

Proving Termination of Normalization Functions for Conditional Expressions

Lawrence C Paulson
Computer Laboratory
University of Cambridge
3 June 1985

Boyer and Moore have discussed a recursive function that puts conditional expressions into normal form [1]. It is difficult to prove that this function terminates on all inputs. Three termination proofs are compared: (1) using a measure function, (2) in domain theory using LCF, (3) showing that its *recursion relation*, defined by the pattern of recursive calls, is well-founded. The last two proofs are essentially the same though conducted in markedly different logical frameworks. An obviously total variant of the normalize function is presented as the ‘computational meaning’ of those two proofs.

A related function makes nested recursive calls. The three termination proofs become more complex: termination and correctness must be proved simultaneously. The recursion relation approach seems flexible enough to handle subtle termination proofs where previously domain theory seemed essential.

1980 Math Classification: 68E10 (computer software correctness)

Keywords: Boyer/Moore Theorem Prover, LCF, total correctness, well-founded relations.

Contents

1 A normalization function	2
2 A proof using a measure function	3
3 A proof in the Logic of Computable Functions	3
4 Proving the recursion relation is well-founded	5
5 An obviously total normalize function	6
6 A normalize function with nested recursion	7
7 The LCF proof revisited	8
8 The recursion relation proof revisited	10
9 Conclusions	12

1 A normalization function

Boyer and Moore have published a machine-assisted proof of the correctness of a tautology checker for propositional logic [1]. Propositions are represented as conditional expressions (henceforth *expressions*). An expression is either an *atom* $At(a)$ for some symbol a , or else has the form $If(x, y, z)$ where x, y, z are themselves expressions. An atom represents a propositional letter, while $If(x, y, z)$ is equal to y if x is true, and equal to z otherwise.

The tautology checker includes a function for putting expressions into normal form. An expression is in *normal form* if it has no *tested Ifs*: subexpressions of the form $If(If(u, v, w), y, z)$. Replacing this by $If(u, If(v, y, z), If(w, y, z))$ preserves the value of the entire expression and removes one tested *If*. However new tested *Ifs* are created whenever v or w begin with *If*.

The following program, written in Standard ML [6], defines the data structure *exp* and the normalize function *norm*. If its argument is a tested *If* then *norm* replaces it as above and calls itself recursively. For any other argument *norm* makes recursive calls on the subexpressions. The ML code should fill in the details:

```
type rec exp = data At of string | If of exp × exp × exp;

fun norm(At(a)) = At(a) |
  norm(If(At(a), y, z)) = If(At(a), norm(y), norm(z)) |
  norm(If(If(u, v, w), y, z)) = norm(If(u, If(v, y, z), If(w, y, z)));
```

It is far from obvious that *norm* terminates. In the *If-If* case it calls itself with a larger expression than it was given. One way of proving termination is to find a *well-founded relation* under which the argument ‘goes down’ in every recursive call [1, 5]. Classically, a relation \prec is well-founded if and only if it has no infinite descending chains $\cdots \prec x_2 \prec x_1 \prec x_0$. The less-than relation $<$ on the set \mathbf{N} of natural numbers is well-founded. Less-than is not well-founded on certain other sets: for the integers, $\cdots < -2 < -1 < 0$, and for the rationals, $\cdots < .01 < .1 < 1$.

A common way of defining a well-founded relation on a set A uses a *measure function* $f : A \rightarrow \mathbf{N}$, defining $a' \prec a \iff f(a') < f(a)$. Then \prec is the *inverse image* of $<$ under f . The *lexicographic combination* of two well-founded relations \prec_A and \prec_B defines a well-founded relation \prec on pairs $\langle a, b \rangle$. Here $\langle a', b' \rangle \prec \langle a, b \rangle$ if and only if $a' \prec_A a$ or $a' = a$ and $b' \prec_B b$.

2 A proof using a measure function

In Boyer and Moore's logic all functions are total. Their theorem prover only accepts a recursive definition if it can show that the function terminates on all arguments. For this purpose it uses well-founded relations consisting of lexicographic combinations of inverse images. Boyer and Moore present a well-founded relation for *norm* involving two measures on expressions. Boyer has also sent me a simpler proof, credited to R. Shostak, using a single measure function:

```
fun m(At(a)) = 1 |
  m(If(x, y, z)) = m(x) + m(y) + m(z).
```

To show that this measure goes down in each of *norm*'s recursive calls is a tedious exercise of expanding and collecting terms. It is important to check the easy *If-At* case, because a clever measure that goes down in the hard *If-If* case may not go down in the easy case. Note that $m(x)$ is positive for all x .

Let $U = m(u)$, $V = m(v)$, etc. The *If-At* case terminates because $Y < 1 + Y + Z$ and $Z < 1 + Y + Z$. For the *If-If* case the recursive call has measure

$$\begin{aligned} m(\text{If}(u, \text{If}(v, y, z), \text{If}(w, y, z))) \\ = U + U(V + VY + VZ) + U(W + WY + WZ) \\ = U + UV + UVY + UVZ + UW + UWY + UWZ \end{aligned}$$

and the original argument has measure

$$\begin{aligned} m(\text{If}(\text{If}(u, v, w), y, z)) \\ = U + UV + UW + (U + UV + UW)Y + (U + UV + UW)Z \\ = U + UV + UW + UY + UVY + UWY + UZ + UVZ + UWZ. \end{aligned}$$

Cancelling common terms, this case terminates because $UY + UZ > 0$.

3 A proof in the Logic of Computable Functions

Jacek Leszczyłowski [4] has proved the termination of *norm* using the theorem prover Edinburgh LCF [3]. LCF's logic, a formalization of domain theory, allows reasoning about partial functions. Leszczyłowski's proof uses a lemma that the termination of *norm* in particular cases implies termination in other cases.

Each domain contains an ‘undefined’ element \perp , representing the result of a divergent computation. There is a *weak equality* predicate \equiv such that $x \equiv y$ iff x and y are both undefined, or both defined and equal. Since quantifiers often range over defined values only, let $\forall_D x. P(x)$ abbreviate $\forall x. x \neq \perp \Rightarrow P(x)$. The statement

‘*norm* is total’ is expressed as $\forall_D x. \text{norm}(x) \not\equiv \perp$. The domain of expressions is *flat* to avoid having infinite expressions [9]. The constructor functions *At* and *If* are total:

$$\forall_D a. \text{At}(a) \not\equiv \perp \quad \forall_D xyz. \text{If}(x, y, z) \not\equiv \perp$$

Structural induction for expressions is

$$\frac{\forall_D a. P(\text{At}(a)) \quad \forall_D xyz. P(x) \wedge P(y) \wedge P(z) \Rightarrow P(\text{If}(x, y, z))}{\forall_D x. P(x)}$$

This rule is often stated with the additional premise $P(\perp)$. Then the conclusion is $\forall x. P(x)$.

The function *norm* is expressed as three equations in LCF:

$$\begin{aligned} \forall_D a. \text{norm}(\text{At}(a)) &\equiv \text{At}(a) \\ \forall_D ayz. \text{norm}(\text{If}(\text{At}(a), y, z)) &\equiv \text{If}(\text{At}(a), \text{norm}(y), \text{norm}(z)) \\ \forall_D uvwyz. \text{norm}(\text{If}(\text{If}(u, v, w), y, z)) &\equiv \text{norm}(\text{If}(u, \text{If}(v, y, z), \text{If}(w, y, z))) \end{aligned}$$

The termination proof involves a lemma that if $\text{norm}(y)$ and $\text{norm}(z)$ terminate, then $\text{norm}(\text{If}(x, y, z))$ terminates.

Lemma. $\forall_D xyz. \text{norm}(y) \not\equiv \perp \wedge \text{norm}(z) \not\equiv \perp \Rightarrow \text{norm}(\text{If}(x, y, z)) \not\equiv \perp$

Proof. By structural induction on x . The *At* case reduces to showing

$$\forall_D ayz. \text{norm}(y) \not\equiv \perp \wedge \text{norm}(z) \not\equiv \perp \Rightarrow \text{If}(\text{At}(a), \text{norm}(y), \text{norm}(z)) \not\equiv \perp$$

which follows because *If* and *At* are total.

The *If* case reduces to showing

$$\text{norm}(y) \not\equiv \perp \wedge \text{norm}(z) \not\equiv \perp \Rightarrow \text{norm}(\text{If}(u, \text{If}(v, y, z), \text{If}(w, y, z))) \not\equiv \perp$$

assuming that u, v, w, y, z are all defined and with induction hypotheses for u , v , and w :

$$\begin{aligned} \forall_D yz. \text{norm}(y) \not\equiv \perp \wedge \text{norm}(z) \not\equiv \perp &\Rightarrow \text{norm}(\text{If}(u, y, z)) \not\equiv \perp \\ \forall_D yz. \text{norm}(y) \not\equiv \perp \wedge \text{norm}(z) \not\equiv \perp &\Rightarrow \text{norm}(\text{If}(v, y, z)) \not\equiv \perp \\ \forall_D yz. \text{norm}(y) \not\equiv \perp \wedge \text{norm}(z) \not\equiv \perp &\Rightarrow \text{norm}(\text{If}(w, y, z)) \not\equiv \perp \end{aligned}$$

Assume $\text{norm}(y) \not\equiv \perp$ and $\text{norm}(z) \not\equiv \perp$. The induction hypotheses for v and w imply

$$\text{norm}(\text{If}(v, y, z)) \not\equiv \perp \quad \text{and} \quad \text{norm}(\text{If}(w, y, z)) \not\equiv \perp .$$

Instantiate the induction hypothesis for u with $y \rightarrow \text{If}(v, y, z)$ and $z \rightarrow \text{If}(w, y, z)$, proving

$$\text{norm}(\text{If}(u, \text{If}(v, y, z), \text{If}(w, y, z))) \not\equiv \perp .$$

Q.E.D.

Termination of *norm* on all inputs follows by induction in $\forall_D x. \text{norm}(x) \not\equiv \perp$.

4 Proving the recursion relation is well-founded

The first termination proof defines a well-founded relation using a measure function, and shows that *norm*'s recursive calls obey that relation. A dual approach is to define a relation \prec in terms of *norm*'s recursive calls, then show that \prec is well-founded. Define $x \prec y$ to be *true* whenever evaluating $\text{norm}(x)$ requires a recursive call $\text{norm}(y)$, and to be *false* otherwise. (It should be *false* whenever possible, since additional relationships between elements could prevent \prec from being well-founded.)

I call \prec the *recursion relation* of *norm*.

Case analysis of *norm* defines its recursion relation:

$$\begin{aligned} x \prec \text{At}(a) &\iff \text{false} \\ x \prec \text{If}(\text{At}(a), y, z) &\iff x = y \vee x = z \\ x \prec \text{If}(\text{If}(u, v, w), y, z) &\iff x = \text{If}(u, \text{If}(v, y, z), \text{If}(w, y, z)) \end{aligned}$$

To show the termination of *norm* it suffices to show that the relation \prec is well-founded. This proof will have a remarkable similarity to the LCF proof, which was conducted in domain theory. This section uses a simple mathematical framework with no partial elements. It uses *constructive* mathematics because this paper is an outgrowth of my study of well-founded relations [10] in Martin-Löf's *Constructive Type Theory* [8].

Showing that a relation is well-founded requires showing the soundness of its rule of *well-founded induction* for an arbitrary predicate P :

$$\frac{\forall x. (\forall x'. x' \prec x \Rightarrow P(x')) \Rightarrow P(x)}{\forall x. P(x)}$$

In constructive reasoning, showing that \prec has no infinite descending chains is insufficient to verify the rule. The rule is verified directly, proving its conclusion from its premise. For the rest of this section assume the *induction step*:

$$\forall x. (\forall x'. x' \prec x \Rightarrow P(x')) \Rightarrow P(x) \tag{1}$$

Termination follows from proving $\forall x. P(x)$; a lemma is helpful.

Lemma. $\forall xyz. P(y) \wedge P(z) \Rightarrow P(\text{If}(x, y, z))$

Proof. By structural induction on x . The *At* case is

$$\forall ayz. P(y) \wedge P(z) \Rightarrow P(\text{If}(\text{At}(a), y, z)) ,$$

which follows from (1) and the definition of \prec . Recall that $\text{If}(\text{At}(a), y, z)$ has only two predecessors, y and z .

The *If* case is $\forall yz. P(y) \wedge P(z) \Rightarrow P(\text{If}(\text{If}(u, v, w), y, z))$ under the induction hypotheses

$$\begin{aligned}\forall yz. P(y) \wedge P(z) &\Rightarrow P(\text{If}(u, y, z)) \\ \forall yz. P(y) \wedge P(z) &\Rightarrow P(\text{If}(v, y, z)) \\ \forall yz. P(y) \wedge P(z) &\Rightarrow P(\text{If}(w, y, z)) .\end{aligned}$$

By (1) it is enough to show

$$P(y) \wedge P(z) \Rightarrow P(\text{If}(u, \text{If}(v, y, z), \text{If}(w, y, z))) .$$

Assume $P(y)$ and $P(z)$. The induction hypotheses for v and w imply $P(\text{If}(v, y, z))$ and $P(\text{If}(w, y, z))$. Instantiate the induction hypothesis for u with $y \rightarrow \text{If}(v, y, z)$ and $z \rightarrow \text{If}(w, y, z)$, proving $P(\text{If}(u, \text{If}(v, y, z), \text{If}(w, y, z)))$. Q.E.D.

Now $\forall x.P(x)$ follows immediately by induction on x .

The previous proof can be translated into this one by replacing $\text{norm}(x) \not\equiv \perp$ by $P(x)$. Each unfolding of norm becomes an appeal to the induction step (1). Perhaps domains and partial objects are not essential even for difficult proofs of termination.

As a sample proof in his higher-order theory of constructions, Thierry Coquand has proved the termination of normalization [2] (pages 46–48). He defines a predicate $N(x)$ to mean ‘ x can be put into normal form,’ and proves $\forall x. N(x)$. Translated from his formalism, the axioms are

$$\begin{aligned}N(\text{At}(a)) \\ N(y) \wedge N(z) \Rightarrow N(\text{If}(\text{At}(a), y, z)) \\ N(\text{If}(u, \text{If}(v, y, z), \text{If}(w, y, z))) \Rightarrow N(\text{If}(\text{If}(u, v, w), y, z))\end{aligned}$$

The connection between $N(x)$ and $\text{norm}(x) \not\equiv \perp$ is obvious. Coquand builds a proof object resembling my Constructive Type theory one, using a similar Lemma.

5 An obviously total normalize function

Constructive Type Theory provides a formal interpretation of propositions as types. One consequence is that every proof by induction involves constructing a proof object by recursion. My Type Theory proof that \prec is well-founded suggests another way of writing the normalize function:

```
fun normif(At(a), y, z) = If(At(a), y, z) |
    normif(If(u, v, w), y, z) = normif(u, normif(v, y, z), normif(w, y, z));

fun norm1(At(a)) = At(a) |
    norm1(If(x, y, z)) = normif(x, norm1(y), norm1(z));
```

The function *normif* is obviously total because it is structural recursive in its first argument, a sort of ‘higher type’ recursion. Although *normif* makes nested recursive calls in its second and third arguments, these have no effect on termination. (Ackermann’s function is another example where termination is obvious despite nested recursive calls.) Note the similarity between *normif*’s recursive calls and the appeals to the induction hypotheses in the proof of the lemma.

Proving in LCF that $\forall_D x. \text{norm}(x) \equiv \text{norm}_1(x)$ constitutes yet another termination proof for *norm*. Our familiar lemma now takes the form

$$\forall_D xyz. \text{norm}(\text{If}(x, y, z)) \equiv \text{normif}(x, \text{norm}(y), \text{norm}(z)) ,$$

with essentially the same proof as before.

There is a pleasing concreteness about the first termination proof. But the measure function offers little intuition. The second and third proofs convey something of what *norm* is actually doing, for they give us the function *normif*.

6 A normalize function with nested recursion

If we modify the *If-If* case of *norm* to make nested recursive calls, proving termination becomes trickier still. Call the new function *norm*₂:

```
fun norm2(At(a)) = At(a) |
  norm2(If(At(a), y, z)) = If(At(a), norm2(y), norm2(z)) |
  norm2(If(If(u, v, w), y, z)) =
    norm2(If(u, norm2(If(v, y, z)), norm2(If(w, y, z))));
```

I sent this function as a challenge to Boyer and Moore. The version of the theorem prover described in their book [1] cannot handle this nested recursion. It could not admit *norm*₂ as a function unless, for some measure *m*₂(*x*), it could prove

$$m_2(\text{If}(u, \text{norm}_2(\text{If}(v, y, z)), \text{norm}_2(\text{If}(w, y, z)))) < m_2(\text{If}(\text{If}(u, v, w), y, z)) .$$

Yet this very statement involves *norm*₂.

Moore informs me that the theorem prover has since been extended. A nested recursive function definition can be admitted by showing that it is equivalent to some already accepted definition. In this case replace *norm*₂ by *norm* in the recursion equations and show that the new equations hold. Thus they have at least one solution: *norm*. Then show that some measure decreases for each recursive call of *norm* in the new equations. Thus the solution is unique: by well-founded induction on the measure, *norm*(*x*) = *norm*₂(*x*) for all *x*. Reasoning about *norm* is possible

because it is already known to be a total function. Moore describes this principle of definition in his paper on the termination of Takeuchi's function [7].

Moore's proof of $norm_2$ has several stages:

- $norm(norm(x)) = norm(x)$ is proved by induction on the measure $m(x)$. The result is used in the *At* case of the next stage.
- $\forall yz. norm(If(x, norm(y), norm(z))) = norm(If(x, y, z))$ is proved, like the Lemma, by structural induction on x . The theorem prover does not allow quantified induction schemes, but any instance of one can be specified.
- Therefore $norm$ is a solution to the equations for $norm_2$.
- A function to count the number of tested *Ifs* in an expression is defined. This differs from the function `IFDEPTH` of the original proof [1], which counts the nesting of tested *Ifs*.
- $norm(x)$ is indeed normal: it contains no tested *Ifs*. Proved by induction on $m(x)$.
- The measure for proving uniqueness is the lexicographic combination of the number of tested *Ifs* and the size of an expression.

7 The LCF proof revisited

In domain theory the termination of $norm_2$ can be proved without any mention of $norm$. Termination and partial correctness must be proved *simultaneously*. Showing termination of the *If-If* case requires showing that the nested calls yield normal expressions.

Define the predicate $ISN(x)$ to hold whenever x is in normal form. ISN is a recursive predicate but the recursion is trivially well-founded:

$$\begin{array}{ll} ISN(\perp) & \iff \text{false} \\ \forall_D a. ISN(At(a)) & \iff \text{true} \\ \forall_D ayz. ISN(If(At(a), y, z)) & \iff ISN(y) \wedge ISN(z) \\ \forall_D uvwyz. ISN(If(If(u, v, w), y, z)) & \iff \text{false} \end{array}$$

The element \perp is not in normal form under this definition; $\forall_D x. ISN(norm_2(x))$ states that $norm_2$ is a total function whose result is always normal. This is not a complete statement of correctness; it mentions no relationship between x and $norm_2(x)$.

The proof resembles that of section 3, replacing each occurrence of $\text{norm}(x) \not\equiv \perp$ by $\text{ISN}(\text{norm}_2(x))$.

Fact. If the argument of norm_2 is normal then so is its result:

$$\forall x. \text{ISN}(x) \Rightarrow \text{ISN}(\text{norm}_2(x)) \quad (2)$$

Proof. By structural induction on x . The \perp and At cases are easy. For If consider two cases. Since $\text{If}(\text{If}(u, v, w), y, z)$ is not normal the result holds vacuously. The $\text{If}(\text{At}(a), y, z)$ case is

$$\text{ISN}(\text{If}(\text{At}(a), y, z)) \Rightarrow \text{ISN}(\text{norm}_2(\text{If}(\text{At}(a), y, z)))$$

which simplifies to

$$\text{ISN}(y) \wedge \text{ISN}(z) \Rightarrow \text{ISN}(\text{norm}_2(y)) \wedge \text{ISN}(\text{norm}_2(z))$$

which follows from the induction hypotheses.

Now we have the usual

$$\text{Lemma. } \forall_D xyz. \text{ISN}(\text{norm}_2(y)) \wedge \text{ISN}(\text{norm}_2(z)) \Rightarrow \text{ISN}(\text{norm}_2(\text{If}(x, y, z)))$$

Proof. By structural induction on x . The At case reduces to the clearly true

$$\forall_D ayz. \text{ISN}(\text{norm}_2(y)) \wedge \text{ISN}(\text{norm}_2(z)) \Rightarrow \text{ISN}(\text{If}(\text{At}(a), \text{norm}_2(y), \text{norm}_2(z)))$$

The If case reduces to showing, under induction hypotheses,

$$\text{ISN}(\text{norm}_2(y)) \wedge \text{ISN}(\text{norm}_2(z)) \Rightarrow \text{ISN}(\text{norm}_2(\text{If}(u, \text{If}(v, y, z), \text{If}(w, y, z)))) .$$

Assume $\text{ISN}(\text{norm}_2(y))$ and $\text{ISN}(\text{norm}_2(z))$. The induction hypotheses for v and w imply $\text{ISN}(\text{norm}_2(\text{If}(v, y, z)))$ and $\text{ISN}(\text{norm}_2(\text{If}(w, y, z)))$. Now comes a clear departure from the section 3 proof: inserting an extra call to norm_2 . The Fact (2) gives

$$\text{ISN}(\text{norm}_2(\text{norm}_2(\text{If}(v, y, z)))) \quad \text{and} \quad \text{ISN}(\text{norm}_2(\text{norm}_2(\text{If}(w, y, z)))) .$$

Instantiate the induction hypothesis for u with $y \rightarrow \text{norm}_2(\text{If}(v, y, z))$ and $z \rightarrow \text{norm}_2(\text{If}(w, y, z))$. Q. E. D.

Again the overall proof for norm_2 is an easy induction using the Lemma. Proving $\forall_D x. \text{ISN}(\text{norm}_2(x))$ rather than $\forall_D x. \text{norm}_2(x) \not\equiv \perp$ is a classic example of strengthening the goal in order to strengthen the induction hypotheses. The Lemma is essentially the inductive step. Its proof requires the Fact (2). Proving $\forall_D x. \text{norm}_2(x) \not\equiv \perp$ would require a Lemma of the form

$$\forall_D xyz. \text{norm}_2(y) \not\equiv \perp \wedge \text{norm}_2(z) \not\equiv \perp \Rightarrow \text{norm}_2(\text{If}(x, y, z)) \not\equiv \perp$$

and a Fact of the form $\forall x.x \not\equiv \perp \Rightarrow \text{norm}_2(x) \not\equiv \perp$. We are going in circles! It would suffice to prove a weaker version of the Fact:

$$\forall x.\text{norm}_2(x) \not\equiv \perp \Rightarrow \text{norm}_2(\text{norm}_2(x)) \not\equiv \perp.$$

An attempted proof of this resembles that of (2) except that the *If-If* case is no longer trivial.

8 The recursion relation proof revisited

The recursion relation proof of section 4 can similarly be adapted to norm_2 . I continue to use the predicate *ISN* for reasoning about normal expressions, though in this section there is no element \perp . The recursion relation \prec_2 is defined like \prec , except that the *If-If* case has three recursive calls instead of one. The outer call involves the results of the inner calls, expressed as existentially quantified variables. The results are *assumed* to be in normal form:

$$\begin{aligned} x \prec_2 \text{At}(a) &\iff \text{false} \\ x \prec_2 \text{If}(\text{At}(a), y, z) &\iff x = y \vee x = z \\ x \prec_2 \text{If}(\text{If}(u, v, w), y, z) &\iff \left(\begin{array}{l} x = \text{If}(v, y, z) \vee \\ x = \text{If}(w, y, z) \vee \\ \exists v'w'.\text{ISN}(v') \wedge \text{ISN}(w') \wedge x = \text{If}(u, v', w') \end{array} \right) \end{aligned}$$

It will be necessary to show inductively that the equations for norm_2 produce normal expressions. First let us show that \prec_2 is well-founded. Assume the induction step for an arbitrary P :

$$\forall x. (\forall x'. x' \prec_2 x \Rightarrow P(x')) \Rightarrow P(x) \quad (3)$$

Fact. The induction step (3) implies $P(x)$ for all x in normal form.

$$\forall x.\text{ISN}(x) \Rightarrow P(x) \quad (4)$$

Proof. By structural induction on x . The *At* case is easy. Since $\text{If}(\text{If}(u, v, w), y, z)$ is not normal, this case is vacuous. The $\text{If}(\text{At}(a), y, z)$ case is

$$\text{ISN}(\text{If}(\text{At}(a), y, z)) \Rightarrow P(\text{If}(\text{At}(a), y, z))$$

which simplifies, using the induction step, to

$$\text{ISN}(y) \wedge \text{ISN}(z) \Rightarrow P(y) \wedge P(z)$$

which follows from the induction hypotheses.

The Lemma is stated just like in section 4:

Lemma. $\forall xyz. P(y) \wedge P(z) \Rightarrow P(If(x, y, z))$

Proof. By structural induction on x . The *At* case is proved as before. The *If* case is

$$\forall yz. P(y) \wedge P(z) \Rightarrow P(If(If(u, v, w), y, z)) .$$

By (3) it is enough to show that $P(y)$ and $P(z)$ imply each of

$$\begin{aligned} &P(If(v, y, z)) \\ &P(If(w, y, z)) \\ \forall v'w'. &ISN(v') \wedge ISN(w') \Rightarrow P(If(u, v', w')) \end{aligned}$$

The induction hypotheses for v and w imply $P(If(v, y, z))$ and $P(If(w, y, z))$. It suffices to show $P(If(u, v', w'))$ for arbitrary normal expressions v' and w' . The Fact (4) implies $P(v')$ and $P(w')$. Instantiate the induction hypothesis for u with $y \rightarrow v'$ and $z \rightarrow w'$. Q.E.D.

The translation from the domain theory proof replaces $ISN(norm_2(x))$ by $P(x)$. The connection between the proofs is weaker than it was for $norm$. Domain theory allows explicit mention of $norm_2$'s recursive calls when instantiating u 's induction hypothesis; the recursion relation hides the calls via quantifiers.

The justification of $norm_2$ still requires simultaneous proofs that it terminates yielding a normal expression. The proof is by well-founded induction on \prec_2 :

- Given $At(a)$ it makes no recursive calls and returns an atom, which is always normal.
- Given $If(At(a), y, z)$ it makes recursive calls on the predecessors y and z . By induction hypotheses these calls return normal expressions so the final result is normal.
- Given $If(If(u, v, w), y, z)$ it makes recursive calls on predecessors $If(v, y, z)$ and $If(w, y, z)$. By induction hypotheses these return normal expressions v' and w' . So $If(u, v', w')$ is a predecessor, justifying the final recursive call. By induction hypothesis this returns a normal expression.

This reasoning about the various cases of $norm_2$ can be formalized in my setting of well-founded recursion operators in Constructive Type Theory [10]. A function application has type $\sum_{y \in exp} ISN(y)$. It returns a *pair* of results: a normal expression y and a proof object of type $ISN(y)$. Each recursive call on an argument z must be justified by exhibiting a proof object of type $z \prec_2 x$. This is passed as an additional argument. In the *If-If* case, the outer call passes a proof object for $If(u, v', w') \prec_2$

$If(If(u, v, w), y, z)$, constructed from proof objects $ISN(u')$ and $ISN(v')$ from the inner calls.

After performing this elaborate construction of a well-founded recursion, the equations for $norm_2$ can be proved as usual in the approach [10].

9 Conclusions

Domain theory allows reasoning about recursion in a most flexible way, but at a heavy cost of complexity. The ideas can be difficult to grasp and introduce theoretical and practical obstacles. LCF is the only major theorem proving project that uses domain theory; even LCF users sometimes prefer to do without domains. The recursion relation proofs of $norm$ and $norm_2$ suggest that domains are not essential for reasoning about many programs. Domain theory still has a vital role to play: there is no alternative for reasoning about compilers and continuously running processes.

Constructive Type Theory is concerned with terminating computations. Domain theory cannot be patched onto it; partial functions are completely antithetical to its view of computation. Using recursion relations it can express termination proofs of $norm$ and $norm_2$, thereby deriving the function $normif$.

It is especially hard to prove the termination of functions involving *nested recursion*, since the termination of an outer recursive call may depend on a property of the results of the inner calls. The example $norm_2$ shows that this can be done without domains. Manna and Waldinger have studied a much more interesting nested recursive function, *UNIFY*, which performs unification [5]. They define a well-founded relation involving the structure of the expressions being unified and the number of distinct variables in those expressions. Under this relation, *UNIFY*'s outer recursive call can be justified only if its inner call returns a most-general, idempotent unifier of its arguments.

I formalized their work using the theorem prover Cambridge LCF, proving the total correctness of *UNIFY* [12]. The proof used a predicate *BEST-UNIFY-TRY* in a role analogous to that of *ISN* in the proof of $norm_2$: to allow the simultaneous proof of termination and correctness. It appears possible to verify *UNIFY* in Constructive Type Theory. Manna and Waldinger's well-founded relation is appropriate; there is no need to consider the recursion relation. *UNIFY* would return a substitution paired with a proof that this substitution had the necessary properties.

Acknowledgements. Robert Boyer and J Moore answered several queries about $norm$ and generously hosted my visit to the University of Texas at Austin. Michael

J. C. Gordon read drafts of this paper.

Appendix: the Cambridge LCF proof

Here is a sequence of commands that causes Cambridge LCF to prove the termination *norm* in domain theory. It is simpler than Leszczyłowski's Edinburgh LCF proof [4] because LCF has developed since then. No special ML code need be written: the data structure *exp* is defined automatically, and the standard rewriting tactic is powerful enough to handle both theorems. I performed this proof in half an hour at the terminal. I also proved the equivalence of *norm* and *norm*₁, and verified *norm*₂. These proofs are similar and not presented here.

Note that the ML used in Cambridge LCF is not Standard ML. An effort is underway to bring this ML up to date with the Standard.

This table gives LCF's printed representation of each logical connective:

!	\forall	universal quantifier
?	\exists	existential quantifier
$/\wedge$	\wedge	conjunction
$\vee\backslash$	\vee	disjunction
$\Rightarrow\Leftarrow$	\Rightarrow	implication
$\Leftarrow\Rightarrow$	$\Leftarrow\Rightarrow$	biconditional
\sim	\neg	negation
UU	\perp	undefined element of domain
\equiv	\equiv	equivalence (weak equality)
\ll	\subseteq	partial ordering on domain

The theory *exp* is declared, with a type operator of the same name. If τ is a type, then τexp is the type of expressions whose atoms have type τ . In LCF, $*$, $**$, ... are type variables. The **struct_axm** command defines expressions as a recursive type with strict constructors **ATOM** and **IF**. The constructor functions are *curried*: we must write **IF** *x* *y* *z* instead of *If*(*x*, *y*, *z*).

```
new_theory 'exp';
new_type 1 'exp';
struct_axm ("*: exp", 'strict',
  ['ATOM', ["a:*"]; 'IF', ["x: * exp"; "y: * exp"; "z: * exp"]]);;
```

The function symbol *norm* is declared, and a new axiom asserts its definition. The quantifier \forall_D is not built in: we must write $!a.\sim a==UU \Rightarrow$ instead of $\forall_D a$.

```

new_constant ('NORM', ": (* exp) -> (* exp)");; 

let NORM_CLAUSES =
new_closed_axiom ('NORM_CLAUSES',
"(!a.~ a==UU : * ==> NORM(ATOM(a)) == ATOM(a)) /\ 
(! a y z.~ a==UU:* /\ ~ y==UU /\ ~ z==UU ==>
    NORM (IF (ATOM a) y z) == IF (ATOM a) (NORM y) (NORM z)) /\ 
(!u v w y z.~ u==UU /\ ~ v==UU /\ ~ w==UU /\ ~ y==UU /\ ~ z==UU ==>
    NORM (IF (IF u v w) y z) ==
    NORM (IF u (IF v y z) (IF w y z)) : * exp)");;

```

An axiom, previously created by `struct_axm`, is bound to the ML identifier `EXP_DEFINED`. The structural induction tactic is instantiated to handle expressions and bound to the ML identifier `EXP_TAC`.

```

let EXP_DEFINED = axiom 'exp' 'DEFINED';;
let EXP_TAC = STRUCT_TAC 'exp' [];;

```

The Lemma is proved by induction followed by rewriting via the equations for *norm* and the totality of the constructors *At* and *If*. I have tweaked the statement of the Lemma to circumvent an annoyance involving admissibility of induction [3].

```

let NORM_LEMMA =
prove_thm ('NORM_LEMMA',
"!x y z. ~ y==UU /\ ~ z==UU /\ ~ NORM(y)==UU /\ ~ NORM(z)==UU ==>
    ~ x==UU ==> ~ NORM(IF x y z)==UU : * exp",
EXP_TAC "x" THEN
ASM_REWRITE_TAC [NORM_CLAUSES; EXP_DEFINED]);;

```

The proof that *norm* is total resembles the proof of the Lemma.

```

let NORM_TOTAL =
prove_thm ('NORM_TOTAL',
"!x. ~ x==UU ==> ~ NORM x ==UU : * exp",
EXP_TAC "x" THEN
ASM_REWRITE_TAC [NORM_CLAUSES; EXP_DEFINED; NORM_LEMMA]);;

```

References

- [1] R. Boyer and J Moore, *A Computational Logic* (Academic Press, 1979).
- [2] T. Coquand, *Une Théorie des Constructions*, Thèse de 3ème cycle (in French), University of Paris VII (1985).
- [3] M. J. C. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation* (Springer, 1979).
- [4] J. Leszczylowski, An experiment with ‘Edinburgh LCF,’ in: W. Bibel and R. Kowalski, editors, *Fifth Conference on Automated Deduction*, Springer LNCS 87 (1980), pages 170–181.
- [5] Z. Manna, R. Waldinger, Deductive synthesis of the unification algorithm, *Science of Computer Programming* **1** (1981), pages 5–48.
- [6] R. Milner, A proposal for Standard ML, *ACM Symposium on Lisp and Functional Programming* (1984), pages 184–197.
- [7] J S. Moore, A mechanical proof of the termination of Takeuchi’s function, *Information Processing Letters* **9** (1979), pages 176–181.
- [8] B. Nordström and J. Smith, Propositions and specifications of programs in Martin-Löf’s type theory, *BIT* **24** (1984), pages 288–301.
- [9] L. C. Paulson, Deriving structural induction in LCF, in: G. Kahn, D. B. MacQueen, G. Plotkin, editors, *International Symposium on Semantics of Data Types* (Springer, 1984), pages 197–214.
- [10] L. C. Paulson, Constructing recursion operators in Intuitionistic Type Theory, Report 57, Computer Lab., University of Cambridge (1984).
- [11] L. C. Paulson, Lessons learned from LCF: A Survey of Natural Deduction Proofs, *Computer Journal* **28** (1985), 474–479.
- [12] L. C. Paulson, Verifying the unification algorithm in LCF, *Science of Computer Programming* **5** (1985), pages 143–170.