Peano Arithmetic for all integers

An extension of Peano's axioms for negative numbers

2021

Peano's axioms have been used to formalize a notion of natural numbers to start counting from 0 up to positive infinity. The axioms introduce their own data type, defined inductively as follows:

1. The set of natural numbers is called $\mathbb{N}$.
2. The first element of the set is 0.
3. Any other element of the set can be constructed by using $S(\cdot)$ on another element already within the set. (Here S stands for successor, meaning that S would provide the 'next' number of the input given.)

For simplicity and brevity the more exhaustive properties have been omitted with just the essence kept. Although the properties listed does not directly provide symbols for the more common use of natural numbers, one can easily identify $S(0)$ as 1, $S(S(0))$ as 2, and so. (One can also use $S(1)$ as 2 if 1 is defined as a symbolic shortcut for $S(0)$.) And so, the set can be listed as a roster as follows:

$$\mathbb{N} = \{0, S(0), S(S(0)), S(S(S(0))), \dots\}$$

Addition and multiplication have then been defined on this number system, as follows:

$$n + 0 = n \tag{1}$$
$$m + S(b) = S(m + b) \tag{2}$$

(Notice that the operations are identified between two numbers m and n, where if one needs to deconstruct the numbers we use the following convention $m = S(a)$ and $n = S(b)$.)

While counting forwards by adding one is simple, (i.e., given any number n that one has, S can be called to construct the next number $S(n)$) counting backwards can be achieved with a function P (for predecessor) as follows:

$$P : \mathbb{N} \to \mathbb{N} \tag{3}$$
$$P(S(a)) = a \tag{4}$$

Notice that notation does not directly allow one to find the predecessor of a number m, instead it has to be identified as the successor of some other number a and then the function S 'unwrapped'.

However, the implementation of the predecessor function creates a problem: the number 0 is not the successor of any other number in the set, and it's predecessor has to be defined differently. Either we identify $P(0) = 0$ as a valid statement, or we say that $P(0)$ is undefined within $\mathbb{N}$, and so not allowing P to be closed within the natural numbers.

Starting with integers

Moreover, notice that subtraction cannot be closed within our set either, or any implementation of the natural numbers.

To get around these problems, we extend Peano's axioms to create a set for all integers.

1. The set of integers is called $\mathbb{Z}$.

2. 0 is in the set of integers.
3. The function $S(\cdot)$ can be used on any element already within the set to produce another element.
4. The function $P(\cdot)$ can be used on any element already within the set to produce another element.

Notice that we've included $P(\cdot)$ within the axioms to allow us negative numbers. This is because we've promoted $P(\cdot)$ from being a function to being a data constructor, a construct that we can use to create new pieces of data under a given data type (in this case, $\mathbb{Z}$). Notice that $S(\cdot)$ wasn't a function either, instead being a data constructor. We have to define them as such because there's no sensible codomain where $S(\cdot)$ and $P(\cdot)$ can map to, and the result of $S(\cdot)$ taking 0 is simply denoted as $S(0)$.

What is also implied by the axioms is that $S(\cdot)$ and $P(\cdot)$ work like inverses. That is, if S and P are called successively on some integer n, the would 'cancel' each other out to produce the first argument. That is, $S(P(n)) = P(S(n)) = n$.[1]

The intuition behind this can be identified with the number line. With axiom 2, we can be placed in the center of the line at 0, and if we call S on ourselves (and imagine ourselves to be a number), we can take one step to the right. And similarly, we can call P on ourselves to go one step to the left. And since we don't have to stop at 0, we can call P there to take one step to the left, landing us at -1 and into the world of negative numbers. The set can thus be illustrated in roster form as (only writing down the integers from -3 to 3 inclusive):

$$\mathbb{Z} = \{\dots, P(P(P(0))), P(P(0)), P(0), 0, S(0), S(S(0)), S(S(S(0))), \dots\}$$

A note about canonical forms

Since P can be used to cancel out S and vice versa, it makes sense to identify all numbers as a chain of calls to P or S until the stack ends in 0 with there being no calls to S after one to P or vice versa. This also allows us to quickly determine whether a number is negative or positive by simply peeking at the top constructor of the stack and seeing whether it is P (negative) or S (positive). However, the axioms stated do give us the freedom to chain the calls whimsically and still get a number out of it. For example, $S(S(S(S(S(P(S(P(S(P(S(P(P(P(S(0)))))))))))))))$ is just as valid an integer, and is in fact equivalent to $S(S(S(0)))$, or 3. However, trying to reason about these complicated stacks become very hard, and we therefore identify a canonical/normal form for a number. We say 0 is in canonical form, and any piece of data that only has P or S in its data construction/call stack is in canonical form. Any other piece of data (that mixes P and S in its stack) is not in canonical form. And the following function has been provided to reduce a number to its canonical form:

$$\begin{aligned}
\texttt{simplify} &: \mathbb{Z} \to \mathbb{Z} \\
\texttt{simplify} &: S(P(a)) \mapsto \texttt{simplify}(a) \\
\texttt{simplify} &: P(S(a)) \mapsto \texttt{simplify}(a) \\
\texttt{simplify} &: P(a) \mapsto P(\texttt{simplify}(a)) \\
\texttt{simplify} &: S(a) \mapsto S(\texttt{simplify}(a))
\end{aligned}$$

Notice that the number of interest here was an integer n, although that was nowhere to be found. Instead, we decomposed n as one of the four forms $S(P(a))$, $P(S(a))$, $P(a)$, or $S(a)$. And since a formula was impossible to provide for $\texttt{simplify}(n) = f(n)$, we used pattern matching on the constructors instead.

Side note to a side note: although this function simplifies the construction stack, it does return the same number, and if someone were cavalier about the stacks they were working with they could simply define $\texttt{simplify} = \mathrm{id}_{\mathbb{Z}}$ and call it a day.

[1]Notice that this would have not been possible under the natural numbers, since the predecessor of 0 would either have been undefined resulting in the whole stack blowing up, or a do-nothing operation in which case S would have incorrectly produced $S(0)$ (or 1) for $S(P(0))$.

The function provided in the box has a form that will appear a lot throughout this write-up. Since it's often impossible to work abstractly over an integer, we have to take a look at the constructors it has and work on deconstructing it. (For the natural numbers P was another such function that worked by unwrapping/deconstructing constructors.)

Another such example can be found with the new predecessor and successor functions we have to define. Since we can take a step to the left by calling P or removing an S, we can formalize the notions as follows:

$$\begin{aligned}
&\texttt{pred} : \mathbb{Z} \to \mathbb{Z} \\
&\texttt{pred} : S(a) \mapsto a \qquad (n = S(a)) \\
&\texttt{pred} : n \mapsto P(n)
\end{aligned}$$

$$\begin{aligned}
&\texttt{succ} : \mathbb{Z} \to \mathbb{Z} \\
&\texttt{succ} : P(a) \mapsto a \qquad (n = P(a)) \\
&\texttt{succ} : n \mapsto S(n)
\end{aligned}$$

The usage of `pred` and `succ` simplifies our lives by letting the call stacks stay uncluttered, but they don't detect if a number is not in canonical form. And so, for the rest of this article, any functions we define will simply assume the numbers are in canonical form to simplify computations.

Another thing to note here is that the order in which the statements appear matter. After the domain and codomain definition, the first line checks whether the number n has the form $P(a)$ or $S(a)$. If it doesn't, we go to the last line of the definition which cares about all other forms the numbers can take. The accompanying file `PeanoInts.hs` makes good use of this, which is provided by the pattern matching construct in Haskell. In Haskell a piece of input data can be matched against several patterns (where all pieces of data are generated by using constructors like the ones here) and the first matching constructor pattern is used. This allows the latter patterns to be simpler, since we can make particular assumptions as earlier patterns have been discarded.[2]

Cutting the monotony with two simple algorithms

To get away from the dry talk of algorithms and functions, I bring you two new algorithms and functions. This time, the ideas are so simple that you can read passively while thinking about whether you filed your tax papers correctly.

The first function we take a look at is negation, identified by `neg` which negates a number. It works as follows:

$$\begin{aligned}
&\texttt{neg} : \mathbb{Z} \to \mathbb{Z} \\
&\texttt{neg}(n) = \begin{cases} 0 & \text{if } n = 0 \\ P(\texttt{neg}(a)) & \text{if } n \text{ has the form } S(a) \\ S(\texttt{neg}(a)) & \text{if } n \text{ has the form } P(a) \end{cases}
\end{aligned}$$

Also, I used the usual mathematical case-by-case description of a function to show what it may look like, but it involves more typing on my part and so I'm foregoing it for all other functions except the two here.

The second function is the absolute value operation, which simply flips all the P in a stack to S.

$$\begin{aligned}
&\texttt{abs} : \mathbb{Z} \to \mathbb{Z} \\
&\texttt{abs}(n) = \begin{cases} S(\texttt{abs}(a)) & \text{if } n \text{ has the form } P(a) \\ n & \text{otherwise} \end{cases}
\end{aligned}$$

[2]The most obvious example of this is found when checking for 0 when we implement our division algorithm.

Starting with the arithmetic

Now that our numbers have been defined and we know how to navigate over them, let's start by abstracting away some of that navigation by defining the four basic arithmetic operations.

Addition

The first one is addition, implemented by the `add` function. The syntax equivalence is $\texttt{add}(a, b) = a + b$.[3]

$$\begin{aligned}
&\texttt{add} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
&\texttt{add} : (n, 0) \mapsto n && \texttt{add} : (0, n) \mapsto n \\
&\texttt{add} : (P(a), S(b)) \mapsto \texttt{add}(a, b) && \texttt{add} : (S(a), P(b)) \mapsto \texttt{add}(a, b) \\
&\texttt{add} : (P(a), P(b)) \mapsto \texttt{add}(a, P(P(b))) \\
&\texttt{add} : (S(a), S(b)) \mapsto \texttt{add}(a, S(S(b)))
\end{aligned}$$

The first line refers to the use of 0 as the additive identity. The second line matches one number as negative and one number as positive. We make the decompositions $m = a - 1$ and $n = b + 1$. This allows $m + n = a - 1 + b + 1 = a + b$. This also minimizes the depth of the stack and eventually it reduces down with either side being 0 and the whole computation being reduced to the additive identity. The (m, n) both negative and (m, n) both positive cases work the same way, where we remove one layer of P or S call from one integer to give to the other integer.

Subtraction

TODO: write-up

$$\begin{aligned}
&\texttt{sub} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
&\texttt{sub} : (m, 0) \mapsto m \\
&\texttt{sub} : (0, m) \mapsto \texttt{neg}(n) \\
&\texttt{sub} : (m, P(b)) \mapsto \texttt{add}(m, \texttt{neg}(P(b))) \\
&\texttt{sub} : (S(a), S(b)) \mapsto \texttt{sub}(a, b) \\
&\texttt{sub} : (P(a), S(b)) \mapsto P(P(\texttt{sub}(a, b)))
\end{aligned}$$

More simply,

$$\texttt{sub} : (m, n) \mapsto \texttt{add}(m, \texttt{neg}(n))$$

Multiplication

TODO: write-up

$$\begin{aligned}
&\texttt{mul} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
&\texttt{mul} : (m, 0) \mapsto 0 && \texttt{mul} : (0, n) \mapsto 0 \\
&\texttt{mul} : (m, S(0)) \mapsto m && \texttt{mul} : (S(0), n) \mapsto n \\
&\texttt{mul} : (m = S(a), S(b)) \mapsto \texttt{add}(m, \texttt{mul}(m, b)) \\
&\texttt{mul} : (m = P(a), n = P(b)) \mapsto \texttt{mul}(\texttt{neg}(m), \texttt{neg}(n)) \\
&\texttt{mul} : (m = S(a), n = P(b)) \mapsto \texttt{neg}(\texttt{mul}(m, \texttt{neg}(n))) \\
&\texttt{mul} : (m = P(a), n = S(b)) \mapsto \texttt{mul}(n, m)
\end{aligned}$$

[3]The definitions on the right are only to make commutativity explicit and can be safely ignored.

Division

TODO: write-up

$$\begin{aligned}
&\texttt{div} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
&\texttt{div} : (m, 0) \mapsto \texttt{undefined} \\
&\texttt{div} : (0, n) \mapsto 0 \\
&\texttt{div} : (m, S(0)) \mapsto m \\
&\texttt{div} : (m = P(a), n = P(b)) \mapsto \texttt{div}(\texttt{neg}(m), \texttt{neg}(n)) \\
&\texttt{div} : (m = P(a), n = S(b)) \mapsto \texttt{neg}(\texttt{div}(\texttt{neg}(m)), n) \\
&\texttt{div} : (m = S(a), n = P(b)) \mapsto \texttt{neg}(\texttt{div}(m, \texttt{neg}(n))) \\
&\texttt{div} : (m, n) \mapsto \begin{cases} 0 & \text{if } m < n \\ S(\texttt{div}(\texttt{sub}(m, n), n)) & \text{otherwise} \end{cases}
\end{aligned}$$

Bonus: Remainder

TODO: write-up

$$\begin{aligned}
&\texttt{rem} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
&\texttt{rem} : (m, 0) \mapsto \texttt{undefined} \\
&\texttt{rem} : (0, n) \mapsto 0 \\
&\texttt{rem} : (m, S(0)) \mapsto 0 \\
&\texttt{rem} : (m = S(a), n = P(b)) \mapsto \texttt{rem}(m, \texttt{neg}(n)) \\
&\texttt{rem} : (m = P(a), n = S(b)) \mapsto \texttt{neg}(\texttt{rem}(\texttt{neg}(m), n)) \\
&\texttt{rem} : (m = P(a), n = P(b)) \mapsto \texttt{neg}(\texttt{rem}(\texttt{neg}(m), \texttt{neg}(n))) \\
&\texttt{rem} : (m, n) \mapsto \begin{cases} m & \text{if } m < n \\ \texttt{rem}(\texttt{sub}(m, n), n) & \text{otherwise} \end{cases}
\end{aligned}$$

Concisely,

$$\texttt{rem}(m, n) = \texttt{sub}(m, \texttt{mul}(n, \texttt{div}(m, n)))$$

Double bonus: Sign/Signum function

TODO: write-up

$$\begin{aligned}
&\texttt{sgn} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
&\texttt{sgn} : 0 \mapsto 0 \\
&\texttt{sgn} : P(a) \mapsto P(1) \\
&\texttt{sgn} : S(a) \mapsto S(1)
\end{aligned}$$