A computational experiment in Lestrade: programming the computation of Fibonacci numbers in Lestrade

Holmes

May 14, 2018

In this file so far I have developed a binary adder in which each rewrite rule is justified as a theorem. This is the first posted test of the `rewrited` command (which I had apparently actually broken in recent upgrades of rewriting: it now works).

```
Lestrade execution:

construct Nat type

>> Nat: type {move 0}


construct 0 in Nat

>> 0: in Nat {move 0}


construct 1 in Nat

>> 1: in Nat {move 0}


declare x in Nat

>> x: in Nat {move 1}


declare y in Nat

>> y: in Nat {move 1}
```

```
construct + x y in Nat

>> +: [(x_1:in Nat),(y_1:in Nat) => (---:in
>>        Nat)]
>>   {move 0}


construct * x y in Nat

>> *: [(x_1:in Nat),(y_1:in Nat) => (---:in
>>        Nat)]
>>   {move 0}


construct = x y prop

>> =: [(x_1:in Nat),(y_1:in Nat) => (---:prop)]
>>   {move 0}


construct Refl x that x=x

>> Refl: [(x_1:in Nat) => (---:that (x_1 = x_1))]
>>   {move 0}


declare pred [x=>prop]

>> pred: [(x_1:in Nat) => (---:prop)]
>>   {move 1}


declare eqev that x=y

>> eqev: that (x = y) {move 1}


declare predev that pred x

>> predev: that pred(x) {move 1}


construct Subs  pred, eqev, predev that pred y

>> Subs: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(eqev_1:
```

```
>>        that (.x_1 = .y_1)),(predev_1:that pred_1(.x_1))
>>        => (---:that pred_1(.y_1))]
>>   {move 0}
```

Declaring the natural numbers, the primitive constants 0 and 1, addition and multiplication, equality, and the rules of equality.

```
Lestrade execution:

define Eqsymm eqev : Subs [y=>y=x],eqev,Refl x

>> Eqsymm: [(.x_1:in Nat),(.y_1:in Nat),(eqev_1:
>>        that (.x_1 = .y_1)) => (Subs([(y_2:in
>>             Nat) => ((y_2 = .x_1):prop)]
>>        ,eqev_1,Refl(.x_1)):that (.y_1 = .x_1))]
>>   {move 0}


declare z in Nat

>> z: in Nat {move 1}


declare eqev2 that y=z

>> eqev2: that (y = z) {move 1}


define Eqtrans eqev eqev2 : Subs [y=>x=y],eqev2,eqev

>> Eqtrans: [(.x_1:in Nat),(.y_1:in Nat),(eqev_1:
>>        that (.x_1 = .y_1)),(.z_1:in Nat),(eqev2_1:
>>        that (.y_1 = .z_1)) => (Subs([(y_2:in
>>             Nat) => ((.x_1 = y_2):prop)]
>>        ,eqev2_1,eqev_1):that (.x_1 = .z_1))]
>>   {move 0}
```

Symmetry and transitivity proved. Note the usefulness of lambda-terms.

```
Lestrade execution:

construct Addcomm x y that (x+y) = y+x
```

```
>> Addcomm: [(x_1:in Nat),(y_1:in Nat) => (---:
>>        that ((x_1 + y_1) = (y_1 + x_1)))]
>>   {move 0}


construct Multcomm x y that (x*y) = y*x

>> Multcomm: [(x_1:in Nat),(y_1:in Nat) => (---:
>>        that ((x_1 * y_1) = (y_1 * x_1)))]
>>   {move 0}


construct Addid x  that (x+0)=x

>> Addid: [(x_1:in Nat) => (---:that ((x_1 +
>>        0) = x_1))]
>>   {move 0}


construct Multid x that (x*1) = x

>> Multid: [(x_1:in Nat) => (---:that ((x_1
>>        * 1) = x_1))]
>>   {move 0}


construct Assocadd x y z that ((x+y)+z) = x+y+z

>> Assocadd: [(x_1:in Nat),(y_1:in Nat),(z_1:
>>        in Nat) => (---:that (((x_1 + y_1) +
>>        z_1) = (x_1 + (y_1 + z_1))))]
>>   {move 0}


construct Assocmult x y z that ((x*y)*z) = x*y*z

>> Assocmult: [(x_1:in Nat),(y_1:in Nat),(z_1:
>>        in Nat) => (---:that (((x_1 * y_1) *
>>        z_1) = (x_1 * (y_1 * z_1))))]
>>   {move 0}


construct Dist x y z that (x*y+z)=(x*y)+x*z

>> Dist: [(x_1:in Nat),(y_1:in Nat),(z_1:in
>>        Nat) => (---:that ((x_1 * (y_1 + z_1))
```

```
>>        = ((x_1 * y_1) + (x_1 * z_1))))]
>>   {move 0}
```

Usual arithmetic axioms introduced.

```
Lestrade execution:

define 2 :1+1

>> 2: [((1 + 1):in Nat)]
>>   {move 0}


define bin x y : x+2*y

>> bin: [(x_1:in Nat),(y_1:in Nat) => ((x_1
>>        + (2 * y_1)):in Nat)]
>>   {move 0}
```

Declared 2 and the numeral building operation `bin`. Because of the preferred order of grouping in Lestrade, I am writing binary numerals backward.

```
Lestrade execution:

define zeropluszero : Addid 0

>> zeropluszero: [(Addid(0):that ((0 + 0) =
>>        0))]
>>   {move 0}


define onepluszero : Addid 1

>> onepluszero: [(Addid(1):that ((1 + 0) = 1))]
>>   {move 0}


define zeroplusone : Subs [x=>x=1],Addcomm 1 0,onepluszero

>> zeroplusone: [(Subs([(x_1:in Nat) => ((x_1
>>             = 1):prop)]
>>        ,(1 Addcomm 0),onepluszero):that ((0
>>        + 1) = 1))]
```

```
>>   {move 0}


define Formfix x y eqev : eqev

>> Formfix: [(x_1:in Nat),(y_1:in Nat),(eqev_1:
>>        that (x_1 = y_1)) => (eqev_1:that (x_1
>>        = y_1))]
>>   {move 0}


define oneplusone1: Formfix 1+1,2,Refl 2

>> oneplusone1: [(Formfix((1 + 1),2,Refl(2)):
>>        that ((1 + 1) = 2))]
>>   {move 0}


define twopluszero : Addid 2

>> twopluszero: [(Addid(2):that ((2 + 0) = 2))]
>>   {move 0}


define zeroplus2 : Subs [x=>x=2],Addcomm 2 0, twopluszero

>> zeroplus2: [(Subs([(x_1:in Nat) => ((x_1
>>             = 2):prop)]
>>        ,(2 Addcomm 0),twopluszero):that ((0
>>        + 2) = 2))]
>>   {move 0}


define zeroplus3 :Subs [x=>(0+x)=2],Eqsymm (Multid 2),zeroplus2

>> zeroplus3: [(Subs([(x_1:in Nat) => (((0 +
>>             x_1) = 2):prop)]
>>        ,Eqsymm(Multid(2)),zeroplus2):that ((0
>>        + (2 * 1)) = 2))]
>>   {move 0}


define defexpand : Formfix (0+2*1,0 bin 1,Refl (0+2*1))

>> defexpand: [(Formfix((0 + (2 * 1)),(0 bin
>>        1),Refl((0 + (2 * 1)))):that ((0 + (2
```

```
>>        * 1)) = (0 bin 1)))]
>>   {move 0}


define oneplusone2: Subs [x=>x=2],defexpand,zeroplus3

>> oneplusone2: [(Subs([(x_1:in Nat) => ((x_1
>>             = 2):prop)]
>>        ,defexpand,zeroplus3):that ((0 bin 1)
>>        = 2))]
>>   {move 0}


define oneplusone : Eqtrans (oneplusone1,Eqsymm oneplusone2)

>> oneplusone: [((oneplusone1 Eqtrans Eqsymm(oneplusone2)):
>>        that ((1 + 1) = (0 bin 1)))]
>>   {move 0}


declare input1 that pred(1+0)

>> input1: that pred((1 + 0)) {move 1}


define Onepluszero input1: Subs pred,onepluszero,input1

>> Onepluszero: [(.pred_1:[(x_2:in Nat) => (---:
>>             prop)]),
>>        (input1_1:that .pred_1((1 + 0))) =>
>>        (Subs(.pred_1,onepluszero,input1_1):
>>        that .pred_1(1))]
>>   {move 0}


rewrited Onepluszero 1+0,1

>> Onepluszero''': [(Onepluszero''''_1:in Nat)
>>        => (---:prop)]
>>   {move 1}



>> Onepluszero'': that Onepluszero'''((1 + 0))
>>   {move 1}
```

```
>> Onepluszero': [(Onepluszero'''_1:[(Onepluszero''''_2:
>>             in Nat) => (---:prop)]),
>>        (Onepluszero''_1:that Onepluszero'''_1((1
>>        + 0))) => (---:that Onepluszero'''_1(1))]
>>   {move 0}



>> Inspector Lestrade says:  Rewrite demonstration succeeded

declare input2 that pred(0+0)

>> input2: that pred((0 + 0)) {move 1}


define Zeropluszero input2: Subs pred,zeropluszero,input2

>> Zeropluszero: [(.pred_1:[(x_2:in Nat) =>
>>             (---:prop)]),
>>        (input2_1:that .pred_1((0 + 0))) =>
>>        (Subs(.pred_1,zeropluszero,input2_1):
>>        that .pred_1(0))]
>>   {move 0}


declare input3 that pred(0+1)

>> input3: that pred((0 + 1)) {move 1}


define Zeroplusone input3: Subs pred,zeroplusone,input3

>> Zeroplusone: [(.pred_1:[(x_2:in Nat) => (---:
>>             prop)]),
>>        (input3_1:that .pred_1((0 + 1))) =>
>>        (Subs(.pred_1,zeroplusone,input3_1):
>>        that .pred_1(1))]
>>   {move 0}


declare input4 that pred(1+1)

>> input4: that pred((1 + 1)) {move 1}
```

```
define Oneplusone input4: Subs pred,oneplusone,input4

>> Oneplusone: [(.pred_1:[(x_2:in Nat) => (---:
>>             prop)]),
>>        (input4_1:that .pred_1((1 + 1))) =>
>>        (Subs(.pred_1,oneplusone,input4_1):that
>>        .pred_1((0 bin 1)))]
>>   {move 0}


rewrited Zeropluszero 0+0,0

>> Zeropluszero''': [(Zeropluszero''''_1:in
>>        Nat) => (---:prop)]
>>   {move 1}



>> Zeropluszero'': that Zeropluszero'''((0 +
>>   0)) {move 1}



>> Zeropluszero': [(Zeropluszero'''_1:[(Zeropluszero''''_2:
>>             in Nat) => (---:prop)]),
>>        (Zeropluszero''_1:that Zeropluszero'''_1((0
>>        + 0))) => (---:that Zeropluszero'''_1(0))]
>>   {move 0}



>> Inspector Lestrade says:  Rewrite demonstration succeeded

rewrited Zeroplusone 0+1,1

>> Zeroplusone''': [(Zeroplusone''''_1:in Nat)
>>        => (---:prop)]
>>   {move 1}



>> Zeroplusone'': that Zeroplusone'''((0 + 1))
>>   {move 1}
```

```
>> Zeroplusone': [(Zeroplusone'''_1:[(Zeroplusone''''_2:
>>             in Nat) => (---:prop)]),
>>        (Zeroplusone''_1:that Zeroplusone'''_1((0
>>        + 1))) => (---:that Zeroplusone'''_1(1))]
>>   {move 0}



>> Inspector Lestrade says:  Rewrite demonstration succeeded

rewrited Oneplusone 1+1,0 bin 1

>> Oneplusone''': [(Oneplusone''''_1:in Nat)
>>        => (---:prop)]
>>   {move 1}



>> Oneplusone'': that Oneplusone'''((1 + 1))
>>   {move 1}



>> Oneplusone': [(Oneplusone'''_1:[(Oneplusone''''_2:
>>             in Nat) => (---:prop)]),
>>        (Oneplusone''_1:that Oneplusone'''_1((1
>>        + 1))) => (---:that Oneplusone'''_1((0
>>        bin 1)))]
>>   {move 0}



>> Inspector Lestrade says:  Rewrite demonstration succeeded

% addition table for digits finished
```

The four rewrite rules which constitute the addition table are presented.

```
Lestrade execution:

% (a bin b) + 0 -> (a bin b)

declare input5 that pred((x bin y)+0)
```

```
>> input5: that pred(((x bin y) + 0)) {move
>>   1}


define Rule1 pred, input5: Subs pred,Addid(x bin y),input5

>> Rule1: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(input5_1:
>>        that pred_1(((.x_1 bin .y_1) + 0)))
>>        => (Subs(pred_1,Addid((.x_1 bin .y_1)),
>>        input5_1):that pred_1((.x_1 bin .y_1)))]
>>   {move 0}


rewrited Rule1 (x bin y)+0, x bin y

>> Rule1''': [(Rule1''''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Rule1'': that Rule1'''(((x bin y) + 0)) {move
>>   1}



>> Rule1': [(Rule1'''_1:[(Rule1''''_2:in Nat)
>>             => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(Rule1''_1:
>>        that Rule1'''_1(((.x_1 bin .y_1) + 0)))
>>        => (---:that Rule1'''_1((.x_1 bin .y_1)))]
>>   {move 0}



>> Inspector Lestrade says:  Rewrite demonstration succeeded

% (a bin b) + 1 -> 1+a bin b

declare input6 that pred((x bin y)+1)

>> input6: that pred(((x bin y) + 1)) {move
>>   1}


define Rule2 pred, input6:  Subs pred,Addcomm x bin y, 1,input6
```

```
>> Rule2: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(input6_1:
>>        that pred_1(((.x_1 bin .y_1) + 1)))
>>        => (Subs(pred_1,((.x_1 bin .y_1) Addcomm
>>        1),input6_1):that pred_1((1 + (.x_1
>>        bin .y_1))))]
>>   {move 0}


rewrited Rule2 (x bin y)+1,1+(x bin y)

>> Rule2''': [(Rule2''''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Rule2'': that Rule2'''(((x bin y) + 1)) {move
>>   1}



>> Rule2': [(Rule2'''_1:[(Rule2''''_2:in Nat)
>>             => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(Rule2''_1:
>>        that Rule2'''_1(((.x_1 bin .y_1) + 1)))
>>        => (---:that Rule2'''_1((1 + (.x_1 bin
>>        .y_1))))]
>>   {move 0}



>> Inspector Lestrade says:  Rewrite demonstration succeeded

% 0 + a bin b -> a bin b

declare input7 that pred(0+(x bin y))

>> input7: that pred((0 + (x bin y))) {move
>>   1}


define Rulea3  pred, input7: Subs pred,Addcomm 0, x bin y,input7

>> Rulea3: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(input7_1:
```

```
>>        that pred_1((0 + (.x_1 bin .y_1))))
>>        => (Subs(pred_1,(0 Addcomm (.x_1 bin
>>        .y_1)),input7_1):that pred_1(((.x_1
>>        bin .y_1) + 0)))]
>>   {move 0}


define Rule3 pred, input7: Subs pred,Addid (x bin y), Rulea3 pred, input7

>> Rule3: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(input7_1:
>>        that pred_1((0 + (.x_1 bin .y_1))))
>>        => (Subs(pred_1,Addid((.x_1 bin .y_1)),
>>        Rulea3(pred_1,input7_1)):that pred_1((.x_1
>>        bin .y_1)))]
>>   {move 0}


rewrited Rule3 0+x bin y,x bin y

>> Rule3''': [(Rule3''''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Rule3'': that Rule3'''((0 + (x bin y))) {move
>>   1}



>> Rule3': [(Rule3'''_1:[(Rule3''''_2:in Nat)
>>             => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(Rule3''_1:
>>        that Rule3'''_1((0 + (.x_1 bin .y_1))))
>>        => (---:that Rule3'''_1((.x_1 bin .y_1)))]
>>   {move 0}



>> Inspector Lestrade says:  Rewrite demonstration succeeded

% 1 + 0 bin a -> 1 bin a

declare input8 that pred(1+ 0 bin x)

>> input8: that pred((1 + (0 bin x))) {move
```

```
>>   1}


define Rule4 pred,input8 : Subs pred,Eqsymm(Assocadd(1,0,2*x)),input8

>> Rule4: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(input8_1:that pred_1((1
>>        + (0 bin .x_1)))) => (Subs(pred_1,Eqsymm(Assocadd(1,
>>        0,(2 * .x_1))),input8_1):that pred_1(((1
>>        + 0) + (2 * .x_1))))]
>>   {move 0}


rewrited Rule4 1+0 bin x,1 bin x

>> Rule4''': [(Rule4''''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Rule4'': that Rule4'''((1 + (0 bin x))) {move
>>   1}



>> Rule4': [(Rule4'''_1:[(Rule4''''_2:in Nat)
>>             => (---:prop)]),
>>        (.x_1:in Nat),(Rule4''_1:that Rule4'''_1((1
>>        + (0 bin .x_1)))) => (---:that Rule4'''_1((1
>>        bin .x_1)))]
>>   {move 0}



>> Inspector Lestrade says:  Rewrite demonstration succeeded

% 1 + 1 bin a  -> 0 bin 1+a

declare input9 that pred(1+1 bin x)

>> input9: that pred((1 + (1 bin x))) {move
>>   1}


define Rulea5 pred,input9 :Subs pred,Eqsymm(Assocadd(1,1,2*x)),input9
```

```
>> Rulea5: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(input9_1:that pred_1((1
>>        + (1 bin .x_1)))) => (Subs(pred_1,Eqsymm(Assocadd(1,
>>        1,(2 * .x_1))),input9_1):that pred_1(((1
>>        + 1) + (2 * .x_1))))]
>>   {move 0}


define Ruleb5 pred,input9: Subs pred,Assocadd(0,2*1,2*x),Rulea5(pred,input9)

>> Ruleb5: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(input9_1:that pred_1((1
>>        + (1 bin .x_1)))) => (Subs(pred_1,Assocadd(0,
>>        (2 * 1),(2 * .x_1)),Rulea5(pred_1,input9_1)):
>>        that pred_1((0 + ((2 * 1) + (2 * .x_1)))))]
>>   {move 0}


define Rulec5 pred,input9 : Subs pred,Addcomm(0,(2*1)+2*x),Ruleb5 pred,input9

>> Rulec5: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(input9_1:that pred_1((1
>>        + (1 bin .x_1)))) => (Subs(pred_1,(0
>>        Addcomm ((2 * 1) + (2 * .x_1))),Ruleb5(pred_1,
>>        input9_1)):that pred_1((((2 * 1) + (2
>>        * .x_1)) + 0)))]
>>   {move 0}


define Ruled5 pred,input9 : Subs pred,Addid((2*1)+2*x),Rulec5(pred,input9)

>> Ruled5: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(input9_1:that pred_1((1
>>        + (1 bin .x_1)))) => (Subs(pred_1,Addid(((2
>>        * 1) + (2 * .x_1))),Rulec5(pred_1,input9_1)):
>>        that pred_1(((2 * 1) + (2 * .x_1))))]
>>   {move 0}


define Rulee5 pred,input9 : Subs pred,Eqsymm(Dist(2,1,x)),Ruled5 pred,input9

>> Rulee5: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(input9_1:that pred_1((1
>>        + (1 bin .x_1)))) => (Subs(pred_1,Eqsymm(Dist(2,
>>        1,.x_1)),Ruled5(pred_1,input9_1)):that
>>        pred_1((2 * (1 + .x_1))))]
```

```
>>   {move 0}


define Rulef5 pred,input9 : Subs pred, Eqsymm(Addid(2*(1+x))),Rulee5 pred,input9

>> Rulef5: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(input9_1:that pred_1((1
>>        + (1 bin .x_1)))) => (Subs(pred_1,Eqsymm(Addid((2
>>        * (1 + .x_1)))),Rulee5(pred_1,input9_1)):
>>        that pred_1(((2 * (1 + .x_1)) + 0)))]
>>   {move 0}


define Rule5 pred,input9 : Subs pred,Addcomm(2*(1+x),0),Rulef5 pred,input9

>> Rule5: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(input9_1:that pred_1((1
>>        + (1 bin .x_1)))) => (Subs(pred_1,((2
>>        * (1 + .x_1)) Addcomm 0),Rulef5(pred_1,
>>        input9_1)):that pred_1((0 + (2 * (1
>>        + .x_1)))))]
>>   {move 0}


rewrited Rule5 1+1 bin x,0 bin 1+x

>> Rule5''': [(Rule5''''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Rule5'': that Rule5'''((1 + (1 bin x))) {move
>>   1}



>> Rule5': [(Rule5'''_1:[(Rule5''''_2:in Nat)
>>             => (---:prop)]),
>>        (.x_1:in Nat),(Rule5''_1:that Rule5'''_1((1
>>        + (1 bin .x_1)))) => (---:that Rule5'''_1((0
>>        bin (1 + .x_1))))]
>>   {move 0}



>> Inspector Lestrade says:  Rewrite demonstration succeeded
```

```
% ((a bin b) + c bin d) -> a + c + 0 bin b+d

declare w in Nat

>> w: in Nat {move 1}


declare input10 that pred((x bin y) + z bin w)

>> input10: that pred(((x bin y) + (z bin w)))
>>   {move 1}


define Rulea6 pred, input10:  Subs pred, Assocadd(x,2*y,z bin w), input10

>> Rulea6: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(.z_1:in
>>        Nat),(.w_1:in Nat),(input10_1:that pred_1(((.x_1
>>        bin .y_1) + (.z_1 bin .w_1)))) => (Subs(pred_1,
>>        Assocadd(.x_1,(2 * .y_1),(.z_1 bin .w_1)),
>>        input10_1):that pred_1((.x_1 + ((2 *
>>        .y_1) + (.z_1 bin .w_1)))))]
>>   {move 0}


define Ruleb6 pred,input10: \
Subs [w=>pred(x+w)],Eqsymm(Assocadd(2*y,z,2*w)),Rulea6 pred,input10

>> Ruleb6: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(.z_1:in
>>        Nat),(.w_1:in Nat),(input10_1:that pred_1(((.x_1
>>        bin .y_1) + (.z_1 bin .w_1)))) => (Subs([(w_3:
>>             in Nat) => (pred_1((.x_1 + w_3)):
>>             prop)]
>>        ,Eqsymm(Assocadd((2 * .y_1),.z_1,(2
>>        * .w_1))),Rulea6(pred_1,input10_1)):
>>        that pred_1((.x_1 + (((2 * .y_1) + .z_1)
>>        + (2 * .w_1)))))]
>>   {move 0}


define Rulec6 pred,input10:\
  Subs pred,Eqsymm(Assocadd(x,(2*y)+z,2*w)),Ruleb6 pred,input10

>> Rulec6: [(pred_1:[(x_2:in Nat) => (---:prop)]),
```

```
>>        (.x_1:in Nat),(.y_1:in Nat),(.z_1:in
>>        Nat),(.w_1:in Nat),(input10_1:that pred_1(((.x_1
>>        bin .y_1) + (.z_1 bin .w_1)))) => (Subs(pred_1,
>>        Eqsymm(Assocadd(.x_1,((2 * .y_1) + .z_1),
>>        (2 * .w_1))),Ruleb6(pred_1,input10_1)):
>>        that pred_1(((.x_1 + ((2 * .y_1) + .z_1))
>>        + (2 * .w_1))))]
>>   {move 0}


define Ruled6 pred,input10:\
 Subs [z=>pred((x+z)+2*w)],Addcomm(2*y,z),Rulec6 pred,input10

>> Ruled6: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(.z_1:in
>>        Nat),(.w_1:in Nat),(input10_1:that pred_1(((.x_1
>>        bin .y_1) + (.z_1 bin .w_1)))) => (Subs([(z_3:
>>             in Nat) => (pred_1(((.x_1 + z_3)
>>             + (2 * .w_1))):prop)]
>>        ,((2 * .y_1) Addcomm .z_1),Rulec6(pred_1,
>>        input10_1)):that pred_1(((.x_1 + (.z_1
>>        + (2 * .y_1))) + (2 * .w_1))))]
>>   {move 0}


define Rulee6 pred,input10 : Subs pred,Assocadd(x,z+2*y,2*w),Ruled6 pred,input10

>> Rulee6: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(.z_1:in
>>        Nat),(.w_1:in Nat),(input10_1:that pred_1(((.x_1
>>        bin .y_1) + (.z_1 bin .w_1)))) => (Subs(pred_1,
>>        Assocadd(.x_1,(.z_1 + (2 * .y_1)),(2
>>        * .w_1)),Ruled6(pred_1,input10_1)):that
>>        pred_1((.x_1 + ((.z_1 + (2 * .y_1))
>>        + (2 * .w_1)))))]
>>   {move 0}


define Rulef6 pred,input10 :\
 Subs [y=>pred(x+y)],Assocadd(z,2*y,2*w),Rulee6 pred,input10

>> Rulef6: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(.z_1:in
>>        Nat),(.w_1:in Nat),(input10_1:that pred_1(((.x_1
>>        bin .y_1) + (.z_1 bin .w_1)))) => (Subs([(y_3:
>>             in Nat) => (pred_1((.x_1 + y_3)):
```

```
>>             prop)]
>>        ,Assocadd(.z_1,(2 * .y_1),(2 * .w_1)),
>>        Rulee6(pred_1,input10_1)):that pred_1((.x_1
>>        + (.z_1 + ((2 * .y_1) + (2 * .w_1))))))]
>>   {move 0}


define Ruleg6 pred,input10 : \
 Subs [w=>pred(x+z+w)],Eqsymm(Dist(2,y,w)),Rulef6 pred,input10

>> Ruleg6: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(.z_1:in
>>        Nat),(.w_1:in Nat),(input10_1:that pred_1(((.x_1
>>        bin .y_1) + (.z_1 bin .w_1)))) => (Subs([(w_3:
>>             in Nat) => (pred_1((.x_1 + (.z_1
>>             + w_3))):prop)]
>>        ,Eqsymm(Dist(2,.y_1,.w_1)),Rulef6(pred_1,
>>        input10_1)):that pred_1((.x_1 + (.z_1
>>        + (2 * (.y_1 + .w_1))))))]
>>   {move 0}


define Ruleh6 pred,input10 : \
Subs [w=>pred(x+z+w)],Eqsymm(Addid(2*(y+w))),Ruleg6 pred,input10

>> Ruleh6: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(.z_1:in
>>        Nat),(.w_1:in Nat),(input10_1:that pred_1(((.x_1
>>        bin .y_1) + (.z_1 bin .w_1)))) => (Subs([(w_3:
>>             in Nat) => (pred_1((.x_1 + (.z_1
>>             + w_3))):prop)]
>>        ,Eqsymm(Addid((2 * (.y_1 + .w_1)))),
>>        Ruleg6(pred_1,input10_1)):that pred_1((.x_1
>>        + (.z_1 + ((2 * (.y_1 + .w_1)) + 0)))))]
>>   {move 0}


define Rule6 pred,input10 : \
Subs [w=>pred(x+z+w)],(Addcomm(2*y+w,0)),Ruleh6 pred,input10

>> Rule6: [(pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(.z_1:in
>>        Nat),(.w_1:in Nat),(input10_1:that pred_1(((.x_1
>>        bin .y_1) + (.z_1 bin .w_1)))) => (Subs([(w_3:
>>             in Nat) => (pred_1((.x_1 + (.z_1
>>             + w_3))):prop)]
```

```
>>        ,((2 * (.y_1 + .w_1)) Addcomm 0),Ruleh6(pred_1,
>>        input10_1)):that pred_1((.x_1 + (.z_1
>>        + (0 + (2 * (.y_1 + .w_1)))))))]
>>   {move 0}


rewrited Rule6 (x bin y) + z bin w,x+z+ 0 bin y+w

>> Rule6''': [(Rule6''''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Rule6'': that Rule6'''(((x bin y) + (z bin
>>   w))) {move 1}



>> Rule6': [(Rule6'''_1:[(Rule6''''_2:in Nat)
>>             => (---:prop)]),
>>        (.x_1:in Nat),(.y_1:in Nat),(.z_1:in
>>        Nat),(.w_1:in Nat),(Rule6''_1:that Rule6'''_1(((.x_1
>>        bin .y_1) + (.z_1 bin .w_1)))) => (---:
>>        that Rule6'''_1((.x_1 + (.z_1 + (0 bin
>>        (.y_1 + .w_1))))))]
>>   {move 0}



>> Inspector Lestrade says:  Rewrite demonstration succeeded
```

The six rewrite rules which handle multidigit calculations are presented. I do not have a rule which drops leading zeroes yet, though it could easily be introduced. At this point, if one defines an identifier as an arbitrary sum of binary notations, it will evaluate the sum as a single binary notation, and all the evaluations are justified by theorems of this Lestrade theory.

It is official: Lestrade with rewriting is a programming language.

```
Lestrade execution:

define test :  (0 bin 1 bin 1) + (1 bin 1 bin 0 bin 1) + 1 bin 0 bin 1

>> test: [((0 bin (1 bin (1 bin (0 bin 1)))):
>>        in Nat)]
>>   {move 0}
```

An example.

We now implement the predecessor function and the Fibonacci sequence. For this, we will cheat. We implement the rewrite rules as axioms. A more principled justification using a rule for constructing recursive functions is probably desirable, but this is a programming test.

```
Lestrade execution:

construct P x in Nat

>> P: [(x_1:in Nat) => (---:in Nat)]
>>   {move 0}


rewritec Pred1 P 1, 0

>> Pred1'': [(Pred1'''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Pred1': that Pred1''(P(1)) {move 1}



>> Pred1: [(Pred1''_1:[(Pred1'''_2:in Nat) =>
>>             (---:prop)]),
>>        (Pred1'_1:that Pred1''_1(P(1))) => (---:
>>        that Pred1''_1(0))]
>>   {move 0}


rewritec Pred2 P (1 bin x), 0 bin x

>> Pred2'': [(Pred2'''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Pred2': that Pred2''(P((1 bin x))) {move
>>   1}
```

```
>> Pred2: [(Pred2''_1:[(Pred2'''_2:in Nat) =>
>>             (---:prop)]),
>>        (.x_1:in Nat),(Pred2'_1:that Pred2''_1(P((1
>>        bin .x_1)))) => (---:that Pred2''_1((0
>>        bin .x_1)))]
>>   {move 0}


rewritec Pred3  P (0 bin x), 1 bin P x

>> Pred3'': [(Pred3'''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Pred3': that Pred3''(P((0 bin x))) {move
>>   1}



>> Pred3: [(Pred3''_1:[(Pred3'''_2:in Nat) =>
>>             (---:prop)]),
>>        (.x_1:in Nat),(Pred3'_1:that Pred3''_1(P((0
>>        bin .x_1)))) => (---:that Pred3''_1((1
>>        bin P(.x_1))))]
>>   {move 0}


rewritec Pred4 P(0 bin 1), 1

>> Pred4'': [(Pred4'''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Pred4': that Pred4''(P((0 bin 1))) {move
>>   1}



>> Pred4: [(Pred4''_1:[(Pred4'''_2:in Nat) =>
>>             (---:prop)]),
>>        (Pred4'_1:that Pred4''_1(P((0 bin 1))))
>>        => (---:that Pred4''_1(1))]
>>   {move 0}
```

The rules for predecessor.

```
Lestrade execution:

construct F x in Nat

>> F: [(x_1:in Nat) => (---:in Nat)]
>>   {move 0}


rewritec Fib1 x, F x, (F(P x))+F(P(P x))

>> Fib1'': [(Fib1'''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Fib1': that Fib1''(F(x)) {move 1}



>> Fib1: [(x_1:in Nat),(Fib1''_1:[(Fib1'''_2:
>>             in Nat) => (---:prop)]),
>>        (Fib1'_1:that Fib1''_1(F(x_1))) => (---:
>>        that Fib1''_1((F(P(x_1)) + F(P(P(x_1))))))]
>>   {move 0}


rewritec Fib2 F 0, 1

>> Fib2'': [(Fib2'''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Fib2': that Fib2''(F(0)) {move 1}



>> Fib2: [(Fib2''_1:[(Fib2'''_2:in Nat) => (---:
>>             prop)]),
>>        (Fib2'_1:that Fib2''_1(F(0))) => (---:
>>        that Fib2''_1(1))]
```

```
>>   {move 0}


rewritec Fib3 F 1,  1

>> Fib3'': [(Fib3'''_1:in Nat) => (---:prop)]
>>   {move 1}



>> Fib3': that Fib3''(F(1)) {move 1}



>> Fib3: [(Fib3''_1:[(Fib3'''_2:in Nat) => (---:
>>             prop)]),
>>        (Fib3'_1:that Fib3''_1(F(1))) => (---:
>>        that Fib3''_1(1))]
>>   {move 0}
```

The recurrence relations for the Fibonacci numbers. It is important that the reductions for F(0) and F(1) appear after the others: this means they are applied first.

```
Lestrade execution:

define ftest5 : F(1 bin 0 bin 1)

>> ftest5: [((0 bin (0 bin (0 bin 1))):in Nat)]
>>   {move 0}


define ftest20 : F(0 bin 0 bin 1 bin 0 bin 1)

>> ftest20: [((0 bin (1 bin (0 bin (0 bin (0
>>        bin (0 bin (1 bin (1 bin (0 bin (1 bin
>>        (0 bin (1 bin (0 bin 1)))))))))))):
>>        in Nat)]
>>   {move 0}


define ftest64 : F(0 bin 0 bin 0 bin 0 bin 0 bin 0 bin 1)

>> ftest64: [((1 bin (0 bin (1 bin (1 bin (1
```

```
>>        bin (0 bin (0 bin (1 bin (1 bin (0 bin
>>        (1 bin (0 bin (0 bin (0 bin (0 bin (1
>>        bin (0 bin (1 bin (0 bin (1 bin (1 bin
>>        (1 bin (1 bin (0 bin (1 bin (0 bin (0
>>        bin (1 bin (0 bin (1 bin (0 bin (0 bin
>>        (1 bin (0 bin (1 bin (1 bin (1 bin (0
>>        bin (0 bin (1 bin (1 bin (1 bin (1 bin
>>        1))))))))))))))))))))))))))))))))))))))))))):
>>        in Nat)]
>>   {move 0}
```

```
define ftest128 :  F(0 bin 0 bin 0 bin 0 bin 0 bin 0 bin 0 bin 1)
```

```
>> ftest128: [((0 bin (1 bin (0 bin (0 bin (0
>>        bin (1 bin (1 bin (1 bin (0 bin (1 bin
>>        (0 bin (1 bin (0 bin (0 bin (1 bin (1
>>        bin (1 bin (1 bin (0 bin (1 bin (0 bin
>>        (1 bin (0 bin (1 bin (0 bin (1 bin (0
>>        bin (1 bin (0 bin (0 bin (1 bin (0 bin
>>        (1 bin (1 bin (0 bin (1 bin (0 bin (1
>>        bin (1 bin (1 bin (0 bin (1 bin (1 bin
>>        (0 bin (0 bin (1 bin (0 bin (0 bin (1
>>        bin (1 bin (0 bin (0 bin (1 bin (1 bin
>>        (1 bin (0 bin (0 bin (0 bin (0 bin (0
>>        bin (1 bin (1 bin (1 bin (0 bin (1 bin
>>        (1 bin (0 bin (1 bin (1 bin (0 bin (1
>>        bin (0 bin (0 bin (1 bin (0 bin (1 bin
>>        (0 bin (1 bin (1 bin (1 bin (0 bin (0
>>        bin (0 bin (0 bin (1 bin (0 bin (1 bin
>>        (0 bin 1))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
>>        in Nat)]
>>   {move 0}
```