pos.scm

Mohammed Slimani

27 November, 2023

Introduction

In the realm of binary arithmetic, the library 'pos.scm' unfolds as a repository encapsulating essential axioms, theorems, and definitions tailored to the landscape of positive binary numbers. Which are a way of representing numbers using only 0s and 1s. In this library, we start by defining what 'One' means in this binary world, and we explore the ideas of 'SZero' and 'SOne,' which help us build and understand positive binary numbers. As we go along, we'll use letters like 'p,' 'q,' and 'r' to represent these numbers in our library.

Algebra: pos

```
(add-algs "pos"
                  '("One"      "pos")
                  '("SZero"    "pos ⇒ pos")
                  '("SOne"     "pos ⇒ pos"))
(add-var-name "p" "q" "r" (py "pos"))
```

We first use the command (add-algs "pos" '("One" "pos") '("SZero" "pos⇒pos") '("SOne" "pos⇒pos")) to create an algebra with constructors **One**, **SZero** and **SOne**, then we use the second command which adds the variables $\mathbf{p}, \mathbf{q}$ and $\mathbf{r}$ as positive binary numbers.

Remark: To develop an understanding of the definition of the algebra "pos" and the functions "PosToNat" and "NatToPos" that I will introduce next, let's explore the following observations:
The binary representation of a positive number is generated from **One** through two successor operations, $S_0 :=$ SZero and $S_1 :=$ SOne, corresponding to the functions n $\mapsto$ 2n and n $\mapsto$ 2n+1, respectively.
We know that every positive number a can be written as:

$$a = \sum_{k=0}^{n} c_k 2^{n-k},$$

where $c_k \in \{0,1\}$ and $c_0 = 1$. This corresponds to the representation: $c_0c_1c_2...c_n$
In the algebra **pos**, this corresponds to the following term:
$S_{c_n}(...S_{c_2}(S_{c_1})...)$

Examples:

decimal representation	binary representation	term of type pos
1	1	1
2	10	$S_0 1$
3	11	$S_1 1$
4	100	$S_0(S_0 1)$
5	101	$S_1(S_0 1)$
6	110	$S_0(S_1 1)$
7	111	$S_1(S_1 1)$
8	1000	$S_0(S_0(S_0 1))$
9	1001	$S_1(S_0(S_0 1))$
10	1010	$S_0(S_1(S_0 1))$

Program constants

PosToNat

We first add the Program constant "NatDouble" through:
(add-program-constant "NatDouble" (py "nat ⇒ nat")), with the following computation rules:

"NatDouble Zero" "Zero"
"NatDouble(Succ n)" "Succ(Succ(NatDouble n))"

Now, we add the program constant "PosToNat" through:
(add-program-constant "PosToNat" (py "pos ⇒ nat"))
The behaviour of the program constant is specified by the following computation

rules:

"PosToNat One" "Succ Zero"
"PosToNat(SZero p)" "NatDouble(PosToNat p)"
"PosToNat(SOne p)" "Succ(NatDouble(PosToNat p))")

NatToPos

We first add the Program constant "NatToPosStep" through:
(add-program-constant
"NatToPosStep" (py "nat ⇒ (nat ⇒ pos) ⇒ pos")),
with the following computation rules:

"NatToPosStep n(nat ⇒ pos)"
"[if (NatEven n) (SZero((nat ⇒ pos)(NatHalf n)))
[if (n=Succ Zero) One
(SOne((nat ⇒ pos)(NatHalf n)))]]")

Now we add the program constant "NatToPos" through:
(add-program-constant "NatToPos" (py "nat ⇒ pos"))

The behaviour of the program constant is specified by the following computation rule:

"NatToPos n" "(GRec nat pos)([n]n)n NatToPosStep")

"GRec" refers to the general recursion operator, which is defined for given algebras "nat" and "pos" as follows:

(GRec nat pos)([n]n) n NatToPosStep :=

NatToPosStep n ([n](GRec nat pos)([n]n) (NatHalf n) NatToPosStep)

cNatPos

The program constant "NatToPos" produces undesired unfoldings as the input, which is a natural number, becomes larger.

To solve this problem, we aim to define another program constant "CNatPos", abbreviating the computational content of "NatToPos", which will be unfolded under normalization.

Compare in Minlog the output of the two following normalizations:

(nt (pt "NatToPos (Succ (Succ n)"))
(nt (pt "cNatPos (Succ (Succ n))"))

PosLt, PosLe

Now, we define on the algebra "pos" the "<" and "≤ "*operators*.

We add the program constants "PosLt" and "PosLe" through:
(add-program-constant "PosLt" (py "pos ⇒ pos ⇒ boole"))
(add-program-constant "PosLe" (py "pos ⇒ pos ⇒ boole"))

Since the computation rules of the two constants tend to depend on each other, we add their computation rules simultaneously.

(add-computation-rules

"p < One"	" False "
"One < SZero p"	" True "
"One < SOne p"	" True "
"SZero p < SZero q"	"p < q"
"SZero p < SOne q"	"p ≤ q"
"SOne p < SZero q"	"p < q"
"SOne p < SOne q"	"p < q")

(add-computation-rules

"One ≤ q"	"$True$ "
"SZero p ≤ One"	"$False$ "
"SOne p ≤ One"	"$False$ "
"SZero p ≤ $SZero\ q$"	"$p \leq q$"
"SZero p ≤ $SOne\ q$"	"$p \leq q$"
"SOne p ≤ $SZero\ q$"	"$p < q$"

"SOne p $\leq$ $SOne\ q$" "$p \leq q$")

(add-computation-rules

"One $\leq$ q"	"$True$"
"SZero p $\leq$ One"	"$False$"
"SOne p $\leq$ One"	"$False$"
"SZero p $\leq$ $SZero\ q$"	"$p \leq q$"
"SZero p $\leq$ $SOne\ q$"	"$p \leq q$"
"SOne p $\leq$ $SZero\ q$"	"$p < q$"
"SOne p $\leq$ $SOne\ q$"	"$p \leq q$")

PosLtToLe

Theorem: $\forall p, q\ (p < q \rightarrow p \leq q)$

Proof: We first proceed by induction (using (ind)), hence we consider have now the two goals A(1) and A(p) $\rightarrow$ A(p + 1), where A(p) = $\forall q\ (p < q \rightarrow p \leq q)$. First consider A(1) = $\forall q\ (1 < q \rightarrow 1 \leq q)$. First normalize the goal (using (ng t)), then instantiate q and then assume "1 < q" (using (strip)), finally use (Truth), hence the claim is proved.

Now consider the claim A(p) $\rightarrow$ A(p + 1). Instantiate p and assume A(p), then consider the cases (i) where p = 1 (ii) where the positive number is of the form SZero(p) and (iii) where the positive number is of the form SOne(p).

Now Consider the case where p = 1, namely: $\forall q(1 \leq q \rightarrow 1<q) \rightarrow \forall q(2<q \rightarrow 2 \leq q)$. After normalizing the goal we get $\forall q(1<q \rightarrow T)$, which is trivial, since we can instantiate "q", assume "1<q" (using (strip)) and conclude using (use "Truth").
Consider now (ii), namely the goal: $\forall p(\forall q(p<q \rightarrow p \leq q) \rightarrow \forall q(\text{SZero } p<q \rightarrow \text{SZero } p \leq q))$, we first instantiate "p" and assume the induction hypothesis, we get $\forall q(\text{SZero } p<q \rightarrow \text{SZero } p \leq q)$. We consider the cases (ii.1) where q = 1 (ii.2) where the positive number is of the form SZero(q) and (ii.3) where the positive number is of the form SOne(q). Now we consider the case where q=1, namely: SZero p<1 $\rightarrow$SZero $p \leq 1$, which is trivial after normalizing the term.
Now consider (ii.2), namely: $\forall q$(SZero p<SZero p $\rightarrow$SZero$p$ $\leq$SZero q), we assume "q", normalize and use the induction hypothesis to conclude.
Now consider (ii.3), namely: $\forall q$(SZero p< SOne q $\rightarrow$SZero $p \leq$ SOne q). After assuming "q" and normalizing, we get $p \leq q \rightarrow p \leq q$, which is trivial, since we can assume $p \leq q$ ans use it to conclude.
Consider now (iii), namely the goal: $\forall p(\forall q(p<q \rightarrow p \leq q) \rightarrow allq(\text{SOne } p<q \rightarrow \text{SOne } p \leq q))$, we first instantiate "p" and assume the induction hypothesis, we get $\forall q(\text{SOne } p<q \rightarrow \text{SOne } p \leq q)$, then consider the cases (iii.1) where q = 1 (iii.2) where the positive number is of the form SZero(q) and (iii.3) where the positive number is of the form SOne(q).
Now we consider the case where q=1, namely: SOne p<1 $\rightarrow$SOne $p \leq 1$. After normalizing the goal we get "$F \rightarrow F$" which is trivial by the same previous explanation.

Now consider (iii.2), namely: $\forall q$(SOne p<SZero q →SOne p ≤SZero q). After instantiating "q" and normalizing, we get $p<q \rightarrow p<q$, which is again trivial. Now consider (iii.3), namely: $\forall q$(SOne p<SOne q → SOne p ≤SOne q). After instantiating "q", normalizing and using induction hypothesis we finish the proof.

```
(ind)
(ng #t)
(strip)
(use "Truth")
(assume "p" "IH")
(cases)
(ng #t)
(assume "f")
(use "f")
(assume "q")
(ng #t)
(use "IH")
(assume "q")
(ng #t)
(assume "p<=q")
(use "p<=q")
(assume "p" "IH")
(cases)
(ng #t)
(assume "f")
(use "f")
(assume "q")
(ng #t)
(assume "p<p")
(use "p<p")
(assume "q")
(ng #t)
(use "IH")
```

PosLtrans

Theorem: $\forall p, q, r\ ((p < q \rightarrow q \leq r \rightarrow p < r)\ andnc$
$(\mathrm{p} \leq q \rightarrow q \leq r \rightarrow p \leq r)\ andnc$
$(\mathrm{p} \leq q \rightarrow q < r \rightarrow p < r))$

Proof:

```
(ind)
(cases)
(assume "r")
```

```
(ng #t)
(split)
(use "Efq")
(split)
(auto)
(assume "q")
(ng #t)
(cases)
(ng #t)
(split)
(auto)
(split)
(auto)
(assume "r")
(ng #t)
(split)
(auto)
(split)
(auto)
(assume "r")
(ng #t)
(split)
(auto)
(split)
(auto)
(assume "q")
(cases)
(ng #t)
(split)
(auto)
(split)
(auto)
(assume "r")
(ng #t)
(split)
(auto)
(split)
(auto)
(assume "r")
(ng #t)
(split)
(auto)
(split)
(auto)
(assume "p" "IH1")
(cases)
```

```
(assume "q")
(ng #t)
(split)
(use "Efq")
(split)
(use "Efq")
(use "Efq")
(assume "q")
(cases)
(ng #t)
(split)
(auto)
(split)
(auto)
(assume "r")
(ng #t)
(split)
(use "IH1")
(split)
(use "IH1")
(use "IH1")
(assume "r")
(ng #t)
(split)
(assume "p<q" "q<=r")
(use "PosLtToLe")
(use-with "IH1" (pt "q") (pt "r") 'left "p<q" "q<=r")
(split)
(use "IH1")
(use "IH1")
(assume "q")
(cases)
(ng #t)
(split)
(auto)
(split)
(auto)
(assume "r")
(ng #t)
(split)
(use "IH1")
(split)
(use "IH1")
(assume "p<=q" "q<r")
(use "PosLtToLe")
(use-with "IH1" (pt "q") (pt "r") 'right 'left "p<=q" "q<r")
```

```
(assume "r")
(ng #t)
(split)
(use "IH1")
(split)
(assume "p<=q" "q<r")
(use "PosLtToLe")
(use-with "IH1" (pt "q") (pt "r") 'right 'left "p<=q" "q<r")
(use "IH1")
(assume "p" "IH1")
(cases)
(assume "q")
(ng #t)
(split)
(use "Efq")
(split)
(use "Efq")
(use "Efq")
(assume "q")
(cases)
(ng #t)
(split)
(auto)
(split)
(auto)
(assume "r")
(ng #t)
(split)
(use "IH1")
(split)
(assume "p<q" "q<r")
(use-with "IH1" (pt "q") (pt "r") 'left "p<q" "?")
(use "PosLtToLe")
(use "q<r")
(use "IH1")
(assume "r")
(ng #t)
(split)
(use "IH1")
(split)
(use "IH1")
(assume "p<q" "q<=r")
(use "PosLtToLe")
(use-with "IH1" (pt "q") (pt "r") 'left "p<q" "q<=r")
(assume "q")
(cases)
```

```
(ng #t)
(split)
(auto)
(split)
(auto)
(assume "r")
(ng #t)
(split)
(assume "p<q" "q<r")
(use-with "IH1" (pt "q") (pt "r") 'left "p<q" "?")
(use "PosLtToLe")
(use "q<r")
(split)
(use "IH1")
(use "IH1")
(assume "r")
(ng #t)
(split)
(use "IH1")
(split)
(use "IH1")
(use "IH1")
```