be verified to compute *G.* |
| background: mathematical simplification rules supplied as theory files or asserted interactively. The verification is relative to the assumed correctness of the simplification rules. |

**Exercise 5.14:**

**(a)** Use the induction tool to verify that if the function `t` is defined by

```
int t(int n, result){
   return (n==0 ?  result :  t(n-1, 2 * result));
}
```

then `t(n,1)` returns $2^n$.

**(b)** Let t(n, r) be the recursive function you verified in (a). What corresponds to the functions *build* and *d* in the transformation from tail- to simple recursion? Show how to use the induction tool to check that your definitions are correct.

### *For-loops*

For-loops which count down by 1 correspond to tail-recursive functions which reduce the recursion variable by 1. The loop

`for(`*X = Initial*`, int` *i = V*`;` *i* `> 0;` *i*`--)` *X = g(*`i,` *X)*`;`

corresponds to the tail-recursive function

*f(V, X)* = if(*V*=0, *X*, *f(V*-1, *g(V, X)))*, with *f(V) = f(V, Initial).*

and so is verified by using the `induction` tool on the input

`tail(if(`*V*=0, *X*, *f(V-1, g(V, X)))*`:` *Goal*, *Initial*`)`.

**Exercise 5.15:** Translate the code fragment

```
for(p = 1,  int i = n; i > 0; i--) p = p*i;
```

to a tail-recursive expression and use the `induction` tool to verify that it computes the factorial function n! for n ≥ 0.

Most for-loops, however, count up. A small variation of the previous pattern allows us to handle this as well. The loop

`for(`*X = Initial*`, int` *i* `= 1;` *i* `<` *V* `+ 1;` *i*`++)` *X = g(i, X)*`;`

corresponds to a tail-recursive expression with an additional parameter:

*f((Index, X, V)* = if(*Index = V*, *X*, *f(Index*+1, *g(Index, X), V))*,

with *f(V)* = *f(*0, *Initial, V*). To verify that *f(V)* computes a particular *Goal* function, we input

`tail(if(`*Index* = V, X, *f(Index*+1, *g(Index, X), V))*: *Goal*, *Initial*`).`

to the `induction` tool which converts it to the simple recursive equivalent:

*f(V)* = if(0, *Initial, g(f(V-1), V)*).

and verifies that.

As an example, we can represent

```
for(p = 1, int i = 1; i < n + 1; i++) p = x*p;
```

as the tail-recursive definition

tail(if(i = n, p, power(i + 1, x*p, n)): x**n, 1)

and input it to `induction`, resulting in the output:

```
Assume * is associative.
X*1=X is valid.
1*X=X is valid.

Verifying: power(n)=if(n=0,1,power(n-1)*x):x**n
Base case: x**0=1 is valid.

Induction step: x**n=x** (n-1)*x is valid.
```

**Exercise 5.16:** Verify that the code fragment

```
for(p = 1,  int i = 1; i < n + 1; ; i++) p = p*i;
```

computes the factorial function n! for n ≥ 0.

(You can use `(n!)` to represent the goal function in the `induction` tool.)

**Exercise 5.17:** Construct verification scenarios for `for`-statements in C/Java, which count either up or down.

### *5.6 Structural induction*

Mathematical induction, as defined so far, is an inference rule for propositional functions with integer arguments ≥ 0. The idea can be extended to induction over recursively defined data structures, in which form it is called *structural induction*.[18]

Structural induction is used to establish the truth of a proposition P(S), for all objects S of a particular compound datatype, such as stack or string.

Assume that the datatype has some minimal element, call it λ—for strings, λ is the empty string; for stacks, it is the empty stack which we earlier called nil. Suppose some operation f is defined on the datatype which has the property that if iterated often enough, it produces the element λ; for stacks, pop is such an operation. Then the principle of structural induction states that if P(λ) is true, and

P(f(S)) implies P(S), for all S,

then P(S) is true for all S.

Notice the analogy with induction over natural numbers: the minimal element is 0 in that case, and f(n) = n - 1.

In structural induction over a stack, the minimal element (corresponding to 0 in induction over the natural numbers) is nil. The induction step, which for integers was an the implication from P(n - 1) to P(n), is now an implication from P(pop(stack)) to P(stack). In structural induction over queues, λ = nil and the operation f = drop.

### *Computing stack size*

As an example, consider a function count which counts the number of elements on a stack. Using the notation of the previous section, we can define this by:

count(stack) = if(stack = nil, 0, 1 + count(pop(stack))).

What does it mean to verify this? We need a specification of a mathematical function, say, size(stack), which count is supposed to compute.

We define size by simplification rules:

```
size(nil) ->> 0.
size(pop(X))+1 ->> size(X).
```

Now `induction` (which knows about structural induction on stacks) can easily establish that count(stack) = size(stack) for all stacks:

[18]See [Dromey, 1989, p. 174.]

```
|: assert((size(nil) ->> 0)).
|: assert((size(pop(Stack))+1 ->> size(Stack))).
|: count(s)= if(s=nil,0,1+count(pop(s))):size(s).
size(nil)=0 is valid.
size(s)=1+size(pop(s)) is valid.
```

You may feel that not much has been established by this verification; the functional expression and the specification are so obviously parallel that there is hardly anything to prove; the induction step turned out to be identical to a defining property of the size function. Indeed, one virtue of programs expressed in functional terms is precisely that they often function as their own specification. To put it another way, specifications expressed in a functional language can be directly executed by an interpreter for that language. This is one easy way to construct programs which are necessarily correct, since there is no distinction between the program code and the description of what the program is supposed to do. Learning to program would, in this case, just involve learning how to take informal specifications expressed, perhaps, in a natural language such as English or in a mixture of English and mathematical notations, and translate them into a language of executable formal specifications.

That conventional programming is so difficult to verify and, as a result, is so prone to error is a direct consequence of the "semantic gap" between the notation of ordinary programming languages and a logical specification language—a gap which seems to the programmer to be required for practical reasons, but whose hidden costs from the difficulty of verification are rarely counted. In the subsequent chapters, we shall see more clearly how this gap arises for procedural languages such as C or Java, and what additional concepts are needed to bridge it.

**Exercise 5.18:** Suppose the stack function double is defined by the equations:

```
    double(nil) = nil
double(push(X, S))  =  push(X, push(X, double(S))).
```

Use `induction` to verify that a recursive function which, when given a stack as argument, calls itself on the result of popping the top element off, followed by pushing the top element twice on the result, computes the function double.

### *Computing the reverse of a list*

Lists, as abstract datatypes, are the same structures as stacks, with relabeled operations:

- the empty list is represented by [];
- the push(X, S) operation becomes cons(X, S);
- top(S) becomes first(S),
- pop(S) becomes rest(S),

and the axioms for lists are just those for stacks translated into the list vocabulary.

A convenient piece of 'syntactic sugar'[19] represents

- a list by [X, Y, . . ],
- cons(X, Y) by [X | Y], and
- cons(X, []) by [X].

We use the list vocabulary in the following example, in which we verify that a particular functional program, that is, a program defined by a functional expression, *reverses* a list. To do this we need an additional definition of the reverse of a list as a mathematical function which we can use in a specification. Again, it is hard to conceive of a simple way to express such a definition which does not itself directly parallel a recursive computation. To make the verification less trivial, we will define the reverse function in one way and the functional program in another, and then use induction over lists to verify that the specification in terms of the mathematical function and the function program agree.

We define first a function append to represent the operation of concatenating the elements of one list onto another. This is given by the equations:

append([], Y) = Y,
append(X, Y) = [first(X) | append(rest(X), Y)] if X is not [].

If the equivalent rewrite rules are entered in this order, the second rule won't need a condition to exclude X = [], since if the first argument to append is empty, the first rule will apply, preventing the second from applying.

Now we can define the reverse of a list by the axioms

reverse([]) = [],
reverse([X | L]) = append(reverse(L), [X]).

Note that in this form, the second equation does not directly define how to compute reverse(L) for an arbitrary list. But since any list is either empty or equals cons(X, L) for some element X and list L, the equations cover all cases.

**Exercise 5.19:** Convert `stack.simp` to `list.simp`, changing `push` to `cons`, `pop` to `rest`, and `top` to `first`. Use the following axioms for append and reverse:

append([X | Y], L) = [X | append(Y, L)].
reverse([X | Y]) = append(reverse(Y), [X]).

Add appropriate rules for the empty list []. Then use the simplify program to evaluate the following list expressions:

append([a, b], [x])
reverse([a, b, c])
reverse(append(reverse([a, b]), [z]))

[19] The notation is borrowed from the Edinburgh syntax for Prolog.

A tail-recursive reversal function rev(list) is defined using the equations

```
rev(list) = rev(list, []).
rev(list, result) =
     if(list = [], result, rev(rest(list), cons(first(list), result))).
```

We want to verify that rev(list) = reverse(list) for all lists. The function rev(list, result) fits the tail recursive pattern, so we can try to verify it using the `induction` tool on the input:

```
|: theory(list).
|: tail(if(list = [], result, rev(rest(list),
       cons(first(list), result))):reverse(list), []).
```

The output from `induction` begins

```
Assume cons is associative.

Cannot prove cons(X,[])=X.

Cannot prove cons([],X)=X.

Verifying:
rev(list)=if(list=[],[],cons(rev(rest(list)),first(list))):
     reverse(list)
```

This shows us that there is a problem with the formulation of the rev function, for the purpose of verification, because, in fact, cons is **not** associative. (Both operands of an associative operation must have the same type, since in the equation

$$(a \circ b) \circ c = a \circ (b \circ c)$$

the middle operand appears both as the second and the first operand of the operation. But cons takes a list *element* as the first argument, and a *list* as the second.)

However, append is associative (although proving it requires a non-trivial inductive argument), and we can show by direct calculation that

cons(X, S) = [X|S] = append([X], S).

So we rewrite rev, replacing the cons with an append, and now get

```
|: theory(list).
Loading /cs/home/fac2/peter/3341/list.simp
|: tail(if(list=[], result, rev(rest(list),
     append([first(list)], result))): reverse(list), []).

tail(if(list=[],result,rev(rest(list),append([first(list)],res
ult))):reverse(list),[])

Assume append is associative.

Cannot prove append(X,[])=X.
append([],X)=X is valid.

Verifying:
rev(list)=if(list=[],[],append(rev(rest(list)),[first(list)]))
:reverse(list)
Base case: reverse([])=[] is valid.

Induction step:
Cannot prove reverse([first(list)|rest(list)])=reverse(list).
```

What does it take to complete the verification?

The first step is to show append(X, []) = X. One approach is to see if we can verify that the recursive definition of append yields the identity function if the second argument is [], and we can use the `induction` tool for this, together with an appropriate `list.simp` file based on the definitions above.

```
|: theory(list).
Loading /cs/home/fac2/peter/3341/list.simp
|: append(x, []) = if(x=[], [], [first(x)|append(rest(x),
[])]):x.

append(x,[])=if(x=[],[],[first(x)|append(rest(x),[])]):x

Base case: []=[] is valid.

Induction step:
 Cannot prove [first(x)|rest(x)]=x.
```

The last line follows immediately from the definitions of first and rest for non-empty lists and asserting the rule

```
[first(S) | rest(S)] ->>S
```

will complete the verification. So we can now add

```
append(X, []) ->> X
```

to the theory of lists. And the rule [first(S) | rest(S)] ->>S will then also complete the verification of reverse as well!

(As with induction over integers, the `induction` tool assumes that in the induction step the induction variable does not equal the base case, so the required rule can be expressed unconditionally.)

Notice that the original definition for rev:

```
rev(list, result) =
        if(list = [], result, rev(rest(list), cons(first(list), result))).
```

is now verified as well since cons(first(rest(list)), result))) equals append([first(rest(list))], result) and so can be substituted for it in the verified definition. Modifying the original definition was only necessary to allow us to reduce the tail-recursion to simple recursion which could then be verified by mathematical induction.

**Exercise 5.20:** Prove that [x | s] = append([x], s). What rules from your list theory file were used and how were they applied?

**Exercise 5.21:** Locate or construct an inductive proof that appending lists is associative and adapt it to make an argument that pushS(X, Y) = append([X], Y) is associative.

**Exercise 5.22:** State the base case and induction step for an induction proof that append is associative. Use `prover` to help prove the induction step.

**Exercise 5.23:** Construct and verify a tail-recursive implementation of the double function in Exercise 5.16.

### *Induction over trees*

In the previous examples of structural induction, the datatype (stack or list) was "1-dimensional", so that only a single function such as pop or rest was needed to compute the predecessor argument, pop(stack) or rest(list), in the induction step. For more complex datatypes such as trees, we need, instead of a single predecessor function, multiple predecessors, $f_1, \ldots, f_n$, and the structural induction step becomes

$$P(f_1(S)) \text{ and } P(f_2(S)) \text{ and } \ldots \text{ and } P(f_n(S)) \text{ implies } P(S) .$$

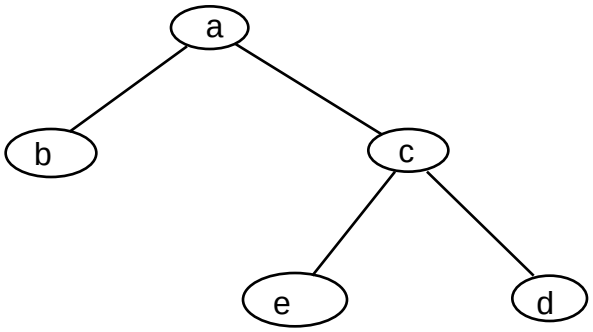
The induction tool can create the appropriate induction step for the case of *binary trees.*, in which every node is either a leaf (no subtrees) or has two subtrees. The base case is specified by the Boolean predicate leaf(t), which is true if t is a leaf node. The structural induction is based on two functions, left and right, which return the two subtrees of their argument. These functions together play the role of a "predecessor" function, because, if repeatedly applied to the values they return, they eventually terminate in a leaf node..

A classic problem for binary trees is proving that the leaves function which counts the number of leaves of a tree is one greater then the nodes function which counts the number of nodes internal to the tree, i. e., not leaves. Before verifying this with `induction` tool, let 's do some computations of the leaves function, using simplify.

We can present the non-nil nodes of a labelled binary tree as lists with either 1 or 3 elements:

> [Label] represents a leaf node; it has no sub-trees.
>
> [Label, Left, Right] represents an internal node with a label; Left and Right are the two sub-trees.

Thus the expression [a, [b], [c, [e], [d]]] represents the tree structure:




The leaves function is defined recursively by the equations

```
leaves([Label]) = 1,
leaves([Label, Left, Right])  = leaves(Left) + leaves(Right).
```

If we turn the equations into a theory file

```
leaves([Label])  ->> 1.
leaves([Label, Left, Right])  ->>  leaves(Left) + leaves(Right).
```

we can count the number of nodes in a tree using `simplify`:

```
% simplify
% loading file /cs/fac/bin/simplify
% simplify loaded in module user, 0.020 sec 79,436 bytes
Version 1.4.10, August 6, 2001.
|: theory(tree).
Loading /cs/home/peter/3341/tree.simp
Loading /cs/dept/course/2000-01/F/3341/arithmetic.simp
|: leaves([a, [b], [c, [e], [d]]]).

leaves([a,[b],[c,[e],[d]]]) ->>
      3
```

We can define the nodes-counting function recursively:

```
nodes(t) = if(leaf(t), 0 , 1 +  nodes(left(t)) + nodes(right(t))).
```

We would like to verify that nodes(t) = leaves(t) - 1 using the induction tool. Since the base case is here specified by a Boolean function leaf, rather than a specific value,

induction treats it differently than in previous cases. It detects the base case leaf(t) and and interprets the induction as being over binary trees. The base case and induction step are constructed as shown in the output:

```
% loading file /cs/fac/bin/induction
% induction loaded in module user, 0.030 sec 90,376 bytes
Version 2.8, January 25, 2002
Loading /cs/dept/course/2000-01/F/3341/arithmetic.simp
Loading /cs/dept/course/2000-01/F/3341/equality.simp
Loading /cs/dept/course/2000-01/F/3341/logic.simp
|: nodes(t)=if(leaf(t), 0, 1 +nodes(left(t))+
       nodes(right(t))):leaves(t)-1.

nodes(t)=if(leaf(t),0,1+nodes(left(t))+nodes(right(t))):
      leaves(t)-1

Base case:
Cannot prove leaf(t)implies leaves(t)=1.

Induction step:
Cannot prove leaves(t)-1=leaves(right(t))-
1+leaves(left(t)).
```

What rules can we use to discharge the two proof obligations? The first is easy; we can certainly assert that leaf(T) implies leaves(T)=1 ->> true. For the second, we use the definition of leaves:

```
|: assert(leaves(right(T)) + leaves(left(T)) ->>
leaves(T)).
```

and we need a rule to simplify the arithmetic equality

```
|: assert(X - Y = Z - Y + W ->> X = Z + W).
```

to yield the desired proof:

```
Base case: leaf(t)implies leaves(t)-1=0 is valid.

Induction step: leaves(t)-1=1+ (leaves(left(t))-1)+
(leaves(right(t))-1) is valid.
```

**Exercise 5.24:** Explain carefully and in detail how the proposition in the induction step is computed, using the notation for substitution *E[S/V]*. (Compare the description of the proof obligations in the verification scenario for recursive functions in Sec. 5.4.)

# 6. VERIFYING ADT IMPLEMENTATIONS

## *6.1 Implementing a stack by an array*

One abstract datatype can often be usefully implemented in terms of another—the example we will explore here is that of a *stack* implemented as an *array*. The question before us is the correctness of the implementation. What do we mean by 'correctness' here? We mean that the defining conditions for the original ADT still hold in the implementation. In Section 5.1, we gave just two conditions for a stack:

```
pop(push(X, S) = S
top(push(X, S)) = X
```

To show that an implementation of this datatype is correct, we must be able to prove that these axioms hold in the implementation.

A more elaborate specification of the stack datatype would include additional conditions, governing for example, the empty stack, as in Exercise 5.1, or defining a size attribute. But omitting such conditions does not invalidate the concept of stack given by the two axioms; it just means that the datatype they define is a *different* version of the stack concept. Correctness of an implementation is not an absolute property of the implementation; it is always relative to the choice of axioms for the ADT being implemented.

### *The array datatype*

Before we can discuss the implementation of stacks in terms of arrays, we have to specify the abstract datatype *array*. Instead of the usual notation X[i] for the i'th value in the array X, we will use a more mathematical notation in which the value at index i is given by the function array(X, i). [20] Assignment of a value to an element of an array is represented by a function change(Array, Index, Value) which returns a copy of the Array argument in which the value at location Index is changed to Value. This is similar to the approach we took with the stack functions; operations defined on the abstract datatypes are *mathematical functions*, not actions which alter the state of a computation—they have no side effects.

A single condition characterizes the datatype array for our purposes:

```
array(change(Array, I, Value), J) = if (J = I , Value, array(Array, J))
```

In any actual implementation of an array, the array index will be constrained to a finite interval, but to simplify our examples, we will just assume that it is non-negative.

Now suppose the datatype *stack* is implemented as an array together with an index that points to the top of the stack. We can package the array and the index together as a pair [Array, Index] with Index > 0 if the stack is not empty.

[20] The array function corresponds to the vector-ref function in Scheme.

We will need some functions which extract the array and index of a stack in its array implementation:

```
a([A, I]) = A
i([A, I]) = I
```

We can define i(nil) to be 0, but what is the array which implements the empty stack? that is, what is a(nil)? We just need a name for it, for example,

```
nil = [nilA, 0].
```

To simplify the specification of the translation from the abstract stack to its array implementation, we will translate the stack operations to a new set of operations pushA, popA, and topA, defined in terms of the array implementation.

The implementation is given by the conditions[21]:

```
pushA(X, S) = [change(a(S), i(S)+1, X), i(S)+1]
popA(S) = [a(S), i(S)-1]
topA(S) = array(a(S), i(S))
```

To verify this implementation, we need to prove that the conditions that relate push, pop and top in the abstract stack datatype still hold when these functions are translated, *via* the implementation, into conditions on arrays.

For this purpose, we construct a theory file for the `prover` program. The first part of the file defines the auxiliary functions a and i, and translates push, pop and top expressions into their proposed implementations:

```
a([A, I]) ->> A.
i([A, I]) ->> I.

push(X, S) ->> pushA(X, [a(S), i(S)]).
pop(S) ->> popA([a(S), i(S)]).
top(S) ->> topA([a(S), i(S)]).
```

The next part of the file shows how the functions pushA, popA and topA are implemented:

[21] To keep the example simple, we omit conditions which would ensure that the stack index is non-negative.

```
pushA(X, S) ->> [change(a(S), i(S)+1, X), i(S)+1].
popA(S) ->> [a(S), i(S)-1] :- prove(i(S) > 0).
topA(S) ->> array(a(S), i(S)).
```

We add rules for the array axioms:

```
array(change(Array, I, Value), J) ->>
     Value if prove(I=J).
array(change(Array, I, Value), J) ->>
     array(Array, J) :- prove(not(I = J)).
```

These conditional rules use a special predicate `prove`, defined in `simplify` and `prover`, which succeeds if its argument simplifies to true. The rules assume that the indices `I` and `J` are ≥ 0 and that there is no upper bound to the value of an index.

**Exercise 6.1:** What is the purpose of the `prove` predicate in the conditions in the above rules ? What if the first condition was changed to just "`I = J and I >= 0`"?

Suggestion: experiment. Try different variations of the rules on array expressions and see what happens.

### *Verifying the pop axiom*

Let us see what pop(push(X, S)) simplifies to (given appropriate rules):

```
|: pop(push(x,s)).

pop(push(x,s)) ->> [change(a(s),i(s)+1,x),i(s)]
```

If we can prove that [change(a(s),i(s)+1,x),i(s)] = s, then we will have verified that the condition

pop(push(X, S) = S

holds in the array implementation of stacks.

But for this we need a definition of equality for the stack implementation—a definition which determines when [A, I] = [B, J].

A reasonable definition of stack equality in the array implementation is that the stack indices should be equal and that corresponding array elements should be equal. More precisely,

[A, I] = [B, J] if I = J and 0 ≤ k ≤ I  implies array(A, k) = array(B, k).

Since the calculation above shows that the index of pop(push(x, s)) in the array implementation is i(s), we only need to prove the equality of the arrays; that is, we need to establish

0<=j and j<=i(s) implies array(change(a(s),i(s)+1,x), j) = array(a(s),j).

We can see that this holds, using a verbal argument rather than a formal calculation, by observing that since 0<=j, j is a valid index for the array. So we can use the rules for array(change(Array, I, Value), J) , and since j ≤ i(s), j must be unequal to i(s)+1, which gives

array(change(a(s),i(s)+1,x), j) = array(a(s),j).

With this as a rationale, we can assert

```
[change(A,I+1,X),I] ->> [A, I].
```

to complete the verification of the pop axiom.

### *Verifying the top axiom*

Finally, we need to verify that top(push(x, s)) = x. This can be calculated directly using the simplification rules given above. (In the example shown below, the rules are loaded from a file `astack.simp`.) As the trace shows, attempting the calculation manually would both error-prone and tedious:

```
% simplify
. . .

|: theory(astack).
|: theory(arithmetic)
|: theory(equality).
|: traceOn.
|:
|: top(push(x,s)).
topA([a(push(x,s)),i(push(x,s))])
array(a([a(push(x,s)),i(push(x,s))]),i([a(push(x,s)),
      i(push(x,s))]))
array(a([a(push(x,s)),i(push(x,s))]),i(push(x,s)))
array(a([a(push(x,s)),i(push(x,s))]),
      i(pushA(x,[a(s),i(s)])))
array(a([a(push(x,s)),i(push(x,s))]),
      i([change(a([a(s),i(s)]),i([a(s),i(s)])+1,)
array(a([a(push(x,s)),i(push(x,s))]),i([a(s),i(s)])+1)
array(a([a(push(x,s)),i(push(x,s))]),i(s)+1)
array(a(push(x,s)),i(s)+1)
array(a(pushA(x,[a(s),i(s)])),i(s)+1)
array(a([change(a([a(s),i(s)]),i([a(s),i(s)])+1,x),
      i([a(s),i(s)])+1]),i(s)+1)
array(change(a([a(s),i(s)]),i([a(s),i(s)])+1,x),i(s)+1)
true
x

top(push(x,s)) ->>
   x
```

**Exercise 6.2:** What use was made of the `arithmetic.simp` module in the above simplification?

Where did the **`true`** in the trace come from; i. e. what term simplified to `true` and what rule was used?

### *6.2 An array implementation of queues*

Verifying an ADT implementation is complicated enough that we better do another example. We sketch the verification of an implementation of *queues* in terms of arrays, with some parts left as exercises for the reader.

To implement a queue with an array, we can use the same approach we used for stacks; namely, we represent a queue as a pair [Array, Index], where Index is an integer ≥ 0 pointing to the last element in the queue. We use nilQ to represent the empty queue and nilA as the empty array, and as we did for stacks, we define "getter" functions to extract the array and index parts of a queue:

```
a([Array, Index]) = Array,
i([Array, Index]) = Index.
```

We can implement the empty queue nilQ in the same way as we used for the empty stack:

nilQ = [nilA, 0].

We need implementations for the queue operations:

add(X, Q),

and

drop(Q).

Let us call the implementations addA and dropA respectively. For addA, a natural definition is

addA(X, Q) = [change(a(Q), i(Q) + 1, X), i(Q) + 1].

Informally, this expresses the idea that if we use an array to represent a queue, then, to add an element, the location of the end of the array is increased by one and the added item is stored in the last location of the array. But note that, from a formal perspective, nothing is "assigned" or "stored"; the definition just describes how to compute a queue data object as the value of addA.

**Exercise 6.3:** Define an array implementation, dropA, of the queue operation drop.

**Exercise 6.4:** Use the array implementation of a queue to define a function last which returns the last element of a queue.

We can now pose the verification issue for this implementation of queues: is it *correct* with respect to the definition of 'queue' as an ADT (Ch. 5, Exercise. 5.4)? For this purpose, we'll need some axioms to characterize queues:

drop(add(X, Q)) = if(empty(Q), nilQ, add(X, drop(Q))).

and

last(add(X, Q)) = X.

The verification of the implementation will require showing that these equalities are still valid when the functions are mapped into their array implementations, which is left as an exercise.

**Exercise 6.5:** Use `prover` and an appropriate theory file to verify that the queue axioms hold when drop is implemented by dropA, add by addA, and last is implemented as in Exercise 6.4.

**Exercise 6.6:** In this exercise, we use the notation of Exercise 5.4 to represent queues as terms constructed from the infix operator -- , and assume an array implementation.

(a) Use `prover` and an appropriate theory file to verify that the axiom for first

first(Q -- X) = if(empty(Q), X, first(Q))

holds, if first(Q) is implemented by firstA(Q) = array(a(Q), 1). (Suggestion: find a general simplification rule to eliminate the if function in expressions of the form if(X, Y, Z) = V.)

(b) Suppose drop is implemented by

dropA(Q) = [shift(a(Q)), i(Q) - 1)]), if Q is not empty,

where shift is an array function defined by

array(shift(A), I) = array(A, I+1) if I ≥ 1.

Verify that the drop axiom

drop(Q -- X) = if(empty(Q), nilQ, drop(Q) -- X)

holds in the array implementation by showing that

shift(change(a(Q), i(Q) + 1, Y)) = change(shift(a(Q), i(Q), Y).

(Since this is an equality between arrays, it is established by showing that the i'th elements of each side are equal for 1 ≤ i ≤ i(Q).)

**Exercise 6.7:** The axiom which characterizes the change function for the array datatype is quite unspecific about what an array index is. The stack implementation assumes an index is an integer (or more precisely, something one can add and subtract 1 from), but the axiom is very general and allows any type of object as an index.

We could, for example, allow an index to be a pair [i, j] for the purposes of defining a 2-dimensional array. These 2-dimensional arrays can then be implemented by 1-dimensional arrays with integer indices.

To define such an implementation in a verifiable form, construct simplification rules which

• translate a two-dimensional array object a into an object of the form [a1, rows], where a1 is a one-dimensional array whose indices are integers, and rows is the number of rows in the 2-dimensional array a,

• translate array(a, i) into an object of the form array1([a1, rows], rows * n + m), where i = [m, n], and array1 is the array function for one dimensional arrays.

• define an assignment function change(a, [i, j], x) for two-dimensional arrays in terms of the one-dimensional array assignment change1(a, k, x).

Add rules for "getter" functions (similar to the functions a and i in the array implementation of stacks above) which extract the parts of a two-dimensional array.

Use prover and your simplification rules to prove that the change axiom

I ≥ 0 implies array(change(Array, I, Value), J) = if (J = I , Value, array(Array, J))

holds in this implementation.

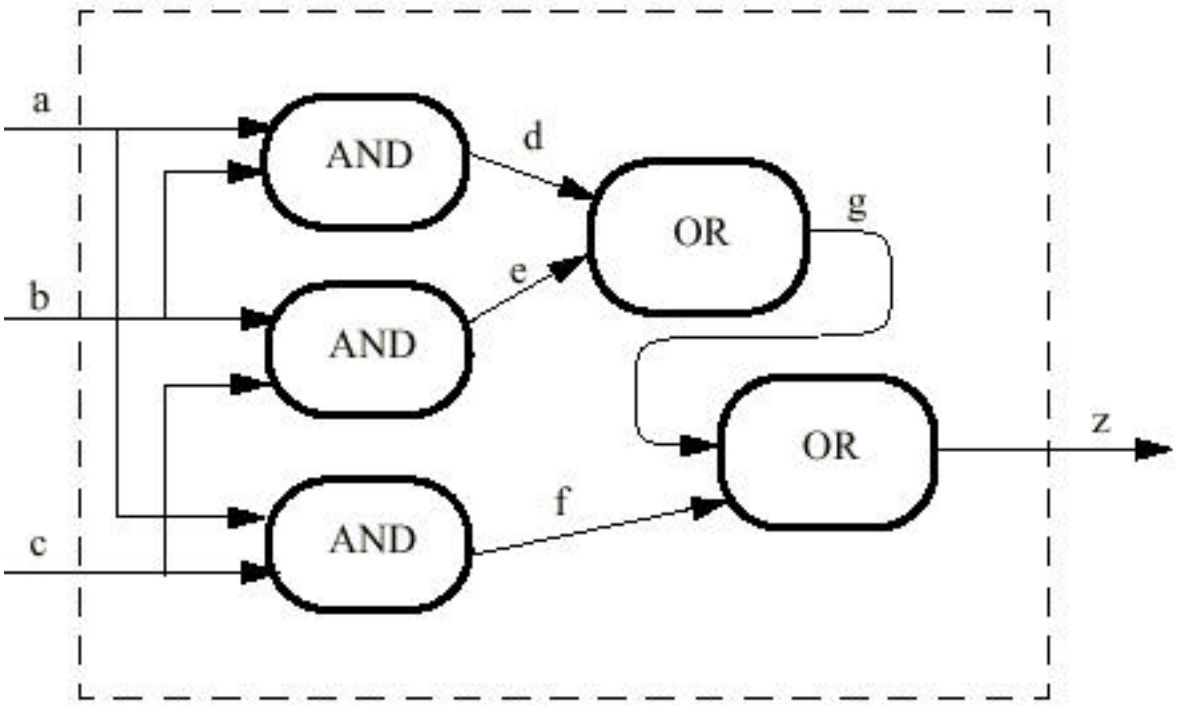
(Note that here the index I is a pair of integers, so 0 <= I will need a definition through a simplification rule such as

0 <= [M, N] ->> 0 <= M and 0 <= N . )

### *6.3 Verifying a logic-gate implementation*[22]

The following diagram shows a logic circuit for a "majority voting circuit". Such circuits are used in nuclear reactors and the electronics in airplanes or spacecraft, where for the sake of reliability, three computers are each given the same task. If at least two computers signal to do the same thing (i.e. at least two of a, b and c are 'high') then z is 'high 'and the task is performed; otherwise z is 'low' and the task is not performed.

[22] The example and diagram are due to J. Ostroff.




The circuit is intended to implement a vote(a, b, c) function which returns true if the two or more of the inputs a, b, c are true. We want to verify this by showing that if the inputs are translated to integers 1 or 0 and summed, the sum of the inputs (as integers) is ≥ 2 iff z is true.

The translation function is specified by the rule

```
int(X) ->> if(X, 1, 0).
```

and we can represent the logic-circuit implementation by the connection of and and or gates:

```
implementation(A, B, C, Z) ->>
        and_gate(A, B, d)
        and and_gate(B, C, e)
        and and_gate(C, A, f)
        and or_gate(d, e, g)
        and or_gate(g, f, Z).
```

The pattern-variables A, B, C, Z in this rule represent the three inputs and the output of the circuit. The values d, e, f, and g represent the internal signals connecting the logic gates. We use constants rather than pattern variables for these because they cannot be given arbitrary values; they are determined by the input values and the gates[23].

[23] In the jargon of clause logic, they are *Skolem constants* and correspond to existentially quantified variables in the predicate calculus.

The two kinds of gates used in the circuit are defined by

```
and_gate(V, W, X) ->> X iff (V and W).
or_gate(V, W, X) ->> X iff (V or W).
```

The implementation is proved correct relative to these definitions if we can prove

implementation(a, b, c, z) implies z = majority(a, b c).

where majority(a, b, c) = (2 <= int(a) + int(b) + int(c)).

We add a few simplification rules to assist the translation from arithmetic into propositional calculus.

```
(A <= X) = Z ->> (A <= X) iff Z.
 A <= W + if(X, Y, Z) ->> (A - Y) <= W and X or A -Z <= W and
not X.
A <= if(X, Y, Z) ->> A <= Y and X or A <= Z and not X.
```

(The need for these rules is determined as usual by looking at what remains unproved when `prover` attempts to prove the implication without them.)

Inputting the theorems to the `prover` tool, and using standard rules for arithmetic and equality:

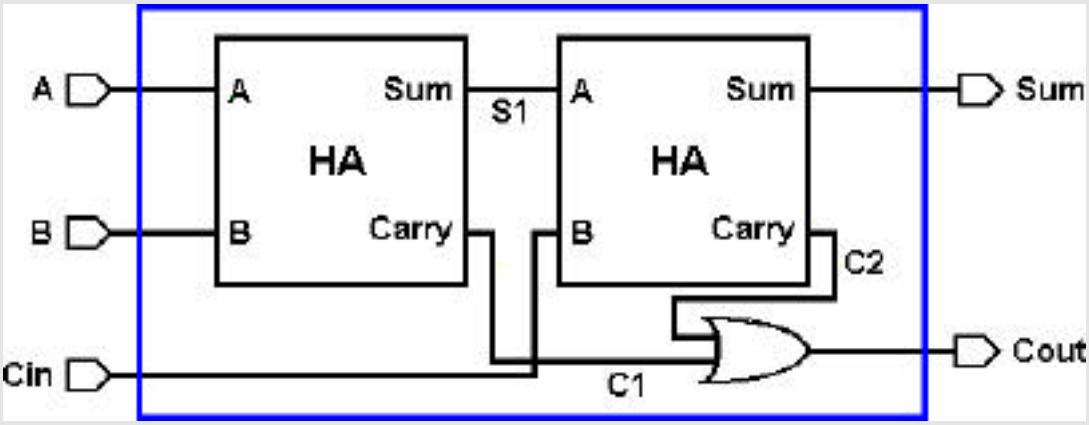
```
|: theory(majority).

|:  implementation(a,b, c, z) implies z = (int(a) + int(b)
   + int(c) >= 2).
implementation(a,b,c,z)implies z= (int(a)+int(b)+int(c)>=2)

* Valid.
```

Turning on the trace shows that simplification reduces the implication to

```
(g or f iff z)and (e or d iff g)and (c and b iff e)and (c and
a iff f)and (b and a iff d)implies (b and c and a or not c and
b and a or not b and c and a or not a and c and b iff z)
```

which the tautology checker finds to be valid.

**Exercise 6.8:** The following diagram[24] shows an implementation of a 1-bit adder using two half-adders and an or-gate.




Define an implementation predicate

adder(A, B, Cin, Sum, Cout)

using the approach explained above for the majority circuit, and verify that

adder(a, b, cin, sum, cout) implies int(cin) + int(a) + int(b) = int(sum) + 2*int(cout)

under the assumption that the half-adders are constructed correctly, i. e., that

half(A,B,Sum,Carry) iff int(A) + int(B) = int(Sum) + 2*int(Carry).

(Convert this equivalence to a simplification rule so that it can be used to eliminate references to half .)

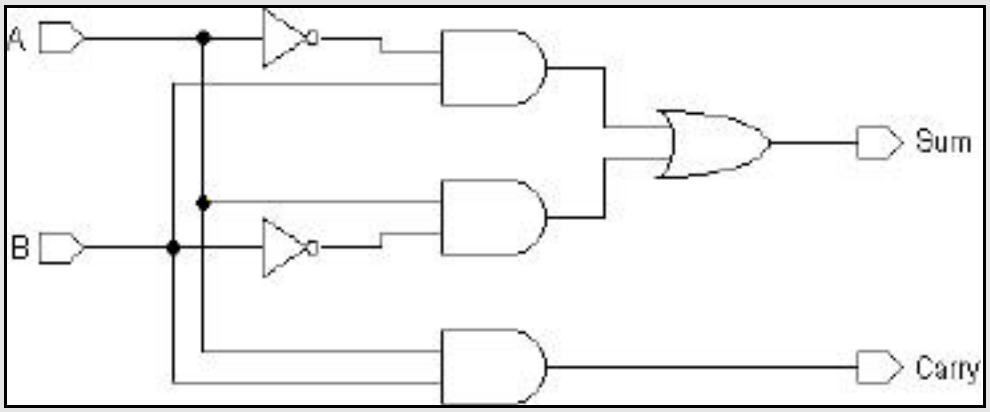
prover will need some help to eliminate the arithmetic expressions. Here are some examples of the sort of rules that will be needed:

```
if(S,1,0) + if(C,1,0) * 2 = 2 ->> not S and C.
if(X, Y, Z) + 1 ->>  if(X, 1 + Y, 1 + Z).
```

As a check on your circuit theory, show that if the or-gate is replaced by an and-gate, the implementation fails.

[24] The figures are taken from Tamura, P., "Jumpstart Introduction to Designing with Alliance", ASIM Labs: 1997.
`http://www-asim.lip6.fr/alliance/doc/jumpstart/`.

**Exercise 6.9:** A half-adder can be implemented using inverters, or- and and-gates:




Verify this implementation.

### *6.4 Verifying that a subclass is a subtype*[25]

Object-oriented languages such as Java and C++ allow the definition of abstract classes in which some or all of the class methods are left as "virtual", that is, unimplemented., with the implementation of such methods deferred to a sub-class. The utility of this approach is that it allows the definition of an abstract datatype whose axioms (the "abstract" aspect of the datatype) are applicable to a range of sub-types, differing in their implementation of the virtual methods, but sharing the relationships among those methods which are specified in the axioms associated with the abstract class. The question as to whether the sub-class methods correctly implement the virtual methods becomes the problem of verifying that the sub-class, in addition to being a sub-class of the given abstract class, is a *subtype* of the abstract datatype.

We illustrate this using the example of Shape and Circle, coded in Java:

```
public abstract class Shape {
     public Point center () {
          return Point ((left()+right())/2,
                        (top()+bottom())/2);
      };

     public void recenter (Point p) { move (p - center()); };
     public abstract double top();
     public abstract double bottom();
     public abstract double left();
     public abstract double right();
     public abstract void move(Point p);
}
```

[25] The material in this section is derived from an example in [Antoy, 1994].

Let us assume that the intended relationships among the Shape methods are specified by requiring that if S is a shape, P is a point, and X and Y are numbers, then the following propositions are true:

```
center(move(S, P)) = center(S) + P
top(move(S, point(X,Y))) = top(S) + X
bottom(move(S, point(X,Y))) = bottom(S) + Y
left(move(S, point(X,Y))) = left(S) + X
right(move(S, point(X,Y))) = right(S) + X
center(S) = point((left(S) + right(S))/2, (top(S) + bottom(S))/2)
recenter(S, P) = move(S, P - center(S))
left(S) <= right(S)
bottom(S) <= top(S)
```

(Notice that the methods associated with a Shape object are translated in these axioms into functions of the object. For example, `move(Point p)` becomes move(S, p), where S is a Shape, and p is a Point.)

Now suppose the sub-class Circle is defined by

```
class Circle extends Shape {
    protected double r:
    protected Point c;

    public Circle(Point center, double radius) {
        this.r = radius;  // radius >= 0
        this.c = center;
     }
    public Point center() {return this.c;}
    public void move(Point p) {this.c = p;}
    public double top() {return (this.c.y() + this.r);}
    public double bottom() {return (this.c.y() - this.r);}
    public double left() {return (this.c.x() - this.r);}
    public double right() {return (this.c.x() + this.r);}
}
```

Translating the methods from code into functional form yields the following definitions:

```
center(circle(C, R)) = C
move(circle(C, R), P) = circle(C + P, R)
top(circle(point(X, Y),R))=Y + R
bottom(circle(point{X,Y),R)) =Y - R
left(circle(point(X,Y),R)) = X - R
right(circle(point(X, Y), R)) = X + R
```

The verification question here is whether Circle is a valid subtype of Shape, that is, does it satisfy the Shape axioms? To test this, we create a theory file from the Circle definitions, along with some lemmas, and and see if `prover` can prove the Shape axioms from them:

```
indigo % cat <circle.simp

center(circle(C, R)) ->> C.
move(circle(C, R), P) ->> circle(C + P, R).
top(circle(point(X, Y), R)) ->> Y + R.
bottom(circle(point(X, Y), R)) ->> Y - R.
left(circle(point(X, Y), R)) ->> X - R.
right(circle(point(X, Y), R)) ->> X + R.
recenter(circle(C, R), P) ->> circle(P, R).

% LEMMAS

X - R <= X + R ->> 0 <=  R.
0 <=  r ->> true. % r is a radius
point(X, Y)*A ->> point(X*A, Y*A).
point(X, Y) + point(U, V) ->> point(X + U, Y + V).
X - Y + Z ->> X + Z - Y.
```

(Note that Shape's recenter method is here overwritten by a method specialized for Circles, whose result, however, should agree with that given by the Shape method.)

The Shape axioms are now not used as rewrite rules, but are translated into a set of propositions o be checked for validity, *when instantiated by Circles and Points*. So the variables S and P in the axioms must be replaced by object constructors such as circle(point(x, y), r) to which the rewrite rules in `circle.simp` can be applied.

The input to `prover` is:

```
indigo % cat <shape.props
      theory(circle).

      center(move(circle(point(x,y), r), p)) =
            center(circle(point(x,y), r)) + p.
      top(move(circle(point(u,v), r), point(x,y))) =
            top(circle(point(u,v), r)) + y.
      bottom(move(circle(point(u,v), r), point(x,y))) =
            bottom(circle(point(u,v), r)) + y.
      left(move(circle(point(u, v), r), point(x,y))) =
            left(circle(point(u,v), r)) + x.
      right(move(circle(point(u,v), r), point(x,y))) =
            right(circle(point(u, v), r)) + x.
      center(circle(point(x,y), r)) = point((left(circle(point(x,y), r))
            + right(circle(point(x,y), r)))/2,
            (top(circle(point(x,y), r)) +
            bottom(circle(point(x,y), r)))/2).
      recenter(circle(point(x,y), r), point(u, v)) =
            move(circle(point(x,y), r), point(u, v) -
            center(circle(point(x,y), r))).
      left(circle(point(x,y), r)) <= right(circle(point(x,y), r)).
      bottom(circle(point(x,y), r)) <= top(circle(point(x,y), r)).
```

with the desired result as shown below:

```
indigo 327 % prover < shape.data
Version 1.6.6SWI, January 3, 2007
Loading /cs/home/peter/3341/arithmetic.simp
Loading /cs/dept/course/2006-07/W/3341/equality.simp
Loading /cs/home/peter/3341/logic.simp
|:Loading /cs/home/peter/3341/circle.simp
center(move(circle(point(x, y), r), p))=center(circle(point(x, y), r))+p

* Valid.

top(move(circle(point(u, v), r), point(x, y)))=top(circle(point(u, v),
r))+y

* Valid.

bottom(move(circle(point(u, v), r), point(x, y)))=bottom(circle(point(u,
v), r))+y

* Valid.

left(move(circle(point(u, v), r), point(x, y)))=left(circle(point(u, v),
r))+x

* Valid.

right(move(circle(point(u, v), r), point(x, y)))=right(circle(point(u,
v), r))+x

* Valid.

center(circle(point(x, y), r))=point((left(circle(point(x, y),
r))+right(circle(point(x, y), r)))/2, (top(circle(point(x, y),
r))+bottom(circle(point(x, y), r)))/2)

* Valid.

recenter(circle(point(x, y), r), point(u, v))=move(circle(point(x, y),
r), point(u, v)-center(circle(point(x, y), r)))

* Valid.

left(circle(point(x, y), r))<=right(circle(point(x, y), r))

* Valid.

bottom(circle(point(x, y), r))<=top(circle(point(x, y), r))

* Valid.
```