Primitive recursive arithmetic in Lestrade

Randall Holmes

starting out, 4/5/2019

In this file, I am going to implement PRA and discuss why I am really just implementing PRA. There is a background question of just how strong the Lestrade background logic is; my reply in principle is "vanishingly weak", but I have to justify that.

```
Lestrade execution:

construct Nat type

>> Nat: type {move 0}


% the type of natural numbers

declare m in Nat

>> m: in Nat {move 1}


declare n in Nat

>> n: in Nat {move 1}


construct =  m n prop

>> =: [(m_1:in Nat),(n_1:in Nat) => (---:prop)]
```

```
>>   {move 0}


construct 0 in Nat

>> 0: in Nat {move 0}
```

Above find the natural number type with its primitive constant 0 and the equality relation, the only constructor for atomic propositions. The successor operation is introduced as a primitive recursive function constructor below.

```
Lestrade execution:

% basic propositional logic

declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}


construct & p q prop

>> &: [(p_1:prop),(q_1:prop) => (---:prop)]
>>   {move 0}


construct ~ p prop

>> ~: [(p_1:prop) => (---:prop)]
>>   {move 0}
```

```
declare pp that p

>> pp: that p {move 1}


declare qq that q

>> qq: that q {move 1}


declare rr that p & q

>> rr: that (p & q) {move 1}


construct Conj pp qq that p & q

>> Conj: [(.p_1:prop),(pp_1:that .p_1),(.q_1:
>>        prop),(qq_1:that .q_1) => (---:that
>>        (.p_1 & .q_1))]
>>   {move 0}


construct Simp1 rr that p

>> Simp1: [(.p_1:prop),(.q_1:prop),(rr_1:that
>>        (.p_1 & .q_1)) => (---:that .p_1)]
>>   {move 0}


construct Simp2 rr that q

>> Simp2: [(.p_1:prop),(.q_1:prop),(rr_1:that
>>        (.p_1 & .q_1)) => (---:that .q_1)]
>>   {move 0}
```

```
declare negp that ~ p

>> negp: that ~(p) {move 1}


declare q2 prop

>> q2: prop {move 1}


construct Contra negp q2 that q2

>> Contra: [(.p_1:prop),(negp_1:that ~(.p_1)),
>>        (q2_1:prop) => (---:that q2_1)]
>>   {move 0}


declare proveall [pp => that q]

>> proveall: [(pp_1:that p) => (---:that q)]
>>   {move 1}


construct Negintro proveall that ~ p

>> Negintro: [(.p_1:prop),(.q_1:prop),(proveall_1:
>>        [(pp_2:that .p_1) => (---:that .q_1)])
>>        => (---:that ~(.p_1))]
>>   {move 0}


declare maybe that ~ ~ p

>> maybe: that ~(~(p)) {move 1}


construct Dneg maybe that p
```

```
>> Dneg: [(.p_1:prop),(maybe_1:that ~(~(.p_1)))
>>        => (---:that .p_1)]
>>   {move 0}
```

Above find primitives of propositional logic based on conjunction and negation. The logic is classical. Is there any actual difference between constructive and classical PRA? Note that equality, conjunction and negation are the only proposition constructors.

```
Lestrade execution:

% properties of equality

construct  Refleq m that m=m

>> Refleq: [(m_1:in Nat) => (---:that (m_1 =
>>        m_1))]
>>   {move 0}


declare pred [m => prop]

>> pred: [(m_1:in Nat) => (---:prop)]
>>   {move 1}


declare misn that m=n

>> misn: that (m = n) {move 1}


declare predm that pred m

>> predm: that pred(m) {move 1}
```

```
construct Subs pred, misn predm that pred n

>> Subs: [(pred_1:[(m_2:in Nat) => (---:prop)]),
>>        (.m_1:in Nat),(.n_1:in Nat),(misn_1:
>>        that (.m_1 = .n_1)),(predm_1:that pred_1(.m_1))
>>        => (---:that pred_1(.n_1))]
>>   {move 0}
```

Here we have the usual primitive constructions for the logic of equality.

```
Lestrade execution:

% functions

construct function type

>> function: type {move 0}


construct operation type

>> operation: type {move 0}


declare f in function

>> f: in function {move 1}


declare g in function

>> g: in function {move 1}


declare o in operation
```

```
>> o: in operation {move 1}


declare m2 in Nat

>> m2: in Nat {move 1}


declare n2 in Nat

>> n2: in Nat {move 1}


% we support only unary and binary functions.  This should be enough.

% to justify use of unary and binary functions, we need enough to bootstrap

% implementation of the pair.

construct App1 f m2 in Nat

>> App1: [(f_1:in function),(m2_1:in Nat) =>
>>        (---:in Nat)]
>>   {move 0}


construct App2 o m2 n2 in Nat

>> App2: [(o_1:in operation),(m2_1:in Nat),(n2_1:
>>        in Nat) => (---:in Nat)]
>>   {move 0}


construct Iter1  f  n2 in function

>> Iter1: [(f_1:in function),(n2_1:in Nat) =>
>>        (---:in function)]
>>   {move 0}
```

```
% an internal successor function and a defined Lestrade successor function

construct S0 in function

>> S0: in function {move 0}


define S n2 : App1 S0 n2

>> S: [(n2_1:in Nat) => ((S0 App1 n2_1):in Nat)]
>>   {move 0}
```

We introduce a type of functions (primitive recursive functions of one argument) and operations (primitive recursive functions of two arguments). We cannot handle functions of arbitrary arity unless we introduce an argument list type, and we should be able to show that this is the same as the usual logic by defining a pair and its projections. We provide the operations for applying a function to a single argument and an operation to two arguments, and we provide the successor function, as a function in the internal sense and as a defined Lestrade function.

```
Lestrade execution:

% constructors for unary functions:  iteration and composition.

% that iteration is enough requires proof, again, using pairs.

declare  x in Nat

>> x: in Nat {move 1}


construct Iteraxiom11 f  n2 that App1(Iter1 f  n2,0) = n2
```

```
>> Iteraxiom11: [(f_1:in function),(n2_1:in
>>        Nat) => (---:that (((f_1 Iter1 n2_1)
>>        App1 0) = n2_1))]
>>   {move 0}
```

```
construct Iteraxiom12  f   n2  x  that App1(Iter1 f  n2,S x) = App1(f,App1(Iter
```

```
>> Iteraxiom12: [(f_1:in function),(n2_1:in
>>        Nat),(x_1:in Nat) => (---:that (((f_1
>>        Iter1 n2_1) App1 S(x_1)) = (f_1 App1
>>        ((f_1 Iter1 n2_1) App1 x_1))))]
>>   {move 0}
```

```
construct Comp1 f g in function
```

```
>> Comp1: [(f_1:in function),(g_1:in function)
>>        => (---:in function)]
>>   {move 0}
```

```
construct Compaxiom1 f g x that App1(Comp1 f g,x) = App1(f,App1 g x)
```

```
>> Compaxiom1: [(f_1:in function),(g_1:in function),
>>        (x_1:in Nat) => (---:that (((f_1 Comp1
>>        g_1) App1 x_1) = (f_1 App1 (g_1 App1
>>        x_1))))]
>>   {move 0}
```

We provide the basic operations on unary functions. We use iteration instead of recursion, which again requires a justification by constructing a pair and its projections. The iteration constructor is provided (taking a function f and a number a to the function $(n \mapsto f^n(a))$) and the basic axioms for the iterator are provided. Composition of unary functions and its basic axiom are provided.

Lestrade execution:

```
% iteration for operations

% again, work on pairs will be needed to show that this is adequate

clearcurrent

declare o in operation

>> o: in operation {move 1}


declare g in function

>> g: in function {move 1}


declare m in Nat

>> m: in Nat {move 1}


declare n in Nat

>> n: in Nat {move 1}


construct Iter2 o g in operation

>> Iter2: [(o_1:in operation),(g_1:in function)
>>        => (---:in operation)]
>>   {move 0}


construct Iteraxiom21 o g m that App2 (Iter2 o g,m,0) = App1(g,m)

>> Iteraxiom21: [(o_1:in operation),(g_1:in
```

```
>>        function),(m_1:in Nat) => (---:that
>>        (App2((o_1 Iter2 g_1),m_1,0) = (g_1
>>        App1 m_1)))]
>>   {move 0}


construct Iteraxiom2 o g m n that App2(Iter2 o g,m,S n) = App2(o,m,App2(Iter2 o

>> Iteraxiom2: [(o_1:in operation),(g_1:in function),
>>        (m_1:in Nat),(n_1:in Nat) => (---:that
>>        (App2((o_1 Iter2 g_1),m_1,S(n_1)) =
>>        App2(o_1,m_1,App2((o_1 Iter2 g_1),m_1,
>>        n_1))))]
>>   {move 0}
```

We provide the iteration operator on operations, taking an operation $\circ$ and a function g to $(mn \mapsto (p \mapsto m \circ p)^n(g(m))$.

```
Lestrade execution:


% compose operations

% unary operations can be padded with an additional argument on either side to

% the binary iteration construct, so there is no need of compositions with unar

clearcurrent

declare o in operation

>> o: in operation {move 1}


declare o1 in operation

>> o1: in operation {move 1}
```

```
declare o2 in operation

>> o2: in operation {move 1}


declare m in Nat

>> m: in Nat {move 1}


declare n in Nat

>> n: in Nat {move 1}


construct Comp2 o o1 o2 in operation

>> Comp2: [(o_1:in operation),(o1_1:in operation),
>>        (o2_1:in operation) => (---:in operation)]
>>   {move 0}


construct Compaxiom2 o o1 o2 m n that App2(Comp2 o o1 o2,m,n) = App2(o,App2 o1

>> Compaxiom2: [(o_1:in operation),(o1_1:in
>>        operation),(o2_1:in operation),(m_1:
>>        in Nat),(n_1:in Nat) => (---:that (App2(Comp2(o_1,
>>        o1_1,o2_1),m_1,n_1) = App2(o_1,App2(o1_1,
>>        m_1,n_1),App2(o2_1,m_1,n_1))))]
>>   {move 0}


% reduction of arity;  I think this is needed (and sufficient).

construct Diag o in function
```

```
>> Diag: [(o_1:in operation) => (---:in function)]
>>   {move 0}


construct Diagaxiom o n that App1(Diag o,n) = App2(o,n,n)

>> Diagaxiom: [(o_1:in operation),(n_1:in Nat)
>>        => (---:that ((Diag(o_1) App1 n_1) =
>>        App2(o_1,n_1,n_1)))]
>>   {move 0}


% now for the induction axiom

clearcurrent

declare p  prop

>> p: prop {move 1}


declare q  prop

>> q: prop {move 1}


define -> p q:~(p & ~q)

>> ->: [(p_1:prop),(q_1:prop) => (~((p_1 & ~(q_1))):
>>        prop)]
>>   {move 0}


declare m in Nat

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

```
declare pred [m=>prop]

>> pred: [(m_1:in Nat) => (---:prop)]
>>   {move 1}


declare basis that pred 0

>> basis: that pred(0) {move 1}


declare ind  [m => that (pred m) -> pred(S m)]

>> ind: [(m_1:in Nat) => (---:that (pred(m_1)
>>        -> pred(S(m_1))))]
>>   {move 1}


declare n in Nat

>> n: in Nat {move 1}


construct Induction pred, basis, ind, n that pred n

>> Induction: [(pred_1:[(m_2:in Nat) => (---:
>>             prop)]),
>>        (basis_1:that pred_1(0)),(ind_1:[(m_3:
>>             in Nat) => (---:that (pred_1(m_3)
>>             -> pred_1(S(m_3))))]),
>>        (n_1:in Nat) => (---:that pred_1(n_1))]
>>   {move 0}
```

I believe that what I have presented here is a complete axiomatization of PRA. It takes a little work to verify this, because my primitives are stripped down. A pairing operation and its projections must be implementable for

everything to work correctly; they should be. These are needed because I must bound arity in Lestrade, and also because I choose to use iteration rather than recursion for both functions and operations.