4

Equality

So far, equality has been treated as just another binary predicate that may be interpreted arbitrarily. However, the role of equality is so central that often we only want to consider interpretations where 'equality means equality'. The previous logical theory and programmed proof procedures are easily modified for the new circumstances, but there are also more efficient and specialized ways of handling equality.

4.1 Equality axioms

In many applications of logic, particularly to mathematical reasoning, equations play a central role. We've partly recognized this by supporting the usual infix notion '$s = t$' instead of '$= (s, t)$'. Moreover, we can define various handy syntax operations for testing if a formula is an equation and for creating and breaking apart equations, e.g.

```
let is_eq = function (Atom(R("=",_))) -> true | _ -> false;;

let mk_eq s t = Atom(R("=",[s;t]));;

let dest_eq fm =
  match fm with
    Atom(R("=",[s;t])) -> s,t
  | _ -> failwith "dest_eq: not an equation";;

let lhs eq = fst(dest_eq eq) and rhs eq = snd(dest_eq eq);;
```

But, logically speaking, equality has just been dealt with as an arbitrary binary predicate; the interpretations we consider when deciding questions of logical validity include those where '=' is interpreted quite differently from equality. In view of the claimed central role of equality, it's natural to investigate restricting the class of models to those where 'equality means

equality', since it is those that we normally have in mind in, say, abstract algebra. We call an interpretation (or model of a particular set of sentences) *normal* if the equality predicate '=' is interpreted as equality on its domain.

Any normal interpretation must satisfy the formulas asserting that equality is an equivalence relation, i.e. is reflexive, symmetric and transitive:

$$\begin{aligned}&\forall x.\ x = x,\\ &\forall x\ y.\ x = y \Leftrightarrow y = x,\\ &\forall x\ y\ z.\ x = y \wedge y = z \Rightarrow x = z,\end{aligned}$$

as well as formulas asserting *congruence* for each n-ary function f in the language under consideration:

$$\forall x_1 \cdots x_n y_1 \cdots y_n.\ x_1 = y_1 \wedge \cdots \wedge x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n),$$

and similarly for each n-ary predicate R:

$$\forall x_1 \cdots x_n y_1 \cdots y_n.\ x_1 = y_1 \wedge \cdots \wedge x_n = y_n \Rightarrow R(x_1, \ldots, x_n) \Rightarrow R(y_1, \ldots, y_n).$$

For a given set of first-order formulas Δ, we write $\texttt{eqaxioms}(\Delta)$ ('the equality axioms for Δ') to mean the equivalence relation formulas together with the congruence formulas for all functions f and predicates R appearing in the formulas of Δ.

We have observed that any normal interpretation satisfies $\texttt{eqaxioms}(\Delta)$, but it's not the case that any interpretation satisfying $\texttt{eqaxioms}(\Delta)$ must be normal. Consider, for example, a language with just the two binary function symbols '$+$' and '$\cdot$' and the constants 0 and 1. Interpreting all these in the usual way in $\mathbb{Z}$ but equality by the relation $x \equiv y \pmod 2$, the equality axioms are still satisfied even though the interpretation is not normal. In fact, *no* set of formulas can constrain its models to be normal, because given any normal model, we can create a non-normal one by picking some a in the domain, adding arbitrarily many additional elements $b_i \in B$ and interpreting all the b_i in the same way as a. Despite this, we *do* have the following key result.

Theorem 4.1 *Any set Δ of first-order formulas has a normal model if and only if the set $\Delta \cup \texttt{eqaxioms}(\Delta)$ has a model.*

Proof One direction is easy: if M is a normal interpretation, it is clear that $\texttt{eqaxioms}(\Delta)$ holds in it; thus in any normal model of Δ, so does $\Delta \cup \texttt{eqaxioms}(\Delta)$.

Conversely, suppose that $\Delta \cup \texttt{eqaxioms}(\Delta)$ has a model M. Define a relation '$\sim$' on the domain D of M by setting $a \sim b$ precisely when $=_M (a, b)$, i.e. when a and b are 'equal' according to the interpretation $=_M$. Because the equivalence axioms hold in M, this is an equivalence relation, so we can partition D into equivalence classes where each $a \in D$ belongs to the equivalence class:

$$[a] = \{b \mid b \sim a\}$$

and $[a] = [b]$ iff $a \sim b$. We will use the set $D' = \{[a] \mid a \in D\}$ of equivalence classes as the domain of a new model M', and interpret each n-ary function symbol f as follows:

$$f_{M'}([a_1], \ldots, [a_n]) = [f_M(a_1, \ldots, a_n)].$$

Note that this is well-defined, i.e. independent of the particular representative of each equivalence class, because if $a'_i \sim a_i$ for $i = 1, \ldots, n$, we also have $f_M(a'_1, \ldots, a'_n) \sim f_M(a_1, \ldots, a_n)$ precisely because the functional congruence axiom holds in M. Similarly, we interpret each n-ary predicate symbol R by $R_{M'}([a_1], \ldots, [a_n]) = R_M(a_1, \ldots, a_n)$. Once again, this is independent of the particular choice of equivalence class representatives because the predicate congruence holds in M.

In particular we have $=_{M'} ([a], [b])$ precisely when $a \sim b$ and so when $[a] = [b]$. Thus M' is a normal interpretation. To see that it satisfies all the formulas in Δ, we essentially need to show that we can 'pull' the equivalence-class forming operation up the semantics of a formula. Note first that:

$$\texttt{termval}\ M'\ \delta'\ t = [\texttt{termval}\ M\ \delta\ t],$$

where $\delta'(x) = [\delta(x)]$ for all variables x. To prove this, simply proceed by structural induction on t. If t is the variable x then we have

$$\begin{aligned} & \texttt{termval}\ M'\ \delta'\ x \\ = \ & \delta'\ x \\ = \ & [\delta(x)] \\ = \ & [\texttt{termval}\ M\ \delta\ x], \end{aligned}$$

while if $t = f(s_1, \ldots, s_n)$, then using the inductive hypothesis and the definition of $f_{M'}$ we have:

$$\begin{aligned} & \texttt{termval}\ M'\ \delta'\ f(s_1, \ldots, s_n) \\ = \ & f_{M'}(\texttt{termval}\ M'\ \delta'\ s_1, \ldots, \texttt{termval}\ M'\ \delta'\ s_n) \\ = \ & f_{M'}([\texttt{termval}\ M\ \delta\ s_1], \ldots, [\texttt{termval}\ M\ \delta\ s_n]) \end{aligned}$$

$$\begin{aligned} &= [f_M(\texttt{termval}\ M\ \delta\ s_1, \ldots, \texttt{termval}\ M\ \delta\ s_n)] \\ &= [\texttt{termval}\ M\ \delta\ f(s_1, \ldots, s_n)]. \end{aligned}$$

Now we claim that for any formula p we have $\texttt{holds}\ M'\ \delta'\ p = \texttt{holds}\ M\ \delta\ p$. Once again, the proof is by structural induction. This is trivial if p is $\bot$ or $\top$, while it holds by definition of $R_{M'}$ when p is an atomic formula. The propositional operations obviously preserve this property, which leaves the quantified formulas as the interesting case. Note that:

$$\begin{aligned} & \texttt{holds}\ M'\ \delta'\ (\forall x.\ p) \\ = \ & \text{for all } A \in D', \texttt{holds}\ M'\ ((x \mapsto A)\delta')\ p \\ = \ & \text{for all } a \in D, \texttt{holds}\ M'\ ((x \mapsto [a])\delta')\ p \\ = \ & \text{for all } a \in D, \texttt{holds}\ M'\ ((x \mapsto a)\delta)'\ p \\ = \ & \text{for all } a \in D, \texttt{holds}\ M\ ((x \mapsto a)\delta)\ p \\ = \ & \texttt{holds}\ M\ \delta\ (\forall x.\ p), \end{aligned}$$

and similarly for the existential quantifier. Thus, since each $p \in \Delta$ holds in M in all valuations δ, it also holds in M' for all valuations ϵ, since ϵ is necessarily of the form δ' for some valuation δ in M (just let $\delta(x)$ be any member of $\epsilon(x)$). □

In our practical applications, we will be concerned with a single formula. Define $\texttt{eqaxiom}(p)$ to be the conjunction of the (necessarily finitely many) equality axioms $\texttt{eqaxioms}(\{p\})$. Then:

Corollary 4.2 *Any formula p is satisfiable in a normal model iff $p \wedge \texttt{eqaxiom}(p)$ is satisfiable.*

Proof By definition of the semantics of conjunction, an interpretation satisfies $p \wedge \texttt{eqaxiom}(p)$ iff it satisfies p and $\texttt{eqaxiom}(\{p\})$. □

We have the following dual result for validity.

Corollary 4.3 *A formula p holds in all normal models iff $\texttt{eqaxiom}(p) \Rightarrow p$ holds in all models.*

Proof Since p holds in a model iff its universal closure does, we can assume without loss of generality that p is closed. Thus it holds in all normal models iff $\neg p$ has no normal model, and so if $\neg p \wedge \texttt{eqaxiom}(\neg p)$ has no model. But $\texttt{eqaxiom}(\neg p) = \texttt{eqaxiom}(p)$ and so $\neg p \wedge \texttt{eqaxiom}(\neg p)$ is logically equivalent to $\neg(p \vee \neg(\texttt{eqaxiom}(p)))$ and so to $\neg(\texttt{eqaxiom}(p) \Rightarrow p)$. This is unsatisfiable iff $\texttt{eqaxiom}(p) \Rightarrow p$ is valid. □

In the abstract treatment above, the equality axioms included a predicate congruence property for equality itself:

$$\forall x_1\ x_2\ y_1\ y_2.\ x_1 = y_1 \wedge x_2 = y_2 \Rightarrow x_1 = x_2 \Rightarrow y_1 = y_2.$$

But we can afford to omit it, because it's a logical consequence of the equivalence axioms. We can economize further by using only two equivalence axioms, reflexivity and a variant of transitivity $\forall x\ y\ z.x = y \wedge x = z \Rightarrow y = z$. (Symmetry follows by instantiating that axiom so that x and z are the same, then using reflexivity.)

OCaml implementation

In Skolemization we used `functions` to find all the functions in a term; similarly the following finds all predicates, again as name–arity pairs:

```
let rec predicates fm = atom_union (fun (R(p,a)) -> [p,length a]) fm;;
```

We can manufacture a congruence axiom for each function symbol by producing the appropriate number of arguments $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ and constructing the formula

$$\forall x_1 \ldots x_n\ y_1 \ldots y_n.x_1 = y_1 \wedge \cdots x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n).$$

We return a list that normally has one member but is empty in the case of a nullary function (i.e. individual constant):

```
let function_congruence (f,n) =
  if n = 0 then [] else
  let argnames_x = map (fun n -> "x"^(string_of_int n)) (1 -- n)
  and argnames_y = map (fun n -> "y"^(string_of_int n)) (1 -- n) in
  let args_x = map (fun x -> Var x) argnames_x
  and args_y = map (fun x -> Var x) argnames_y in
  let ant = end_itlist mk_and (map2 mk_eq args_x args_y)
  and con = mk_eq (Fn(f,args_x)) (Fn(f,args_y)) in
  [itlist mk_forall (argnames_x @ argnames_y) (Imp(ant,con))];;
```

for example:

```
# function_congruence ("f",3);;
- : fol formula list =
[<<forall x1 x2 x3 y1 y2 y3.
     x1 = y1 /\ x2 = y2 /\ x3 = y3 ==> f(x1,x2,x3) = f(y1,y2,y3)>>]
# function_congruence ("+",2);;
- : fol formula list =
[<<forall x1 x2 y1 y2. x1 = y1 /\ x2 = y2 ==> x1 + x2 = y1 + y2>>]
```

An analogous function for predicates is almost the same, except that we use implication of formulas rather than equality of terms in the consequent:

```
let predicate_congruence (p,n) =
  if n = 0 then [] else
  let argnames_x = map (fun n -> "x"^(string_of_int n)) (1 -- n)
  and argnames_y = map (fun n -> "y"^(string_of_int n)) (1 -- n) in
  let args_x = map (fun x -> Var x) argnames_x
  and args_y = map (fun x -> Var x) argnames_y in
  let ant = end_itlist mk_and (map2 mk_eq args_x args_y)
  and con = Imp(Atom(R(p,args_x)),Atom(R(p,args_y))) in
  [itlist mk_forall (argnames_x @ argnames_y) (Imp(ant,con))];;
```

As planned, we use this variant of the equivalence properties:

```
let equivalence_axioms =
  [<<forall x. x = x>>; <<forall x y z. x = y /\ x = z ==> y = z>>];;
```

Now we define a function that returns `eqaxiom`$(p) \Rightarrow p$ for an input formula p. It leaves p alone if it doesn't involve equality at all, since there is then no distinction between its normal and non-normal models.

```
let equalitize fm =
  let allpreds = predicates fm in
  if not (mem ("=",2) allpreds) then fm else
  let preds = subtract allpreds ["=",2] and funcs = functions fm in
  let axioms = itlist (union ** function_congruence) funcs
                      (itlist (union ** predicate_congruence) preds
                              equivalence_axioms) in
  Imp(end_itlist mk_and axioms,fm);;
```

The upshot of Corollary 4.3 is that we can test the validity of p in first-order logic with equality by testing the validity of `equalitize`(p) in ordinary first-order logic. Thus, we can just apply `equalitize` as a preprocessing step for any of our existing proof procedures. Note, by the way, that we will avoid creating congruence axioms for the Skolem functions, which only appear later in the underlying proof procedure. It's hard to predict whether it would be more efficient to add congruences for Skolem functions: it means more hypotheses, but perhaps allows shortcuts in proofs. Observe also that the equality axioms are Horn clauses (Section 3.14), so whenever Δ is a set of Horn clauses, so is $\Delta \cup$ `eqaxioms`(Δ). Thus, we can also extend the Prolog-like proof procedure `hornprove` from Section 3.14 to a complete prover for Horn problems in logic with equality just by adding the equality axioms in a preprocessing step in the same way. And since `meson` reduces to Prolog-type search on Horn problems, it will continue to do so when combined with the preprocessing step.

For a first example, consider the following formula given by Dijkstra (1997), who shows how its validity underlies a proof of Morley's theorem in geometry.

```
# let ewd = equalitize
   <<(forall x. f(x) ==> g(x)) /\
     (exists x. f(x)) /\
     (forall x y. g(x) /\ g(y) ==> x = y)
     ==> forall y. g(y) ==> f(y)>>;;
...
```

We can prove it by any of the main methods developed earlier, including model elimination, resolution and even tableaux with splitting, e.g.

```
# meson ewd;;
...
- : int list = [6]
```

We thus conclude that the original formula is valid in first-order logic with equality, i.e. holds in all normal models. Another example, which the author learned from Wishnu Prasetya,[†] is that for any two functions $f : A \to B$ and $g : B \to A$ there is a unique x such that $x = f(g(x))$ iff there is a unique y such that $y = g(f(y))$.

```
let wishnu = equalitize
 <<(exists x. x = f(g(x)) /\ forall x'. x' = f(g(x')) ==> x = x') <=>
   (exists y. y = g(f(y)) /\ forall y'. y' = g(f(y')) ==> y = y')>>;;
```

The resulting formula is solvable by MESON, but already it takes a significant amount of time. So, although just adding equality axioms allows us to re-use existing procedures, one might wonder if there are more effective ways of dealing with equality. This is a matter to which we will return before too long.

4.2 Categoricity and elementary equivalence

Thanks to Theorem 4.1, the theoretical results in Chapter 3 can also be adapted quite easily to consider only normal models. Arguably, they are more interesting in this context, since it is usually normal models we have in mind when thinking about mathematical structures. In fact, many of the structures studied in abstract algebra are precisely the normal models of some first-order formula or set of first-order formulas. For example, a *group*

[†] See his message to the info-hol mailing list on 18 October 1993, available on the Web as `ftp://ftp.cl.cam.ac.uk/.aftp/hvg/info-hol-archive/15xx/1574`.

is essentially just a normal model of the following formula:

$$(\forall x\ y\ z.\, m(x, m(y, z)) = m(m(x, y), z)) \wedge$$
$$(\forall x.\, m(x, 1) = x \wedge m(1, x) = x) \wedge$$
$$(\forall x.\, m(x, i(x)) = 1 \wedge m(i(x), x) = 1).$$

It's not difficult to come up with similar axiomatizations for many other structures such as partial orders and rings. Thus, in the model theory of first-order logic, we have a suitable mathematical generalization taking in various specific mathematical structures. This enables us to define notions like 'substructure' and 'homomorphism', such that for example 'subgroup' and 'ring homomorphism' are special cases of the general concept. We give the general definition of 'isomorphism' shortly,† and starting in Section 5.6 we will take a closer look at various algebraic systems.

Metatheorems

First, we can easily adapt the compactness theorem to logic with equality.

Theorem 4.4 *If every finite subset Δ of a set Σ of formulas has a normal model, then Σ itself has a normal model.*

Proof If each finite $\Delta \subseteq \Sigma$ has a normal model, then each $\Delta \cup \mathtt{eqaxioms}(\Delta)$ for finite Δ has a model. However, every finite $\Delta' \subseteq \Sigma \cup \mathtt{eqaxioms}(\Sigma)$ is a subset of some such $\Delta \cup \mathtt{eqaxioms}(\Delta)$ for finite Δ, and consequently each finite $\Delta' \subseteq \Sigma \cup \mathtt{eqaxioms}(\Sigma)$ has a model. By the compactness theorem for arbitrary models, $\Sigma \cup \mathtt{eqaxioms}(\Sigma)$ has a model and therefore, by Theorem 4.1, Σ has a normal model. □

The equalitarian version of the downward Löwenheim–Skolem theorem can be derived similarly.

Theorem 4.5 *If a countable set of formulas Σ has a normal model M, then it has a countable (either finite or countably infinite) normal model.*

Proof If Σ has a normal model, $\Sigma \cup \mathtt{eqaxioms}(\Sigma)$ has a model, and so by the original downward LS Theorem 3.49, it has a model with a countable domain D. The corresponding normal model of Σ that we constructed in the

† There is actually some divergence in general definitions of *homomorphism*, with two standard texts by Enderton (1972) and Mendelson (1987) differing over whether just implication or full equivalence is demanded between interpreted predicates. Also, note that in general these concepts can depend on whether the axioms contain operation symbols or just existence assertions (Hodges 1993b).

proof of Theorem 4.1 has as its domain equivalence classes of elements of D. The cardinality of this set of equivalence classes is at most the cardinality of D (since each equivalence class contains at least one element of D) and so is countable too. □

Constructing *larger* models than a given model is no longer trivial, because we can't just add new domain elements and retain normality. However, by cleverly exploiting compactness, we can still find a way to grow models. For example:

Theorem 4.6 *If a set of sentences S has normal models of arbitrarily large finite cardinality, then it has an infinite normal model.*

Proof Consider the following sentences B_i, which intuitively mean 'there are at least i distinct elements'.

$$\begin{aligned} B_2 &= \exists x\ y.\ x \neq y, \\ B_3 &= \exists x\ y\ z.\ x \neq y \wedge x \neq z \wedge y \neq z, \\ B_4 &= \exists w\ x\ y\ z.\ w \neq x \wedge w \neq y \wedge w \neq z \wedge x \neq y \wedge x \neq z \wedge y \neq z, \\ B_5 &= \ldots \end{aligned}$$

Write $B = \bigcup_{i \in \mathbb{N}} B_i$. Since, by hypothesis, S has models of arbitrarily large finite cardinality, all finite subsets of $S \cup B$ are satisfiable. Therefore by compactness so is $S \cup B$, but clearly any model of these sentences must be infinite. □

Using a closely related technique, one can prove the upward Löwenheim–Skolem theorem (actually due to Tarski), analogous to Theorem 3.50 but much more interesting: if a set of formulas Σ has a normal model with infinite domain D, then it has a model of any infinite cardinality $\geq |D|$. The proof is simply to add enough new constants c_i that do not already occur in Σ, and apply compactness to the set $\Sigma \cup \{c_i \neq c_j \mid i, j \in S,\ i \neq j\}$. However, we will not present this in detail since we have not proved compactness for uncountable languages. Indeed, the upward Löwenheim–Skolem theorem requires the machinery of the Axiom of Choice.†

We will, however, give an example of how to construct 'nonstandard' models using compactness. Consider some language for the real numbers, maybe including addition, multiplication, negation, inversion, the constants

† The formula $\forall x\ y\ x'\ y'.\ p(x,y) = p(x',y') \Rightarrow x = x' \wedge y = y'$ has a model with domain $\mathbb{N}$, e.g. interpreting p as the pairing function $\langle x, y \rangle$ in Section 7.2. The upward LS theorem then implies that this has models of arbitrary infinite cardinality, and hence that $\kappa^2 \leq \kappa$ for any infinite κ. This is known to be equivalent to AC (Jech 1973).

0 and 1, and special functions like sin. Let Σ be the set of all formulas in this language that are true in $\mathbb{R}$ with the intended interpretation, a.k.a. the 'standard model', Consider the set:

$$\Sigma' = \Sigma \cup \{1 < c, 1 + 1 < c, 1 + 1 + 1 < c, \ldots\},$$

where c is a constant symbol not appearing in Σ. Any finite set of these has a model, for the reals are a model of Σ and we can then interpret c by some suitably large number. Thus by compactness there is a 'nonstandard model' of Σ in which c behaves like an infinite number, with $n < c$ for each natural number n. Indeed, this gives rise to other larger infinite numbers like $c + c$ and infinitesimal numbers like $1/c$ (despite the fact that we can also, by the Downward Löwenheim–Skolem theorem, assume it to be countable). Yet this strange menagerie of numbers obeys all the first-order properties that the 'real reals' do. This observation is a possible starting point for non-standard analysis (A. Robinson 1966) which exploits nonstandard models to prove standard results using infinite and infinitesimal elements. For more on this, see Cutland (1988), Davis (1977) or Hurd and Loeb (1985).

Consequences

In the axiomatic approach to mathematics, one starts from a set of axioms and derives conclusions without making any additional assumptions. If we are concerned with properties expressible in first-order logic, we might formalize this idea by allowing from axioms Σ the deduction of any first-order consequence of the axioms Σ. We will sometimes abbreviate the set $\{p \mid \Sigma \models p\}$ of first-order consequences of a set of first-order 'axioms' Σ by $\mathrm{Cn}(\Sigma)$.

Part of the appeal of the axiomatic method is that it isolates the assumptions that are actually necessary, so that the full generality of the results is seen. For this to be significant, we actually want Σ to have several interesting models. For example, the group axioms are satisfied by addition of integers or reals, multiplication of nonzero reals, composition of permutations on a set and so on. Sometimes, however, we want to use a set of axioms almost as a *definition* of a particular structure, such that all structures obeying the axioms are essentially the same. In fact, this use of axioms predated the general idea of the axiomatic method. For example, it used to be believed that the traditional axioms for geometry (without the axioms of parallels) had this property, but it later turned out that there were unexpected non-Euclidean models.

Given two interpretations M and M' of a first-order language with

respective domains D and D', we say that M and M' are *isomorphic* if there are mappings $i : D \to D'$ and $j : D' \to D$ such that for all $x \in D$, $j(i(x)) = x$, for all $x' \in D'$, $i(j(x')) = x'$, for each n-ary function symbol f in the language:

$$i(f_M(a_1, \ldots, a_n)) = f_{M'}(i(a_1), \ldots, i(a_n))$$

and for each n-ary predicate symbol

$$R_M(a_1, \ldots, a_n) = R_{M'}(i(a_1), \ldots, i(a_n))$$

for any $a_1, \ldots, a_n \in D$. The functions i and j are said to set up an *isomorphism*, or sometimes themselves to be *isomorphisms*. Intuitively, isomorphic interpretations are 'essentially the same' but for using a different underlying set, and indeed the word literally means something like 'equal shape'. A set of formulas (or 'axioms') Σ is said to be *categorical* if any two models are isomorphic. (One usually assumes also that it *has* at least one model.) The Löwenheim–Skolem theorems imply that if a set of first-order formulas has some infinite model, it has models of a different cardinality, which are therefore certainly not isomorphic (since an isomorphism is also a bijection). Thus, for first-order formulas, categoricity only arises for sets of formulas with just finite models, which are often the less interesting ones.

However there are at least two natural ways in which we can weaken the idea of categoricity. First, we might say that even though the cardinality of models of Σ may not be fixed, at least all models of some particular cardinality κ are determined up to isomorphism. In this case Σ is said to be *κ-categorical*. A number of interesting instances of this phenomenon are known, many predating the formal articulation of the concept using first-order logic. For example, Steinitz (1910) proved that any two algebraically closed fields of a given characteristic with the same uncountable cardinality are isomorphic. However, we will not dwell on the theory of κ-categoricity here.

Another idea is to say that since Σ consists only of first-order statements, it's unreasonable to expect to be able to prove that all its models are *isomorphic*. It's much more reasonable just to demand that all models satisfy the same first-order sentences, i.e. are all *elementarily equivalent*. (It's not too hard to show that isomorphic models are also elementarily equivalent, though the example of nonstandard models shows that the converse is false in general.) This is essentially the notion of *completeness of a theory*, which we study in detail in Section 5.6.

4.3 Equational logic and completeness theorems

Consider purely equational logic, where we start from a set Δ of (implicitly universally quantified) equations and ask whether another equation $s = t$ holds in all normal models of Δ, i.e. whether $\Delta \models s = t$ in first-order logic with equality. A famous theorem due to Birkhoff (1935) relates this to a set of *proof rules* or *inference rules* for generating equational conclusions. Given a set of equations Δ we define '$s = t$ is provable from Δ', written $\Delta \vdash s = t$, inductively (see Appendix 1) by the following rules:

$$\frac{(s = t) \in \Delta}{\Delta \vdash s = t}\text{AXIOM}$$

$$\frac{\Delta \vdash s = t}{\Delta \vdash \texttt{subst}\ i(s = t)}\text{INST}$$

$$\frac{}{\Delta \vdash t = t}\text{REFL}$$

$$\frac{\Delta \vdash s = t}{\Delta \vdash t = s}\text{SYM}$$

$$\frac{\Delta \vdash s = t \quad \Delta \vdash t = u}{\Delta \vdash s = u}\text{TRANS}$$

$$\frac{\Delta \vdash s_1 = t_1 \quad ... \quad \Delta \vdash s_n = t_n}{\Delta \vdash f(s_1, ..., s_n) = f(t_1, ..., t_n)}\text{CONG}$$

Theorem 4.7 *$\Delta \models s = t$, i.e. an equation $s = t$ holds in all normal models of a set Δ of equations, if and only if $\Delta \vdash s = t$, i.e. the equation $s = t$ is derivable from Δ by repeated use of Birkhoff's rules.*

Proof We first consider the right-to-left direction. Note that each proof rule applied to logically valid hypotheses gives logically valid conclusions; for example for transitivity we just need to observe that if $\Delta \models s = t$ and $\Delta \models t = u$ then also $\Delta \models s = u$. So by induction, whenever $\Delta \vdash s = t$ we also have $\Delta \models s = t$ in first-order logic with equality.

Conversely, if $\Delta \models s = t$, then $\Delta' = \Delta \cup \neg(s = t)$ has no normal model, and therefore $\Delta' \cup \texttt{eqaxioms}(\Delta')$ is unsatisfiable. As noted earlier, all these formulas are Horn clauses, so there is a Prolog-style proof of $\bot$ from them, as explained in Section 3.14. This must start with the formula $s = t \Rightarrow \bot$ to get the subgoal $s = t$, and thereafter divide into subgoals ending either in instances of reflexivity or (possibly instantiation of) formulas in Δ. The internal nodes simply apply transitivity, symmetry and congruence. They

therefore correspond exactly to Birkhoff's rules; all we have done is consider instances of the equality axioms as inference rules in themselves. □

This vindicates a naive expectation that if one equational formula is a logical consequence of others, one can get it by rewriting forwards, backwards and at depth, the kind of manipulative techniques we learn at school. Birkhoff originally approached the problem more directly, and later Maltsev (1936) and others realized that many of the nice properties of equational logic discovered by Birkhoff still hold in the more general setting of Horn clauses.

Soundness and completeness

Birkhoff's theorem is an important case where a *semantic* notion $\Delta \models s = t$ is shown equivalent to a *syntactic* notion $\Delta \vdash s = t$ of 'provability'. In general, we say that such a provability relation '$\vdash$' is:

- *sound* if whenever $\Delta \vdash p$ we also have $\Delta \models p$;
- *complete* if whenever $\Delta \models p$ we also have $\Delta \vdash p$.

Birkhoff's theorem asserts that the rules above are both sound and complete provided we restrict ourselves just to equations. They are definitely incomplete if we consider first-order formulas in general, however, since they can only deduce equational conclusions. We can also consider the resolution rule from Section 3.11 as defining a proof system. However, the reader should register an important mathematical distinction and another, purely psychological, one.

Completeness and refutation completeness Birkhoff's theorem assures us that any equation that holds semantically can be derived syntactically. This is in contrast with, say, the resolution calculus, where we merely showed that if a set of clauses is unsatisfiable, we can derive the empty clause from it. This implies $\Delta \models p$ iff $\Delta \vdash p$ only for the special case $p = \bot$, a property we called *refutation* completeness. As noted in Section 3.11, the example of $P \models P \vee Q$ shows that resolution is not complete in the stronger sense.

Naturalness As mentioned earlier, Birkhoff's theorem confirms our natural intuition and the Birkhoff rules formalize steps that a human attempting to prove the same theorem might make. By contrast, the resolution calculus, which J. A. Robinson (1965b) explicitly categorized as a *machine-oriented*

principle, is remote from the methods people typically use when proving theorems, with its Skolemizing steps and insistence on clausal form.†

We will describe a more human-oriented proof system that is complete for full first-order logic in Section 6.3.

The difficulty of equational proofs

Although in some respects equational logic has turned out to be 'tamer' than full first-order logic, there is a precise sense in which it is just as difficult, by virtue of an embedding of full first-order logic in equational logic due to McKenzie (1975).‡ Indeed, the reader with any experience of finding equational proofs in relatively simple axiom systems will know that it can be astonishingly difficult (Kapur and Zhang 1991). For example, the following problem is often set as an exercise in courses on group theory. We are given '1-sided' versions of the identity and inverse axioms, and are required to deduce that left inverses are also right inverses. Our existing setup for equality handling can solve this problem, but it takes many hours; a more efficient approach is discussed in Section 4.8.

```
(meson ** equalitize)
 <<(forall x y z. x * (y * z) = (x * y) * z) /\
   (forall x. 1 * x = x) /\
   (forall x. i(x) * x = 1)
   ==> forall x. x * i(x) = 1>>;;
```

The reader may like to try competing against the machine! Here is a reasonably human-oriented proof:

$$\begin{aligned} x \cdot i(x) &= 1 \cdot (x \cdot i(x)) \\ &= (i(i(x)) \cdot i(x)) \cdot (x \cdot i(x)) \\ &= i(i(x)) \cdot (i(x) \cdot (x \cdot i(x))) \\ &= i(i(x)) \cdot ((i(x) \cdot x) \cdot i(x)) \\ &= i(i(x)) \cdot (1 \cdot i(x)) \\ &= i(i(x)) \cdot i(x) \\ &= 1. \end{aligned}$$

We found this by tracing the proof MESON found, and rearranging the order of some of the Birkhoff rules to turn it into a simple transitivity chain for easier presentation in a linear format. In fact, Birkhoff proofs in some

† Note, however, the suggestion of A. Robinson (1957) that Skolem functions have their analogue in construction lines used in traditional geometrical proofs.

‡ On the other hand, an embedding of first-order logic in the theory of Boolean rings was actually suggested by Hsiang (1985) as a workable approach to first-order proof.

stronger canonical form can be easier to find, just as, say, linear resolution can cut down the search space compared to unrestricted resolution (Section 3.13). And some of the results we present next can be proved using canonical transformations of Birkhoff proofs (Exercise 4.2).

4.4 Congruence closure

Consider equational logic in the special case of ground terms, i.e. deciding $E \models s = t$ where $s = t$ and all members of E are equations not containing variables. In the light of Birkhoff's Theorem 4.7, this is equivalent to $E \vdash s = t$. But since no variables are involved, the Birkhoff instantiation rule is clearly not necessary. The highlight of this section is the observation that we can further restrict the Birkhoff proofs to those where all terms appearing in intermediate equations are subterms of the terms in the original problem, which implies that the problem is decidable.

In what follows, we assume some set G of terms that is closed under subterms, i.e. if $t \in G$ and s is a subterm of t then $s \in G$. The following can serve as the implementation and the formal definition of the set of subterms of a term:

```
let rec subterms tm =
  match tm with
    Fn(f,args) -> itlist (union ** subterms) args [tm]
  | _ -> [tm];;
```

We say that a binary relation $\sim$ on G is a *congruence* if it is reflexive, symmetric and transitive (i.e. an equivalence relation) and satisfies the congruence property: for each n-ary function symbol f, if $s_1 \sim t_1$, ..., $s_n \sim t_n$ then also $f(s_1, \ldots, s_n) \sim f(t_1, \ldots, t_n)$, whenever all those terms are in G. Note that given any binary relation $R \subseteq G \times G$ there is a unique smallest congruence extending R, and this is known as the *congruence closure* of R. It can be defined inductively (see Appendix 1) by starting with R and adding rules for closure under the equivalence and congruence properties.

Theorem 4.8 *Suppose all s_i, t_i, s and t are ground terms, and G consists of those terms and all their subterms. Let '$\sim$' be the congruence closure on G of $\{(s_1, t_1), \ldots, (s_n, t_n)\}$. Then the following are equivalent:*

(i) $\{s_1 = t_1, \ldots, s_n = t_n\} \models s = t$;
(ii) $s \sim t$;
(iii) *there is a Birkhoff proof of $s = t$ from $s_1 = t_1$, ..., $s_n = t_n$ whose intermediate steps involve only terms in G;*
(iv) $\{s_1 = t_1, \ldots, s_n = t_n\} \vdash s = t$.

Proof By Birkhoff's Theorem 4.7, (i) and (iv) are equivalent. If (iii) then (iv), since it is just a more restricted case of the same thing. If (ii) then (iii), since the set of pairs (s, t) that have a restricted Birkhoff proof from $s_1 = t_1, \ldots, s_n = t_n$ contains $\{(s_1, t_1), \ldots, (s_n, t_n)\}$ and is closed under equivalence and congruence because of the Birkhoff rules, and therefore must include the smallest such relation '$\sim$'. To complete the circle of equivalents, we need to show that (ii) follows from (i).

In fact we show the contrapositive, assuming $s \not\sim t$ and exhibiting an interpretation M where each $s_i = t_i$ holds but $s = t$ does not. The domain of M is the set of equivalence classes of G under '$\sim$'. Each constant c is interpreted by itself. An n-ary function f for $n \geq 1$ is interpreted as $f_M(C_1, \ldots, C_n) = C$, where C is the equivalence class containing $f(u_1, \ldots, u_n)$ for some representatives $u_i \in C_i$ if such a class exists, and some fixed but arbitrary equivalence class otherwise. (There may indeed be no such C containing a suitable $f(u_1, \ldots, u_n)$, because we are restricted to terms in G, but if there is one, it is uniquely defined independent of the representatives u_i, precisely because $\sim$ is a congruence.) This is indeed a (normal) interpretation, and by induction on terms `termval` $M\ \sigma\ u \sim u$ for all $u \in G$. Therefore for all $u, v \in G$, `holds` $M\ \sigma\ (u = v)$ is equivalent to $u \sim v$. Consequently each $s_i = t_i$ holds in M but not $s = t$, so $\{s_1 = t_1, \ldots, s_n = t_n\} \not\models s = t$ as required. □

Implementation of congruence closure

Our implementation of congruence closure will take an existing congruence relation and extend it to a new one including a given equivalence $s \sim t$. This can then be iterated starting with the empty congruence to find the congruence closure of $\{(s_1, t_1), \ldots, (s_n, t_n)\}$ as required. We will use a standard union-find data structure described in Appendix 2 to represent equivalences, so closure under the equivalence properties will be automatic and we'll just have to pay attention to closure under congruences. So suppose we have an existing congruence $\sim$ and we want to extend it to a new one $\sim'$ such that $s \sim' t$. We need to merge the corresponding equivalence classes $[s]$ and $[t]$, and may also need to merge others such as $[f(s, t, f(s, s))]$ and $[f(t, t, f(s, t))]$ to maintain the congruence property. We can test whether two terms 'should be' equated by a 1-step congruence by checking if all their immediate subterms are already equivalent:

```
let congruent eqv (s,t) =
  match (s,t) with
    Fn(f,a1),Fn(g,a2) -> f = g & forall2 (equivalent eqv) a1 a2
  | _ -> false;;
```

For the main algorithm, as well as the equivalence relation itself, `eqv`, we maintain a 'predecessor function' `pfn` mapping each canonical representative s of an equivalence class C to the set of terms of which some $s' \in C$ is an immediate subterm. We can then direct our attention at the appropriate terms each time equivalence classes are merged. It is this `(eqv,pfn)` pair that is updated by the following `emerge` operation for a new equivalence $s \sim t$.

First we normalize $s \to s'$ and $t \to t'$ based on the current equivalence relation, and if they are already equated, we need do no more. Otherwise we obtain the sets of predecessors, `sp` and `tp`, of the two terms. We update the equivalence relation to `eqv'` to take account of the new equation, and combine the predecessor sets to update the predecessor function to `pfn'` (mapped from the new canonical representative `st'` in the new equivalence relation). Then we run over all pairs from `sp` and `tp`, recursively performing an `emerge` operation on terms that should become equated as a result of a single congruence step.

```
let rec emerge (s,t) (eqv,pfn) =
  let s' = canonize eqv s and t' = canonize eqv t in
  if s' = t' then (eqv,pfn) else
  let sp = tryapplyl pfn s' and tp = tryapplyl pfn t' in
  let eqv' = equate (s,t) eqv in
  let st' = canonize eqv' s' in
  let pfn' = (st' |-> union sp tp) pfn in
  itlist (fun (u,v) (eqv,pfn) ->
                if congruent eqv (u,v) then emerge (u,v) (eqv,pfn)
                else eqv,pfn)
         (allpairs (fun u v -> (u,v)) sp tp) (eqv',pfn');;
```

At least this algorithm must terminate, because each time it gets past the initial $s' = t'$ test it reduces the total number of equivalence classes, of which there can only be a finite number. We need to show that if the initial `eqv` is a congruence and `pfn` maps canonical representatives to the predecessor sets, the resulting equivalence relation is the congruence closure of `eqv` and the new equivalence $s \sim t$, and `pfn` is correspondingly updated.

The last part is easy, since `pfn` is always modified in step with direct changes in the equivalence relation from `equate`. As for congruence closure, we can see that the new equivalence relation certainly includes the original `eqv`, since all we do is add to it, and it also contains (s, t) because unless these terms were already equated, the very first `equate` call equates them. Moreover, because of the representation of equivalence classes, it is automatically closed under equivalence properties. We only need to show that it is also closed under congruences. Supposing otherwise, there must be two

terms of the form $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$ that are not equivalent, yet each pair (s_i, t_i) for $1 \leq i \leq n$ is. Since, by hypothesis, the initial `eqv` was congruence closed, at least one of these equivalences $s_i = t_i$ must have resulted from a call to `equate` from within `emerge`, and there must have been some such `equate` call at which *all* the pairs (s_i, t_i) became equated for the first time. However, by construction, this would be followed by a congruence check that would equate $f(s_1, \ldots, s_n)$ and $f(t_1, \ldots, t_n)$, a contradiction.

Equality decision procedure

We can use congruence closure to give a complete decision procedure for validity of universal formulas $\forall x_1, \ldots, x_n.\, P[x_1, \ldots, x_n]$ where $P[x_1, \ldots, x_n]$ involves no predicates besides equality, but may involve arbitrary function symbols. Such a formula is valid iff its negation $\exists x_1, \ldots, x_n.\, \neg P[x_1, \ldots, x_n]$ is unsatisfiable, and so, by Skolemization as usual, if $\neg P[c_1, \ldots, c_n]$ is unsatisfiable for new constants $c_1, \ldots, c_n$. If we put $\neg P[c_1, \ldots, c_n]$ into DNF:

$$Q_1[c_1, \ldots, c_n] \vee \cdots \vee Q_k[c_1, \ldots, c_n],$$

then, since no variables are involved, the whole formula is satisfiable precisely if one of the $Q_i[c_1, \ldots, c_n]$ is. Each such formula is just a conjunction of equations and inequations:

$$s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge u_1 \neq v_1 \wedge \cdots \wedge u_m \neq v_m.$$

Returning to validity by negation, we need to test validity of

$$s_1 = t_1 \wedge \cdots \wedge s_n = t_n \Rightarrow u_1 = v_1 \vee \cdots \vee u_m = v_m.$$

If $m = 1$, we know from Theorem 4.8 that this can be tested by forming the congruence closure of $\sim$ of $\{(s_1, t_1), \ldots, (s_n, t_n)\}$ and testing if $u_1 \sim v_1$. We now observe that for general m, the formula is valid precisely if for some $1 \leq i \leq m$ the formula $s_1 = t_1 \wedge \cdots \wedge s_n = t_n \Rightarrow u_i = v_i$ is valid, by the convexity property for Horn clauses (Theorem 3.43), since we can consider the problem as deduction in first-order logic without equality from the (Horn) equality axioms and the hypotheses $s_k = t_k$. Alternatively, the proof of Theorem 4.8 extends easily to cover this generalization.

To set up the initial 'predecessor' function we use the following, which updates an existing function `pfn` with a new mapping for each immediate subterm `s` of a term `t`:

```
let predecessors t pfn =
  match t with
    Fn(f,a) -> itlist (fun s f -> (s |-> insert t (tryapplyl f s)) f)
                      (setify a) pfn
  | _ -> pfn;;
```

Hence, the following tests if a list `fms` of ground equations and inequations is satisfiable. This list is partitioned into equations (`pos`) and inequations (`neg`), which are mapped into lists of pairs of terms `eqps` and `eqns` for easier manipulation. All the left-hand and right-hand sides are collected in `lrs`, and the predecessor function `pfn` is constructed to handle all their subterms. (Note that it is only `pfn` that determines the overall term set.) Then congruence closure is performed starting with the trivial equivalence relation `unequal`, and iteratively calling `emerge` over all the positive equations. Then it is tested whether all the lefts and rights of all the negated equations are inequivalent.

```
let ccsatisfiable fms =
  let pos,neg = partition positive fms in
  let eqps = map dest_eq pos and eqns = map (dest_eq ** negate) neg in
  let lrs = map fst eqps @ map snd eqps @ map fst eqns @ map snd eqns in
  let pfn = itlist predecessors (unions(map subterms lrs)) undefined in
  let eqv,_ = itlist emerge eqps (unequal,pfn) in
  forall (fun (l,r) -> not(equivalent eqv l r)) eqns;;
```

The overall decision procedure now becomes the following:

```
let ccvalid fm =
  let fms = simpdnf(askolemize(Not(generalize fm))) in
  not (exists ccsatisfiable fms);;
```

Let us try a few examples. In this one, the first disjunct always holds, but we include another disjunct to show that we can deal with arbitrary formulas.

```
# ccvalid <<f(f(f(f(f(c))))) = c /\ f(f(f(c))) = c
            ==> f(c) = c \/ f(g(c)) = g(f(c))>>;;
- : bool = true
```

On the other hand, the following is not valid:

```
# ccvalid <<f(f(f(f(c)))) = c /\ f(f(c)) = c ==> f(c) = c>>;;
- : bool = false
```

The congruence closure algorithm and its proof that we have presented essentially follows Nelson and Oppen (1980). There are asymptotically faster

algorithms for congruence closure (Downey, Sethi and Tarjan 1980), but the Nelson–Oppen algorithm seems adequate for most typical examples. One drawback is that we need to decide the term universe once and for all based on the hypotheses and the goal. For some applications, it's preferable to be able to maintain the equivalence relation incrementally so that the relation can be augmented with new equalities *and* the term universe expanded as new goals are encountered, in which case another algorithm due to Shostak (1978) may be preferable.

The earliest decision procedure for this problem was given by Ackermann (1954) using a slightly different technique. He observed that matters can be reduced to the theory of equality without functions by introducing new variables for all subterms and adding new constraints to reflect congruence properties. For example, given the problem $f(f(f(c))) = c \wedge f(f(c)) = c \Rightarrow f(c) = c$, we could introduce variables $x_k = f^k(c)$ for $0 \leq k \leq 3$ and consider the problem:

$$\begin{array}{l}(x_0 = x_1 \Rightarrow x_1 = x_2) \wedge \\ (x_0 = x_2 \Rightarrow x_1 = x_3) \wedge \\ (x_1 = x_2 \Rightarrow x_2 = x_3) \wedge \\ \Rightarrow x_3 = x_0 \wedge x_2 = x_0 \Rightarrow x_1 = x_0.\end{array}$$

This *Ackermann reduction* can be taken still further by replacing the equations $s = t$ between variables by propositional atoms $P_{s,t}$ and adding further constraints to reflect equivalence properties like $P_{s,t} \wedge P_{t,u} \Rightarrow P_{s,u}$, so reducing the problem simply to propositional tautology checking (Exercise 4.4).

4.5 Rewriting

In the more general case of nonground equations, matters are no longer so simple. In order to find a Birkhoff proof of $s = t$ from hypotheses E, we may have to use arbitrarily large and complex intermediate terms. However, a lot of everyday equational reasoning is very straightforward, mostly using equations in a predictable direction. For example, we would normally think of using the group axiom $i(x) \cdot x = 1$ left-to-right in order to make expressions 'simpler'. It's precisely when we have to use it backwards to make a larger intermediate term that proofs tend to become much harder. (See the group theory puzzle in Section 4.3 for an example.) Admittedly the definition of what is 'simpler' can be subtle. For instance, in algebra we often regard using distributive laws to transform:

$$(u + v)(x + y) \rightarrow \cdots \rightarrow ux + uy + vx + vy$$

as a simplification. This makes the term larger, but it does makes it easier to perform subsequent cancellation operations. Using equations in a directional fashion like this is called *rewriting*, because equations are used to 'rewrite' one term into another.† More precisely, if t is a term, and $l = r$ an equation, we say that t' results from rewriting t with $l = r$ if t' is t with a subterm that is an instance of l replaced by a corresponding instance of r. Note that a single rewriting step only transforms a *single* subterm. For instance, the equation $x + x = 2x$ can rewrite the term $(a + a) + (b + b)$ into either $2a + (b + b)$ or to $(a + a) + 2b$, but not (in a single step) to $2a + 2b$.

Given a set R of equations to be considered as left-to-right rewrite rules, we write $t \rightarrow_R t'$ iff there is some equation $(l = r) \in R$ which rewrites t to t'. When the set of rewrites R is clear from the context, we may just write $t \rightarrow t'$. Note that rewriting is logically sound, in the sense that $t = t'$ holds in any model of the equations R, and we could if we wish decompose each rewriting step into a series of Birkhoff rule applications.

If we're trying to prove that $E \Rightarrow s = t$ where E is closed (a conjunction of universally quantified equations in the present situation), then by Theorem 3.11 we're justified in replacing all free variables in s and t by new constants. So we can if we wish always assume that the terms we're rewriting are ground. In principle, rewrite rules might have variables on the RHS that do not occur in the LHS (e.g. $y \cdot 0 = 0 \cdot x$), and this could make intermediate terms non-ground. However, as the reader might expect, these tend to spoil the nice properties of rewriting, and we will never use rewriting with such terms. In fact, many authors define a *rewrite rule* to be an equation $l = r$ where $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$ and l is not a variable. (A term with a variable LHS could be applied to any term, and is hence not likely to be controllable.)

Nevertheless, it's quite convenient to be able to rewrite arbitrary terms, first so that we don't have to transform the initial problem, and also because we sometimes want to rewrite some of the rewrite rules themselves with others. On the other hand, even if it does involve variables, we *don't* want to permit instantiation of the term being rewritten, since that would spoil the idea that we are simplifying a fixed term. The extension of rewriting to allow instantiation of the term being rewritten is known as *narrowing* (Fay 1979; Hullot 1980); it is a special case of *paramodulation* which we consider later.

† The first explicit use of rewriting seems to have been described by Wos, Robinson, Carson and Shalla (1967), and the original term 'demodulation' from that paper is still used instead of 'rewriting' in some parts of the resolution theorem proving community; see Section 4.9.

Canonical rewrite systems

Sometimes, a simplification procedure has the property that all 'equivalent' expressions reduce to the *same* simplified form. In such cases we can decide whether s and t are equivalent by reducing both s and t to their simplified forms s' and t' and then comparing s' and t' syntactically (Evans 1951). In equational reasoning with hypotheses E, it is natural to call s and t equivalent iff $E \models s = t$. We call E a *canonical* or *convergent* rewrite system when it can be decided whether $E \models s = t$ by treating E as a set of rewrite rules, repeatedly rewriting s and t as much as possible to give s' and t' respectively, and comparing the results. That is, we can rewrite each term to a 'canonical' or 'normal' form, so that all terms s and s' with $E \models s = s'$ have the same normal form. For example, the following set of rewrite rules can be thought of as embodying evaluation rules for addition of numbers written in terms of 0 and a successor operation S, though they have other models:

$$\{m + 0 = m, 0 + n = n, m + S(n) = S(m + n), S(m) + n = S(m + n)\}.$$

No intelligence or creativity is required: even where there are several possible ways of reducing a term, we cannot make an irrevocable wrong decision that will lead us away from the canonical form, e.g. reducing $S(0) + S(S(0))$ in this way:

$$\begin{aligned} S(0) + S(S(0)) &\to S(0 + S(S(0))) \\ &\to S(S(S(0))), \end{aligned}$$

or another:

$$\begin{aligned} S(0) + S(S(0)) &\to S(S(0) + S(0)) \\ &\to S(S(S(0) + 0)) \\ &\to S(S(S(0 + 0))) \\ &\to S(S(S(0))). \end{aligned}$$

Of course, from the point of view of *efficiency*, it may matter which rewrite we choose (e.g. if we have a rule $0 \cdot x = 0$, it makes sense to apply it to a term $0 \cdot E$ without performing reductions on E). And there are surprisingly simple rewrite systems that, although terminating in principle, can lead to infeasibly lengthy reduction sequences, e.g. (Hofbauer and Lautemann 1989):

$$\begin{aligned} \{\ & f(x) + (y + z) = x + (f(f(y)) + z), \\ & f(u) + (v + (w + x)) = u + (w + (v + x))\}. \end{aligned}$$

Let us neglect efficiency for now, and ask how canonicality can fail completely. Using the singleton set $\{x + y = y + x\}$ any subterm $a + b$ can be

rewritten indefinitely, and for this reason that set is not canonical:

$$a + b \to b + a \to a + b \to b + a \to a + b \to \cdots$$

Rewriting with the following rewrite set:

$$\{\ x \cdot (y + z) = x \cdot y + x \cdot z, (x + y) \cdot z = x \cdot z + y \cdot z\}$$

can never be continued indefinitely (we will prove this later), but we may not get a well-defined result in that even the same term can sometimes be rewritten to different irreducible forms, e.g.

$$\begin{aligned}(a + b) \cdot (c + d) &\to a \cdot (c + d) + b \cdot (c + d) \\ &\to (a \cdot c + a \cdot d) + b \cdot (c + d) \\ &\to (a \cdot c + a \cdot d) + (b \cdot c + b \cdot d)\end{aligned}$$

or

$$\begin{aligned}(a + b) \cdot (c + d) &\to (a + b) \cdot c + (a + b) \cdot d \\ &\to (a \cdot c + b \cdot d) + (a + b) \cdot d \\ &\to (a \cdot c + b \cdot d) + (a \cdot d + b \cdot d).\end{aligned}$$

Abstract reduction relations

The examples above hint at two critical properties we need, roughly speaking:

- termination – starting from any term, we must eventually reach a form that can no longer be further reduced;
- confluence – starting from any term, if we apply the simplification rules in different orders to get different intermediate results, we can subsequently 'rejoin' them by further reductions.

We will now define these more precisely and show that together they give us the results we need. However, it's convenient to work in the more general context of an arbitrary binary relation on a set, rather than merely rewrite relations over terms. This helps to clarify the essential theoretical features without introducing technical complications, and also allows us to re-use some of the key results in a different context later on.† Our view is fairly pragmatic and we only scratch the surface of the subject; for a more thorough treatment see, for example, Klop (1992).

† See Section 5.11 on Gröbner bases. Many of these concepts were first articulated in contexts other than rewriting, e.g. reductions in untyped lambda calculus (Barendregt 1984; Hindley and Seldin 1986).

An abstract reduction relation is simply a binary relation R on a set X, though we jog our intuition by writing $x \to y$ instead of $R(x, y)$, and the reader may like to keep in mind the special case of rewrite relations. In the following, we denote by $\to^+$ the transitive closure of $\to$ and by $\to^*$ its reflexive transitive closure (see Appendix 1). That is, $x \to^+ y$ if there is a possibly-empty sequence of elements $x_i \in X$ with $x \to x_1 \to \cdots \to x_n \to y$, and $x \to^* y$ if $x \to^+ y$ or $x = y$.

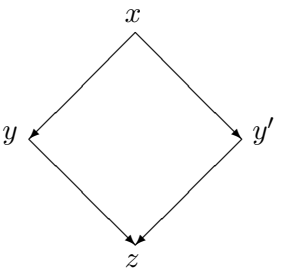
An $x \in X$ is said to be in *normal form* iff there is no $y \in X$ with $x \to y$. In the context of rewriting, a term is in normal form w.r.t. $\to_R$ precisely when no rewrites from R can be applied to it. A reduction relation is said to be *terminating*, *strongly normalizing* (SN) or *noetherian* iff there is no infinite reduction sequence $x_0 \to \cdots \to x_n \to \cdots$.[†]

Considering the reverse relation defined by $x < y =_{\text{def}} y \to x$, we see that x is in normal form iff it is minimal with respect to $<$, and $\to$ is terminating precisely if $<$ is wellfounded. Thus, the two concepts just defined are familiar in another guise, and we can take over corresponding theorems with trivial changes. For example, the transitive closure of a terminating relation is also terminating, and we can perform induction over a terminating relation: if $\to$ is terminating and we can establish that $P(x)$ holds whenever $P(y)$ holds for all y such that $x \to y$, then we may conclude $P(x)$ for all $x \in X$. We'll apply this principle shortly. (Note that this includes the degenerate case of establishing $P(x)$ for all x in normal form.)

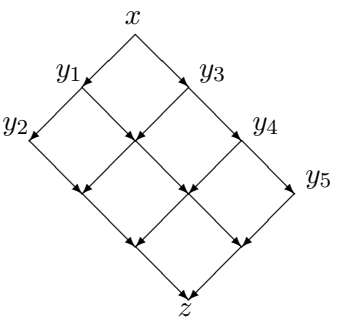
An abstract reduction relation is said to have the *diamond property* iff whenever $x \to y$ and $x \to y'$, there is a z such that $y \to z$ and $y' \to z$. It is said to be *confluent* if $\to^*$ has the diamond property. It is said to be *weakly confluent* if whenever $x \to y$ and $x \to y'$, there is a z such that $y \to^* z$ and $y' \to^* z$. We say for short that x and y are *joinable*, and write $x \downarrow y$, to mean that there is a z with $x \to^* z$ and $y \to^* z$, so we can express confluence as 'if $x \to^* y_1$ and $x \to^* y_2$ then $y_1 \downarrow y_2$' and weak confluence as 'if $x \to y_1$ and $x \to y_2$ then $y_1 \downarrow y_2$'.

The name 'diamond property' comes from the convenient diagrammatic representation of reductions as descending diagonal lines moving from the first element to the second. Thus the forms of confluence all assert that given reductions from x to both y and y', there is a z with reductions from both y and y' to z; the forms only differ in whether we have $\to$ or $\to^*$ at the top or bottom.

[†] Weak normalization (WN) means that for each x there is a y in normal form such that $x \to^* y$. We won't use this concept but it seems worth noting the distinction in case the reader wants to delve deeper into such material.



All the variations on a theme of confluence are closely interrelated. If $\rightarrow$ has the diamond property, it is weakly confluent, since $y \rightarrow z$ trivially implies $y \rightarrow^* z$. For similar reasons, confluence implies weak confluence. It is not much harder to see that the diamond property implies confluence, by double induction on the lengths of the initial reduction sequences $x \rightarrow^* y$ and $x \rightarrow^* y'$. For example, if we have a 2-step reduction $x \rightarrow y_1 \rightarrow y_2$ and a 3-step reduction $x \rightarrow y_3 \rightarrow y_4 \rightarrow y_5$ we can show that there is a z with $y_2 \rightarrow^* z$ and $y_5 \rightarrow^* z$ by repeatedly using the diamond property to fill in the internal lines in this diagram, starting at the top and ending with some suitable z:



On the other hand, weak confluence does not in general imply confluence; the following is a particularly simple counterexample due to Hindley. (One can think of this as specifying a term rewriting system where a, b, c and d are all constants, or simply as an exhaustive enumeration of an abstract binary relation.)

$$\begin{aligned} b &\rightarrow a \\ b &\rightarrow c \\ c &\rightarrow b \\ c &\rightarrow d \end{aligned}$$

Still, for a *terminating* reduction relation, weak confluence does imply confluence. This key result is known as *Newman's lemma*. The original proof (Newman 1942) was rather complicated, and it was only much later that Huet (1980) pointed out the following relatively straightforward proof, exploiting the fact that when $\rightarrow$ is terminating we can perform wellfounded induction.

Theorem 4.9 *If $\rightarrow$ is terminating and weakly confluent, then it is confluent.*

Proof Since $\rightarrow$ is terminating, all reduction sequences terminate, so we just need to prove that if $x \rightarrow^* y$ and $x \rightarrow^* y'$ with y and y' in normal form, then $y = y'$. We will prove this by wellfounded induction: suppose x is the minimal element such that for some y and y' this fails.

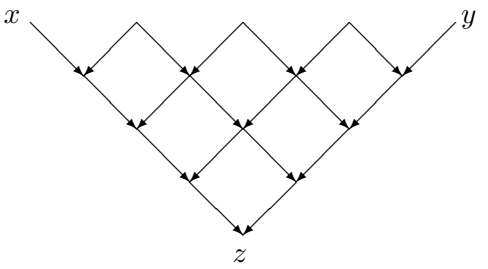
The assertion is vacuous if $x = y$ or $x = y'$, so we can assume the existence of w and w' such that $x \rightarrow w \rightarrow^* y$ and $x \rightarrow w' \rightarrow^* y'$. Weak confluence tells us that there's a z with $w \rightarrow^* z$ and $w' \rightarrow^* z$; by continuing the reduction as much as possible we can assume z to be in normal form. But by the fact that y and y' are successors of x and x was the minimal case where the key property fails, we have $y = z$ and $y' = z$, and so $y' = y$ as required. □

Let us write $\leftrightarrow^*$ for the reflexive *symmetric* transitive closure of $\rightarrow$. We say that $\rightarrow$ is *Church–Rosser* if whenever $x \leftrightarrow^* y$ then $x \downarrow y$.[†] We will prove in fact that the Church–Rosser property is equivalent to confluence, so the two terms may be, and sometimes are, used synonymously. In one direction this is easy, since confluence is a special case of the Church–Rosser property: if $x \rightarrow^* y_1$ and $x \rightarrow^* y_2$ then $y_1 \leftrightarrow^* y_2$. In the other direction, if $x \leftrightarrow^* y$ then we can get from x to y by a series of steps that we can separate into alternating 'forward' and 'backward' segments,

$$x \cdots \rightarrow^* x_i \leftarrow^* x_{i+1} \rightarrow^* x_{i+2} \leftarrow^* \cdots y.$$

Because of confluence, we can at each stage find a suitable z_i such that $x_i \rightarrow^* z_i$ and $x_{i+2} \rightarrow^* z_i$ and hence successively reduce the number of segments, filling in the internal sides in the diagram until we eventually reach a final z with $x \rightarrow^* z$ and $y \rightarrow^* z$.

[†] The peculiar name 'Church–Rosser' arises from the fact that the first significant instance was proved for the case of β-reduction in lambda calculus by Church and Rosser (1936).



In what follows, we recast this argument as a formal induction. Note that we do *not* need to assume termination to show that interconvertible elements are joinable.

Theorem 4.10 *Confluence is equivalent to the Church–Rosser property, i.e.* $\rightarrow$ *is confluent if and only if for any* x *and* y *we have* $x \leftrightarrow^* y$ *iff* $x \downarrow y$.

Proof Since $x \downarrow y$ is a special case of $x \leftrightarrow^* y$, we just need to prove that confluence is equivalent to 'if $x \leftrightarrow^* y$ then $x \downarrow y$'. As noted above, the right-to-left direction is easy because confluence is a special case of the Church–Rosser property. For the other direction, we proceed by induction on the definition $x \leftrightarrow^* y$. If we actually have $x \rightarrow y$ then trivially $x \downarrow y$ because $x \rightarrow^* y$ and $y \rightarrow^* y$. Even more trivially, if x and y are identical, they are joinable. If $x \leftrightarrow^* y$ is obtained by symmetry from $y \leftrightarrow^* x$, then by the inductive hypothesis $y \downarrow x$, and since joinability is symmetric between x and y we have $x \downarrow y$. Finally, if $x \leftrightarrow^* y$ arises by transitivity from $x \leftrightarrow^* z$ and $z \leftrightarrow^* y$, we have by the inductive hypothesis some u and v with $x \rightarrow^* u$, $z \rightarrow^* u$ and $z \rightarrow^* v$, $y \rightarrow^* v$. Using confluence, there is a z such that $u \rightarrow^* z$ and $v \rightarrow^* z$. By transitivity of $\rightarrow^*$, we therefore have $x \rightarrow^* z$ and $y \rightarrow^* z$ as required. □

Another useful lemma about joinability is the following.

Lemma 4.11 *A reduction relation* $\rightarrow$ *is confluent iff the corresponding joinability relation is transitive, i.e. for all* x, y *and* z *such that* $x \downarrow y$ *and* $y \downarrow z$ *we have* $x \downarrow z$.

Proof If $\rightarrow$ is confluent, the previous result shows that $x \downarrow y$ coincides with $x \leftrightarrow^* y$, and the latter is clearly transitive. (It's also easy to reason more directly.)

Conversely, suppose joinability is transitive. If $p \to^* q_1$ and $p \to^* q_2$ then $p \downarrow q_1$ and $p \downarrow q_2$. Using the obvious symmetry and assumed transitivity of $\downarrow$, we see that $q_1 \downarrow q_2$ so the relation is confluent. □

We say that a reduction relation is *canonical* when it is both terminating and confluent. Note that if $\to$ is canonical, then whenever $x \to^* x'$ and $y \to^* y'$ with x' and y' in normal form, we have $x \leftrightarrow^* y$ iff $x' = y'$. In the special case of a rewrite relation, this justifies exactly the kind of process for testing $E \models s = t$ that we outlined at the start of this section, by virtue of the following theorem.

Theorem 4.12 *For a rewrite relation $\to_R$ generated by a set of rewrites R, for all terms s and t we have $s \leftrightarrow^*_R t$ iff $R \models s = t$.*

Proof One way is relatively easy: if $s \to_R t$ then $R \models s = t$ because t results from replacing s according to an equation in R. By induction, the same applies when $s \leftrightarrow^*_R t$.

Conversely, if $R \models s = t$ then by Theorem 4.7 we have $R \vdash s = t$. We will show by induction on the Birkhoff rules that if $R \vdash s = t$ then also $s \leftrightarrow^*_R t$. Closure of $\leftrightarrow^*_R$ under reflexivity, symmetry and transitivity is immediate, and if $(s = t) \in R$ then by a trivial rewrite step $s \leftrightarrow^*_R t$. We will be finished if we can establish that $\leftrightarrow^*_R$ is closed under congruence and instantiation. Both of these follow (formally, by another induction) by systematically applying the congruence or instantiation to all elements in the transitivity chain, since the core rewrite relation $\to_R$ is closed in this way. □

Implementing rewriting

To rewrite a term t at the top level with an equation $l = r$ we just attempt to match l to t and apply the corresponding instantiation to r; the following does this with the first in a list of equations to succeed:

```
let rec rewrite1 eqs t =
  match eqs with
    Atom(R("=",[l;r]))::oeqs ->
     (try tsubst (term_match undefined [l,t]) r
      with Failure _ -> rewrite1 oeqs t)
  | _ -> failwith "rewrite1";;
```

Our interest is in rewriting at all subterms, and repeatedly, to normalize a term w.r.t. a set of equations. Although, for theoretical reasons, in particular for applying Newman's Lemma, it's important to single out the 'one-step'

(though at depth) rewrite relation $\rightarrow_R$, from an implementation point of view we needn't bother isolating it. The following function simply applies rewrites at all possible subterms and repeatedly until no further rewrites are possible. The user is responsible for ensuring that the rewrites terminate, and if this is not the case this function may loop indefinitely. Where several rewrites could be applied, the leftmost outermost subterm in the term being rewritten is always preferred, and thereafter the first applicable equation in the list of rewrites. Alternative strategies such as choosing the innermost rewritable subterm would work equally well in our applications.

```
let rec rewrite eqs tm =
  try rewrite eqs (rewrite1 eqs tm) with Failure _ ->
  match tm with
    Var x -> tm
  | Fn(f,args) -> let tm' = Fn(f,map (rewrite eqs) args) in
                  if tm' = tm then tm else rewrite eqs tm';;
```

Here's a simple example, evaluating $3 * 2 + 4$ in the zero-successor representation of numerals:

```
rewrite [<<0 + x = x>>; <<S(x) + y = S(x + y)>>;
         <<0 * x = 0>>; <<S(x) * y = y + x * y>>]
        <<|S(S(S(0))) * S(S(0)) + S(S(S(S(0))))|>>;;
- : term = <<|S(S(S(S(S(S(S(S(S(S(S(S(0))))))))))|>>
```

It is in general undecidable whether a particular set of equations, used as a rewrite system, is terminating, either for some particular reduction strategy or for all strategies. Indeed, one can express arbitrary algorithms as rewrite systems in a manner not unlike the clausal pattern-matching that is typical in functional programming languages.† The analogy is not exact, since functional languages tend to have many additional constructs and a particular evaluation strategy. On the other hand, in one respect the standard clausal function definitions are simpler than general rewrite rules because they are *linear*, meaning that each variable occurs at most once on the left-hand side. (For example, OCaml will reject a function definition '`function (x,x) -> 0`' because the variable `x` is bound twice in the pattern.) There is a substantial literature on the theory of linear rewrite rules; they turn out to be in certain respects 'better behaved' than general rewrite rules. In particular, it is more straightforward to analyze their

† To see that *any* algorithm can be suitably encoded, one can observe that SK combinator reduction is just a pair of rewrite rules, and it is known that SK combinators can encode all computable functions (Hindley and Seldin 1986). In practice one can often use more direct encodings (see Exercise 4.7).

confluence without assuming termination. The connection with functional programming is examined in detail by Huet and Lévy (1991).

4.6 Termination orderings

One way of showing that a reduction $\rightarrow$ is terminating is to show that it is included in another relation $>$ (i.e. whenever $s \rightarrow t$ we also have $s > t$) that is itself terminating. For a suitable $>$, this can be more tractable than a direct attack on $\rightarrow$. In particular, for a rewrite relation, things are much more straightforward when it suffices to consider $l > r$ for the equations $(l = r) \in R$ themselves, rather than the induced rewrite relationship, which may involve instantiations and substitution at an arbitrary (single) subterm. This motivates the following definition.

Definition 4.13 *A binary relation $>$ on terms is said to be a* rewrite order *if it is transitive and irreflexive and is closed under instantiation and simple congruences (within a fixed set of function symbols understood implicitly), i.e.*

- *it is never the case that $t > t$,*
- *if $s > t$ and $t > u$ then $s > u$,*
- *if $s > t$ then* `tsubst` *i $s >$* `tsubst` *i t,*
- *if $s > t$ then*

$$f(u_1, \ldots, u_{i-1}, s, u_{i+1}, \ldots, u_n) > f(u_1, \ldots, u_{i-1}, t, u_{i+1}, \ldots, u_n).$$

A rewrite order that is terminating is said to be a *reduction order*. Note that in this case the irreflexivity clause is redundant since a wellfounded relation is automatically irreflexive (if $t > t$ then $t > t > t > \cdots$ would be an infinite descending chain).

Lemma 4.14 *If $>$ is a reduction order and $l > r$ for each equation $(l = r) \in R$, then the rewrite relation $\rightarrow_R$ is terminating.*

Proof By definition $s \rightarrow_R t$ if there is some instantiation $l' = r'$ of an equation $(l = r) \in R$ such that t results from s by replacing a single instance of l' with r'. By hypothesis, $l > r$, and since $>$ is closed under instantiation $l' > r'$. Repeatedly using the fact that $>$ is closed under simple congruences, we see that $s > t$. Therefore, the rewrite relation $\rightarrow_R$ is included in the relation $>$ and is consequently also terminating. □

Measure-based orders

How do we find a suitable reduction order for a given rewrite set? One of the standard techniques for generating wellfounded relations is to use a measure function to map into a familiar wellfounded set such as $\mathbb{N}$, using the fact that if $<$ is wellfounded then so is the relation defined by $x \prec y =_{\text{def}} m(x) < m(y)$. In our context, a natural idea is to consider the 'size' of terms. Denote by $|t|$ the number of variables and function symbols in t, which we can compute like this:

```
let rec termsize tm =
  match tm with
    Var x -> 1
  | Fn(f,args) -> itlist (fun t n -> termsize t + n) args 1;;
```

We might hope to define a reduction order $s > t$ by $|s| > |t|$. Since the size is always a positive integer, this is wellfounded and is also transitive and obeys the congruence property. However, it fails the instantiation property; for example $f(x, x, x) > g(x, y)$ but if we instantiate y to $f(x, x, x)$ we have $f(x, x, x) \not> g(x, f(x, x, x))$. A little thought will convince the reader that it's the presence of variables that occur more often in the smaller term than the larger term that is the source of the problem. One can fix this by defining $s > t$ if both $|s| > |t|$ *and* $|s|_x \geq |t|_x$ for each $x \in \text{FVT}(t)$, where $|t|_x$ denotes the number of occurrences of x in t. However, although this does yield a reduction order (as the reader can confirm), it's poorly suited to the kinds of equations we often encounter in algebraic theories. Two typical examples are associative and distributive laws:

- $(x \cdot y) \cdot z = x \cdot (y \cdot z)$,
- $x \cdot (y + z) = x \cdot y + x \cdot z$.

Both sides of the associative law have equal measure, so we can't use the size-based ordering whichever way round it's written. And for the distributive law things are even worse: the right-hand side is larger than the left, despite the fact that we might want to consider expanding using it left-to-right.

Lexicographic path orders

These problems with simple measure-based orders suggest that to deal with typical algebraic examples, we need first to be able to:

- treat the arguments to functions asymmetrically, so that applying the associative law in one preferred direction is possible;

- treat the function symbols asymmetrically so that we can say, for example, that replacing the top-level function symbol f by g represents 'progress', even if the term grows in size.

It is possible to do both of these things with more elaborate measure-based orderings. However, the most direct method is simply to define an ordering on terms by recursion, explicitly designed to 'force' the required properties. To deal with the associative law, for example, we can say that:

- $f(s_1, \ldots, s_m) > f(t_1, \ldots, t_m)$ if the sequence $s_1, \ldots, s_m$ is lexicographically greater than $t_1, \ldots, t_m$, i.e. if $s_i = t_i$ for all $i < k \leq m$ and $s_k > t_k$ under the same ordering.

This ensures that $(x \cdot y) \cdot z > x \cdot (y \cdot z)$ provided $x \cdot y > x$. It's natural to also arrange more generally that $s > t$ whenever t is a proper subterm of s. It's more in keeping with the structurally recursive nature of the other clauses if we just specify it for immediate subterms; the general result then follows by induction. Note that this includes the special case that if t is a variable x we have $s > x$ whenever $x \in \mathrm{FVT}(s)$, excluding the reflexive case when $s = x$.

- $f(s_1, \ldots, s_n) > t$ whenever $s_i \geq t$.

Finally, in order to impose a precedence on function symbols, allowing us to deal with the distributive law by 'preferring' '$\cdot$' to '$+$' or vice versa, we can stipulate:

- $f(s_1, \cdots, s_m) > g(t_1, \ldots, t_n)$ if $f > g$ according to some specified precedence ordering of the function symbols, without further analysis of the s_i and t_i.

These desiderata are almost enough to allow us to define the ordering directly by recursion. However, as it stands the requirements are stated too bluntly and are not enough to ensure termination. For example, instead of the correct distributive law, consider $x \cdot (y+z) = x \cdot (z+y)+z$. The LHS is still greater than the RHS according to the ordering as specified so far, but it is nonterminating. We therefore refine things slightly to ensure that the proper subterms of the RHS must also be less than the starting term on the left, i.e. that $f(s_1, \ldots, s_m) > g(t_1, \ldots, t_n)$ (whether or not $f = g$) only if in addition $f(s_1, \ldots, s_m) > t_i$ for each $1 \leq i \leq n$. It isn't immediately obvious that this fix is enough to ensure termination, but we will prove it below. The resulting order is called the *lexicographic path order* (LPO). More properly, it specifies a whole class of LPOs parametrized by the particular 'weighting' of function

symbols chosen. We can render the definition in OCaml quite directly. First we define the general lexicographic extension of an arbitrary relation `ord`. It always returns falsity when applied to lists of different lengths; this feature is exploited below.

```
let rec lexord ord l1 l2 =
  match (l1,l2) with
    (h1::t1,h2::t2) -> if ord h1 h2 then length t1 = length t2
                       else h1 = h2 & lexord ord t1 t2
  | _ -> false;;
```

Now we define the irreflexive and reflexive versions of the LPO, both of which are parametrized by a 'weighting' `w` on function symbols, where `w` (f, n) (g, m) decides whether the n-ary function f is 'bigger' than the m-ary function symbol g. We will sloppily write $f > g$ for this below, but note from a formal point of view that we treat as distinct function symbols with the same name but different arity.†

```
let rec lpo_gt w s t =
  match (s,t) with
    (_,Var x) ->
        not(s = t) & mem x (fvt s)
  | (Fn(f,fargs),Fn(g,gargs)) ->
        exists (fun si -> lpo_ge w si t) fargs or
        forall (lpo_gt w s) gargs &
        (f = g & lexord (lpo_gt w) fargs gargs or
         w (f,length fargs) (g,length gargs))
  | _ -> false

and lpo_ge w s t = (s = t) or lpo_gt w s t;;
```

Specifying the ordering on function symbols, arities and all, is quite a tedious business. We define the following function to generate a weight function from a more convenient starting point: a list of function symbols in increasing order of precedence. In the (unexpected) case when functions are identical but arities different, we disambiguate by treating functions with larger arity as 'greater':

```
let weight lis (f,n) (g,m) = if f = g then n > m else earlier lis g f;;
```

† This is just for theoretical reasons; we will never actually work with terms containing identically-named function symbols with different arities. In fact we could ignore arities for our present purposes. But for some applications, it is important that the LPO be total on ground terms, and $f(c, c)$ and $f(c)$ would be incomparable if we ignored arities. A common alternative is to use a more general notion of lexicographic extension.

Properties of the LPO

Although the LPO is a more or less natural embodiment of the desiderata we outlined, with fixes to counter the obvious failures of termination, it isn't at all obvious that the final result is terminating, or indeed satisfies other reduction order properties such as transitivity. In fact, if there are infinitely many function symbols with a nonterminating sequence of weights $w(f_1) > w(f_2) > \cdots$, then the LPO is not terminating, but we usually implicitly assume a finite set of function symbols, those that occur in the finitely many formulas we are dealing with. In this case, we will establish that the LPO is a reduction order. Most of the proofs that follow are by induction on the (total) sizes of the terms involved followed by an analysis of the cases in the LPO definition.

Lemma 4.15 *If $s > t$ then $FVT(t) \subseteq FVT(s)$.*

Proof By induction on $|s| + |t|$. If t is a variable x then $s > x$ means that $x \in \mathrm{FVT}(s)$ and therefore $\mathrm{FVT}(x) = \{x\} \subseteq \mathrm{FVT}(s)$, so the result holds. If s is a variable then $s > t$ is false and the result holds trivially. Otherwise we can assume s is of the form $f(s_1, \ldots, s_n)$ and t of the form $g(t_1, \ldots, t_m)$. One way that $s > t$ can arise is if some $s_i \geq t$. But then $\mathrm{FVT}(t) \subseteq \mathrm{FVT}(s_i)$ by the inductive hypothesis and since $\mathrm{FVT}(s_i) \subseteq \mathrm{FVT}(s)$ we have $\mathrm{FVT}(t) \subseteq \mathrm{FVT}(s)$ as required. Otherwise, whatever the relation between f and g we always have $s > t_i$ for $1 \leq i \leq m$. Consequently, by the inductive hypothesis each $\mathrm{FVT}(t_i) \subseteq \mathrm{FVT}(s)$ and therefore $\mathrm{FVT}(t) = \bigcup_{1 \leq i \leq n} \mathrm{FVT}(t_i) \subseteq \mathrm{FVT}(s)$ as required. □

Theorem 4.16 *The LPO is transitive.*

Proof By induction on the total term size $|s| + |t| + |u|$, we show that if $s > t$ and $t > u$ then $s > u$. We sometime use variants of the inductive hypothesis such as the inference that if $s > t \geq u$ then $s > u$. This is an easy consequence since if $t \geq u$ either $t = u$ or $t > u$.

Suppose first that u is a variable x. In this case we have $x \in \mathrm{FVT}(t)$ and $x \neq t$ by definition. But by Lemma 4.15 we also have $\mathrm{FVT}(t) \subseteq \mathrm{FVT}(s)$ and so $x \in \mathrm{FVT}(s)$. We can also rule out $x = s$ because $x > t$ could not then hold. Consequently $s > u$ in this case.

Now assume u is of the form $h(u_1, \ldots, u_p)$. Since we never have $x > u$ it must be the case that t is also of the form $g(t_1, \ldots, t_n)$ and similarly s of the form $f(s_1, \ldots, s_m)$. We now consider the various ways in which $s > t$ and $t > u$ could arise.

First, suppose $f(s_1, \ldots, s_m) > g(t_1, \ldots, t_n)$ arises because for some $1 \leq i \leq m$ we have $s_i \geq g(t_1, \ldots, t_n) = t$. By the inductive hypothesis, $s_i \geq t > u$ implies $s_i > u$, so a fortiori $s_i \geq u$ and therefore also $s > u$ by the definition of the LPO. There now just remains the case where, whatever the relation between f and g, we have $s > t_i$ for each $1 \leq i \leq n$.

Now suppose $g(t_1, \ldots, t_n) > h(u_1, \ldots, u_p)$ arises because for some $1 \leq i \leq n$ we have $t_i \geq h(u_1, \ldots, u_p) = u$. Since $s > t_i$ the inductive hypothesis yields $s > u$ as required.

Otherwise, we may now assume $t > u_i$ for each $1 \leq i \leq p$, and also that $f \geq g \geq h$. By the inductive hypothesis we have $s > u_i$ for each $1 \leq i \leq p$, so the additional condition on $s > u$ is satisfied. If $f > h$, therefore, we have $s > u$ immediately. Otherwise we have $f = g = h$, $m = n = p$ and the lexicographic relations:

$$(s_1, \ldots, s_p) >_{\text{LEX}} (t_1, \ldots, t_p) >_{\text{LEX}} (u_1, \ldots, u_p).$$

By the inductive hypothesis, $s_i > t_j$ and $t_j > u_k$ implies $s_i > u_k$ for any such triple from these subterms. Therefore we also have transitivity of the lexicographic extension and $(s_1, \ldots, s_p) >_{\text{LEX}} (u_1, \ldots, u_p)$, yielding $s > u$ as required. □

Theorem 4.17 *The LPO has the subterm property, i.e. if t is a proper subterm of s then $s > t$.*

Proof Now that we know $>$ is transitive, the result follows by induction on the size of s if we can prove the special case $f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_n) > t$. If t is a variable this holds by definition. Otherwise it is also immediate from the definition since $t \geq t$. □

Theorem 4.18 *The LPO is closed under substitutions, i.e. if $s > t$ then for any instantiation σ we have* `tsubst` σ $s >$ `tsubst` σ t.

Proof Fix an instantiation σ; for any term u we will consistently abbreviate $u' =$ `tsubst` σ u. We proceed by induction on $|s| + |t|$. If t is a variable x we have $x \in \text{FVT}(s)$ so x' is a subterm of s'; since we also have $x \neq s$ it is a *proper* subterm and the result follows from the subterm property.

Otherwise, neither s nor t can be a variable, so we can suppose that s is of the form $f(s_1, \ldots, s_m)$ and t is also of the form $g(t_1, \ldots, t_n)$. Consider the ways in which $s > t$ can arise. If $s_i > t$ for $1 \leq i \leq m$ we have by the inductive hypothesis that $s'_i > t'$. Since s'_i is a proper subterm of s', it follows by transitivity that $s' > t'$. Otherwise the auxiliary condition $s > t_i$

for $1 \leq i \leq n$ implies by the inductive hypothesis that the corresponding condition $s' > t'_i$ holds. If $f > g$ then the required result is immediate. If $f = g$, $m = n$ then we have $(s_1, \ldots, s_m) >_{\mathrm{LEX}} (t_1, \ldots, t_n)$. This means that there is some $1 \leq i \leq n$ such that $s_j = t_j$ for $j < i$ and $s_i > t_i$. Trivially, then $s'_j = t'_j$ for $j < i$ and by the inductive hypothesis $s'_i > t'_i$, thus showing $(s'_1, \ldots, s'_m) >_{\mathrm{LEX}} (t'_1, \ldots, t'_n)$ and hence $s' > t'$ as required. □

Theorem 4.19 *The LPO is a congruence w.r.t. the function symbols, i.e. if $t > u$ then $f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_n) > f(s_1, \ldots, s_{i-1}, u, s_{i+1}, \ldots, s_n)$*

Proof $(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_n) >_{\mathrm{LEX}} (s_1, \ldots, s_{i-1}, u, s_{i+1}, \ldots, s_n)$ since $t > u$ and all preceding terms are identical. Moreover, most of the auxiliary condition follows from the fact that $f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_n) > s_j$ for $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$, while $f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_n) > u$ is immediate from transitivity given the hypothesis $t > u$ and the subterm property $f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_n) > t$ proved previously. □

Theorem 4.20 *The LPO is irreflexive, i.e. $t > t$ never holds.*

Proof By induction on the size of t. If t is a variable then $t > t$ is false by definition because of the $x \neq t$ clause in the definition. If on the other hand we have $t = f(t_1, \ldots, t_n)$, then $t > t$ can only arise because of lexicographic extension $(t_1, \ldots, t_n) >_{\mathrm{LEX}} (t_1, \ldots, t_n)$. But by the inductive hypothesis we never have $t_i > t_i$ for $1 \leq i \leq n$ and there could be no 'first' i such that this holds. □

Tedious as those proofs were, they were mostly a question of following one's nose. Termination, however, is a bit more subtle, though not much more difficult if approached in the right way, using a *minimality* trick. Our proof here is inspired by Ferreira and Zantema (1995); for another relatively short proof see Buchholz (1995).

Theorem 4.21 *The LPO,* restricted to terms based on a finite set of function symbols*, is terminating.*

Proof If there exists an infinite descending chain at all, there exists one $t_0 > t_1 > t_2 > \cdots$ that is *minimal* in the sense that each term has minimal size among those that could possibly appear at that point in an infinite descending chain. More precisely, let us say that a term t is *nonwellfounded* if there is an infinite descending chain starting with t. We will show that if

there is a descending chain, then there is one $t_0 > t_1 > t_2 > \cdots$ with the following properties:

- $|t_0| \leq |s|$ for all nonwellfounded terms s,
- $|t_{i+1}| \leq |s|$ for all nonwellfounded terms s with $t_i > s$.

To show that such a chain exists, proceed by recursion on i. If there is an infinite descending chain, then there is some nonwellfounded element. Let t_0 be one of minimal size (this is not in general unique). Now, having defined a sequence $t_0 > t_1 > \cdots > t_i$ with t_i nonwellfounded, there must be some nonwellfounded s with $t_i > s$ (otherwise t_i would be wellfounded). Again, we can simply pick the minimal one as t_{i+1}.

Now, we never have $t > x$ for a variable x, and so no variable is nonwellfounded and so none of the t_i can be a variable. And since the number of function symbols is by hypothesis finite, there must be at least one function symbol (with particular arity n) that occurs infinitely often as the top-level function in the t_i. We can define a subsequence, i.e. an increasing function $k : \mathbb{N} \to \mathbb{N}$, such that each t_{k_i} is of the form $f(u_1^i, \ldots, u_n^i)$. Now, by the minimality hypothesis, none of the u_j^i can be nonwellfounded, and by transitivity we have $f(u_1^i, \ldots, u_n^i) > f(u_1^{i+1}, \ldots, u_n^{i+1})$ for each i.

Consider the ways in which this can happen according to the definition of the LPO. We cannot have any $u_j^i > f(u_1^{i+1}, \ldots, u_n^{i+1})$, for that would contradict minimality of t_{k_i}. Since the function symbols are the same, we must have $(u_1^i, \ldots, u_n^i) > (u_1^{i+1}, \ldots, u_n^{i+1})$ lexicographically for each i. However the LPO restricted to all the terms u_j^i is wellfounded, and therefore so is its lexicographic extension. We thus arrive at a contradiction. □

A rewrite order with the subterm property ($s > t$ whenever t is a proper subterm of s) is said to be a *simplification order*. Surprisingly, a simplification order turns out to be automatically terminating and hence a reduction order (Dershowitz 1979); by appealing to this result, we could have avoided the direct proof that the LPO is terminating.Typically, one proves relations wellfounded by means of mappings into a wellfounded set like $\mathbb{N}$. But provided the properties of a simplification order hold, mappings into other sets like $\mathbb{R}$ can be useful.

4.7 Knuth–Bendix completion

Suppose we know, perhaps via a suitable ordering as in the previous section, that a rewrite system R is terminating. This is a great help in deciding confluence, because of Newman's lemma (Theorem 4.9): $\rightarrow_R$ is confluent, and hence canonical, iff it is locally confluent. Analyzing local confluence

can be much more tractable than a direct attack on full confluence, because we only need to consider two individual rewrite steps $s \to_R t_1$ and $s \to_R t_2$ and decide whether $t_1 \downarrow_R t_2$. Consider, for example, the following axioms for groups, which can be seen to constitute a terminating rewrite set R using a suitable LPO:

$$\begin{aligned} (x \cdot y) \cdot z &= x \cdot (y \cdot z), \\ 1 \cdot x &= x, \\ i(x) \cdot x &= 1. \end{aligned}$$

We can rewrite the term $(1 \cdot x) \cdot y$ in two different ways, either by the first equation to:

$$(1 \cdot x) \cdot y \to_R 1 \cdot (x \cdot y)$$

or by the second equation to:

$$(1 \cdot x) \cdot y \to_R x \cdot y.$$

However, these are joinable, because we can make an additional rewrite to the first result by the second equation and get $1 \cdot (x \cdot y) \to_R x \cdot y$. On the other hand, if we start from the term $(i(x) \cdot x) \cdot y$, we can rewrite with the first equation to get

$$(i(x) \cdot x) \cdot y \to_R i(x) \cdot (x \cdot y)$$

or by the third to get

$$(i(x) \cdot x) \cdot y \to_R 1 \cdot y.$$

The first term is already in R-normal form, and the only further reduct of the second term is $1 \cdot y \to y$, which is not the same. Consequently, the terms are not joinable so R is not (even locally) confluent.

This example suggests how, given any terminating rewrite set (with a finite number of equations) we can decide its local confluence. We need to discover whether any starting terms s give rise via $s \to_R t_1$ and $s \to_R t_2$ to non-joinable reducts t_1 and t_2. Because R is terminating, joinability of any given t_1 and t_2 can be shown to be decidable, since there are only finitely many possible terms to which each can be rewritten.[†] In fact, with confluence as the overall aim, the situation is even simpler: we need only reduce t_1 and t_2 in some arbitrary way to normal forms t_1' and t_2' and compare them. If they are the same, this particular pair of terms is

[†] This follows at once from König's lemma, which states that a finitely-branching tree without an infinite path has only finitely many nodes. This can be proved simply by wellfounded induction.

joinable, while if they are different we can conclude at once that the whole rewrite set is non-confluent (and hence not locally confluent either) without examining any other possibilities.

Critical pairs

At first sight, this still doesn't help much because we need to consider an arbitrary starting term s, of which there are infinitely many. However it turns out that we can decide local confluence by examining a finite number of critical situations where rewrites can interfere with each other and lead to the failure of local confluence. When $s \to_R t_1$ and $s \to_R t_2$ we can distinguish three possibilities.

- The two rewrites apply to disjoint subterms, for example $(1 \cdot x) \cdot (i(y) \cdot y)$ to $x \cdot (i(y) \cdot y)$ and to $(1 \cdot x) \cdot 1$,
- One rewrite applies to a term that is a (not necessarily proper) subterm of a term to which a variable is instantiated in the other rewrite. For example $((1 \cdot x) \cdot y) \cdot z$ can be rewritten either to $(1 \cdot x) \cdot (y \cdot z)$ or to $(x \cdot y) \cdot z$, but the subterm $1 \cdot x$ to which the second rewrite is applied is exactly the subterm to which x is instantiated in the first rewrite $(x \cdot y) \cdot z \to x \cdot (y \cdot z)$.
- One rewrite applies to a term that is inside the term to which the other rewrite applies, but is not at or below a variable position. Examples include the two rewrites to $(1 \cdot x) \cdot y$ given near the start of this section.

It is only the third situation, when the rewritten subterms are said to 'overlap',[†] that non-confluence can occur, because in the first two cases the subterm to which the other rewrite is applicable is not structurally changed by the chosen rewrite, though in the second case it may be removed or duplicated. Let us analyze this more precisely. Consider the application of two rewrite rules $l_1 = r_1$ and $l_2 = r_2$ to subterms l'_1 and l'_2 of a term s, replacing them with r'_1 and r'_2 respectively. Note that in general we need to consider the case where the two rewrites are identical or are applied to the same subterm. However, if the rewrites *and* the subterm are both identical, we evidently get the same results immediately so confluence is not an issue.

First, if the rewrites are applied to disjoint subterms of $s = s[l'_1, \ldots, l'_2]$ to give $t_1 = s[r'_1, \ldots, l'_2]$ and $t_2 = s[l'_1, \ldots, r'_2]$, we may rejoin t_1 and t_2 by applying the other rewrite to the undisturbed subterm. Thus, in the first case t_1 and t_2 are always joinable.

[†] The terminology is perhaps unfortunate. Despite the misleading impression the concrete syntax might give, two subterms are either disjoint or one is a subterm of the other.

Second, consider the case where one rewrite is applied below the variable position in another. Without loss of generality we will consider the case where $l_2 = r_2$ occurs inside $l_1 = r_1$, the other being symmetric. That is, there is some variable x occurring in $l_1[\ldots, x, \ldots, x, \ldots]$ that is instantiated in l'_1 to some term $u[l'_2]$:

$$l'_1[\ldots, u[l'_2], \ldots, u[l'_2], \ldots],$$

and the other rewrite is applied to one of the subterms (indeed, there may be several of them) $u[l'_2]$. The result of applying $l_2 = r_2$ to one of these subterms, say the first, is:

$$l'_1[\ldots, u[r'_2], \ldots, u[l'_2], \ldots].$$

On the other hand, if we apply $l_1 = r_1$, at the top level we get the following term, where the number of instances of $u[l'_2]$ depends on how many times x occurs in r_1; we choose three as a paradigmatic example:

$$r'_1[\ldots, u[l'_2], \ldots, u[l'_2], \ldots, u[l'_2], \ldots].$$

These two terms are always joinable. To the first we can apply $l_2 = r_2$ repeatedly until *all* the terms $u[l'_2]$ substituted for x are modified to $u[r'_2]$, then apply $l_1 = r_1$ to the whole term. To the second, we can apply $l_2 = r_2$ to all the subterms $u[l'_2]$ and the end result is the same, namely:

$$r'_1[\ldots, u[r'_2], \ldots, u[r'_2], \ldots, u[r'_2], \ldots].$$

We see here the advantages of only needing to prove local confluence: we just make a single rewrite step from s to t_1 and t_2, but are allowed arbitrarily many subsequent steps to rejoin them.

Therefore, in order to decide confluence, we only need to consider non-variable 'critical overlaps', which as the initial examples showed may or may not turn out to be joinable. This is much more appealing, because there are only finitely many essentially different ways that one left-hand side can be overlapped with another: one LHS cannot go below the variable position of the other. The points of overlap may depend on the instantiation, but we can always find the most general instantiation that allows overlap at a given position, if any, via most general unifiers (MGUs), as we will now show.

Definition 4.22 *Suppose $l_1 = r_1$ and $l_2 = r_2$ are two rewrite rules (we assume the variables of the LHSs are disjoint, i.e. $FVT(l_1) \cap FVT(l_2) = \emptyset$). If l'_2 occurs at least once as a non-variable subterm of $l_1 = l_1[l'_2, \ldots, l'_2, \ldots, l'_2]$, and σ is a most general unifier of l_2 and l'_2, then the pair of terms:*

$$(\texttt{tsubst}\ \sigma\ r_1,\ \texttt{tsubst}\ \sigma\ l_1[l'_2, \ldots, r_2, \ldots, l'_2])$$

is said to be a critical pair *of* $l_1 = r_1$ *and* $l_2 = r_2$.

Critical pairs are intended to be 'most general' representatives of the ways in which two rewrites can overlap. Indeed, we have the following key properties.

Lemma 4.23 *Let* $l_1 = r_1$ *and* $l_2 = r_2$ *be two equations with no common variables. If* $s \to_{l_1=r_1} t_1$ *and* $s \to_{l_2=r_2} t_2$ *with* t_1 *and* t_2 *not joinable, then* t_1 *and* t_2 *differ only in two subterms* u_1 *and* u_2 *(i.e.* $t_1 = u[\ldots, u_1, \ldots]$ *and* $t_2 = u[\ldots, u_2, \ldots]$*) such that either* (u_1, u_2) *or* (u_2, u_1) *is an instance of a critical pair.*

Proof The above discussion makes clear that the two rewrites cannot be applied at disjoint positions, nor one at or below a variable subterm of another, for otherwise t_1 and t_2 would be joinable, contrary to hypothesis. Thus there is a nontrivial overlap in the rewrites; without loss of generality we will suppose that $l_2 = r_2$ rewrites inside l_1. Since the two equations have no variables in common, we can assume the same instantiation θ for both l_1 and l_2 in the rewrites. Thus, l_1 has a subterm l_2' that is unifiable with l_2, say $l_1 = l_1 = l_1[\ldots, l_2', \ldots]$, with $\texttt{tsubst}\ \theta\ l_2' = \texttt{tsubst}\ \theta\ l_2$. The two rewrites on the term $\texttt{tsubst}\ \theta\ l_1[\ldots, l_2', \ldots]$ result in $u_1 = \texttt{tsubst}\ \theta\ r_1$ and $u_2 = \texttt{tsubst}\ \theta\ l_1[\ldots, r_2, \ldots]$. Since l_2 and l_2' are unifiable, they have a most general unifier σ, and so $(\texttt{tsubst}\ \sigma\ r_1, \texttt{tsubst}\ \sigma\ l_1[\ldots, r_2, \ldots])$ is a critical pair. By the MGU property, (u_1, u_2) is an instance of this critical pair. □

Theorem 4.24 *A term rewriting system is locally confluent iff all its critical pairs are joinable.*

Proof If a system is locally confluent, then since critical pairs (t_1, t_2) all arise by applying two 1-step rewrites to some starting term s, i.e. $s \to t_1$ and $s \to t_2$, it follows at once that t_1 and t_2 are joinable.

Conversely, suppose all critical pairs are joinable. Now, given any term s, suppose $s \to u_1$ and $s \to u_2$; we will show that u_1 and u_2 are joinable. There are two equations (possibly the same) with $s \to_{l_1=r_1} u_1$ and $s \to_{l_2=r_2} u_2$. Now, by the previous lemma, either u_1 and u_2 are joinable, or u_1 and u_2 differ only in corresponding subterms v_1 and v_2 where (v_1, v_2) is an instance of a critical pair (t_1, t_2). By hypothesis t_1 and t_2 are joinable. Since reduction is closed under substitution (whenever $s \to t$ we also have $\texttt{tsubst}\ \theta\ s \to \texttt{tsubst}\ \theta\ t$), v_1 and v_2 are joinable. Since rewriting allows arbitrary subterms, so are u_1 and u_2. □

Corollary 4.25 *A terminating term rewriting system is confluent iff all its critical pairs are joinable.*

Proof Since the system is terminating, Newman's lemma shows that confluence and local confluence are equivalent, so the result is immediate from the previous theorem. □

We now turn to implementation. As with resolution, we start with the tedious business of preparing for unification by renaming variables. For simplicity, we replace the variables in two given formulas by schematic variables of the form `x_n`:

```
let renamepair (fm1,fm2) =
  let fvs1 = fv fm1 and fvs2 = fv fm2 in
  let nms1,nms2 = chop_list(length fvs1)
                            (map (fun n -> Var("x"^string_of_int n))
                                 (0--(length fvs1 + length fvs2 - 1))) in
  subst (fpf fvs1 nms1) fm1,subst (fpf fvs2 nms2) fm2;;
```

Now we come to finding all possible overlaps. This is a little bit trickier than it looks, because we want to ensure that the MGU discovered at depth eventually gets applied to the whole term. The following function defines all ways of overlapping an equation `l = r` with another term `tm`, where the additional argument `rfn` is used to create each overall critical pair from an instantiation `i`.

The function simply recursively traverses the term, trying to unify `l` with each non-variable subterm and applying `rfn` to any resulting instantiations to give the critical pair arising from that overlap. During recursive descent, the function `rfn` is itself modified correspondingly. For updating `rfn` across the list of arguments we define the auxiliary function `listcases`, which we will re-use later in a different situation:

```
let rec listcases fn rfn lis acc =
  match lis with
    [] -> acc
  | h::t -> fn h (fun i h' -> rfn i (h'::t)) @
            listcases fn (fun i t' -> rfn i (h::t')) t acc;;

let rec overlaps (l,r) tm rfn =
  match tm with
    Fn(f,args) ->
        listcases (overlaps (l,r)) (fun i a -> rfn i (Fn(f,a))) args
                  (try [rfn (fullunify [l,tm]) r] with Failure _ -> [])
  | Var x -> [];;
```

In order to present a nicer interface, we accept equational formulas rather than pairs of terms, and return critical pairs in the same way, by appropriately setting up the initial `rfn`:

```
let crit1 (Atom(R("=",[l1;r1]))) (Atom(R("=",[l2;r2]))) =
  overlaps (l1,r1) l2 (fun i t -> subst i (mk_eq t r2));;
```

For the overall function, we need to rename the variables in the initial formula then find all overlaps of the first on the second and vice versa, unless the two input equations are identical, in which case only one needs to be done:

```
let critical_pairs fma fmb =
  let fm1,fm2 = renamepair (fma,fmb) in
  if fma = fmb then crit1 fm1 fm2
  else union (crit1 fm1 fm2) (crit1 fm2 fm1);;
```

As a simple example, which also illustrates how an equation can have non-trivial overlaps with itself, consider the following:

```
# let eq = <<f(f(x)) = g(x)>> in critical_pairs eq eq;;
- : fol formula list = [<<f(g(x0)) = g(f(x0))>>; <<g(x1) = g(x1)>>]
```

Because of the fairly naive implementation, which doesn't check the trivial case of overlapping identical equations on the same subterm, we get reflexive results. But the other critical pair $(f(g(x_0)), g(f(x_0)))$, arising from two rewrites to $f(f(f(x_0)))$, is non-trivial. Since both terms are in normal form, it shows that the initial 1-element rewrite set is not confluent.

Completion

We could now code up a function to decide if a terminating rewrite system is confluent by finding all the critical pairs $\{(s_i, t_i) \mid 1 \leq i \leq n\}$ between pairs of equations, and for each such (s_i, t_i) reducing the terms to some normal forms s_i' and t_i'. The resulting system is confluent iff all corresponding pairs of terms s_i' and t_i' are syntactically equal. However, rather than merely doing this, we can be more ambitious.

If (s_i', t_i') is a normalized critical pair, then it is a logical consequence of the initial equations, since it results from repeated rewriting with those equations of a common starting term. Thus, we could add $s_i' = t_i'$ or $t_i' = s_i'$ as a new equation, retaining logical equivalence with the old axiom set. It may turn out that with this addition, the set will become confluent. If not, we can repeat the process with remaining critical pairs and any arising from the

new equation. This idea is known as *completion*, and was first systematically investigated by Knuth and Bendix (1970), who demonstrated that it can be a remarkably effective technique for arriving at a canonical rewrite set for many interesting algebraic theories such as groups. It should be noted, however, that success of the procedure is not guaranteed; two things can go wrong.

First, adding $s'_i = t'_i$ or $t'_i = s'_i$ may cause the resulting rewrite set to become nonterminating. To try and avoid this, we will keep a fixed term ordering in mind, and try to orient the equation so that it respects the ordering, but it may turn out that *neither* direction respects the ordering.

Second, although the new equation $s'_i = t'_i$ or $t'_i = s'_i$ trivially means that the originating critical pair (s_i, t_i) is now joinable in the new system, the new equation will in general create new critical pairs, with the existing equations and perhaps even with itself. It's entirely possible that the creation of new critical pairs will 'outrun' their processing into new rules, so that the overall process never terminates.

Despite these provisos, let us implement completion and see it in action. The central component is a procedure that takes an equation $s = t$, normalizes both s and t to give s' and t', and attempts to orient these terms into an equation respecting the given ordering `ord`, failing if this is impossible. We assume `ord` is the reflexive form of ordering, so failure will not occur in the case where s' and t' are identical.

```
let normalize_and_orient ord eqs (Atom(R("=",[s;t]))) =
  let s' = rewrite eqs s and t' = rewrite eqs t in
  if ord s' t' then (s',t') else if ord t' s' then (t',s')
  else failwith "Can't orient equation";;
```

The central completion procedure maintains a set of equations `eqs` and a set of pending critical pairs `crits`, and successively examines critical pairs, normalizing and orienting resulting equations and adding them to `eqs`. However, since the order in which we examine critical pairs is arbitrary, we try to avoid failing too hastily by storing equations that cannot as yet be oriented on a separate 'deferred' list `def`.

Only at the end, by which time these troublesome equations may normalize to the point of joinability, or at least orientability, do we reconsider them, putting the first orientable one back in the main list of critical pairs. The following auxiliary function is used to conditionally emit a report on current status, so that the user gets an idea what's going on.

```
let status(eqs,def,crs) eqs0 =
  if eqs = eqs0 & (length crs) mod 1000 <> 0 then () else
  (print_string(string_of_int(length eqs)^" equations and "^
                string_of_int(length crs)^" pending critical pairs + "^
                string_of_int(length def)^" deferred");
   print_newline());;
```

In the main completion loop, if there is a critical pair left to be examined, we attempt to normalize and orient it; if it is nontrivial (i.e. not of the form $t = t$) we add it to the equations, and augment the critical pairs (at the tail end) with new critical pairs from this new equation and itself plus those already present. If the orientation fails, then we just add the critical pair to the 'deferred' list. Finally, if there are no critical pairs left, we attempt to orient and deal with the deferred critical pairs, starting with any found to be orientable. If we are ultimately left with some that are non-orientable, we fail. Otherwise we terminate with success and return the new equations.

```
let rec complete ord (eqs,def,crits) =
  match crits with
    (eq::ocrits) ->
        let trip =
          try let (s',t') = normalize_and_orient ord eqs eq in
              if s' = t' then (eqs,def,ocrits) else
              let eq' = Atom(R("=",[s';t'])) in
              let eqs' = eq'::eqs in
              eqs',def,
              ocrits @ itlist ((@) ** critical_pairs eq') eqs' []
          with Failure _ -> (eqs,eq::def,ocrits) in
        status trip eqs; complete ord trip
  | _ -> if def = [] then eqs else
         let e = find (can (normalize_and_orient ord eqs)) def in
         complete ord (eqs,subtract def [e],[e]);;
```

The main loop maintains the invariant that all critical pairs from pairs of equations in `eqs` that are not joinable by `eqs` are contained in `crits` and `def` together, so when successful termination occurs, since `crits` and `def` are both empty, there are no non-joinable critical pairs, and so by Corollary 4.25 successful the system is confluent. Moreover, since the original equations are included in the final set and we have only added equational consequences of the original equations, they give a logically equivalent set. In order to get started, we just have to set `crits` to the critical pairs for the original equations and also `def = []`, so the invariant is true to start with.

Before considering refinements, let's try a simple example: the axioms for groups. For the ordering we choose the lexicographic path ordering, with 1 having smallest precedence and the inverse operation the largest. The

intuitive reason for giving the inverse the highest precedence is that it will tend to cause the expansion $(x \cdot y)^{-1} = y^{-1} \cdot x^{-1}$ to be applied (when it is eventually derived), leading to more opportunities for cancellation of multiple inverse operations. Indeed, if we try this out:

```
# let eqs =
   [<<1 * x = x>>; <<i(x) * x = 1>>; <<(x * y) * z = x * y * z>>];;
...
# let ord = lpo_ge (weight ["1"; "*"; "i"]);;
...
# let eqs' = complete ord
   (eqs,[],unions(allpairs critical_pairs eqs eqs));;
```

the completion algorithm terminates successfully after a little computation, and the inverse property is one of the equations deduced as part of the final complete set (first in the list that follows):

```
val eqs' : fol formula list =
  [<<i(x4 * x5) = i(x5) * i(x4)>>; <<x1 * i(x5 * x1) = i(x5)>>;
   <<i(x4) * x1 * i(x3 * x1) = i(x4) * i(x3)>>;
   <<x1 * i(i(x4) * i(x3) * x1) = x3 * x4>>;
   <<i(x3 * x5) * x0 = i(x5) * i(x3) * x0>>;
   <<i(x4 * x5 * x6 * x3) * x0 = i(x3) * i(x4 * x5 * x6) * x0>>;
   <<i(x0 * i(x1)) = x1 * i(x0)>>; <<i(i(x2 * x1) * x2) = x1>>;
   <<i(i(x4) * x2) * x0 = i(x2) * x4 * x0>>;
   <<x1 * i(x2 * x1) * x2 = 1>>;
   <<x1 * i(i(x4 * x5) * x1) * x3 = x4 * x5 * x3>>;
   <<i(x3 * i(x1 * x2)) = x1 * x2 * i(x3)>>;
   <<i(i(x3 * i(x1 * x2)) * i(x5 * x6)) * x1 * x2 * x0 = x5 * x6 * x3 *
     x0>>;
   <<x1 * x2 * i(x1 * x2) = 1>>; <<x2 * x3 * i(x2 * x3) * x1 = x1>>;
   <<i(x3 * x4) * x3 * x1 = i(x4) * x1>>;
   <<i(x1 * x3 * x4) * x1 * x3 * x4 * x0 = x0>>;
   <<i(x1 * i(x3)) * x1 * x4 = x3 * x4>>;
   <<i(i(x5 * x2) * x5) * x0 = x2 * x0>>;
   <<i(x4 * i(x1 * x2)) * x4 * x0 = x1 * x2 * x0>>; <<i(i(x1)) = x1>>;
   <<i(1) = 1>>; <<x0 * i(x0) = 1>>; <<x0 * i(x0) * x3 = x3>>;
   <<i(x2 * x3) * x2 * x3 * x1 = x1>>; <<x1 * 1 = x1>>;
   <<i(1) * x1 = x1>>; <<i(i(x0)) * x1 = x0 * x1>>;
   <<i(x1) * x1 * x2 = x2>>; <<1 * x = x>>; <<i(x) * x = 1>>;
   <<(x * y) * z = x * y * z>>]
```

And, indeed, this complete set gives an effective canonical simplifier for groups based on rewriting, e.g.

```
# rewrite eqs' <<|i(x * i(x)) * (i(i((y * z) * u) * y) * i(u))|>>;;
- : term = <<|z|>>
```

Interreduction

Although `eqs'` does form a canonical rewrite set, it seems to be an unnecessarily large and redundant one. For example, the two sides of $i(x_3 \cdot x_5) \cdot x_0 = i(x_5) \cdot i(x_3) \cdot x_0$ are joinable from the simple inverse law noted above and the associative law. The fact that one equation is joinable by others may mean that the critical pair giving rise to it was processed before the equations that allow it to be joined were derived. Or, since we just blindly normalized them using an essentially arbitrary choice of rewrites at a time when the rewrite set was not confluent, we may just have been unlucky and taken the wrong path even when there was a way to join them.

Whatever their genesis, it's natural to filter out afterwards equations whose two sides are joinable by others. We might even go further by simplifying both sides of each equation using all the others. Plausible as this looks, we need first to satisfy ourselves that the result remains canonical. Indeed, reducing the LHS of an equation may cause it to become mis-oriented, or even non-orientable. Fortunately, however, it turns out that if the LHS of an equation in a canonical term rewriting system is reducible by the other equations, then both sides are automatically joinable by the other equations and it may be discarded. Thus (Métivier 1983) we can simply:

- discard any equation whose LHS is reducible by any of the others (excluding itself);
- reduce the RHS of any equation with all the equations (including itself).

Both these facts follow quite easily from the following general theorem about arbitrary reduction relations.

Theorem 4.26 *Let $\rightarrow_R$ be a canonical (terminating and confluent) reduction relation on a set X (this can be any relation, though the reader may care to think of it as a rewrite relation generated by R). Suppose another reduction relation $\rightarrow_S$ has the following two properties:*

- *for any $x, y \in X$, if $x \rightarrow_S y$ then $x \rightarrow_R^+ y$;*
- *for any $x, y \in X$, if $x \rightarrow_R y$ then there is a $y' \in X$ with $x \rightarrow_S y'$.*

Then $\rightarrow_S$ is also canonical and defines the same equivalence, i.e. two objects are joinable by $\rightarrow_R$ iff they are joinable by $\rightarrow_S$.

Proof First we will prove the lemma that if y is in normal form w.r.t. $\rightarrow_R$, then for any x with $x \rightarrow_R^* y$ we also have $x \rightarrow_S^* y$. Since $\rightarrow_R$ is terminating, we can prove this by wellfounded induction on x, keeping y fixed. Suppose $x \rightarrow_R^* y$. If $x = y$ the result follows at once; otherwise there is a $u \in X$

with $x \to_R u \to_R^* y$. Using the hypotheses relating $\to_R$ and $\to_S$, we deduce that there is some $v \in X$ with $x \to_S v$, and that $x \to_R^+ v$ and so a fortiori $x \to_R^* v$. Since $\to_R$ is confluent, there is therefore a $z \in X$ with $y \to_R^* z$ and $v \to_R^* z$. Since y is in normal form w.r.t. $\to_R$ we must in fact have $z = y$. Therefore we have $v \to_R^* y$. By the inductive hypothesis, $v \to_S^* y$ and by definition of reflexive transitive closure we have $x \to_S^* y$ as required.

Because $\to_S$ is a subrelation of the transitive closure $\to_R^+$, which is itself terminating because $\to_R$ is, $\to_S$ is terminating. To show that it is also confluent, then, we need only prove local confluence and appeal to Newman's lemma. So suppose $x \to_S y_1$ and $x \to_S y_2$. Then by hypothesis $x \to_R^+ y_1$ and $x \to_R^+ y_2$. Since $\to_R$ is confluent, we have some z, which we can by termination assume to be in normal form, such that $y_1 \to_R^* z$ and $y_2 \to_R^* z$. But by the lemma established at the beginning of this proof, $y_1 \to_S^* z$ and $y_2 \to_S^* z$, establishing local and hence full confluence of $\to_S$.

Finally, we need to show that for any $x, y \in X$, $x \downarrow_R y$ iff $x \downarrow_S y$. The right-to-left implication is almost immediate, because $\to_S$ is contained in $\to_R^+$ and therefore $\to_S^*$ is contained in $\to_R^*$. For the other direction, if $x \downarrow_R y$ we can assume by termination that there is a z in normal form w.r.t. $\to_R$ such that $x \to_R^* z$ and $y \to_R^* z$. But now by the lemma at the start of the proof, we also have $x \to_S^* z$ and $y \to_S^* z$. □

Corollary 4.27 *If R is a canonical term rewriting system and $(l = r) \in R$, then if l is reducible by the other equations, the system $R - \{l = r\}$ is also canonical and is logically equivalent.*

Proof We simply need to check that the conditions of Theorem 4.26 are satisfied, with $\to_R$ generated by R and $\to_S$ by $S = R - \{l = r\}$. It is immediate that if $s \to_S t$ then $s \to_R t$, and hence $s \to_R^+ t$, since S is a subset of R. Moreover, if $s \to_R t$ then since l is reducible by $\to_S$, so is s. □

Corollary 4.28 *If R is a canonical term rewriting system and $(l = r) \in R$, let S be the result of replacing the equation $l = r$ in R with $l = r'$ where r' is the R-normal form of r. Then S is also canonical and logically equivalent to R.*

Proof Again, we just need to check the conditions of Theorem 4.26. Suppose first that $s \to_S t$. If this reduction uses the new rule $l = r'$, then there is a transition $s \to_R u \to_R^* t$, where the first step corresponds to the original rewrite $l = r$ and the remaining steps to the normalization of r, with the appropriate subterm and instantiation. This exactly means that $s \to_R^+ t$. On

the other hand, if the reduction does not use the new rule, then trivially $s \to_R t$ and so $s \to_R^+ t$. Now suppose $s \to_R t$. Either this reduction involves $l = r$, in which case it can also be reduced by $l = r'$ and hence by $\to_S$, or it does not, in which case $s \to_S t$ anyway. □

To implement this, we just transfer equations from the input list `eqs` to the output list `dun` as needed, reversing at the end to maintain the order:

```
let rec interreduce dun eqs =
  match eqs with
    (Atom(R("=",[l;r])))::oeqs ->
        let dun' = if rewrite (dun @ oeqs) l <> l then dun
                   else mk_eq l (rewrite (dun @ eqs) r)::dun in
        interreduce dun' oeqs
  | [] -> rev dun;;
```

Applying this to the complete set obtained above, we get a much more elegant and manageable result. In fact, it can be shown (Métivier 1983) that the interreduced set is essentially unique once the reduction ordering is fixed.

```
# interreduce [] eqs';;
- : fol formula list =
[<<i(x4 * x5) = i(x5) * i(x4)>>; <<i(i(x1)) = x1>>; <<i(1) = 1>>;
 <<x0 * i(x0) = 1>>; <<x0 * i(x0) * x3 = x3>>; <<x1 * 1 = x1>>;
 <<i(x1) * x1 * x2 = x2>>; <<1 * x = x>>; <<i(x) * x = 1>>;
 <<(x * y) * z = x * y * z>>]
```

Let us now set up a slightly more convenient interface to completion, so that input equations are oriented, the initial critical pairs are generated automatically, and interreduction is applied afterwards.

```
let complete_and_simplify wts eqs =
  let ord = lpo_ge (weight wts) in
  let eqs' = map (fun e -> let l,r = normalize_and_orient ord [] e in
                           mk_eq l r) eqs in
  (interreduce [] ** complete ord)
  (eqs',[],unions(allpairs critical_pairs eqs' eqs'));;
```

Instead of waiting till the end of the completion process to perform interreduction, it's usually significantly more efficient to simplify and perhaps delete or reorient equations *during* the completion process. Nevertheless, justifying such optimizations is significantly more complicated, particularly in connection with simplification of existing equations on the left (Huet 1981; Baader and Nipkow 1998). And our simple algorithm is already enough to handle most of the examples from the original paper by Knuth and Bendix (1970). One of the more surprising is the following single-axiom system. If one asserts $i(x) \cdot (x \cdot y) = y$, it also follows that $x \cdot (i(x) \cdot y) = y$, and vice versa, without any other assumptions such as associativity. Knuth and Bendix remark that 'this fact can be used to simplify several proofs which appear in the literature, for example in the algebraic structures associated with projective geometry'.

```
# complete_and_simplify ["1"; "*"; "i"]
  [<<i(a) * (a * b) = b>>];;
2 equations and 4 pending critical pairs + 0 deferred
3 equations and 9 pending critical pairs + 0 deferred
3 equations and 0 pending critical pairs + 0 deferred
- : fol formula list =
[<<x0 * i(x0) * x3 = x3>>; <<i(i(x0)) * x1 = x0 * x1>>;
 <<i(a) * a * b = b>>]
```

Knuth and Bendix also demonstrate in their paper some techniques for extending the approach to non-equational axioms. Consider the quite typical 'cancellation' property $\forall x\ y\ z.\ x \cdot y = x \cdot z \Rightarrow y = z$. Although this isn't an equation, it is logically equivalent to $\forall x\ z.\ \exists w.\ \forall y.\ z = x \cdot y \Rightarrow w = y$, as we can confirm automatically:

```
# (meson ** equalitize)
   <<(forall x y z. x * y = x * z ==> y = z) <=>
     (forall x z. exists w. forall y. z = x * y ==> w = y)>>;;
...
- : int list = [5; 4]
```

If we Skolemize this equivalent form we get $\forall x\ y\ z.\ z = x \cdot y \Rightarrow f(x, z) = y$, which is logically equivalent to $\forall x\ y.\ f(x, x \cdot y) = y$, a purely equational property. Thus we can introduce a new operator f and an axiom $\forall x\ y.\ f(x, x \cdot y) = y$, and by the conservativity property of Skolemization (see Section 3.6) anything we can prove that does not involve f must still be true in the original system. Similarly, the language can sometimes be expanded to accommodate otherwise non-orientable rules. For example, if an equation $g(w, x, y) = g(w, x, z)$ is derived, this is an indication that the third argument is irrelevant and we can replace g with a binary function.

Dealing with commutativity

Despite tricks for extending the scope of completion, certain standard algebraic axioms give rise to difficult problems. In particular the commutativity law $x \cdot y = y \cdot x$ cannot be oriented according to any rewrite order, since any such order has to be closed under the instantiation $x \mapsto y$, $y \mapsto x$. There are several approaches to dealing with commutativity, either on its own or in conjunction with other properties such as associativity.

The most sophisticated is to change the notions of matching and unification to treat as equal all associative and commutative rearrangements of the same term. This process is usually called associative–commutative (AC) unification or matching. There are algorithms for these operations, but they are a bit more complicated than regular unification; indeed the first full AC-unification algorithm (Stickel 1981) was only proved to terminate some years after it was first introduced (Fages 1984). Moreover, in contrast to simple unification, single MGUs may not exist, though there are always finitely many; even in matching, for example, $1 \cdot (x \cdot y)$ can be matched to $(2 \cdot 1) \cdot 3$ either by $x \mapsto 2, y \mapsto 3$ or $x \mapsto 3, y \mapsto 2$, neither of which is an instance of the other. The idea of AC-unification can be generalized from unification modulo associative and commutative laws to unification modulo any set of equational axioms (regular unification being the special case of the empty set), and this was actually discussed by Plotkin (1972) some years before algorithms for specific cases like AC were developed. In the general case, however, unification may be undecidable and there may not even be an *infinite* set of most general unifiers (Fages and Huet 1986). Nevertheless, this is an important technique, playing a role in some of the most impressive achievements in automated equational reasoning such as the solution by McCune (1997) of the Robbins conjecture.

A simpler alternative is to re-examine a key idea motivating the definition of rewrite orderings, that we just need to orient an equation $l = r$ once and for all rather than separately considering each individual instance $l' = r'$. Appealing as this is, we can consider dropping it and constraining rewriting by an ordering on the *instances*. This idea seems to have first been used by Boyer and Moore (1977), who used a system like the following to implement associative–commutative normalization for an operator '+':

$$\begin{aligned} x + y &= y + x, \\ x + (y + z) &= y + (x + z), \\ (x + y) + z &= x + (y + z). \end{aligned}$$

Applying these rewrites subject to a suitable ordering constraint on the instances will normalize terms to be right-associated, and also ordered via a kind of 'bubblesort', e.g.

$$(1 + 4) + (3 + 2) \rightarrow 1 + (4 + (3 + 2)) \rightarrow 1 + (3 + (4 + 2))$$
$$\rightarrow 1 + (3 + (2 + 4)) \rightarrow 1 + (2 + (3 + 4)).$$

Assuming that the ordering we use is wellfounded, termination is assured, so to show confluence we just need to demonstrate local confluence. For many common orderings such as LPO, testing local confluence with ordering constraints on instances is decidable (Comon, Narendran, Nieuwenhuis and Rusinowitch 1998). In general it can still be difficult, though in typical cases a fairly straightforward approach based on analyzing all the possible orderings of the subterms in the instances works well; see Exercise 4.15 for the automation of such case analysis and checking. Martin and Nipkow (1990) demonstrate confluence of ordered rewrite systems for many important systems of algebraic axioms using such techniques.

Unfailing completion

Ordered rewriting can also be used to generalize completion to *unfailing completion* (Bachmair, Dershowitz and Plaisted 1989), which will never fail owing to non-orientable equations, but rather will use them with ordered rewriting based on some term ordering, typically an LPO.

Moreover, if implemented appropriately, one can show that even if it never finds a canonical rewrite system, it will eventually find a rewrite system capable of proving $s = t$ by rewriting whenever $s = t$ follows from the starting axioms. Thus, it can form a complete proof procedure for equational logic. This shift in emphasis from finding canonical systems to proving equations is quite natural. After all, if we try to complete the axioms for groups where $x^2 = 1$, then we do not meet with success:

```
complete_and_simplify ["1"; "*"; "i"]
 [<<(x * y) * z = x * (y * z)>>;
  <<1 * x = x>>; <<x * 1 = x>>; <<x * x = 1>>];;
```

If we trace through successive loops of the completion procedure (using `#trace complete;;` before execution), we find that the critical pair $x_2 \cdot x_0 = x_0 \cdot x_2$ is generated, and subsequently put in the deferred list since it is non-orientable. This immediately dooms the standard completion procedure to failure or nontermination, since this equation will never be oriented or rewritten away. Yet from the point of view of first-order theorem proving, we have

rapidly drawn an interesting conclusion (such a group must be commutative) and so this should be considered a success rather than a failure.

4.8 Equality elimination

Many of the ideas from equational logic, such as orienting rewrites into a favoured direction and considering only proper overlaps, can be generalized to full first-order logic. However, the theoretical justification becomes significantly more difficult, and we will not dwell on it. However, we will consider a few approaches to equality handling other than just adding the equality axioms in a preprocessing step. In this section, we briefly consider avoiding equality altogether, then examine a more sophisticated way of preprocessing the input formulas to incorporate the necessary equality properties.

Predicate formulations

One technique that was popular for encoding group theory etc. in the early days of automated reasoning was to use, rather than a 2-argument function symbol, a 3-argument predicate symbol, the idea being that $P(x, y, z)$ stands for $x \cdot y = z$. Now we can render the axioms of identity and inverse as $\forall x.\ P(1, x, x)$ and $\forall x.\ P(i(x), x, 1)$. By introducing auxiliary variables for subexpressions, we can express the associative law, e.g. as

$$\forall u, v, w, x, y, z.\ P(x, y, u) \wedge P(y, z, w) \Rightarrow (P(x, w, v) \Leftrightarrow P(u, z, v)).$$

Admittedly, there are several important properties of the group operation that aren't captured by the three axioms for P so far, e.g. $\forall x\ y. \exists! z. P(x, y, z)$. Nevertheless, it turns out that some properties of groups can still be derived just from these properties. The problem of proving that a group where $x^2 = 1$ is abelian ($x \cdot y = y \cdot x$) works particularly nicely, because we don't need to postulate an inverse operation, each element being its own inverse:

```
# meson
   <<(forall x. P(1,x,x)) /\
     (forall x. P(x,x,1)) /\
     (forall u v w x y z. P(x,y,u) /\ P(y,z,w)
                          ==> (P(x,w,v) <=> P(u,z,v)))
     ==> forall a b c. P(a,b,c) ==> P(b,a,c)>>;;
...
- : int list = [13]
```

Effective though this method can be, and interesting as it is to see how weaker axioms suffice for many purposes, it has a rather ad hoc flavour, and obliges us to code up the natural notions in a rather peculiar fashion. Indeed, it was mainly popular before more effective equality reasoning methods had been developed. Nevertheless, the idea of breaking down terms like $(x \cdot y) \cdot z$ by the introduction of auxiliary variables will reappear in a slightly different form below.

Equivalence elimination

Our main interest is in the equality relation, but we'll consider equality-like properties of an arbitrary binary relation R in what follows. Besides giving greater generality, it might actually be clearer since the notation won't tempt the reader to make special assumptions about equality. Note that in contrast to most of this chapter, we're concerned with arbitrary interpretations here, not necessarily normal ones.

Consider the axiom 'Equiv' asserting that a binary relation R is an equivalence relation, i.e. is reflexive, symmetric and transitive.

$$\begin{array}{l}(\forall x.\ R(x,x)) \wedge \\ (\forall x\ y.\ R(x,y) \Rightarrow R(y,x)) \wedge \\ (\forall x\ y\ z.\ R(x,y) \wedge R(y,z) \Rightarrow R(x,z)).\end{array}$$

This is equivalent to simply $\forall x\ y.\ R(x,y) \Leftrightarrow (\forall z.\ R(x,z) \Leftrightarrow R(y,z))$; the reader can verify this, or we can leave it to the machine:

```
# meson
   <<(forall x. R(x,x)) /\
     (forall x y. R(x,y) ==>  R(y,x)) /\
     (forall x y z. R(x,y) /\ R(y,z) ==> R(x,z))
     <=> (forall x y. R(x,y) <=> (forall z. R(x,z) <=> R(y,z)))>>;;
...
- : int list = [4; 3; 9; 3; 2; 7]
```

Similarly, an assertion of reflexivity and transitivity (without symmetry) is equivalent to $\forall x\ y.\ R(x,y) \Leftrightarrow (\forall z.\ R(y,z) \Rightarrow R(x,z))$, while symmetry of R alone is equivalent to $\forall x\ y. R(x,y) \Leftrightarrow R(x,y) \wedge R(y,x)$. These equivalences are all of the form

$$\forall x\ y.\ R(x,y) \Leftrightarrow R^*[x,y],$$

so we can think of them as rules for replacing each instance of $R(s,t)$ in a formula by $R^*[s,t]$. After making such replacements, we will prove shortly that the corresponding axioms about R are no longer needed. Consider the case of full equivalence; the reflexivity–transitivity and symmetry cases work

similarly. Given an atomic formula $R(s,t)$, write $R^*[s,t]$ for $\forall w.\ R(s,w) \Leftrightarrow R(t,w)$ where $w \notin \mathrm{FV}(s) \cup \mathrm{FV}(t)$.

Theorem 4.29 *$P \wedge$ Equiv is satisfiable iff the formula P^* that results from replacing each subformula $R(s,t)$ in P with $R^*[s,t]$ is satisfiable.*

Proof We noted above that Equiv $\Leftrightarrow (\forall x\ y.\ R(x,y) \Leftrightarrow R^*[x,y])$ and so for any terms s and t we have Equiv $\Rightarrow (R(s,t) \Leftrightarrow R^*[s,t])$. Hence Equiv $\wedge\, P \Leftrightarrow$ Equiv $\wedge\, P^*$. This means that if Equiv $\wedge\, P$ is satisfiable, so is Equiv $\wedge\, P^*$ and a fortiori P^*. Note that this works equally well if we choose only to replace *some* formulas $R(s,t)$ in P with $R^*[s,t]$, not necessarily all of them.

Now suppose that P^* is satisfiable, say in an interpretation M with domain D where R is interpreted by R_M. Define a new interpretation N that is the same except that $R_N(a,b)$ is defined to hold precisely when $R_M(a,c)$ and $R_M(b,c)$ are equivalent for all $c \in D$. By design, `holds` $N\ v\ (R(s,t)) =$ `holds` $M\ v\ (R^*[s,t])$, so since P^* holds in M, P holds in N. By construction R_N is an equivalence relation, so Equiv also holds in N. □

This approach is generalized by Ohlbach, Gabbay and Plaisted (1994) to a large class of 'killer transformations', so called because they 'kill' certain axioms. The proofs here of the key equisatisfiability properties were suggested by Rob Arthan.

Brand's S- and T-modifications

An earlier equality elimination method (Brand 1975) similarly eliminates symmetry and transitivity, but keeps the reflexivity axiom $\forall x.\ R(x,x)$. The advantage of doing this is that one may then perform the expansive transformation only on *positive* occurrences of $R(s,t)$, while negative occurrences $\neg R(u,v)$ can be left alone. We can adapt the proof of Theorem 4.29 as follows. Assume the formula $P[\ldots, R(s,t), \ldots, \neg R(u,v), \ldots]$ whose satisfiability is at issue is in NNF, so we can distinguish positive and negative occurrences simply by whether they are directly covered by a negation operation. All are treated in the way indicated for the paradigmatic examples $R(s,t)$ and $\neg R(u,v)$. Write as before

$$P^* = P[\ldots, R^*[s,t], \ldots, \neg R^*[u,v], \ldots]$$

but also

$$P' = P[\ldots, R^*[s,t], \ldots, \neg R(u,v), \ldots].$$

The first part of the proof works equally well to show that if Equiv $\wedge$ P is satisfiable, so is Equiv $\wedge$ P' and therefore $(\forall x.\ R(x,x)) \wedge P'$. Conversely, $(\forall x. R(x,x)) \Rightarrow R^*[u,v] \Rightarrow R(u,v)$, so $(\forall x. R(x,x)) \Rightarrow \neg R(u,v) \Rightarrow \neg R^*[u,v]$ and therefore $(\forall x. R(x,x)) \wedge P' \Rightarrow (\forall x. R(x,x)) \wedge P^*$. Thus if $(\forall x. R(x,x)) \wedge P'$ is satisfiable, so is P^* and, by the same proof as before, so is P.

Restricted to the special case of a formula in clausal form with R being the equality relation, these ways of eliminating symmetry and transitivity give exactly Brand's S-modification and T-modification respectively. Doing these successively works out the same as doing equivalence-elimination once and for all, but we'll keep them separate both to emphasize the correspondence with Brand's work and to modularize the implementation. In the clausal context we can also recognize positivity or negativity trivially. If we keep the same predicate symbol, namely $=$, then we can just leave negative literals untouched in each case, and only modify positive equations. The S-transformation on a clause with n positive equations (written at the beginning for simplicity):

$$s_1 = t_1 \vee \cdots \vee s_n = t_n \vee C$$

leads to

$$(s_1 = t_1 \wedge t_1 = s_1) \vee \cdots \vee (s_n = t_n \wedge t_n = s_n) \vee C.$$

This is no longer in clausal form, but we can redistribute and arrive at 2^n resulting clauses:

$$\begin{aligned}
&s_1 = t_1 \vee \cdots \vee s_{n-1} = t_{n-1} \vee s_n = t_n \vee C,\\
&s_1 = t_1 \vee \cdots \vee s_{n-1} = t_{n-1} \vee t_n = s_n \vee C,\\
&s_1 = t_1 \vee \cdots \vee t_{n-1} = s_{n-1} \vee s_n = t_n \vee C,\\
&s_1 = t_1 \vee \cdots \vee t_{n-1} = s_{n-1} \vee t_n = s_n \vee C,\\
&\cdots\\
&t_1 = s_1 \vee \cdots \vee t_{n-1} = s_{n-1} \vee t_n = s_n \vee C,
\end{aligned}$$

which essentially cover all possible combinations of forward and backward equations in the original clause. Admittedly, if n is large, this exponential blowup in the number of clauses is not very appealing, but it can be made manageable using a few extra tricks (see Exercise 4.4). Here is the implementation on a clause represented as a list of literals:

```
let rec modify_S cl =
  try let (s,t) = tryfind dest_eq cl in
      let eq1 = mk_eq s t and eq2 = mk_eq t s in
      let sub = modify_S (subtract cl [eq1]) in
      map (insert eq1) sub @ map (insert eq2) sub
  with Failure _ -> [cl];;
```

For the T-modification, we need to replace each equation $s_i = t_i$ in a clause:

$$s_1 = t_1 \vee \cdots \vee s_n = t_n \vee C$$

as follows:

$$(\forall w.\, t_1 = w \Rightarrow s_1 = w) \vee \cdots \vee (\forall w.\, t_n = w \Rightarrow s_n = w) \vee C.$$

We can pull out the universal quantifiers to retain clausal form, but we then need to use distinct variable names w_i instead of a single w in each equation. We also transform $t_1 = w \Rightarrow s_1 = w$ into $\neg(t_i = w) \vee s_i = w$ to return to clausal form, resulting in:

$$\neg(t_1 = w_1) \vee s_1 = w_1 \vee \cdots \vee \neg(t_n = w_n) \vee s_n = w_n \vee C.$$

We can implement this directly, just running through the literals successively, recursively transforming the tail and picking a new variable w that is neither in the transformed tail nor the unmodified literal being considered:

```
let rec modify_T cl =
  match cl with
    (Atom(R("=",[s;t])) as eq)::ps ->
        let ps' = modify_T ps in
        let w = Var(variant "w" (itlist (union ** fv) ps' (fv eq))) in
        Not(mk_eq t w)::(mk_eq s w)::ps'
  | p::ps -> p::(modify_T ps)
  | [] -> [];;
```

Brand's E-modification

We have shown how the equivalence axioms can be eliminated by incorporating new structure into the other formulas. We now proceed to do the same with the congruence axioms

$$\forall x_1 \cdots x_n y_1 \cdots y_n.\, x_1 = y_1 \wedge \cdots \wedge x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

and

$$\forall x_1 \cdots x_n y_1 \cdots y_n.\, x_1 = y_1 \wedge \cdots \wedge x_n = y_n \Rightarrow P(x_1, \ldots, x_n) \Rightarrow P(y_1, \ldots, y_n)$$

for the function symbols f and predicates P appearing in the initial formulas. We will actually perform this transformation first, and so we can assume the equivalence axioms. The basic idea is to repeatedly pull out non-variable immediate subterms t of function and predicate symbols (other than equality) using the following, which are clearly equivalences in the presence of the congruence and reflexivity axioms:

$$\begin{array}{rcl} f(\ldots,t,\ldots) = s & \Leftrightarrow & \forall w.\, t = w \Rightarrow f(\ldots,w,\ldots) = s, \\ s = f(\ldots,t,\ldots) & \Leftrightarrow & \forall w.\, t = w \Rightarrow s = f(\ldots,w,\ldots), \\ P(\ldots,t,\ldots) & \Leftrightarrow & \forall w.\, t = w \Rightarrow P(\ldots,w,\ldots). \end{array}$$

We can repeat this transformation until function symbols (including constants) only appear as arguments to the equality predicate, not other predicates nor other functions. A formula with this property is said to be *flat* and we will describe the transformation as *flattening*. For example, we might transform the associative law as follows, assuming all free variables to be implicitly universally quantified:

$$\begin{array}{l} (x \cdot y) \cdot z = x \cdot (y \cdot z), \\ x \cdot y = w_1 \Rightarrow w_1 \cdot z = x \cdot (y \cdot z), \\ x \cdot y = w_1 \wedge y \cdot z = w_2 \Rightarrow w_1 \cdot z = x \cdot w_2. \end{array}$$

It turns out that for flat quantifier-free formulas, the congruence axioms are not necessary, in the following precise sense.

Theorem 4.30 *Suppose a quantifier-free formula P is flat, E asserts the equivalence properties of equality and C is the collection of congruences for the functions and predicates appearing in P. Then $P \wedge E \wedge C$ is satisfiable iff $P \wedge E$ is.*

Proof One way is immediate. So suppose $P \wedge E$ is satisfiable; we will show that $P \wedge E \wedge C$ is too. If M is a model of $P \wedge E$ with domain D, then since it is a fortiori a model of E, the interpretation $=_M$ of equality is an equivalence relation. For any $a \in D$, let $\overline{a}$ be some fixed canonical representative of the equivalence class $[a]_{=_M}$. Thus, for any $a, b \in D$ we have $=_M (a, b)$ iff $\overline{a} = \overline{b}$. We now define a new model M' with the same domain D interpreting the function symbols as follows:

$$f_{M'}(a_1,\ldots,a_n) = f_M(\overline{a_1},\ldots,\overline{a_n}),$$

equality in the same way, $=_M$, and the other predicate symbols like this:

$$P_{M'}(a_1, \ldots, a_n) = P_M(\overline{a_1}, \ldots, \overline{a_n}).$$

We claim that M' is a model of $P \wedge E \wedge C$. It is a model of E since we have not changed the interpretation of the equality symbol nor the domain, and no function symbols or other predicates appear in E. To see that it is also a model of C, note that the function congruence axiom

$$x_1 = y_1 \wedge \cdots \wedge x_n = y_n \Rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

holds in M' under a valuation mapping each $x_i \mapsto a_i$ and $y_i \mapsto b_i$ precisely if whenever $a_i =_M b_i$ for $1 \leq i \leq n$, then $f_{M'}(a_1, \ldots, a_n) = f_{M'}(b_1, \ldots, b_n)$. But $a_i = b_i$ implies, as noted above, that $\overline{a_i} = \overline{b_i}$, and since by definition $f_{M'}(a_1, \ldots, a_n) = f_M(\overline{a_1}, \ldots, \overline{a_n})$ and similarly for b_i, the result follows. The predicate congruences hold for similar reasons.

All that remains is to show that M' is a model of P as well, and this is where the flatness of P is critical. Let v be any valuation, and define $\overline{v}(x) = \overline{v(x)}$. We claim that for any flat atomic formula p we have $\texttt{holds}\ M'\ v\ p = \texttt{holds}\ M\ \overline{v}\ p$. Note first that for each term consisting of a function applied to (not necessarily distinct) variables we have

$$\begin{aligned}
& \texttt{termval}\ M'\ v\ (f(x_1, \ldots, x_n)) \\
= \ & f_{M'}(\texttt{termval}\ M'\ v\ x_1, \ldots, \texttt{termval}\ M'\ v\ x_n) \\
= \ & f_{M'}(v(x_1), \ldots, v(x_n)) \\
= \ & f_M(\overline{v(x_1)}, \ldots, \overline{v(x_n)}) \\
= \ & f_M(\overline{v}(x_1), \ldots, \overline{v}(x_n)) \\
= \ & f_M(\texttt{termval}\ M\ \overline{v}\ x_1, \ldots, \texttt{termval}\ M\ \overline{v}\ x_n) \\
= \ & \texttt{termval}\ M\ \overline{v}\ (f(x_1, \ldots, x_n)).
\end{aligned}$$

The same result does not hold for variables alone, but at least the two values $\texttt{termval}\ M'\ v\ x = v(x)$ and $\texttt{termval}\ M\ \overline{v}\ x = \overline{v}(x) = \overline{v(x)}$ are equivalent under $=_M$ by definition. Thus if t is a 'flat term', either a variable or function applied to variables, we have

$$=_M (\texttt{termval}\ M'\ v\ t, \texttt{termval}\ M\ \overline{v}\ t).$$

Consequently, since $=_M$ is an equivalence relation we can see that for an equation between two such terms:

$$\begin{aligned} & \texttt{holds}\ M'\ v\ (s = t) \\ = \quad & =_M\ (\texttt{termval}\ M'\ v\ s, \texttt{termval}\ M'\ v\ t) \\ = \quad & =_M\ (\texttt{termval}\ M\ \overline{v}\ s, \texttt{termval}\ M\ \overline{v}\ t) \\ = \quad & \texttt{holds}\ M\ \overline{v}\ (s = t). \end{aligned}$$

For other predicate symbols applied to variables, we similarly have:

$$\begin{aligned} & \texttt{holds}\ M'\ v\ (P(x_1, \ldots, x_n)) \\ = \quad & P'_M(\texttt{termval}\ M'\ v\ x_1, \ldots, \texttt{termval}\ M'\ v\ x_n)) \\ = \quad & P'_M(v(x_1), \ldots, v(x_n)) \\ = \quad & P_M(\overline{v(x_1)}, \ldots, \overline{v}(x_n)) \\ = \quad & P_M(\overline{v}(x_1), \ldots, \overline{v}(x_n)) \\ = \quad & P_M(\texttt{termval}\ M\ \overline{v}\ x_1, \ldots, \texttt{termval}\ M\ \overline{v}\ x_n) \\ = \quad & \texttt{holds}\ M\ \overline{v}\ (P(x_1, \ldots, x_n)). \end{aligned}$$

It now follows by induction on the structure of P that we can extend the basic result to the whole formula (which is quantifier-free by hypothesis):

$$\texttt{holds}\ M'\ v\ P = \texttt{holds}\ M\ \overline{v}\ P$$

However, since M is a model of P, the RHS is simply 'true', and therefore so is the left. But v was arbitrary, and therefore the theorem is proved. □

Brand's 'E-modification' applies the flattening transformation to clauses, adding new negative literals $\neg(t = w_i)$ for the extra variable definitions included. It follows that if we perform E-modification and then S- and T-modifications, the resulting set of clauses plus the reflexive law $x = x$ has a model iff the original formula has a normal model. We have thus succeeded in transforming the input clauses to eliminate the need for any equality axioms besides reflexivity.

Implementation

First we define functions to identify non-variables:

```
let is_nonvar = function (Var x) -> false | _ -> true;;
```

and hence find a nested non-variable subterm where possible:

```
let find_nestnonvar tm =
  match tm with
    Var x -> failwith "findnvsubt"
  | Fn(f,args) -> find is_nonvar args;;
```

Now we can identify a non-variable subterm that we want to pull out in flattening; in the case of equality this is a *nested* non-variable subterm, while for the other predicate symbols it is any non-variable subterm:

```
let rec find_nvsubterm fm =
  match fm with
    Atom(R("=",[s;t])) -> tryfind find_nestnonvar [s;t]
  | Atom(R(p,args)) -> find is_nonvar args
  | Not p -> find_nvsubterm p;;
```

Having found such a non-variable subterm, we want to replace it with a new variable. We don't have a general function to replace subterms (`tsubst` and `subst` only replace *variables*), so we define one, first for terms:

```
let rec replacet rfn tm =
  try apply rfn tm with Failure _ ->
  match tm with
    Fn(f,args) -> Fn(f,map (replacet rfn) args)
  | _ -> tm;;
```

and then for other formulas (here we only care about literals, and can treat quantified formulas without regard to variable capture):

```
let replace rfn = onformula (replacet rfn);;
```

To E-modify a clause, we try to find a nested non-variable subterm; if we fail we are already done, and otherwise we replace that term with a fresh variable w, add the new disjunct $\neg(t = w)$ and call recursively:

```
let rec emodify fvs cls =
  try let t = tryfind find_nvsubterm cls in
      let w = variant "w" fvs in
      let cls' = map (replace (t |=> Var w)) cls in
      emodify (w::fvs) (Not(mk_eq t (Var w))::cls')
  with Failure _ -> cls;;
```

The `fvs` parameter tracks the free variables in the clause so far, so we just need to set its initial value:

```
let modify_E cls = emodify (itlist (union ** fv) cls []) cls;;
```

The overall Brand transformation now applies E-modification, then S-modification and T-modification, then finally includes the reflexive clause $x = x$:

```
let brand cls =
  let cls1 = map modify_E cls in
  let cls2 = itlist (union ** modify_S) cls1 [] in
  [mk_eq (Var "x") (Var "x")]::(map modify_T cls2);;
```

We insert Brand's transformation into MESON's clausal framework to give `bmeson`:

```
let bpuremeson fm =
  let cls = brand(simpcnf(specialize(pnf fm))) in
  let rules = itlist ((@) ** contrapositives) cls [] in
  deepen (fun n ->
     mexpand rules [] False (fun x -> x) (undefined,n,0); n) 0;;

let bmeson fm =
  let fm1 = askolemize(Not(generalize fm)) in
  map (bpuremeson ** list_conj) (simpdnf fm1);;
```

For easy comparison, we'll define a similar version of MESON that just uses the equality axioms.

```
let emeson fm = meson (equalitize fm);;
```

The relative performance of these two methods depends on the application. For example, on the `wishnu` problem from the end of Section 4.1, Brand's transformation is substantially slower than just adding the equality axioms. But on our group theory examples, Brand's transformation is much better, e.g. only a few minutes here while `emeson` takes far longer:

```
# bmeson
   <<(forall x y z. x * (y * z) = (x * y) * z) /\
     (forall x. e * x = x) /\
     (forall x. i(x) * x = e)
     ==> forall x. x * i(x) = e>>;;
- : int list = [19]
```

Since Brand's original work, several variant methods have been proposed that are often more efficient. Moser and Steinbach (1997) suggest a version that avoids equations with variables on their left-hand sides, which tends to reduce the number of possible unifications. However, this comes at the cost of needing to split negative equations as well as positive ones in the analogue of the T-modification. A further refinement based on imposing term ordering constraints was proved complete by Bachmair, Ganzinger and

Voronkov (1997) and shown to be substantially more efficient on a number of examples.

4.9 Paramodulation

So far we have handled equality by using standard first-order proof methods on modified formulas, resulting either from adding equality axioms or using the more sophisticated modification methods in the previous section. Preprocessing has several advantages: we can re-use proof procedures intended for pure first-order logic without internal modification, and can also transfer results like compactness to the equality case without new theoretical difficulties. However, it is also possible to augment one of the standard first-order theorem proving techniques with additional rules for equality, rather than modifying the input formulas themselves. It seems more straightforward to add new inference rules in the context of bottom-up procedures like resolution, though some authors have also introduced special equality-handling methods for top-down methods such as tableaux (Fitting 1990), model elimination (Moser, Lynch and Steinbach 1995), model evolution (Baumgartner and Tinelli 2005) and others.

The first equality-based inference rule to be introduced was *demodulation* (Wos, Robinson, Carson and Shalla 1967), which uses unit equality clauses like $x + 0 = x$ as rewrite rules to simplify other clauses. The name arises because it is typically used to remove 'modulations' of essentially the same fact, e.g. $P(x)$, $P(0 + x)$, $P(x - 0)$ etc. Although useful in practice, it is not complete. However, the more general rule of *paramodulation* introduced a little later (G. Robinson and Wos 1969) gives, when used together with the standard resolution rule, a theoretically complete method of handling equality. Even in its unrestricted initial form it was often found to be far more effective than adding equality axioms, and it has subsequently been extensively refined, in particular by introducing ordering notions from term rewriting. Paramodulation is the following inference rule, where $s \doteq t$ may be either $s = t$ or $t = s$:

$$\frac{C \vee s \doteq t \quad D \vee P[s']}{\texttt{subst}\ \sigma\ (C \vee D \vee P[t])}\ \text{Paramodulation},$$

where σ is a MGU of s and the indicated term instance s'. Paramodulation generalizes rewriting in several respects that make it look more like the resolution rule itself: we can use equations that occur disjoined with additional literals C to rewrite with, the rewrite may be applied in either direction, and the identification of the terms s and s' is done by full unification, not

just matching. It's relatively easy to see that the rule is sound, i.e that the conclusion holds in any normal model in which the hypotheses do. The issue of its refutation completeness as a method of equality handling is subtler.

Refutation completeness of paramodulation

It is *not* the case that if a set of clauses has no normal model then it can be refuted by resolution plus paramodulation, as the example of $\{\neg(x = x)\}$ shows. This suggests that, as with Brand's method, we may not need all the equality axioms but we *do* at least need to add reflexivity to the input clauses. In fact, we will demonstrate refutation completeness on the stronger assumption that we also add all the *functional reflexive axioms* of the form:

$$f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n),$$

one for each function symbol f appearing in the input clauses. (This looks strange, but the reason will become clearer below.) Our proof of refutation completeness rests on the fact that a hyperresolution proof assuming equality axioms can be simulated by resolution and paramodulation with the functional reflexive axioms. In order to simplify the proof, we will adopt instead of the usual congruence rules the 1-instance variants:

$$\neg(x = x') \vee f(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, x', x_{i+1}, \ldots, x_n)$$

for each n-ary function f in the clauses S and for each $1 \leq i \leq n$, and similarly:

$$\neg(x = x') \vee \neg P(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n) \vee P(x_1, \ldots, x_{i-1}, x', x_{i+1}, \ldots, x_n)$$

for each n-ary predicate P in the clauses S and for each $1 \leq i \leq n$, together with the usual combined symmetry–transitivity rule:

$$\neg(x = y) \vee \neg(x = z) \vee (y = z)$$

and simple reflexivity

$$x = x.$$

We refer to these collectively as $\texttt{eqaxioms}'(S)$. They are logically equivalent to $\texttt{eqaxioms}(S)$, since we can derive the multiple-instance congruence rules by repeated use of the one-instance rule put together by transitivity, while the converse follows by reflexivity. We let R be simple reflexivity together with the functional reflexive axioms, one for each function symbol in S:

$$f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n).$$

Theorem 4.31 *If S has no normal model, then $S \cup R$ has a refutation by resolution and paramodulation.*

Proof Since S has no normal model, $S \cup \texttt{eqaxioms}'(S)$ is unsatisfiable (by the above remarks and Theorem 4.1). It therefore has a refutation by positive hyperresolution (see Section 3.13). We will show that all conclusions obtainable by positive hyperresolution from $S \cup \texttt{eqaxioms}'(S)$ can also be obtained by resolution and paramodulation from $S \cup R$.

We will establish this by induction on the steps of a hyperresolution proof. We need only consider hyperresolution steps where at least one input clause is taken from the set $R' = \texttt{eqaxioms}'(S) - R$, since otherwise the conclusion holds at once. And since there are no all-positive clauses in R', we must by the definition of positive hyperresolution have *exactly* one input clause from R'. If this input clause is a function-congruence axiom, then the resolution must be of the following form. (In such cases, we can assume that only the left-hand hypothesis is instantiated, in this case with a unifier $x \mapsto s$ and $x' \mapsto t$, because x and x' are just variables.)

$$\frac{\neg(x = x') \vee f(\ldots, x, \ldots) = f(\ldots, x', \ldots) \quad C \vee s = t}{C \vee f(\ldots, s, \ldots) = f(\ldots, t, \ldots)}$$

This can be simulated by a paramodulation inference using the functional reflexive axiom:

$$\frac{f(\ldots, x, \ldots) = f(\ldots, x, \ldots) C \vee s = t}{C \vee f(\ldots, s, \ldots) = f(\ldots, t, \ldots)}.$$

Now, if the input is a predicate-congruence axiom, then any hyperresolution consisting of two successive positive resolution steps (in the order shown here or vice versa):

$$\frac{\dfrac{\neg(x = x') \vee \neg P(\ldots, x, \ldots) \vee P(\ldots, x', \ldots) \quad C \vee s = t}{C \vee \neg P(\ldots, s, \ldots) \vee P(\ldots, t, \ldots)} \quad D \vee P(\ldots, s', \ldots)}{\texttt{subst}\ \sigma\ (C \vee D \vee P(\ldots, t, \ldots))},$$

where σ is an MGU of s and s', can be simulated directly by a single paramodulation:

$$\frac{C \vee s = t \quad D \vee P(\ldots, s', \ldots))}{\texttt{subst}\ \sigma(C \vee D \vee P(\ldots, t, \ldots))},$$

Finally, a hyperresolution with the symmetry–transitivity axiom, again either in the order shown here or vice versa:

$$\dfrac{\dfrac{\neg(x = y) \vee \neg(x = z) \vee (y = z) \quad C \vee s = t}{C \vee \neg(s = z) \vee (t = z)} \qquad D \vee s' = t'}{\texttt{subst}\ \sigma\ (C \vee D \vee t = t')},$$

with σ a MGU of s and s', can be simulated by a single paramodulation as follows:

$$\dfrac{C \vee s = t \quad D \vee s' = t'}{\texttt{subst}\ \sigma\ (C \vee D \vee t = t')}.$$

□

This proof exploits the fact that many conclusions can be derived by paramodulation with the functional reflexive axioms. But for exactly the same reason, it's not clear that this combination in practice is actually any better controlled than direct hyperresolution with the equality axioms (Kowalski 1970a). Moreover, the apparent need for the functional reflexive axioms, all of which are just instances of $x = x$, shows that the kind of 'lifting' arguments underlying resolution do not generalize, and suggests that subsumption for paramodulation may be subtle.

For a long time it was an open question whether simple reflexivity $x = x$ is enough to ensure refutation completeness of resolution with paramodulation.† Eventually Brand (1975) presented an analogous simulation argument based on his equality transformation (Section 4.8), showing not only that simple reflexivity suffices but also that paramodulation can be restricted in other ways without losing refutation completeness. In particular, there is almost no need to paramodulate into variables, i.e. unify the left of the paramodulating equation with a variable subterm of the literal being paramodulated. However, when using many of the most effective refinements of resolution like set-of-support, the functional reflexive axioms are necessary once again for refutation completeness. Consider, for example, the following set of clauses, including simple reflexivity:

$$\{\neg(x < x), f(a) < f(b), a = b, x = x\}.$$

The entire set is unsatisfiable, but the set with $\neg(x < x)$ removed is satisfiable. However, if we attempt to find a proof by resolution and paramodulation with set of support $\neg(x < x)$, no proof can be found. On the other hand,

† A footnote in G.G. Robinson and Wos (1969) remarks: 'In the two years that paramodulation has been under study, no counterexample has been found to the *R*-refutation completeness of paramodulation and resolution for simply-reflexive systems'.

if we add the functional reflexive axiom $f(x) = f(x)$, we can paramodulate with $\neg(x < x)$ to yield $\neg(f(x) < f(x))$ and quickly arrive at a refutation. Despite such examples, it is common to leave the functional reflexive axioms out when attempting theorem proving in the hope that their theoretical necessity will not arise in the particular case under consideration. In our implementation, we will just use simple reflexivity and also disallow paramodulation into variables, in line with Brand's result.

Implementation

The key operation in paramodulation is not unlike that of finding a critical pair in Knuth–Bendix completion (Section 4.7), except that we need to consider overlaps inside an arbitrary literal, not just another term. It's similar enough that we can re-use some of the code such as the `overlaps` function. (To allow paramodulation into variables the last line '`Var x -> []`' could be replaced by '`Var x -> [rfn (fullunify [l,tm]) r]`'.) We then define an analogous function to find overlaps within literals. The code is very similar, the main change being that we don't attempt overlaps at the top level (which is a formula, not a term) and include a separate clause for negations.

```
let rec overlapl (l,r) fm rfn =
  match fm with
    Atom(R(f,args)) -> listcases (overlaps (l,r))
                                 (fun i a -> rfn i (Atom(R(f,a)))) args []
  | Not(p) -> overlapl (l,r) p (fun i p -> rfn i (Not(p)))
  | _ -> failwith "overlapl: not a literal";;
```

We lift this to an operation on a whole clause, i.e. a list of literals:

```
let overlapc (l,r) cl rfn acc = listcases (overlapl (l,r)) rfn cl acc;;
```

Now to apply paramodulation to a clause `ocl` using all the positive equations in a paramodulating clause `pcl`, we treat each positive equation `eq` in turn, considering it as both $l = r$ and $r = l$. In each case we apply `overlapc`, with the reconstruction function set up to disjoin the other clauses and apply the final instantiation to each.

```
let paramodulate pcl ocl =
  itlist (fun eq -> let pcl' = subtract pcl [eq] in
                    let (l,r) = dest_eq eq
                    and rfn i ocl' = image (subst i) (pcl' @ ocl') in
                    overlapc (l,r) ocl rfn ** overlapc (r,l) ocl rfn)
         (filter is_eq pcl) [];;
```

Now to generate all paramodulants between clauses, we just rename the clauses to avoid variable clashes in unification, as usual, and then perform paramodulation of each clause within the other.

```
let para_clauses cls1 cls2 =
  let cls1' = rename "x" cls1 and cls2' = rename "y" cls2 in
  paramodulate cls1' cls2' @ paramodulate cls2' cls1';;
```

Now we modify the main resolution loop from Section 3.11 to incorporate both resolution and paramodulation:

```
let rec paraloop (used,unused) =
  match unused with
    [] -> failwith "No proof found"
  | cls::ros ->
        print_string(string_of_int(length used) ^ " used; "^
                     string_of_int(length unused) ^ " unused.");
        print_newline();
        let used' = insert cls used in
        let news =
          itlist (@) (mapfilter (resolve_clauses cls) used')
            (itlist (@) (mapfilter (para_clauses cls) used') []) in
        if mem [] news then true else
        paraloop(used',itlist (incorporate cls) news ros);;
```

and then set up the top-level function as before, remembering to add simple reflexivity to the clause set:

```
let pure_paramodulation fm =
  paraloop([],[mk_eq (Var "x") (Var "x")]::
               simpcnf(specialize(pnf fm)));;

let paramodulation fm =
  let fm1 = askolemize(Not(generalize fm)) in
  map (pure_paramodulation ** list_conj) (simpdnf fm1);;
```

This implementation is at least enough to deal with some simple equality problems we've already encountered, as well as some others like the following (Dijkstra 1996):

```
# paramodulation
   <<(forall x. f(f(x)) = f(x)) /\ (forall x. exists y. f(y) = x)
     ==> forall x. f(x) = x>>;;
...
- : bool list = [true]
```

However, our rather simple-minded implementation cannot really demonstrate the full power of paramodulation. It works best in conjunction with strong restrictions on applicability, e.g. applying equations in a preferred

direction based on orderings in the style of term rewriting. Moreover, resolution itself, and paramodulation even more so, work best with more intelligent strategies for choosing the next application rather than the naive round-robin approach that we have implemented. In fact, by encoding atomic formulas $P(t_1, \ldots, t_n)$ as equations $f_P(t_1, \ldots, t_n) = \mathrm{T}$ (where 'T' is thought of as 'true'; see Exercise 4.3), one can essentially perform all logical inference via equational techniques like paramodulation, obviating the need for resolution or similar principles. This idea underlies the *superposition* method (Bachmair and Ganzinger 1994), implemented efficiently in the E theorem prover (Schulz 1999).

Further reading

The branch of model theory focusing on equational logic is also known as *universal algebra*, and there are several texts on the subject such as Cohn (1965) and Burris and Sankappanavar (1981). Almost all books on model theory cited in the last chapter also contain something about the theoretical material described here. More information, historical and otherwise, on the concept of categoricity is given by Corcoran (1980). Two more difficult theorems about κ-categoricity are Morley's theorem, which asserts that a theory categorical in one uncountable cardinal is categorical in them all, and the Ryll–Nardzewski theorem, which gives an attractive algebraic characterization of $\aleph_0$-categorical theories. Both these theorems can be found in Hodges (1993b).

For pure equational reasoning based on rewriting techniques, see the book by Baader and Nipkow (1998) and the survey articles by Huet and Oppen (1980), Klop (1992) and Plaisted (1993). Dershowitz's result that a simplification order is terminating is usually deduced from (a simple case of) *Kruskal's theorem* (Kruskal 1960; Nash-Williams 1963); an accessible account can be found in Baader and Nipkow (1998). In implementing the LPO we paid no attention to efficiency, but this question is carefully analyzed by Löchner (2006).

Methods for deciding validity of universal formulas in logic with equality have significant applications in verification (Burch and Dill 1994). This has led to the exploration of various alternative algorithms to congruence closure. For further refinements of the approach based on Ackermann reduction, see Goel, Sajid, Zhou, Aziz and Singhal (1998), Velev and Bryant (1999) and Lahiri, Bryant, Goel and Talupur (2004).

Paramodulation is discussed in some of the automated theorem proving texts already mentioned, including Chang and Lee (1973) and Loveland

(1978). Again, books such as Wos, Overbeek, Lusk and Boyle (1992) by the Argonne group cover the use of paramodulation to solve non-trivial problems. Bachmair and Ganzinger (1994) is a survey of paramodulation and related ideas, and Degtyarev and Voronkov (2001) of equality reasoning in top-down free-variable calculi like tableaux.

The TPTP problem library (Sutcliffe and Suttner 1998) includes many equational problems, and provides tools to add equality axioms for provers that do not handle equality directly. Some of the most impressive applications of automated reasoning to hard problems are in the general area of equational logic. The most famous example is the Robbins conjecture, which resisted proof attempts by many notable mathematicians including Tarski, yet was solved automatically by McCune (1997) using the EQP prover. This is just one particularly well-known case where automated reasoning programs have answered open questions. Some more can be found in the monographs by McCune and Padmanabhan (1996) and Wos and Pieper (2003), and on the Web.†

Exercises

4.1 Recall that a set of formulas is said to be κ-categorical if (it has a model and) all its models of cardinality κ are isomorphic. Prove a version of the *Łoś–Vaught test*: if a countable set of formulas is κ-categorical for some infinite κ then all models are elementarily equivalent. (You may find it useful to use the upward Löwenheim-Skolem theorem.)

4.2 Show that a Birkhoff proof can be rearranged so that all instantiation and symmetry is applied immediately above the leaves, then congruence rules where necessary and at the top level a right-associated transitivity chain such that no two adjacent equations in a transitivity chain are derived by a congruence. Hence deduce in another way that congruence closure of the subterms in the input problem is a complete approach to the equational theory of a set of ground equations.

4.3 We can reduce validity of arbitrary formulas in first-order logic with equality to a language with equality as the only predicate by the device of turning each $P(t_1, \ldots, t_n)$ to a term $f_P(t_1, \ldots, t_n) = \mathrm{T}$ for some new n-ary function symbol f_P and a new constant T for 'true'. For example, this allows us to decide the full universal theory of first-order logic with equality using standard congruence closure. Under

† See `http://www-unix.mcs.anl.gov/AR/new_results/`

what circumstances does this transformation preserve validity? (Take care over 1-element interpretations!)

4.4 Rigorously justify the Ackermann reduction from universal formulas in logic with equality to the corresponding problem without functions, and so all the way to propositional logic. Implement this idea, using some method such as DPLL to solve the resulting formulas, and test it against congruence closure on examples.

4.5 We say that two abstract reduction relations $\rightarrow_\alpha$ and $\rightarrow_\beta$ on a set X *commute* if whenever $a \rightarrow_\alpha^* b$ and $a \rightarrow_\beta^* b'$ there is a c with $b \rightarrow_\beta^* c$ and $b' \rightarrow_\alpha^* c$. Thus, in particular, a reduction relation is confluent iff it commutes with itself. Prove that if a set of reduction relations $\{\rightarrow_\alpha | \ \alpha \in A\}$ on a set X has the property that any two (not necessarily distinct) $\rightarrow_\alpha$ and $\rightarrow_\beta$ commute, then the union relation $\rightarrow$, defined by $a \rightarrow b$ iff there is an $\alpha \in A$ with $a \rightarrow_\alpha b$, is confluent (Hindley 1964).

4.6 Prove that if two abstract reduction relations $\rightarrow_\alpha$ and $\rightarrow_\beta$ on a set X are such that the union relation $\rightarrow$, i.e. $a \rightarrow b$ iff either $a \rightarrow_\alpha b$ or $a \rightarrow_\beta b$, is transitive, then $\rightarrow$ is terminating iff both $\rightarrow_\alpha$ and $\rightarrow_\beta$ are (Geser 1990). You may find Ramsey's theorem useful. Extend this to the case of n different component relations. For an application to termination analysis of programs see Cook, Podelski and Rybalchenko (2006).

4.7 The Collatz conjecture (Lagarias 1985) is that the following recursive function (assuming unlimited range for the integer n) always terminates. Encode this definition as a rewrite system:

```
let rec collatz n =
  if n <= 1 then n
  else if n mod 2 = 0 then collatz (n / 2) else collatz(3 * n + 1);;
```

4.8 Show that the singleton set of rewrite rules $\{f(f(x)) = f(g(f(x)))\}$ is terminating, but this cannot be shown via any simplification order.

4.9 Complete the following rewrite sets taken from Baader and Nipkow (1998): (a) $\{f(g(f(x))) = g(x)\}$ and (b) $\{f(f(x)) = f(x), g(g(x)) = f(x), f(g(x)) = g(x), g(f(x)) = f(x)\}$. Can you characterize the normal forms? You may like to analyze the examples by hand before running completion.

4.10 Suppose E_1 and E_2 are two separate sets of equations, considered as rewrite rules, that have disjoint signatures, i.e. such that the function (including constant) symbols in E_1 do not occur in E_2 and vice versa. Show that if E_1 and E_2 both have the weak normalization

property (every term has a normal form), then so does the combined set $E_1 \cup E_2$. However, give a counterexample to show that even if E_1 and E_2 are terminating (*strongly* normalizing) $E_1 \cup E_2$ may fail to be (Toyama 1987a). Also prove (more difficult) that if E_1 and E_2 are confluent, so is $E_1 \cup E_2$ (Toyama 1987b).

4.11 You will probably find that our present implementation cannot complete the following axioms for 'near rings' in a reasonable time:

$$\begin{aligned} 0 + x &= x, \\ -x + x &= 0, \\ (x + y) + z &= x + (y + z), \\ (x \cdot y) \cdot z &= x \cdot (y \cdot z), \\ (x + y) \cdot z &= x \cdot z + y \cdot z. \end{aligned}$$

Nevertheless, finding a completion is quite feasible (Aichinger 1994). Try optimizing our completion algorithm so that left-reducible rules are put back into the critical pair list, and see if you can then solve it. Can you justify the completeness of this refinement?

4.12 Instead of running completion with a simple queue of critical pairs, an alternative (Lescanne 1984) would be to run the procedure for a while, select the most 'interesting' equations derived – perhaps those with the simplest structure, e.g. $i(i(x)) = x$ above $i(i(x \cdot i(y))) = i(y \cdot i(x))$ – and restart the procedure with the original equations and the interesting ones selected. Implement this idea and see how it works on typical examples. This idea is not restricted to equational reasoning, but could be used for any bottom-up procedure. Try implementing a similar approach to resolution theorem proving and test its effectiveness.

4.13 Although we've exclusively used versions of the LPO as the ordering in rewriting and completion, Knuth and Bendix (1970) originally used somewhat different orderings, now known as Knuth–Bendix orderings. Try these out following Knuth and Bendix's original paper, and try to convince yourselves theoretically that they have the required properties for a simplification order. Take care over the restrictions on the 'weights'.

4.14 Prove that the LPO is total on ground terms (or terms where weights are assigned to the variables as if they were constants).

4.15 Implement basic automated confluence analysis for ordered rewrite systems as follows. Generate all the possible orderings for the (terms substituted for) the variables on the left of a rewrite rule, e.g. for

$(x + y) + z = x + (y + z)$ the orders include $x = y = z$, $x = y < z$, $y < x = z$ and $y < z < z$. Implement a variant of `lpo_gt` that uses these orderings as hypotheses and deduces the ordering of terms built up from them. For each case, analyze critical pairs, exclude those that are ruled out by orderings and try to verify that the feasible critical pairs are joinable subject to the same constraints. Try your code out on the examples from Martin and Nipkow (1990).

4.16 Paramodulation was based on the idea of a special rule for equality, rather than modification of the input formula. We might also consider modifying top-down methods such as tableaux with special equality-handling methods. Study the methods presented by Fitting (1990) and implement and test them on some equality problems. Can you use similar techniques with model elimination?