Declaring Numbers

Bernd Braßel, Frank Huch and Sebastian Fischer

Department of Computer Science, University of Kiel, Germany

WFLP 2007, Paris, France

I'm going to present joint work with my colleagues Bernd Braßel and Frank Huch. We developed a declarative implementation of numeric operations.

The topic of this talk is not particularly deep. I will show that numeric operations currently do not fit into the FLP paradigm, why they should, and how they can.

Numbers are external

Münster Curry Compiler (MCC)

```
foreign import ccall "prims.h primAddInt"
  (+) :: Int -> Int -> Int
```

Portland Aachen Kiel Curry System (PAKCS)

```
--- Adds two integers.
(+)   :: Int -> Int -> Int
x + y = (prim_Int_plus $# y) $# x

prim_Int_plus :: Int -> Int -> Int
prim_Int_plus external
```

Currently, numeric operations are implemented as external functions in all available Curry systems.

For example, the Münster Curry Compiler uses externally defined C functions. The Curry system PAKCS uses the numeric operations of the underlying Prolog system.

Don't be confused by the details of the definition. You only need to understand that (+) is implemented by an external function.

Arithmetic suspends on free variables

MCC

```
> let x,y free in x+y
Suspended
```

PAKCS

```
> let x,y free in x+y
*** Goal suspended (use ":set +suspend" for details)!
```

Because they are implemented externally, numeric operations cannot handle free variables as parameters. Both MCC and PAKCS suspend the computation, when arguments of numeric operations are unbound.

Functional Logic Programming in Curry

Functions

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)
```

Equality Constraints

```
last :: [a] -> a
last xs | xs =:= ys++[x] = x
 where x,ys free
```

Function (++) does not suspend but **guesses** free parameters

Curry programmers are accustomed to a different behavior.

As this is the Workshop on FLP, I will not spend too much of my time in explaining the syntax of Curry. Rather, I will explain important concepts along the way.

In Curry, functions can be defined by multiple rules with pattern matching. The function (++) appends two lists by recursively descending the first.

The logic features of Curry can be employed with equality constraints that guard specific rules. For example, the function `last` *reuses* (++) to search for an appropriate splitting of the input list.

This programming style is not possible with functions that suspend on free parameters. Rather, it relies on functions to guess them.

FLP not possible with numbers

No guessing

Pythagorean triples

```
pythagoras :: (Int,Int,Int)
pythagoras | a*a + b*b =:= c*c = (a,b,c)
 where a,b,c free

> pythagoras
Suspended
```

Explicit enumeration of search space necessary but **tedious!**

Consider this example for a definition that relies on numeric operations to guess unbound parameters.

Three numbers form a Pythagorean triple if they fulfill the Pythagorean theorem. The computation of such triples in Curry could be straightforward as this definition shows.

Unfortunately, it is not sufficient in current Curry systems. Instead, the programmer needs to supply additional definitions to bind the free variables `a`,`b` and `c`.

FLP not possible with numbers

No laziness

Generate list of variables

```
vars :: Int -> [a]
vars n | n =:= length vs = vs
 where vs free

> vars 7
[_a,_b,_c,_d,_e,_f,_g]
More solutions? [Y(es)/n(o)/a(ll)] Yes
Not enough free memory after garbage collection
```

Solution found but **infinite loop** due to lack of laziness

Externally defined numbers are conceptually represented as infinitely many distinct constructors. Therefore, an equality constraint cannot fail, when it compares a known value to an unbounded number of different values.

Take your time to examine the definition of the function `vars`. It *reuses* `length` (the length-function on lists) to guess a list of free variables of length `n`.

If we apply this function in current Curry systems, a list of free variables is computed correctly. However, the answer 'Yes' to the question for more solutions is evil. The program continues forever – or until exceeding memory – to check for larger and larger lists whether their length equals 7.

Peano Numbers

Zero and Successor

```
data Num = Z | S Num

(+) :: Num -> Num -> Num
Z   + m = m
S n + m = S (n+m)

(*) :: Num -> Num -> Num
Z   * _ = Z
S n * m = m + n*m
```

Well known example for FLP paradigm

```
(-) :: Num -> Num -> Num
n - m | n =:= m+k = k
 where k free
```

A declarative representation of numbers without the discussed drawbacks is named after Peano's axioms.

Peano numbers are represented in unary notation as values of the recursive datatype `Num` with two constructors – `Z` for *zero* and `S` for *successor*.

In Curry, new datatypes can be defined using the keyword `data`. The different constructors of a datatype are introduced along with their argument types separated by a vertical bar.

The implementation of arithmetic operations on Peano numbers is straightforward. The implementation of `(-)` in terms of `(+)` belongs – together with the definition of `last` shown earlier – to the best known examples in our community that demonstrate how the FLP paradigm can enhance *reuse*.

Pythagoras revisited

Enumerating Pythagorean triples

```
pythagoras | a*a + b*b =:= c*c = (a,b,c)
 where -- a,b,c > 0
  a = S x; b = S y; c = S z; x,y,z free

triples n =  getSearchTree pythagoras >>=
             mapIO_ print . take n . allValuesB

> triples 2
(S (S (S Z)),S (S (S (S Z))),S (S (S (S (S Z)))))
(S (S (S (S Z))),S (S (S Z)),S (S (S (S (S Z)))))
```

Let's revisit the computation of Pythagorean triples in the context of Peano numbers.

In the definition shown earlier we forgot to state that all components of the triple should be greater than zero. Therefore, the definition shown here is a bit more complicated.

Do not spend too much time with the definition of `triples`. It uses a fair strategy to enumerate the search space and print the first `n` triples found.

Although we still do not provide definitions to bind `a,b` and `c` explicitly, we can guess Pythagorean triples using the defined arithmetic operations on Peano numbers.

As we forgot to state that `a,b` and `c` should be in ascending order, we get the first Pythagorean triple twice – (3,4,5) and (4,3,5) are computed.

Laziness

A very big number

```
infinity = S infinity

> Z =:= infinity
No solution
```

Generate variables again

```
vars (length vs) = vs -- beautiful function patterns!

> vars (S (S (S (S (S (S (S Z)))))))
[_a,_b,_c,_d,_e,_f,_g]
More solutions? [Y(es)/n(o)/a(ll)] Yes
No more solutions
```

Another disadvantage of externally defined numbers was their lack of laziness.

Peano numbers are represented with a finite number of constructors. Therefore equality constraints can fail without evaluating both arguments completely. For example, unifying zero with infinity – defined as cyclic successor term – fails without running out of memory.

Reconsider the function `vars` that computes a list of free variables. This definition uses a function pattern and is equivalent to the definition shown earlier. The definition of the length-function that returns a Peano number is omitted.

Well, that's a nice definition, isn't it? Don't you want to be able to program like this?

With Peano arithmetic, the green 'Yes' is not evil anymore. The computation fails nicely after computing the first and only result.

Comparison

Two different worlds

	External	Peano
Guessing	**no**	**yes**
Laziness	**no**	**yes**
Performance	**good**	**bad**
Implementation	**external**	**lightweight**
Size of Numbers	**system dependent**	**unbounded**

Table: External vs. Peano Numbers

Peano numbers have **no practical relevance**!
Implementation of numbers that fits nicely in FLP paradigm with moderate performance overhead?

When comparing external with Peano numbers, we see many advantages for the latter.

1. External numbers do not support guessing – Peano numbers do.
2. External numbers do no support incremental comparison – Peano numbers do.
3. The implementation of external numbers is additional work for compiler writers – Peano numbers are implemented in Curry.
4. Whether there are MIN- and MAXINT values depends on the employed Curry system – Peano numbers can be arbitrarily large.

However, there is one disadvantage of a unary encoding of numbers. In fact, this is a very big disadvantage. Peano numbers are tremendously inefficient.

The performance overhead is **the** reason why standard numeric operations are not implemented lightweight but externally defined.

Can we define numeric operations with all the advantages of Peano numbers and acceptable performance?

Here it is!

Numbers in binary notation

```
data Nat = One   | O Nat    | I Nat
data Int = Zero  | Pos Nat  | Neg Nat
```

least significant bit first

most significant bit: `One`

$$\texttt{One} \mathrel{\hat{=}} 1$$
$$\texttt{O n} \mathrel{\hat{=}} 2 * n$$
$$\texttt{I n} \mathrel{\hat{=}} 2 * n + 1$$

$$\texttt{Pos (O (O One))} \mathrel{\hat{=}} 4$$
$$\texttt{Neg (O (I (O (I (O One)))))} \mathrel{\hat{=}} \text{-}42$$

Values of type `Nat` are positive integers in binary notation. The datatype `Int` adds zero and negative numbers.

Binary Arithmetic

Successor function

```
succ :: Nat -> Nat
succ One    = O One          -- 1+1      = 2
succ (O n)  = I n            -- 2*n+1    = 2*n+1
succ (I n)  = O (succ n)     -- 2*n+1+1  = 2*(n+1)
```

- completely defined
- comments show correctness of rules
- yields head-normal form if argument in head-normal form
- consumes argument up to first zero

We can easily define a successor function on `Nat` values.

Binary Arithmetic

Addition

```
(+) :: Nat -> Nat -> Nat
One + m   = succ m        -- 1    + m        = m + 1
O n + One = I n           -- 2*n + 1        = 2*n + 1
O n + O m = O (n+m)       -- 2*n + 2*m      = 2*(n+m)
O n + I m = I (n+m)       -- 2*n + (2*m+1)  = 2*(n+m)+1
I n + One = O (succ n)    -- (2*n+1) + 1    = 2*(n+1)
I n + O m = I (n+m)       -- (2*n+1) + 2*m  = 2*(n+m)+1
I n + I m = O (succ n+m)  -- (2*n+1)+(2*m+1)=2*(n+1+m)
```

- descends on both arguments
- larger argument rarely consumed completely
- yields hnf if both arguments in hnf

Addition is a bit more complex but straightforward.

Again, the function is completely defined and the correctness of the rules is justified in the corresponding comments.

The part of the larger argument that is not matched by `(+)` is consumed by `succ` up to the first zero.

Binary Arithmetic

Multiplication

```
(*) :: Nat -> Nat -> Nat
One * m = m              -- 1         * m = m
O n * m = O (n*m)        -- (2*n)     * m = 2*(n*m)
I n * m = O (n*m) + m    -- (2*n+1)   * m = 2*(n*m) + m
```

- descends on first argument
- (+) consumes second argument
- yields hnf if both arguments in hnf

In the paper: remaining operations and extension to `Int` type

Our implementation of multiplication uses addition. If you are aware of a better implementation, please let me know!

The second argument of (*) is either returned in the first rule or consumed by (+) in the last rule.

Performance

Slowdown

`(+)`	is about 3 times slower
`(-)`	is about 7 times slower
`(*)`	is about 10 times slower
`div`	is about 20 times slower
`mod`	is about 20 times slower

Table: Slowdown of numeric operations measured in PAKCS

Hard to find good benchmarks
Even in programs concerned with numbers (prime sieve, sorting)
surrounding computation dominates numeric calculations

$\rightsquigarrow$ high performance numeric operations not crucial for FLP

A binary implementation of numbers imposes moderate performance overhead.

Conclusions

Comparison

	External	Peano	Binary
Guessing	**no**	**yes**	**yes**
Laziness	**no**	**yes**	**yes**
Performance	**good**	**bad**	**acceptable**
Implementation	**external**	**lightweight**	**lightweight**
Size of Numbers	**depends**	**unbounded**	**unbounded**

Table: Binary Numbers Enter the Comparison

Helps writing beautiful code
Enables solving of simple equations
No replacement for constraint solving

In the paper: termination and solvability issues
Future Work: declarative replacement for floats

We believe that a seamless integration of numeric operations into the FLP paradigm is worth spending the overhead. Therefore, we implemented binary numbers with an algebraic datatype in Curry.

Decide for yourself, whether their performance is acceptable, given the benefits of a lightweight implementation.