Declaring Numbers

Bernd Braßel, Sebastian Fischer, Frank Huch[1,2]

Institute of Computer Science, University of Kiel
Olshausenstr. 40, 24098 Kiel, Germany

Abstract

Most implementations of functional and functional logic languages treat numbers and the basic numeric operations as external entities. The main reason for this is efficiency. However, this basic design decision has many unfortunate consequences for all programs using numbers. We present an approach to model numbers in a declarative way and argue that the loss in efficiency is compensated by the newly gained possibilities. Functional logic languages benefit the most from this proposal because all the numeric operations become fully narrowable. This enables the solving of simple equations on numbers in an efficient way without having to resort to external constraint solvers.
The presented approach can either be used as a library for purely declarative numbers or it can be employed as a basic data type of functional (logic) languages. Indeed, we have integrated the presented data structures as the only numbers available in our compiler for the functional logic language Curry and have experienced several months of satisfactory performance.

1 Introduction

Implementations of functional and functional logic languages usually provide numbers as an external data type and reuse the default implementation of the underlying language (e.g. C) for the implementation of operation like `(+)`, `(-)`, `(<=)`, `(==)`. This provides a very efficient implementation of complex arithmetic computations within the high-level language.

However, in the context of functional logic languages, this approach results in a major drawback: numbers cannot be guessed by means of narrowing. As a consequence, semantic extensions, like *residuation* [4,5], have been proposed which allow the user to use some restricted logical features in combination with numbers. The idea is that all (externally defined) functions on numbers suspend on free variables as long as their value is unbound. These residuated functions have to be combined with a generator which specifies all possible numbers of a free variable. As an example, we consider Pythagorean triples (a, b, c) with $a^2 + b^2 = c^2$. This problem can be implemented in PAKCS by means of the test and generate pattern [1]:

[1] This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-2.

[2] Email: `{bbr,sebf,fhu}@informatik.uni-kiel.de`

```
pyt | a*a+b*b=:=c*c &
      c =:= member [1..] & b =:= member [1..c] & a =:= member [1..c]
     = (a,b,c)
  where a,b,c free

member (y:ys)  = y ? member ys
```

First, the search tests whether a combination of a, b and c is a Pythagorean triple. Since PAKCS cannot guess natural numbers this test is suspended by means of residuation for external functions `(+)` and `(*)`. Now the programmer has to add good generator functions which efficiently enumerate the search space.

If, like in PAKCS, a depth first search strategy is used, it is especially important to restrict the search space for a and b to a finite domain. In practical applications, finding good generator functions is often much more complicated than in this simple example.

Moreover, residuation sacrifices completeness and detailed knowledge of internals is required to understand why the following very similar definition produces a run-time error.

```
pyt' | a*a + b*b =:= c*c = generate a b c where a,b,c free
generate a b c
  | c =:= member [1..] & b =:= member [1..c] & a =:= member [1..c]
   = (a,b,c)
```

On the other hand, the most beautiful standard examples for the expressive power of narrowing are defined for *Peano numbers*:

```
data Peano = O | S Peano

add O     m = m
add (S n) m = S (add m n)

mult O     _ = O
mult (S n) m = add m (mult m n)
```

These functions cannot only be used for there intended purpose. We can also invert them to define new operations. For instance, the subtraction function can be defined by means of `add`:

```
sub n m | add r m =:= n = r
  where r free
```

Note, that we obtain a partial function, since we did not define a Peano representation of negative numbers. By means of narrowing, a solution for the constraint `add r m =:= n` generates a binding for `r`. This binding is the result of `sub`.

For a solution of the Pythagorean triples, it is much easier to use Peano numbers instead of predefined integers. We can easily define a solution to this problem as follows:

```
pyt | (a `mult` a) `add` (b `mult` b) =:= (c `mult` c) = (a,b,c)
  where a,b,c free
```

It is not necessary to define any generator functions at all. In combination with breadth first search, all solutions are generated.

Furthermore, if the result c of the Pythagorean equation is already known (e.g., $c = 20$) and you are only interested in computing a and b, then it is even possible to compute this information with depth first search. The given equation already restricts the search space to a finite domain.

```
pyt | c =:= intToPeano 20 &
      (a ‘mult‘ a) ‘add‘ (b ‘mult‘ b) =:= (c ‘mult‘ c) = (a,b,c)
  where a,b,c free
```

We obtain the solutions: $(a, b) \in \{(0, 20), (12, 16), (16, 12), (20, 0)\}$.

Unfortunately, using Peano numbers is not appropriate for practical applications, as a simple computation using Peano numbers in PAKCS shows: computing the square of 1000 already takes 7 seconds. As a consequence, the developers of Curry [7] proposed the external type `Int` in combination with residuating, external functions for numbers.

In this paper we present an alternative approach which is as flexible as Peano numbers but is, in practical applications, almost as efficient as externally defined numbers: a binary encoding of natural numbers. In a second step we extend these natural numbers with an algebraic sign and add zero to represent arbitrary integers.

We do not intend to provide means of solving complex numeric equations ("number crunching") by narrowing. The purpose of the presented approach is to improve the integration of numbers into the language. Using our representation of numbers, the programmer does no longer have to worry about floundering goals when applying functional logic programming techniques like those presented in [1].

2 Numbers and Numeric Operations

2.1 Natural Numbers

Our representation of natural numbers, i.e., numbers greater than zero, is defined as follows. We consider zero and negative numbers in Subsection 2.2.

```
data Nat = IHi | O Nat | I Nat
```

Natural numbers are represented in binary notation with the *least significant bit* first. The *most significant bit* is always one and, therefore, denoted `IHi`. The main reason to define natural numbers without zero is the unique representation. When zero is present as a terminal constructor of type `Nat` any number is representable in infinitely many ways with leading zeros. This is a serious obstacle for functional logic languages since the search spaces to derive numbers are always infinite in that case. The values of type `Nat` can be related to natural numbers as follows:

$$\begin{aligned} \texttt{IHi} &\mathrel{\hat{=}} 1 \\ \texttt{O}\ (nat(n)) &\mathrel{\hat{=}} 2n \\ \texttt{I}\ (nat(n)) &\mathrel{\hat{=}} 2n+1 \end{aligned}$$

It is straight forward to define the successor function on this data type.

```
succ :: Nat -> Nat
succ IHi   = O IHi          1 + 1 = 2 · 1
succ (O n) = I n            2n + 1 = 2n + 1
succ (I n) = O (succ n)     (2n + 1) + 1 = 2 · (n + 1)
```

After each rule in the definition of succ we give an equational reasoning for its correctness. This is one part of the proof that our functions implement the intended semantics. In general, the proof consists in combining three considerations:

- completeness of pattern matching
  This is in most cases directly observable in the given definitions.
- correctness of defined rules
  This is demonstrated by equations added to each rule. To check the equations, it is important to remember that all occurring variables stand for *positive natural numbers.*
- termination
  Because of laziness our functions are normally also defined on infinite numbers. Therefore, we distinguish two categories of termination. In this section these categories only apply to deterministic expressions. Considerations with respect to narrowing will follow in Section 3.

(i) nf-termination:
We call an expression nf-terminating iff it evaluates to a *normal form* in finitely many steps. We call a function f "nf-terminating for arguments of categories $C_1 \dots C_n$" iff the application of f to arguments of that categories is a nf-terminating expression. For instance, we will show below that succ nf-terminates if its argument is nf-terminating. All of the defined functions nf-terminate for nf-terminating arguments because they inductively descend on at least one of these arguments.

(ii) hnf-termination:
We call an expression hnf-terminating iff it evaluates to a *head normal form* of arbitrary depth in finitely many steps. Analog to nf-termination we will also say that a function hnf-terminates for certain categories of arguments. E.g., succ hnf-terminates if its argument hnf-terminates.

The function succ nf-terminates for nf-terminating input since it descends on its only argument. Moreover, each rule directly produces a head normal form, yielding that succ hnf-terminates for hnf-terminating input.

The function succ is used to easily build add on natural numbers.

```
add :: Nat -> Nat -> Nat
IHi `add` m   = succ m                 1 + m = m + 1
O n `add` IHi = I n                    2n + 1 = 2n + 1
O n `add` O m = O (n `add` m)          2n + 2m = 2 · (n + m)
O n `add` I m = I (n `add` m)          2n + (2m + 1) = 2 · (n + m) + 1
I n `add` IHi = O (succ n)             (2n + 1) + 1 = 2 · (n + 1)
I n `add` O m = I (n `add` m)          (2n + 1) + 2m = 2 · (n + m) + 1
I n `add` I m = O (succ n `add` y)     (2n + 1) + (2m + 1) = 2 · (n + 1 + m)
```

For implementations making use of symmetric pattern matching, the first equation should be spelled out to match the second and fifth equation plus a case for `IHi` and `IHi`. The languages considered here all perform a left-to-right matching for this example and, hence, the given definition is sufficient.

Function `add` nf-terminates for nf-terminating input as it inductively descends both arguments. Hnf-termination is also given for hnf-terminating arguments, as each rule directly produces an hnf or calls the hnf-terminating function `succ`.

```
mult :: Nat -> Nat -> Nat
IHi `mult` m = m                        1 · m = m
O n `mult` m = O (n `mult` m)           (2n) · m = 2 · (n · m)
I n `mult` m = O (n `mult` m) `add` m   (2n + 1) · m = 2 · (n · m) + m
```

This definition is clearly asymmetric. However, the following consideration shows that the result symmetrically relies on both arguments.

`mult` nf-terminates iff both arguments nf-terminate as it descends the first argument and finally yields the second argument as a result. Hnf-termination is also given for hnf-terminating arguments, since either one of those arguments is returned (first rule) or a constructor is produced directly (second rule) and because `add` directly produces an hnf for the call `O` e_1 `` `add` `` e_2 if e_2 is hnf-terminating, regardless of e_1 (third rule).

Next is comparing two numbers. Traditionally, comparing in a functional logic language maps to the ordering type.

```
data Ordering = LT | EQ | GT
```

Respectively, the constructors denote the results "less than", "equal" and "greater than". We also use the tests corresponding to this data definition, i.e., `isEQ`, `isLT`, `isGT :: Ordering -> Bool`. It is convenient to simply map two arguments on this type and formulate the relations `(<=)`,`(<)`,`(>=)`,`(>)` in terms of a comparison function like `compNat` or `compare` which is defined later. `compNat` compares two natural numbers as follows. We denote by $\circ$ any of the relations $<, =, >$.

```
compNat :: Nat -> Nat -> Ordering
compNat IHi IHi   = EQ                1 = 1
compNat IHi (O _) = LT                1 < 2n
compNat IHi (I _) = LT                1 < 2n + 1
compNat (O _) IHi = GT                2n > 1
compNat (O n) (O m) = compNat n m     2n ∘ 2m ⇔ n ∘ m
compNat (O n) (I m)
  | isEQ cmpnm = LT                   n = m ⇒ 2n < 2m + 1
  | otherwise  = cmpnm                (n > m ⇒ 2n > 2m + 1)∧
 where cmpnm = compNat n m            (n < m ⇒ 2n < 2m + 1)
compNat (I _) IHi = GT                2n + 1 > 1
compNat (I n) (O m)
  | isEQ cmpnm = GT                   n = m ⇒ 2n + 1 > 2m
  | otherwise  = cmpnm                (n > m ⇒ 2n + 1 > 2m)∧
 where cmpnm = compNat n m            (n < m ⇒ 2n + 1 < 2m)
compNat (I n) (I m) = compNat n m     (2n + 1) ∘ (2m + 1) ⇔ n ∘ m
```

Because the result type `Ordering` is not a complex type hnf-termination and nf-termination coincide and we simply speak of termination. Since `compNat` *simultaneously* descends on both arguments, it terminates whenever at least one of its arguments is nf-terminating and the other is at least hnf-terminating. Note that the comparison only takes as many steps as the size of *the smaller* argument. This is a nice property helping on efficiency.

Now we define `(<=),(<),(>=),(>)` as follows:

```
n <  m = isLT (compare n m)
n >  m = isGT (compare n m)
n <= m = not (n > m)
n >= m = not (n < m)
```

For Curry, note that by defining the relations on base of `compare`, we obtain a narrowable definition of the comparing operations instead of a rigid one. We think that having numbers as external base type is one of the main reasons why these operations where defined rigidly in Curry in the first place.

We can do better, however, than defining

```
n == m = isEQ (compare n m)
n /= m = not (n == m)
```

by sticking to the usual definition of `(==)` which directly compares the constructors. For non-equal values this implementation can even yield a result without evaluating any of its arguments completely.

We now turn to the last operator solely defined on natural numbers. This is the function `pred` which computes the predecessor of a given natural number. Of course, this operator can only be defined partially. This partiality will then lead to the extension of our numbers to *integers* in the next section.

```
pred (O IHi)      = IHi          2 − 1 = 1
pred (O n@(O _))  = I (pred n)   2 · 2n − 1 = 2 · (2n − 1) + 1
pred (O (I n))    = I (O n)      2 · (2n + 1) − 1 = 2 · 2n + 1
pred (I n)        = O n          2 · n + 1 − 1 = 2 · n
```

`pred` nf-terminates on nf-terminating input because it descends on its argument. For hnf-terminating input it hnf-terminates as each rule produces a constructor immediately. `pred` fails if its argument is `IHi`.

The function `pred` is a good example to illustrate that our categorization captures subtle differences. Should we, for instance, use the alternative definition

```
pred' x | succ y =:= x = y where y free
```

hnf-termination for hnf-terminating input would not be given. The reason is that `(=:=)` demands the full evaluation of `x` to normal form.

`pred` is important for two reasons. For one it can be used to define the useful `(n+1)` patterns in an unproblematic way. These patterns can either simply be taken as syntactic sugar for a *function pattern* [2] `(succ n)` or be translated to a simple case distinction:

```
f (n+1) = e     ⇒   f (I n')     = let n=O n'    in e
                    f n'@(O _)   = let n=pred n' in e
```

Note, that in the context of *positive* natural numbers, this pattern has a slightly different semantics than in Haskell: the smallest value which `n` can be bound to is 1, not 0.

Secondly, `pred` is used in the definition of subtraction. Subtraction naturally leads to the domain of integers.

2.2 Integers

We define integers as *signed* natural numbers.

```
data Int = Pos Nat | Zero | Neg Nat
```

We first define the basic operations $(+1), (-1)$ and $(\cdot 2)$ from which we will build the more complex operations. In the equations we employ the usual signed notation for integers, i.e., $-n$ for the negative number with absolute value n.

```
inc, dec, mult2 :: Int -> Int
inc Zero = Pos IHi                 0 + 1 = 1
inc (Pos n) = Pos (succ n)         n + 1 = n + 1
inc (Neg IHi) = Zero               −1 + 1 = 0
inc (Neg (O n)) = Neg (pred (O n)) −2n + 1 = −(2n − 1)
inc (Neg (I n)) = Neg (O n)        −(2n + 1) + 1 = −2n

dec Zero = Neg IHi                 0 − 1 = −1
dec (Neg n) = Neg (succ n)         −n − 1 = −(n + 1)
dec (Pos IHi) = Zero               1 − 1 = 0
dec (Pos (O n)) = Pos (pred (O n)) 2n − 1 = 2n − 1
dec (Pos (I n)) = Pos (O n)        2n + 1 − 1 = 2n

mult2 Zero    = Zero               0 · 2 = 0
mult2 (Pos n) = Pos (O n)          n · 2 = 2n
mult2 (Neg n) = Neg (O n)          −n · 2 = −(2n)
```

All three functions `inc`, `dec` and `mult2` perform a simple pattern matching before directly returning a head normal form. They therefore inherit hnf- and nf-termination from their arguments and the functions `pred` and `succ`, respectively.

Employing the three basic operations we can define subtraction on natural numbers in a concise way. Subtraction maps two natural numbers to an integer.

```
sub :: Nat -> Nat -> Int
IHi `sub` m   = inc (Neg m)                1 − m = (−m) + 1
O n `sub` IHi = Pos (pred (O n))           2n − 1 = 2n − 1
O n `sub` O m = mult2 (n `sub` m)          2n − 2m = 2 · (n − m)
O n `sub` I m = dec (mult2 (n `sub` m))    2n − (2m + 1) = 2 · (n − m) − 1
I n `sub` IHi = Pos (O n)                  2n + 1 − 1 = 2n
I n `sub` O m = inc (mult2 (n `sub` m))    2n + 1 − 2m = 2 · (n − m) + 1
I n `sub` I m = mult2 (n `sub` m)          2n + 1 − (2m + 1) = 2(n − m)
```

If both arguments nf-terminate then `sub` also nf-terminates as `sub` descends on both arguments and calls only functions which also nf-terminate for this category of arguments. Moreover, if the first argument nf-terminates and the second at least hnf-terminates then `sub` hnf-terminates because it descends on the first argument, finally calling the hnf-terminating function `inc` (first rule) and it calls only hnf-terminating functions on the right hand sides of the remaining rules.

Employing `add,sub,mult` on natural numbers, we can simply define their counterparts on integers.

```
(+), (-), (*) :: Int -> Int -> Int
Pos n + Pos m = Pos (n `add` m)    n + m = n + m
Neg n + Neg m = Neg (n `add` m)    −n + (−m) = −(n + m)
Pos n + Neg m = n `sub` m          n + (−m) = n − m
Neg n + Pos m = m `sub` n          −n + m = m − n
Zero  + n     = n                  0 + n = n
n@(Pos _) + Zero = n               n + 0 = n
n@(Neg _) + Zero = n               n + 0 = n

n - Neg m = n + Pos m              n − (−m) = n + m
n - Pos m = n + Neg m              n − m = n + (−m)
n - Zero  = n                      n − 0 = n

Pos n * Pos m = Pos (n `mult` m)   n · m = n · m
Pos n * Neg m = Neg (n `mult` m)   n · (−m) = −(n · m)
Neg n * Neg m = Pos (n `mult` m)   −n · (−m) = n · m
Neg n * Pos m = Neg (n `mult` m)   −n · m = −(n · m)
Zero  * _     = Zero               0 ∗ n = 0
Pos _ * Zero  = Zero               n ∗ 0 = 0
Neg _ * Zero  = Zero               n ∗ 0 = 0
```

All termination properties are inherited from the operations on natural numbers.

Likewise we can build `compare` on integers on the comparison of natural numbers.

```
compare :: Int -> Int -> Ordering
compare Zero    Zero    = EQ          0 = 0
compare Zero    (Pos _) = LT          0 < n
compare Zero    (Neg _) = GT          0 > −n
compare (Pos _) Zero    = GT          n > 0
compare (Pos n) (Pos m) = compNat n m n ∘ m ⇔ n ∘ m
compare (Pos _) (Neg _) = GT          n > −n
compare (Neg _) Zero    = LT          −n < 0
compare (Neg _) (Pos _) = LT          −n < m
compare (Neg n) (Neg m) = compNat m n −n ∘ −m ⇔ m ∘ n
```

Again all termination properties are inherited.

2.3 *Division and Modulo*

Last we want to introduce division and modulo operations in the following sense.

$$n = m * d + r \wedge r < m \Leftrightarrow div(n, m) = d \wedge mod(n, m) = r$$

We only add these definitions for completeness but we are not yet satisfied with the implementation. Therefore we omit completeness, termination and narrowing results for the division and modulo operation.

Analogous to the previous sections we first define simple basic operations to build on. First the partially defined shift operation corresponds to what is known as a "logical right shift".

```
shift :: Nat -> Nat
shift (O n) = n
shift (I n) = n
```

Obviously, shift nf- (hnf-)terminates whenever its argument nf- (hnf-) terminates.

Next, we define the special case of modulo 2.

```
mod2 :: Nat -> Int
mod2 IHi   = Pos IHi  $1 = 2 \cdot 0 + 1 \wedge 1 < 2$
mod2 (O _) = Zero     $2n = 2n + 0 \wedge 0 < 2$
mod2 (I _) = Pos IHi  $2n + 1 = 2n + 1 \wedge 1 < 2$
```

`mod2` nf-terminates if its argument hnf-terminates.

We can now specify the general operation which computes div and mod for given natural numbers.

```
divmod :: Nat -> Nat -> (Int,Int)
divmod x y
  | y==IHi    = (Pos x,Zero)          $x = x \cdot 1 + 0 \wedge 0 < 1$
  | otherwise =
      case compNat x y of
        EQ -> (Pos IHi,Zero)          $x = y \Rightarrow x = 1 \cdot y + 0 \wedge 0 < y$
        LT -> (Zero, Pos x)           $x < y \Rightarrow x = 0 \cdot y + x \wedge x < y$
        GT -> case divmod (shift x) y of
                (Zero,_) -> (Pos IHi,x `sub` y)
                (Pos d,Zero) -> (Pos (O d),mod2 x)
                (Pos d,Pos m) ->
                  case divmod (carryover x m) y of
                    (Zero,m')   -> (Pos (O d),m')
                    (Pos d',m') -> (Pos (O d `add` d'),m')
  where
    carryover (O _) n = O n
    carryover (I _) n = I n
```

3 Narrowing

With respect to narrowing we are interested in the kind of equations we are able to solve. For that purpose we make some conjectures enabling us to deduce solvability properties from the termination properties which were given with the operation definitions. The general discussion of these conjectures is beyond the scope of this paper.

First, we extend our categories of termination.

- We call an expression *finitely branching* if its evaluation will induce only a finite number of non-failing non-deterministic branches.
- We add to the notion of nf-termination of an expression e the requirement that each branch induced by the evaluation of e nf-terminates.
- We add to the notion of hnf-termination of an expression e the requirement that the evaluation of e only introduces a finite number of non-deterministic branches before producing an hnf in each non-failing branch.

Note that the search space of an hnf-terminating expression may still be infinite. For example the expression `nat` with the following definition is hnf-terminating.

```
nat = O ? S nat
```

In the following we will denote the combination "nf-terminating and finitely branching" by simply saying "finite". A simple example of an expression which is nf-terminating but not finite is `zeros` with the definition

```
zeros = O ? zeros
```

Moreover, because free variables can only be bound to finite ground terms, we classify them as nf-terminating, although not as "finitely branching".

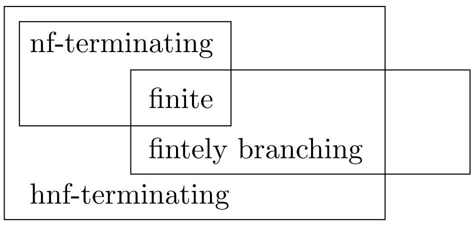

Fig. 1. The Categories and their Relations of Inclusion

For convenience, Figure 1 displays the relations between the categories introduced.

Conjecture 3.1 (Terminating Search)
Let e be a finite expression. Then a depth first search will produce all values of e in finite time and finally terminate.

For example, the unification `(=:=)` built-in for Curry is finite if one of its arguments is finite and the other one is hnf-terminating.

If the conjecture holds, information about nf-termination directly corresponds to solvability of equations. For ease of reference we have collected this kind of information about the numeric operations presented in this paper in the following table.

Expression built with	Argument requirement for nf-terminating result
`succ, inc, pred, dec, mult2`	nf-terminating
`add, (+)`	both nf-terminating
`mult, (*)`	both nf-terminating
`compNat, compare`	any nf-terminating, other hnf-terminating
`(<), (<=), (>), (>=)`	any nf-terminating, other hnf-terminating
`sub, (-)`	both nf-terminating
`(=:=)`	any nf-terminating, other hnf-terminating

Conjecture 3.2 (Complete Search)
Let e be an nf-terminating or hnf-terminating expression. Then an or-fair strategy like breadth first search will produce all values of e but may not terminate after producing the last value.

`(=:=)` terminates (hnf and nf coincide for type `Success`) if one of its arguments nf-terminates and the other one is hnf-terminating.

The following table lists the hnf-termination properties of the defined numeric operations.

Expression built with	Requirement to Argument for hnf-terminating result
`succ, inc, pred, dec, mult2`	hnf-terminating
`add, (+)`	both hnf-terminating
`mult, (*)`	both hnf-terminating
`compNat, compare`	any nf-terminating, other hnf-terminating
`(<), (<=), (>), (>=)`	any nf-terminating, other hnf-terminating
`sub, (-)`	first nf-terminating, other hnf-terminating
`(=:=)`	any nf-terminating, other hnf-terminating

If both conjectures hold, we can now infer whether or not certain equations can be solved and by which strategy. For example, we again consider the Pythagorean triples. Breadth first search completely finds the values of the following expression, where we assume `a,b,c` to be free variables:

```
a*a + b*b =:= c*c &> (a,b,c)
```

For a fixed `c`, e.g. beforehand bound by `c =:= 20`, all solutions are also computed by depth first search.

On the other hand, the following expression does not terminate after all solutions are computed:

```
x+x=:=x
```

4 Experiments

We have made some experiments to evaluate the efficiency of the arithmetic operations. Our benchmarks were run on an AMD Athlon^{TM} XP 3000+ with 2 GHz. It is no surprise that our implementation of integers is clearly outperformed by primitive arithmetic operations. It is, however, efficient enough to justify its use in a functional logic language – given the additional possibilities outlined in the previous sections. We think that high performance of arithmetic computations is not as important as a seamless integration of arithmetics in the functional logic paradigm.

We have compared a Haskell implementation of our operations with GHC's native implementation of `Integers`[3]. In addition, we have compared a Curry implementation of our operations with the primitive arithmetic operations integrated in PAKCS [6]. As we do not aim at comparing the Haskell with the Curry implementation, we use different input values to obtain significant results in both systems. To measure only the time necessary for arithmetic operations, we employ the following test function:

```
test _  x 1     1     = True
test op x 1     (m+1) = test op x x m
test op x (n+1) m     = eval (op n m) && test op x n m

eval n = n==n

bench op x = test op x x x
```

We have called `bench` with the arithmetic operators `(+)`, `(-)`, `(*)`, `div` and `mod`. To measure the overhead imposed by this test function, we have called `bench` with the tuple constructor `(,)` and subtracted the runtime for this computation from all runtimes for the arithmetic operations.

We have employed the latest versions of GHC and PAKCS for our benchmarks. For GHC we have instantiated the parameter `x` of `bench` with 4000 and for PAKCS with 300 to obtain significant results – they are depicted in Table 1. The run times of our arithmetic operations and the built-in functions – after subtracting overhead – are shown in one column. The slowdown of our library is depicted next to the run times.

The measured slowdown imposed by multiplication using GHC is significantly larger than the slowdown for the other operations. We believe that it stems from the larger results of the computation that have to be consumed by `eval`. It also suggests that our definition of multiplication is far from optimal and we are open to suggestions. In PAKCS – where the largest computed value is significantly smaller – there is no such slowdown.

We can see that our arithmetic library is roughly up to 20 times slower than

[3] Note that the size of `Int` values is limited in Haskell.

	GHC `bench op 4000`		PAKCS `bench op 300`	
`op`	seconds	slowdown	seconds	slowdown
`(+)`	9.74/0.45	21.6	3.35/1.03	3.25
`(-)`	9.39/1.15	8.17	7.03/1.01	6.96
`(*)`	62.1/0.52	119	12.8/1.02	12.5
`div`	12.9/2.24	5.77	19.4/1.02	19.0
`mod`	20.5/2.24	9.13	21.9/1.04	21.0

Table 1
Experimental Results

primitive arithmetic operations. Also, our library compiled with GHC is faster than the primitive arithmetic operations of PAKCS. As Curry programmers do not complain about the arithmetic performance of PAKCS, we feel confident that they would also not complain about a Haskell-based Curry system [3] that exclusively uses our narrowable arithmetic operations. Because of the additional support for narrowing, it could even be an option for PAKCS to use our library instead of a primitive implementation of integers. In addition, the time spent in arithmetic computations is usually only a fraction of the run time of functional (logic) programs. Even in our benchmarks, which were designed to make mostly arithmetic operations, the overhead measured with `(,)` as `op` was much larger than the time used to compute the result of the arithmetic computations in most cases.

5 Conclusions

We propose a different implementation of arithmetic operations for functional logic programming languages. Currently, declarative programming languages employ external functions to do arithmetics, which results in serious limitations in the context of functional logic programming. Free variables cannot be guessed in arithmetic operations and need to be generated explicitly.

We show that a binary encoding of integers as algebraic datatype eliminates this need. We define all usual arithmetic operations on this datatype and show that our implementation can be used to narrow unknown arguments to these operations. Practical experiments show that our implementation produced significant overhead compared with primitive operations. However, we feel that programmers willingly accept this overhead given the new opportunities of narrowing.

Our implementations of the arithmetic operations are fairly straightforward. We curiously expect suggestions to improve them in future work. Also, our conjectures w.r.t. the termination analysis may lead to interesting future research.

References

[1] Antoy, S. and M. Hanus, *Functional logic design patterns*, in: *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)* (2002), pp. 67–87.

[2] Antoy, S. and M. Hanus, *Declarative programming with function patterns*, in: *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)* (2005), pp. 6–22.

[3] Braßel, B. and F. Huch, *Translating Curry to Haskell*, in: *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)* (2005), pp. 60–65.

[4] Hanus, M., *On the completeness of residuation*, in: *Proc. Joint International Conference and Symposium on Logic Programming* (1992), pp. 192–206.

[5] Hanus, M., *The integration of functions into logic programming: From theory to practice*, Journal of Logic Programming **19&20** (1994), pp. 583–628.

[6] Hanus, M. et al., *PAKCS: The Portland Aachen Kiel Curry System (version 1.8.0)*, Available at URL `http://www.informatik.uni-kiel.de/~pakcs/` (2007).

[7] Hanus (ed.), M., *Curry: An integrated functional logic language (vers. 0.8.2)*, Available at `http://www.informatik.uni-kiel.de/~curry` (2006).