Program Transformation and Proof Transformation

by

Wilfried Sieg and Stanley S. Wainer

June 1994

Report CMU-PHIL-56

Philosophy
Methodology
Logic

Pittsburgh, Pennsylvania 15213-3890

Program Transformation and Proof Transformation

Wilfried Sieg
(Carnegie Mellon University, Pittsburgh USA)

Stanley S. Wainer *
(University of Leeds UK)

Abstract. *A "linear - style" sequent calculus makes it possible to explore the close structural relationships between primitive recursive programs and their inductive termination proofs, and between program transformations and their corresponding proof transformations. In this context the recursive - to - tail - recursive transformation corresponds proof theoretically to a certain kind of cut elimination, called here "call by value cut elimination".*

1 Introduction.

An old and well-known theorem of logic, due variously to Kreisel, Parsons, Mints, Takeuti and others, states that the primitive recursive functions are exactly those which can be proved to terminate in the fragment of arithmetic with induction restricted to existential (Σ_1) formulas, and more generally as shown in Sieg (1991), in the fragment with (Π_2) - induction provided any side assumptions are less complex, i.e. at worst Σ_2. This is an "extensional" result, characterizing a certain class of number-theoretic *functions* .

What we are looking for here is something more "intensional", i.e. a logic which allows us to distinguish between different kinds of primitive re-

*The second author thanks the Department of Philosophy at Carnegie Mellon University for generous hospitality during a sabbatical year as a Fulbright Scholar 1992-93.

cursive *programs* according to the structure of their respective termination proofs. Preferably it should provide a clear correspondence between proofs and programs, and also at the higher level between proof-transformations and program- transformations, so that "program-complexity" is measurable directly in terms of "proof complexity". See Feferman (1992) for another development of similar ideas.

Note that work of e.g. Goad (1980), Pfenning (1990), Schwichtenberg (1992), Madden (1992), and Anderson (1993) already illustrates the potential applicability of proof-transformation as a means to synthesize and analyze useful program-transformations. However our present concern lies rather in the general proof-theoretic principles which underly such applications. Thus we will be very restrictive in considering only programs over the natural numbers N since they already serve to illustrate the essential logical features, but with the least amount of syntactic "fuss".

The logic we arrive at below is a strictly "linear" one (no contraction, no weakening and no exchange !) obtained simply by analyzing just what one needs to prove the totality of primitive recursive definitions. The absence of exchange rules means that two cut rules are needed - an ordinary one and another one which we call "call-by-value cut" for reasons which will be obvious. It then turns out that, in the appropriate setting, the transformation from recursive to tail-recursive programs is precisely call-by-value-cut-elimination !

Definition. A *recursive program* over N, the natural numbers, is a finite sequence of function-definitions

$$f_i\,(\ldots) \;=\; T_i\,(f_0,\ldots,f_{i-1},f_i\,;x_1,\ldots,x_k)$$

where the arguments ... of f_i are the variables $x_1,\ldots,x_k$, or successors of them, or the constant 0; and T_i is a term built up from those variables and the constant 0 by arbitrary applications of the successor function, the previously defined functions $f_0,\ldots,f_{i-1}$ and possibly the function f_i itself.

Evaluation is by "substitution" or "call-by-value". That is, on given numerical inputs $\vec{n} = n_1,\ldots,n_k$ we are allowed successively to replace any subterm $f_j(\vec{n})$ of T_i by its previously-computed value until eventually a numerical value for $f_i(\vec{n})$ is obtained. However termination is not guaranteed in general.

A *(partial) recursive function* is then one which is defined by a recursive

program, according to the usual least-fixed-point semantics.

Example. Given **a program defining** *g,* **add to** it the new equations :

$$\begin{aligned} h(0,y,z) &= y \\ h(x+1,y,z) &= z \\ f(x) &= h(g(x), x, f(x+1)) \end{aligned}$$

Then on input x := n, and assuming $g(n)$, $g(n+1)$, ... defined, we have

$$\begin{aligned} f(n) &= n && \text{if } g(n) = 0 \\ &= f(n+1) && \text{if } g(n) \neq 0 \\ f(n+1) &= n+1 && \text{if } g(n+1) = 0 \\ &= f(n+2) && \text{if } g(n+1) \neq 0 \end{aligned}$$

etcetera, and therefore if g has a "zero" above n then / computes it :

$$f(n) = \text{least } m \geq n \text{ such that } g(m) = 0 .$$

Note on the other hand that if the third equation defining / is simply regarded as a left-to-right term rewrite then $f(n)$ can be expanded continually without ever giving a value.

Definition. Call a recursive program defining a function / *provably recursive* or *verifiable* in a given logic L if

$$L \vdash \forall x \exists y [f(x) = y].$$

Obviously the more restrictive the logic, the more restricted will be the class of recursive programs we can prove to terminate in it. The aim here is to impose simple logics on the equation calculus in such a way that there is a clear and precise structural correspondence between termination proofs and known subclasses of recursive programs. We concentrate here on primitive recursive programs, though the ideas have a much wider range of application.

2 Primitive Recursive Programs.

Definitions. A *primitive recursive* program is one in which every defining equation has one of the five forms "zero", "successor", "projection", "explicit

definition" and "primitive recursion" as follows :

$$
\begin{array}{lllll}
(Z) & f_i(x) & = & 0 \\
(S) & /<(*) & = & ar + 1 \\
(P) & /«(*) & = & {}^{x}i \\
(E) & f_i(\vec{x}) & = & {}^{r}(/o, \ldots, f_{i-1}; \vec{x}) \\
(PRo) & f_i(0, \vec{x}) & = & f_{i_0}(\vec{x}) \\
(PR_1) & f_i(z+1, \vec{x}) & = & f_{i_1}(z, \vec{x}, f_i(z, \vec{x}))
\end{array}
$$

where in the *(PR)* scheme to, t'i < t.

A *generalized primitive recursive* program is one in which the primitive recursion equation *(PRi)* is generalized to allow substitution of terms for the parameters $\vec{x}$ in the recursive call /,-(z, $\vec{x}$) as follows :

$$(GPRi) \quad fi(z + l, \vec{x}) = fi_1(z, \vec{x}, fi(z, fi_2(z\vec{x}), \ldots, /,-_{k+1}(z, \vec{x})))$$

where to, ti, t2, ..., tjk+i < t.

A primitive *tail recursive* program is one in which generalized primitive recursion is allowed, but only in the following restricted context, where the recursive call on /i(z,...) is the final function-call made in the evaluation of $f_i(z+1, \vec{x})$:

$$(TR_1) \quad f_i(z+1, \vec{x}) = f_i(z, f_{i_1}(z, \vec{x}), \ldots, f_{i_k}(z, \vec{x})) .$$

Remark. Tail recursive programs

$$
\begin{array}{lll}
f(0, x) & = & g(x) \\
f(z + l, x) & = & /(*, *(*, *))
\end{array}
$$

axe "efficient" since they can immediately be recast as while-loops :

$$\text{while } z \wedge 0 \text{ do } z := z - 1;\ x := A(z, x) \text{ od} \ ;\ / := g(x) .$$

The following transformations are either explicit or implicit in the classic R. Péter (1967) which contains a wealth of information on the reduction of various kinds of recursions to simpler forms. See also for example, Wainer (1993) for a survey of further basic information on primitive recursion.

Theorem 2.1 *Every generalized primitive recursive program can be transformed into a primitive tail recursive program defining the same function. Every primitive tail recursive program can be transformed into an ordinary primitive recursive program defining the same function.*

Proof. (i) A generalized primitive recursion (i.e. with parameter substitution) such as

$$\begin{aligned} f(0,x) &= g(x) \\ f(z+1,x) &= h(z,x,f(z,p(z,x))) \end{aligned}$$

can be transformed into a tail recursive program as follows (note however that three tail recursions seem to be needed - the two given here plus another one implicitly used in order to define the "modified minus" from the predecessor)

$$\begin{array}{lll} (TR_0) & f_0(0,z,x) = & x \\ (TR_1) & f_0(n+1,z,x) = & f_0(n,z\dot{-}1,p(z\dot{-}1,x)) \\ (E) & f_1(n,z,x,y) = & h(z\dot{-}(n+1),f_0(n,z,x),y) \\ (TR_0) & f_2(0,z,x,y) = & y \\ (TR_1) & f_2(n+1,z,x,y) = & f_2(n,z,x,f_1(n,z,x,y)) \\ (E) & f_3(z,x) = & f_2(z,z,x,g(f_0(z,z,x)) \end{array}$$

The devoted (!) reader with a taste for intricate inductions might now like to verify that

$$\forall z\, \forall x\, (\, f_3(z,x) = f(z,x)\,)\,.$$

Hint : one needs first to check the following identities

$$f_1(n+1,z+1,x,y) = f_1(n,z,p(z,x),y)$$

$$f_2(n+1,z+1,x,y) = h(z,x,f_2(n,z,p(z,x),y))$$

and then a further induction on z yields the desired result.

(ii) A primitive tail recursion such as

$$\begin{aligned} f(0,x) &= g(x) \\ f(z+1,x) &= f(z,p(z,x)) \end{aligned}$$

can be transformed into an ordinary primitive recursion as follows

$$\begin{array}{lll} (PR_0) & f_0(0,z,x) = & x \\ (PR_1) & f_0(n+1,z,x) = & p(z\dot{-}n,f_0(n,z,x)) \\ (E) & f_1(z,x) = & g(f_0(z,z\dot{-}1,x))\,. \end{array}$$

The verification needs a preliminary induction on n to show

$$f_0(n+1,z,x) = f_0(n,z\dot{-}1,p(z,x))$$

and then by a further induction on z,

$$VzVx(f_x(z,x) = f(z,x)).$$

Notice that the above program - equivalences are all provable by inductions on quantifier - free equational formulas, or on universally quantified equational formulas, i.e. Ill formulas.

We are now going to devise a logic exactly tailored to proofs about primitive recursive and generalized primitive recursive programs.

3 The Logic of Primitive Recursion (LPR).

Formulas A, *B*, C,... will be either atoms of the form $f(\vec{x}) \simeq y$ with y a variable, meaning $f(\vec{x})$ is defined with value y, or Ei -formulas $\exists y(f(\vec{x}) \simeq y)$ or n_2 -formulas $\forall \vec{x} \exists y(f(\vec{x}) \simeq y)$.

The axioms are of two kinds, the principal ones being purely relational sequents or "logic programs" describing the order of evaluation of individual equations in a primitive recursive program, thus for example

$$(Ax) \quad f_0(\vec{x}) \simeq y_0,\ f_1(\vec{x}, y_0) \simeq y_1,\ \ldots,\ f_m(\vec{x}, y_0, \ldots, y_{m-1}) \simeq y_m \vdash f(\vec{x}) \simeq y_m$$

corresponds to an explicit definition

$$f_m(\vec{x}, f_0(\vec{x}), f_1(\vec{x}, f_0(\vec{x})), \ldots) .$$

The other axioms simply express that the zero, successor and projection functions are defined :

$$(N\text{-}Ax) \quad \vdash \exists y(0 \simeq y),\ \vdash \exists y(x + 1 \simeq y),\ \vdash \exists y(x \simeq y) .$$

The logic rules are the sequent rules for $\exists$ and $\forall$:

$$(\exists \vdash) \ \frac{\ldots, A(y) \vdash B}{\ldots, \exists y A(y), \ldots \vdash B} \qquad (\vdash \exists) \ \frac{\ldots \vdash B, \exists y B(y), \ldots}{\ldots \vdash \exists y B(y)}$$

$$(\forall \vdash) \ \frac{\ldots, A(t), \ldots \vdash B}{\ldots, \forall x A(x), \ldots \vdash B} \qquad (\vdash \forall) \ \frac{\ldots \vdash B, \ldots}{\ldots \vdash \forall x S(x)}$$

with the usual "eigenvariable" conditions on (3 h) and (h V), i.e. the quantified variable can not occur free in the "side formulas".

In addition there are two cut rules :

$$(C)\ \frac{\vdash C \quad C, \ldots \vdash B}{\cdots \vdash B} \qquad (CVC)\ \frac{\ldots \wedge \ldots \ldots jC, \ldots \vdash B}{\ldots \ldots \ldots \ldots V\ B}$$

and the induction rule :

$$(IND)\ \frac{\exists\ B(0) \quad B(z) \vdash B(z+1)}{\wedge W)}\ *$$

Note. What you see is all there is ! The dots ... denote arbitrary finite sequences of assumptions and the logic is strictly linear in the sense that there are no hidden structural rules - no Contraction, no Weakening, and furthermore no Exchange ! Hence the need for two Cut rules, the second of which applies a cut "in context" and is called a "call by value" cut for reasons which will shortly become obvious. Note also that there are no other assumptions in the induction rule besides the induction hypothesis *B(z).*

Definition. LPR(3) and LPR(V3) denote the logics restricted to Ex and U_2 formulas respectively. LPR(V3)-(CVC) denotes the logic LPR(V3) without call-by-value cuts.

Theorem 3.1 • *Primitive Recursive* ≡ *LPR(3) - verifiable.*

- *Generalized Primitive Recursive* ≡ *LPRfi/3) - verifiable.*
- *Primitive Tail Recursive* ≡ *LPR(43)-(CVC) - verifiable.*

Proof. We do not give a completely detailed proof here, but sufficient to display the basic relationships.

(i) That primitive recursive programs are LPR(3) - verifiable is easily seen. Suppose for example that / is defined explicitly from *g* and *h* by

$$f(x) = g(h(x))$$

where *g* and *h* are already assumed to be LPR(3) - verifiable. Then the starting axiom is

$$h(x) \simeq y\ ,\ g(y) \simeq z \vdash f(x) \simeq z\ .$$

By ($\vdash \exists$) followed by ($\exists \vdash$) we then obtain

$$h(x) \simeq y \ , \ \exists z \ (g(y) \simeq z) \ \vdash \ \exists z \ (f(x) \simeq z) \ .$$

From this and the assumption $\exists z \ (g(y) \simeq z)$ we then have by a call by value cut (CVC),

$$h(x) \simeq y \ \vdash \ \exists z \ (f(x) \simeq z)$$

and then by ($\exists \vdash$),

$$\exists y \ (h(x) \simeq y) \ \vdash \ \exists z \ (f(x) \simeq z) \ .$$

Thus by the assumption $\exists y \ (h(x) \simeq y)$ and an ordinary cut (C),

$$\vdash \ \exists z \ (f(x) \simeq z) \ .$$

Note how the eigenvariable conditions on ($\exists \vdash$) rules completely determine the order of events in the above proof, so that the call by value cut was essential.

As a further example, suppose f is defined primitive recursively from g and h as follows ;

$$\begin{array}{lcl} f(0,x) & = & g(x) \\ f(z+1,x) & = & h(z,x,f(z,x)) \ . \end{array}$$

where $\vdash \exists y \ (g(x) \simeq y)$ and $\vdash \exists u \ (h(z,x,y) \simeq u)$ are assumed. Then the starting axioms are

$$g(x) \simeq y \ \vdash \ f(0,x) \simeq y$$

and

$$f(z,x) \simeq y \ , \ h(z,x,y) \simeq u \ \vdash \ f(z+1,x) \simeq u \ .$$

Concentrating on the induction step first, we have by ($\vdash \exists$) and ($\exists \vdash$),

$$f(z,x) \simeq y \ , \ \exists u \ (h(z,x,y) \simeq u) \ \vdash \ \exists y \ (f(z+1,x) \simeq y) \ .$$

Then by a call by value cut,

$$f(z,x) \simeq y \ \vdash \ \exists y \ (f(z+1,x) \simeq y)$$

and by ($\exists \vdash$),

$$\exists y \ (f(z,x) \simeq y) \ \vdash \ \exists y \ (f(z+1,x) \simeq y) \ .$$

Applying ($\vdash \exists$), ($\exists \vdash$) and an ordinary cut to the first axiom we easily obtain $\vdash \exists y\,(f(0,x) \simeq y)$, and so by the induction rule we have

$$\vdash \exists y\,(f(z,x) \simeq y)$$

as required.

(ii) Next we show why LPR($\forall\exists$) - verifiable programs are generalized primitive recursive. Suppose we had a proof of

$$\vdash \forall x\,\exists y\,(f(z,x) \simeq y)$$

by induction on z. The induction step would therefore be

$$\forall x\,\exists y\,(f(z,x) \simeq y) \vdash \forall x\,\exists y\,(f(z+1,x) \simeq y)\,.$$

This deduction presumably used some recursive calls on "given" functions, so let us assume it came about by means of one ordinary cut on a function p and a call by value cut on a function h from :

$$\forall x\,\exists u\,(p(z,x) \simeq u)\,,\ \ \forall x\,\exists y\,(f(z,x) \simeq y)\,,\ \forall x\,\forall y\,\exists v\,(h(z,x,y) \simeq v) \vdash \forall x\,\exists y\,(f(z+1,x) \simeq y)\,.$$

The eigenvariable conditions place heavy restrictions on how this could have been derived. Essentially it must have come about by applying ($\exists \vdash$), ($\forall \vdash$), ($\vdash \forall$), in that order (!) to :

$$p(z,x) \simeq u\,,\ \ \forall x\,\exists y\,(f(z,x) \simeq y)\,,\ \forall x\,\forall y\,\exists v\,(h(z,x,y) \simeq v) \vdash \exists y\,(f(z+1,x) \simeq y)\,.$$

Stripping away the quantifiers prefixing $f(z,x) \simeq y$ we now see that this would have come from

$$p(z,x) \simeq u,\ f(z,u) \simeq y,\ \forall x\,\forall y\,\exists v\,(h(z,x,y) \simeq v) \vdash \forall x\,\exists y\,(f(z+1,x) \simeq y)$$

by applying ($\exists \vdash$) and then ($\forall \vdash$) with u as witnessing variable (the only other possible witnessing variables would have been z or x but these are less general). Now we can strip away the quantifier prefix on $h(z,x,y) \simeq v$ to see that this last line would have come about by applying ($\exists \vdash$) and ($\forall \vdash$) to :

$$p(z,x) \simeq u\,,\ f(z,x) \simeq y\,,\ h(z,x,y) \simeq v \vdash \exists y\,(f(z+1,x) \simeq y)\,.$$

Finally, this would have arisen by ($\vdash \exists$) from the axiom :

$$p(z,x) \simeq u \ , \ f(z,x) \simeq y \ , \ h(z,x,y) \simeq v \ \vdash \ f(z+1,x) \simeq v$$

describing a generalized primitive recursion :

$$\begin{array}{lcl} f(0,x) & = & g(x) \\ f(z+1,x) & = & h(z,x,f(z,p(z,x))) \ . \end{array}$$

By reversing the above we also obtain the converse, that every generalized primitive recursion is LPR($\forall\exists$) - verifiable. Note that if we took apart an LPR($\exists$) - inductive proof in a similar way then we would be prevented (by the absence of the $\forall \vdash$ rule) from substituting $p(z,x)$ for the variable x and so an ordinary primitive recursive program would be the only possible result. Hence the converse to part (i).

(iii) The only other crucial thing to note is that if call by value cuts were disallowed in the derivation in part (ii) above, then the h function could not appear and so the extracted program would have to be a tail recursion :

$$\begin{array}{lcl} f(0,x) & = & g(x) \\ f(z+1,x) & = & f(z,p(z,x)) \ . \end{array}$$

This completes the proof.

Theorem 3.2 *Hence the transformation from generalized primitive recursive programs to primitive tail recursive programs corresponds exactly to the elimination of call by value cuts in LPR($\forall\exists$).*

Remarks.

A careful analysis of the above termination proofs in LPR should convince the reader of the close correspondence between the proof - structure and the computation - structure of the given program. By reading the termination proof in a goal - directed way, one sees how the order of $\forall\exists$ - eliminations exactly reflects the intended order of evaluation.

Although the transformation to tail recursion corresponds to elimination of call by value cuts in LPR(V3), the actual transformation itself takes place at the equational rather than the logical level, as given by Theorem 2.1. Thus most of the complexity of the transformation is tied up in the IIi - inductive proofs of program - equivalence associated with 2.1, rather than in the structural complexity of changing call by value cuts into ordinary ones, since this only amounts to an implicit use of the exchange rule to swap the order of cut - formulas in a sequent ! However it is Theorem 2.1 that tells us this is indeed possible, and furthermore what the new exchanged cut formulas should be !

It should be clear by now that the form of the induction rule severely restricts the kinds of recursion that can be verified in the given logic. The simple form we have used so far, in which the induction step requires just one use of the premise *B(x)* to derive $B\{x+1)$, limits the corresponding forms of verifiable recursions to those in which only *one* recursive call is made. If we wish to verify a recursion with two recursive calls, then the linear - style logic requires an induction rule in which the premise *B(x)* of the induction step is explicitly written twice ! In this way the logic reflects the fine structural distinctions between various kinds of recursive programs. To illustrate, we consider some well known examples below.

4 Example : The Minimum Function.

Colson (1989) points out that the minimum function min(x,y) cannot be computed by an ordinary primitive recursive program in time O(min(x,y)). This is essentially because one of the variables would have to be chosen as the recursion variable, and the other one would then remain unchanged throughout the course of the recursion, so the number of computation steps - irrespective of the additional subsidiary functions needed to define it - would still be at least either *x* or y. He notes however that it can be computed in time O(min(x, y)) by a generalized primitive recursion, say on y, with the predecessor $x\dot{-}1$ substituted for the parameter x, thus

$$\min(x, 0) = 0$$
$$\min(x, y+1) = \textbf{if } x = \textbf{0 then 0 else } \min(x\dot{-}1, y) + 1$$

and he comments that this should really be regarded as a higher type "functional" form of recursion.

In our sense, *the efficiency is gained by virtue of a necessary increase in the quantifier complexity of the inductive termination proof, from* Ei *up to* Π_2.

Note also the use of the "cases" function here. But this can be verified easily by a degenerate form of our induction rule, in which the premise *B(x)* of the induction step is not used.

5 Example : Term Evaluation.

This is a classic example of nested recursion, requiring two calls on the induction hypothesis in its termination proof. We show how to transform it to a tail recursion, and again try to assess the incurred "cost".

Consider a system of applicative numerical terms (or straight - line programs) *t(x)* with one free variable x_y generated inductively from a collection of "basic" terms 6(x), including the identity term x, by the rules :

(i) each basic term is a term of height 0,

(ii) if t_0 and *ti* are terms of height h_0 and $/i_x$, then $t := (Aar.t_o)(<i)$ is a term of height max(^o, *hi)* + *l.*

Suppose we are given two term - decomposition functions / and r such that (i) if £ is a basic term then *l(t)* = *x* and *r(t)* = *t,* and (ii) if $t := (Arr.<_0)(<i)$ then *l(t)* = t_0 and *r(t)* = t_x. Suppose also we are given a function *v* which evaluates basic terms outright, i.e. if *t* is a basic term then for every natural number n, v(<, n) gives the value of term *t(x)* under assignment *x* := n.

Now let / be the term - evaluation function such that for arbitrary terms *t* of height $\leq$ z, /(z, *t,* n) gives the value of *t(x)* under assignment *x* := n. Then / is obviously definable by the following nested recursion over *z :*

$$
\begin{aligned}
/(\mathrm{O},*,\mathrm{n}) &= v(t,n) \\
f(z+1,t,n) &= f(z,l(t),f(z,r(t),n)) \ .
\end{aligned}
$$

Notice that the LPR derivation of the induction step in the termination proof for f begins with

$$l(t) \simeq y_0,\ r(t) \simeq y_1,\ f(z, y_1, x) \simeq y_2,\ f(z, y_0, y_2) \simeq y_3 \ \vdash\ f(z+1, t, x) \simeq y_3$$

and then by quantifier rules and ordinary cuts on the formulas $\exists y(l(t) \simeq y)$ and $\exists y(r(t) \simeq y)$ we obtain

$$\forall t\, \forall x\, \exists y\ (f(z,t,x) \simeq y), \forall t\, \forall x\, \exists y\ (f(z,t,x) \simeq y) \vdash \forall t\, \forall x\, \exists y\ (f(z+1,t,x) \simeq y)$$

Since LPR does not allow contraction, the only way in which we can now derive

$$\vdash\ \forall t\, \forall x\, \exists y\ (f(z,t,x) \simeq y)$$

is by an extended induction rule :

$$(IND_2)\ \frac{\vdash\ B(0) \qquad B(z)\ ,\ B(z) \vdash\ B(z+1)}{\vdash\ B(z)}$$

which explicitly allows two uses of the induction hypothesis.

The lesson is of course, that each new form of recursion must carry its own new form of induction in LPR. However we can still try to transform the recursion to, say, a tail recursion, and thereby bring the termination proof back into the original logic LPR($\forall\exists$). In the case of the term - evaluation function f this is an interesting and informative thing to do.

To obtain a primitive tail recursive definition of f we make use of the binary representation of numbers. Let $f(z,t)$ denote the function $\lambda n.f(z,t,n)$.

Define

$$g(0,t,n)\ :=\ n$$

and for each

$$a+1\ =\ 2^{z_0} + 2^{z_1} + \ldots + 2^{z_k}$$

where $z_0 > z_1 > \ldots > z_k$ define

$$g(a+1,t,n)\ :=\ f(z_0, l(t)) \circ f(z_1, l(r(t))) \circ \ldots \circ f(z_k, l(r^k(t)))\ (n)\ .$$

Then if $a+1$ is odd, $z_k = 0$, so $f(z_k, l(r^k(t)), n) = v(l(r^k(t)), n)$ and hence

$$g(a+1,t,n)\ =\ g(a,t,v(l(r^k(t)),n))\ .$$

On the other hand if $a + 1$ is even then $z/* > 0$. In this case we first do a little surgery inside the term t, replacing its subterm $r^k(t)$ by $l(r^k(t))$. Call the resulting term t^f. Then for each $t < k$ we have $/(r^{f"}(t)) = /(r^{f"}(f))$, whereas $l(r^k(t)) = r^k(t')$. Now by unravelling the nested recursive definition of / we see that

$$f(z_k, l(r^k(t)), n) = /(** - 1, /(r*(<0)) \circ /(«* - \% /(r^{*l}(*0)) \circ \ldots \ldots \circ f(0, l(r^{k+z_k-1}(t'))) (v(r*^{+z_k}(t'), n)$$

Combining this with the above definition of $g(a+1, t, n)$, and noting that if $a + 1$ is even then

$$a = 2^{*\circ} + 2^{Z1} + \ldots + 2^{2*"1} + 2^{Z*\sim 2} + \ldots + 2^{\circ}$$

we thus obtain

$$g(a+1, t, n) = g(a, t', v(r^{k+z_k}(t'), n)) .$$

Therefore p can be defined by a primitive tail recursion :

$$\begin{aligned} g(0,t,n) &= n \\ g(a+1,t,n) &= g(a, p(a,t), q(a,t,n)) \end{aligned}$$

where if a is even then $p(a,t) := t$ and $q(a,t,n) := v(Z(r*(<)),n)$, and if a is odd then $p(a,t) := f$ and $q(a,t,n) := v \wedge \wedge * \wedge) \wedge)$.

Furthermore the original evaluation function / can now be extracted from 9 by

$$/(*,*,n) = 0(2*, (A*.*)*, n).$$

Thus a doubly nested recursion over N can be transformed to a tail recursion over "2^N".

It is a quite general principle that nested recursions can be transformed to tail recursions, but at the cost of an "exponential increase" in the complexity of the well - ordering used. See Fairtlough and Wainer (1992).

6 Example : Ackermann - Péter Function.

This is a more complex nested recursion over the lexicographic ordering on N^2:

$$\begin{aligned} F(0,n) &= n+1 \\ F(m+1,0) &= F(m,1) \\ F(m+1,n+1) &= F(m,F(m+1,n)). \end{aligned}$$

The induction step in the LPR termination proof for F begins with

$$F(x+1,y)\sim z_0,\quad F(x,z_0)\sim z_1 \;\vdash\; F(x+1,y+1)\sim z_1$$

and by quantifier rules proceeds to

$$\exists z(F(x+1,y)\simeq z),\quad \forall y\exists z(F(x,y)\simeq z) \;\vdash\; \exists z(F(x+1,y+1)\simeq z).$$

At this point we can go no further without augmenting LPR with a still more complex induction rule, having an additional Π2 assumption C thus :

$$\frac{C,\ B(0),\quad B(y),C \;\vdash\; B(y+1)}{C \;\vdash\; B(y)}$$

With $C = \forall y\exists z(F(x,y)\simeq z)$ we can now derive

$$\forall y\exists z(F(x,y)\simeq z) \;\vdash\; \exists z(F(x+1,y)\simeq z)$$

and then

$$\forall y\exists z(F(x,y)\simeq z) \;\vdash\; \forall y\exists z(F(x+1,y)\simeq z)$$

and finally, by a further Π2 induction in the original form,

$$\vdash \forall y\exists z(F(x,y)\simeq z).$$

Again it is possible to transform the nested recursive definition over N^2 to a tail recursion. But now the complexity of the ordering goes up to N^N. See Fairtlough and Wainer for details.

REFERENCES.

(1) P. Anderson, *Program Derivation by Proof Transformation*, Carnegie Mellon Technical Report CS - 93 - 206, 1993.

(2) L. Colson, *About Primitive Recursive Algorithms*, in Proceedings ICALP '89, Springer Lecture Notes in Computer Science 372, 194 - 206.

(3) M.V. Fairtlough and S.S. Wainer, *Ordinal Complexity of Recursive Definitions*, Information and Computation Vol. 99, 1992, 123 - 153.

(4) S. Feferman, *Logics for Termination and Correctness of Functional Programs II, Logics of Strength PRA*, in "Proof Theory", Eds. P.Aczel, H. Simmons, S. Wainer; Cambridge 1992, 195 - 225.

(5) J.Y. Girard, *Linear Logic*, Theor. Comp. Science Vol.50, 1987, 1 - 102.

(6) C. Goad, *Computational Uses of the Manipulation of Formal Proofs*, Stanford Technical Report CS - 80 - 819, 1980.

(7) P. Madden, *Automatic Program Optimization through Proof Transformation*, Proc. 11th. Intl. Conf. on Automatic Deduction, Ed. D. Kapur, Springer-Verlag LNAI 607, 1992, 446 - 460.

(8) R. Péter, *Recursive Functions*, Academic Press 1967.

(9) F. Pfenning, *Program Development through Proof Transformation*, AMS Contemporary Mathematics Vol.106, 1990, 251 - 262.

(10) H. Schwichtenberg, *Proofs as Programs*, in "Proof Theory", Eds. P.Aczel, H. Simmons, S. Wainer; Cambridge 1992, 79 - 113.

(11) W. Sieg, *Herbrand Analyses*, Archive Math Logic Vol.30, 1991, 409 - 441.

(12) S.S. Wainer, *Four Lectures on Primitive Recursion*, in "Logic and Algebra of Specification", Eds. F. Bauer, W. Brauer, H. Schwichtenberg; Springer-Verlag ASI Series F Vol. 94, 1993, 377 - 410.