On Computing with Types

Paul Tarau
Department of Computer Science and Engineering
University of North Texas
tarau@cs.unt.edu

David Haraburda
Department of Computer Science and Engineering
University of North Texas
dh0142@unt.edu

ABSTRACT

We express in terms of binary trees seen as Gödel System **T** types (with the empty type as the only primitive type) arithmetic computations within time and space bounds comparable to binary arithmetic and derive an efficiently testable total ordering on types, isomorphic to the ordering of natural numbers.

A few novel algorithms are derived in the process, that enable arithmetic computations with type trees.

The use of a Haskell type class describing the "axiomatization" of the shared structure present in System **T**'s type language and natural numbers, together with Haskell instances representing "twin" interpretations connected by an iso-functor that transports operations between the two instances, provides instant empirical testability.

The self-contained source code of the paper is available at http://logic.cse.unt.edu/tarau/research/2012/stypes.hs .

Categories and Subject Descriptors

D.3.3 [**PROGRAMMING LANGUAGES**]: Language Constructs and features—*Data types and structures* [Polymorphism] [Recursion]

General Terms

Algorithms, Languages, Theory

Keywords

arithmetic computations with type trees, polymorphic "axiomatizations" with type classes, Gödel System **T** types

1. INTRODUCTION

In [12, 13] a framework has been introduced to organize isomorphisms between data types as a *groupoid*, i.e. a category [9] where every morphism is an isomorphism, with objects provided by the data types and morphisms provided by their bijective transformations.

SAC'12, Riva del Garda (Trento), Italy, March 26-30, 2012

In their basic form, such isomorphisms show up as *encodings* to/from some simpler and easier to manipulate representations (for instance, natural numbers).

Among their practical uses, encodings between data types can provide a variety of services ranging from free iterators and random objects to data compression and succinct representations.

In [15] it has been shown that Haskell *type classes* [17] can be used to share various operations between isomorphic instances with emphasis on a "shared axiomatization" of hereditarily finite sets and Peano arithmetic.

In this paper we focus on binary trees with empty leaves, seen as a minimalist type language, based on Gödel's System **T**. We show that they provide, through a type class based abstraction layer, universal building blocks representing as instances both natural numbers and types trees. We extend this mechanism in the form of a chain of type classes and express arithmetic computations as well as in Turing-equivalent combinator reductions.

Choosing the simplest possible type language (specifically, the *free magma* generated by the *arrow* operation on an empty basic type) as a foundation for data and code representations is partly motivated by the use of type theory as a constructive alternative to conventional predicate calculus based axiomatizations. This has resulted in powerful proof assistants like Coq [10], based entirely on constructive formalizations of type theory [2]. For instance, one can use the isomorphism between types and natural numbers, that provides a total ordering on types, as a generic means to provide termination proofs for type inference algorithms and type-based proof systems.

At the same time, with practical uses for arbitrary size integer arithmetic in mind, we will focus on keeping the asymptotic complexity of various operations similar to that of operations on conventional bitstrings.

Two other applications focus on instantiating our chain of type classes to parenthesis languages and to provide a name-free encoding of lambda expressions.

The paper is organized as follows. Section 2 introduces the use of type classes and their instances as a mechanism to share specifications and implementations between heterogeneous data types. Section 3 describes the type language used as a basic building block through the paper. Section 4 defines generic successor and predecessor operations on types and and proves their correctness. Section 5 provides implementations of arithmetic operations, with efficiency comparable to ordinary binary arithmetic. Section 6 derives an efficiently testable total ordering on types. Section 7 provides

encodings for the S,K and Rosser's X combinator calculi. Sections 8 and 9 describe encodings of lambda expressions and parenthesis languages. Sections 10 and 11 discuss related work and conclude the paper.

2. TESTABILITY AND CORRECTNESS WITH "TWIN INSTANCES" OF TYPE CLASSES

Haskell's *type classes* [17] are a good approximation of axiom systems as they allow one to describe properties and operations generically i.e. in terms of their action on objects of a parametric type. Haskell's *instances* approximate *interpretations* [6] of such axiomatizations by providing implementations of primitive operations and by refining and possibly overriding derived operations with more efficient equivalents.

Traditionally correctness is proven by matching "implementations" against a "specification". Our specification will be provided by a type class. A pair of instances will provide alternative implementations constrained to be isomorphic by sharing generic constructors and destructors.

Empirical correctness/instant testability is simply agreement between the two instances on various generic operations. Formal correctness, for instance inductive proofs of termination and other program properties, can be derived by using the instance with known properties (for instance $\mathbb{N}$) as a witness for corresponding operations in the implementation of the other instance.

3. A MINIMALIST TYPE LANGUAGE

The class `PureTypes` assumes only the `Read/Show` superclasses needed for input/output. An instance of this class is required to implement the following primitive operations:

```
class (Read n,Show n) ⇒ PureTypes n where
  empty :: n
  isEmpty :: n→Bool
  arrow :: n→n→n
  from,to :: n→n
```

The `PureTypes` type class also provides to its instances generic implementations of the following derived operations:

```
  isArrow :: n→Bool
  isArrow = not . isEmpty

  eq :: n → n → Bool
  eq x y | isEmpty x && isEmpty y = True
  eq x y | isEmpty x || isEmpty y = False
  eq x y = eq (from x) (from y) && eq (to x) (to y)
```

While one could have also derived equality from the Haskell `Eq` class, we have defined it here to clarify our assumptions. The following properties describe the "axioms" connecting these operations:

```
pure_type_prop1 :: (PureTypes n) ⇒ n → Bool
pure_type_prop2 :: (PureTypes n) ⇒ n → n → Bool

pure_type_prop1 z = isEmpty z ||
  eq z (arrow (from z) (to z))
pure_type_prop2 x y = eq x (from z) && eq y (to z)
  where z = arrow x y
```

It is convenient at this point, as we target multiple interpretations materialized as Haskell instances, to provide a polymorphic converter between two different instances of the type class `PureTypes`. The function `view` allows converting between two different PureTypes instances, generically.

```
view :: (PureTypes a,PureTypes b)⇒a→b
view x | isEmpty x = empty
view x = arrow (view (from x)) (view (to x))
```

Note also that `view` provides iso-functors that commute with `arrow`, `from` and `to` between instances of this type class, i.e. `prop_view` holds $\forall x, y$ of type `PureTypes a, PureType b`.

```
prop_view :: (PureTypes a,PureTypes b) ⇒ a→b→Bool
prop_view x y = iso view view x y where
  iso :: (PureTypes a,PureTypes b) ⇒
         (a→b)→(b→a)→a→b→Bool
  iso f g x y = eq ((f . g) y) y && eq ((g . f) x) x
```

We can build a *model* of the "axiomatization" provided by the type class `PureTypes` as a rooted ordered binary tree type, implemented by `data T`. We can see these binary trees as a representation of System **T** types with an empty base type or simply as the *free magma* of rooted ordered binary trees with one generator, representing empty leaves.

```
infixr 5 :→
data T = E|T :→ T deriving (Read, Show)

instance PureTypes T where
  empty = E

  isEmpty E = True
  isEmpty _ = False

  arrow = (:→)

  from E = undefined
  from (x :→ _) = x

  to E = undefined
  to (_ :→ y) = y

t :: (PureTypes a)⇒a→T
t = view
```

Through a possible Haskell language extension one could think about using `arrow` and `empty` as patterns when occurring on the left side of a definition and as data constructors when occurring on the right side of a definition in a way similar to the handling of data constructors `:->` and `E`.

We will next define another instance where `arrow`, `from` and `to` are implemented as computations on natural numbers.

After adding the type synonym

```
type N = Integer
```

we observe[1] that usual arithmetic can also be seen as a *model* of the "axiomatization" implicitly described in terms of the constructors and destructors of the type class `PureTypes`.

```
instance PureTypes N where
  empty=0

  isEmpty 0 = True
  isEmpty _ = False

  arrow x y = (2^x)*(2*y+1)

  from x | x>0 =
    if odd x then 0 else 1+(from  (x `div` 2))
```

[1]Assuming the reader will run the code in the paper with `ghci -XTypeSynonymInstances`.

```
  from _ = undefined

  to x | x>0 =
    if odd x then (x-1) `div` 2 else to (x `div` 2)
  to _ = undefined

n :: (PureTypes a)⇒a→N
n = view
```

One can experiment right away with the isomorphism induced by the "twin" views provided by `N` and `T`.

```
*SystemT> map t [0..4]
[E,E :→ E,(E :→ E) :→ E,E :→ (E :→ E),
 ((E :→ E) :→ E) :→ E]
*SystemT> map n it
[0,1,2,3,4]

*SystemT> t 2012
((E :→ E) :→ E) :→ (E :→ (E :→ ((E :→ E)
    :→ (E :→ (E :→ (E :→ (E :→ E)))))))
*SystemT> n it
2012
```

Fig 1 shows the type tree (represented as a DAG by sharing identical subtrees) generated by `t 2012`. Note that `from` edges are labeled with `0` and `to` edges are labeled with `1`.

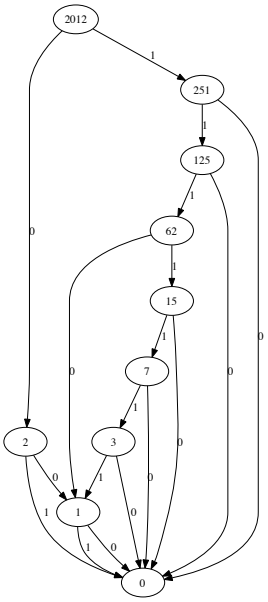

Figure 1: The type tree associated to 2012

Besides providing intuitions on the next steps involving arithmetic operations with instances of the class `PureTypes`, this interpretation suggests deriving a generic implementation of Peano arithmetic.

4. PEANO ARITHMETIC: SUCCESSOR AND PREDECESSOR OPERATIONS

As the first step of an "axiomatization" of Peano arithmetic we implement the successor and predecessor functions `s` and `p` in a class extending `PureTypes`.

```
class PureTypes n⇒PeanoArith n where
  s :: n→n

  s z | isEmpty z = arrow empty empty
  s z | isEmpty (from z) = arrow (s (from (s (to z))))
    (to (s (to z)))
  s z = arrow empty (arrow (p (from z)) (to z))

  p :: n→n

  p z | isEmpty (from z) && isEmpty (to z) = empty
  p z | isEmpty (from z) = arrow (s (from (to z)))
    (to (to z))
  p z = arrow empty (p (arrow (p (from z)) (to z)))
```

It can be proven by structural induction[2] that the following holds

PROPOSITION 1. $\forall\ x\ p\ (s\ x) = x$ *and* $\forall\ x\ x \neq e \Rightarrow s\ (p\ x) = x$.

and more generally that Peano's axioms hold for arbitrary instances of PureTypes.

After adding `T` and `N` as instances of `PeanoArith` one can observe experimentally that that they agree on `s` and `p`.

```
instance PeanoArith T
instance PeanoArith N
```

The following proposition states the correctness of the definitions of `s` and `p`.

PROPOSITION 2. *Let* `t,t'` *be of type* `T` *and* `n,n'` *of type* `N` *such that* `t,t'` *are obtained in* `T` *using the same arrow operations as* `n,n'` *in N. Then* `t'=s t` *if and only if* `n'=s n` *and* `t'=p t` *if and only if* `n'=p n`.

PROOF. Given that `N` and `T` are instances of the same type class connected by the isomorphism defined by the function `view`, we will describe the computations on instance `N` using standard mathematical notation. As `s` and `p` are mutually recursive, we conduct our proof using simultaneous structural induction. For convenience we first introduce several new notations for our operations on the type `N`. First we denote the `arrow` operation on `N` as

$$\langle x, y\rangle = 2^x(2y+1),$$

calling $\langle x, y\rangle$ the pairing of x and y. From this pairing we define the `from` and `to` operations as projection functions respectively,

$$\pi_x = \nu_2(z)$$
$$\pi_y = \frac{2^{-\nu_2(z)}z - 1}{2}.$$

The notation $\nu_p(z)$ is called the (additive) *p-adic valuation* or the *p-adic order* of z, and is used here as a succinct way to write "the highest power of 2 that divides z."

Basis. Given an empty instance which is zero by definition, the function `s` "pairs" two empty instances:

$$\langle 0, 0\rangle = 2^0\,[2(0)+1] = 1.$$

Similarly, `p` acts on the inverse—given that both `from` and `to` are empty, the empty instance (zero) is returned—so our base case holds on both `s` and `p`.

Case 1 (Induction). `(from z)` is empty. In this instance, z is an odd number because for any y used in the pairing to construct z we have

$$\langle 0, y\rangle = (2y+1),$$

[2]The induction principle used is: "given a predicate P ranging over data objects of an instance of PureTypes, if $P(empty)$ and $\forall x, y\ P(x)$ and $P(y)$ implies $P(arrow\ x\ y)$, then $\forall t\ P(t)$ holds."

the definition of an odd number. Now, to show that `s` and `p` act as expected, we write them using our new notation which is easily manipulated algebraically.

Starting with `s`, we note that `(s (to z))` is used in both arguments to the `arrow`, so we assign its value to an intermediate variable m,

$$m = \frac{2^{-\nu_2(z)}z - 1}{2} + 1.$$

If we then let l and r be the first and second arguments of the `arrow` operation respectively, they may be written in terms of m:

$$\begin{aligned} l &= \nu_2(m) + 1 \\ r &= \frac{2^{-\nu_2(m)}m - 1}{2}. \end{aligned}$$

Now, we simply perform the pairing (the application of `arrow`) and back substitute for m.

$$\begin{aligned} \langle l, r\rangle = 2^l(2r+1) &= 2^{\nu_2(m)+1}\left[2\left(\frac{2^{-\nu_2(m)}m - 1}{2}\right) + 1\right] \\ &= 2^{\nu_2(m)+1}\left[2^{-\nu_2(m)}m\right] \\ &= 2m \\ &= 2\left[\frac{2^{-\nu_2(z)}z - 1}{2} + 1\right] \\ &= 2^{-\nu_2(z)}z + 1 \end{aligned}$$

Finally, recall that we have established z as an odd number, meaning there is no power of 2 that divides it. With $\nu_2(z) = 0$, the equation reduces to $z + 1$.

We handle `p` in the same fashion, letting the m be the value of `(to x)` and l and r the first and second arguments of `arrow`.

$$\begin{aligned} m &= \frac{2^{-\nu_2(z)}z - 1}{2} \\ l &= \nu_2(m) + 1 \\ r &= \frac{2^{-\nu_2(m)}m - 1}{2} \end{aligned}$$

Perform the pairing,

$$\begin{aligned} 2^l(2r+1) &= 2^{\nu_2(m)+1}\left[2\left(\frac{2^{-\nu_2(m)}m - 1}{2}\right) + 1\right] \\ &= 2^{\nu_2(m+1)}\left[2^{-\nu_2(m)}m\right] \\ &= 2m \\ &= 2\left[\frac{2^{-\nu_2(z)}z - 1}{2}\right] \\ &= 2^{-\nu_2(z)}z - 1, \end{aligned}$$

which again reduces to $z + 1$.

Case 2 (Induction). `(from z)` is *not* empty. Here, we recall that a number n is even if there exists another number k such that $n = 2k$. Again, if we consider the pairing $\langle x, y\rangle$ used to construct z and the fact that $x > 0$ (otherwise it would be covered by the previous case), we see that z is indeed a factor of 2.

For `s` we again assign the arguments of the inner `arrow` operation to l and r respectively and let m be their pairing.

$$\begin{aligned} l &= \nu_2(z) - 1 \\ r &= \frac{2^{-\nu_2(z)}z - 1}{2} \\ m = \langle l, r\rangle &= 2^l(2r+1) \\ &= 2^{\nu_2(z)-1}\left[2\left(\frac{2^{-\nu_2(z)}z - 1}{2}\right) + 1\right] \\ &= 2^{\nu_2(z)-1}\left[2^{-\nu_2(z)}z\right] \\ &= 2^{-1}z \end{aligned}$$

Perform the pairing for the outer `arrow` operation:

$$\begin{aligned} \langle 0, m\rangle &= 2^0\left[2(2^{-1}z) + 1\right] \\ &= z + 1. \end{aligned}$$

Conveniently, $\nu_2(z)$ in the exponent cancels out.

The final case for `p` is handled in the same way.

$$\begin{aligned} l &= \nu_2(z) - 1 \\ r &= \frac{2^{-\nu_2(z)}z - 1}{2} \\ m = \langle l, r\rangle &= 2^l(2r+1) \\ &= 2^{\nu_2(z)-1}\left[2\left(\frac{2^{-\nu_2(z)}z - 1}{2}\right) + 1\right] \\ &= 2^{\nu_2(z)-1}\left[2^{-\nu_2(z)}z\right] \\ &= 2^{-1}z \\ \langle 0, m\rangle &= 2^0\left[2(2^{-1}z - 1) + 1\right] \\ &= z - 1. \end{aligned}$$

□

Assuming that `N` is an interpretation of Peano's axioms, one can establish a correspondence between proofs of program properties through an iso-functor from `N` to `T` that transports successors and predecessors. Given that `N` can be seen as a model of the *free successor algebra* with one generator and `T` a model for the *free magma* of binary trees with empty leaves, the operations `n` and `t` provide the two sides of an isomorphism between these two free objects. Note however that `view` implements generically such isomorphisms between instances of `PureTypes`.

5. EMULATING BINARY ARITHMETIC

Our next refinement adds efficient arithmetic operations in the form of a type class extending `PeanoArith`. We start with a few operations that, by deepening the analogy with their twin instance `N`, will provide a view of `T` objects as binary numbers. This view will ensure that arithmetic operations can be performed in the two instances within comparable asymptotic time and space complexity bounds. We first define recognizers for "odd" and "even" objects:

```
class PeanoArith n⇒FastArith n where
  one :: n
  one = arrow empty empty

  isOdd,isEven :: n→Bool
  isOdd x = isArrow x && isEmpty (from x)
  isEven x = isArrow x && isArrow (from x)
```

We also add two constructors that build such "even" and "odd" objects:

```
makeOdd,makeEven :: n→n
makeOdd x = arrow empty x
makeEven = s . makeOdd
```

A destructor that reverses the action of both constructors follows:

```
trim :: n→n

trim x | isEmpty (from x) = to x
trim x = p (arrow (p (from x)) (to x))
```

Using the "twin instance" method, we can test that they do indeed, on integers, what we expect:

```
*SystemT> map t [0..3]
[E, E :→ E, (E :→ E) :→ E, E :→ (E :→ E)]
*SystemT> map isOdd it
[False,True,False,True]
*SystemT> map isOdd [0..3]
[False,True,False,True]

*SystemT> makeOdd 3
7
*SystemT> makeEven 3
8
*SystemT> trim 7
3
*SystemT> trim 8
3
```

The last example shows that `makeOdd` and `makeEven` work on instance `N` as if implemented by $\lambda x \to 2x+1$ and $\lambda x \to 2x+2$ while `trim` works by undoing their action. We can now implement addition and subtraction as follows:

```
add :: n→n→n
add x y | isEmpty x = y
add x y | isEmpty y = x
add x y | isOdd x && isOdd y = makeEven
   (add (trim x) (trim y))
add x y | isOdd x && isEven y = makeOdd
   (s (add (trim x) (trim y)))
add x y | isEven x && isOdd y = makeOdd
   (s (add (trim x) (trim y)))
add x y | isEven x && isEven y = makeEven
   (s (add (trim x) (trim y)))

sub :: n→n→n
sub x y | isEmpty x && isEmpty y = empty
sub x y | not(isEmpty x) && isEmpty y = x
sub x y | not (isEmpty x) && eq x y = empty
sub y x | isEven y && isOdd x = makeOdd
   (sub (trim y) (trim x))
sub y x | isOdd y && isOdd x = makeEven
   (sub (trim y) (s (trim x)))
sub y x | isOdd y && isEven x = makeOdd
   (sub (trim y) (s (trim x)))
sub y x | isEven y && isEven x = makeEven
   (sub (trim y) (s (trim x)))
```

We can define a concept of "size" by adding up the "number of" leaves of the ordered binary tree obtained by recursive decomposition with `from` and `to`.

```
lsize :: n→n
lsize x | isEmpty x = one
lsize x = add (lsize (from x)) (lsize (to x))
```

Efficient multiplication and power computations follow:

```
multiply :: n→n→n
multiply x _ |isEmpty x = empty
multiply _ x |isEmpty x = empty
multiply x y = arrow
  (add (from x) (from y)) (add a m) where
     (tx,ty)=(to x,to y)
     a=add tx ty
     m=double (multiply tx ty)

half,double :: n→n
double = p . makeOdd
half = trim . s

exp2 :: n→n
exp2 x = arrow x empty

pow :: n→n→n
pow _ y | isEmpty y = one
pow x y | isOdd y =
  multiply x (pow (multiply x x) (trim y))
pow x y | isEven y =
  multiply x' (pow x' (trim y)) where
    x'=multiply x x
```

Note the simplicity of `exp2` which is clearly a constant time operation when working on instance `T`. After adding:

```
instance FastArith T
instance FastArith N
```

we can try out and note the agreement on various arithmetic operations between the "twin" views:

```
*SystemT> t 3
E :→ (E :→ E)
*SystemT> n (add (t 3) (t 4))
7
*SystemT> n (multiply (t 3) (t 4))
12
*SystemT> n (sub (t 101) (t 100))
1
*SystemT> n (pow (t 3) (t 4))
81
*SystemT> map exp2 [0..7]
[1,2,4,8,16,32,64,128]
```

Computing with huge numbers

We can confirm empirically that our computations work with very large numbers as follows:

```
*SystemT> let googol = t (pow 10 100)
*SystemT> exp2 (exp2 (exp2 googol))
((((((E :→ E) :→ E)...))))))))))
```

Note also that exp2 is $O(1)$ time and *space* when using a symbolic representation like the instance `T`. This explains why this representation easily accommodates the "tower of exponents" needed for numbers like `exp2 (exp2 (exp2 googol))` which obviously would not fit in the memory of any computer (of the size of the known universe included!) when using bitstring representations.

6. DEFINING A TOTAL ORDER

Ordering can now be provided as a new class `Ordered`. Comparison proceeds by case analysis, the interesting cases being when the order relation is strengthened from `EQ` to `LT` or `GT`:

```
class FastArith n ⇒ Ordered n where
  cmp :: n→n→Ordering
```

```
cmp x y | isEmpty x && isEmpty y = EQ
cmp x y | isEmpty x && not(isEmpty y) = LT
cmp x y | not(isEmpty x) && isEmpty y = GT
cmp x y | isOdd x && isOdd y =
  cmp (trim x) (trim y)
cmp x y | isEven x && isEven y =
  cmp (trim x) (trim y)
cmp x y | isOdd x && isEven y =
  downeq (cmp (trim x)
    (trim y)) where
      downeq EQ = LT
      downeq b = b
cmp x y | isEven x && isOdd y =
  upeq (cmp (trim x)
    (trim y)) where
      upeq EQ = GT
      upeq b = b
```

Finally, boolean comparison operators are defined as follows:

```
lt, gt :: n→n→Bool

lt x y = LT==cmp x y
gt x y = GT==cmp x y
```

We can now implement a fast division and reminder algorithm (returned as a pair):

```
div_and_rem :: n→n→(n,n)

div_and_rem x y | lt x y = (empty,x)
div_and_rem x y | gt y empty = (add (exp2 qt) u,v) where
  exp2leq_then n m = try_to_double n m empty where
    try_to_double x y k | lt x y = p k
    try_to_double x y k = try_to_double x (double y) (s k)
  divstep n m = (q, sub n n') where
    q = exp2leq_then n m
    n' = multiply (exp2 q) m
  (qt,rm) = divstep x y
  (u,v) = div_and_rem rm y

divide,reminder :: n→n→n

divide n m = fst (div_and_rem n m)
reminder n m = snd (div_and_rem n m)
```

After adding:

```
instance Ordered T
instance Ordered N
```

the following can be proven by structural induction using the constructors and destructors defined in class `PureTypes`:

PROPOSITION 3. *Arithmetic operations and order relations as defined in the classes* `FastArith` *and* `Ordering` *induce isomorphisms of additive and multiplicative semigroups and totally ordered sets between instances* `N` *and* `T` *representing respectively the set of natural numbers and the free magma of binary trees with empty leaves. These operations and relations are computed within time complexity bounds equivalent to their conventional bitstring implementations.*

7. REPRESENTING COMBINATORS

We will now move from data representations to code representations and explore encodings for Turing-equivalent formalisms.

7.1 The S,K combinator calculus

The combinators $Kxy = x$ and $Sxyz = (xz)(yz)$ are known to be complete i.e. able to express computations with arbitrary (untyped) lambda terms. We can encode the S and K combinators as follows.

```
class Ordered n ⇒ Combinators n where
  cS,cK :: n
  cS = arrow empty one
  cK = arrow one empty

  isS,isK :: n→Bool
  isS x = isArrow x && isEmpty (from x) && isArrow (to x)
    && isEmpty (from (to x)) && isEmpty (to (to x))
  isK x = isArrow x && isEmpty (to x) && isArrow (from x)
    && isEmpty (from (from x)) && isEmpty (to (from x))
```

Note that we have ensured that the "op-codes" for `cS` and `cK` for S and K are not one a subtree of the other - to avoid spurious reductions. Note also that function application is represented directly as the `arrow` constructor. We can now define a polymorphic evaluator:

```
redSK :: n → n
redSK t | isK t = t
redSK t | isS t = t
redSK t | isArrow t && isArrow (from t)
    && isK (from (from t)) = redSK (to (from t))
redSK t | isArrow t && isArrow (from t) &&
    isArrow (from (from t)) &&
    isS (from (from (from t))) = xzyz where
      x = to (from (from t))
      (y,z) = (to (from t),to t)
      (xz,yz) = (arrow x z,arrow y z)
      xzyz = redSK (arrow xz yz)
redSK t | isArrow t = t' where
    (x,y) = (from t,to t)
    (x',y') = (redSK x,redSK y)
    (z,z') = (arrow x y,arrow x' y')
    t'= if (eq z z') then z' else redSK z'
redSK t = t
```

After adding the I combinator `cI` and booleans `cT` and `cF`

```
cI,cT,cF :: n
cI = arrow (arrow cS cK) cK
cT = cK
cF = arrow cK cI
```

some simple properties can be tested with

```
test_t,test_f:: n
test_t = redSK (arrow (arrow cT cK) cS)
test_f = redSK (arrow (arrow cF cK) cS)
```

7.2 Rosser's `X` combinator calculus

Rosser's `X` combinator [4] is defined as $\lambda f.fKSK$. Assuming that `X` is encoded as `empty`, `X` expresses `S` and `K` as exactly the same binary trees we have used for our encoding of `S` and `K`, i.e. `S = X (X X)` and `K = (X X) X`. We can therefore represent the combinator `X` as an empty leaf, application as the `arrow` constructor, and reuse `redSK` as an evaluator for `X`-expression trees.

```
cX :: n
cX = empty

isX :: n→Bool
isX = isEmpty

redX :: n → n
redX = redSK
```

After adding the instance declarations

```
instance Combinators T
instance Combinators N
```

we can observe that the reduction `redX` matches the definitions of `S` and `K` as expected.

```
*SystemT> redX (arrow (arrow cX cX) cX) :: T
(E :→ E) :→ E
*SystemT> redX (arrow cX (arrow cX cX)) :: T
E :→ (E :→ E)
*SystemT> cK :: T
(E :→ E) :→ E
*SystemT> cS :: T
E :→ (E :→ E)
```

8. A "NAME-FREE" ENCODING OF VARIABLES IN LAMBDA EXPRESSIONS

We encode an occurrence of a variable as a path in the rooted ordered binary type tree leading to an empty leaf. A "variable" in a lambda expression can then be represented simply as a set of such paths. Note that as in the case of *de Bruijn terms*, this representation does not require alpha-conversion as no "names" are used.

The predicate `occursIn`, starting a new type class `LambdaTerms`, checks if x describes a path to an empty leaf seen as the "place" where it occurs. Note that navigation along the path follows the `isOdd` and `isEven` predicates directing respectively left and right. Note also that the invariant that a path should reach empty leaves is enforced by the first two cases of the definition.

```
class Combinators n ⇒ LambdaTerms n where
  occursIn :: n → n → Bool
  occursIn expr x | isEmpty expr && isEmpty x = True
  occursIn expr x | isEmpty x || isEmpty expr = False
  occursIn expr x | isOdd x = occursIn (from expr) (trim x)
  occursIn expr x | isEven x = occursIn (to expr) (trim x)
```

Beta-conversion can be expressed as uniform substitution of a list of occurrences, after checking with `occursIn` that all paths defining a lambda variable lead to empty leaves. Once this constraint has been checked `foldl` can be used to implement uniform substitution, using the function `substWith`.

```
  applyLambda :: [n] → n → n → Maybe n
  applyLambda xs expr val | all (occursIn expr) xs =
   Just (foldl (substWith val) expr xs)
  applyLambda _ _ _ = Nothing

  substWith :: n → n → n → n
  substWith val expr x | isEmpty expr = val
  substWith val expr x | isOdd x = arrow l r where
    l =  substWith val (from expr) (trim x)
    r = to expr
  substWith val expr x| isEven x = arrow l r where
    l = from expr
    r = substWith val (to expr) (trim x)
```

After adding the instance declarations

```
instance LambdaTerms T
instance LambdaTerms N
```

one can test the (single occurrence) substitution operation `substWith`

```
*SystemT> t 1
E :→ E
*SystemT> t 2
(E :→ E) :→ E
*SystemT> t 3
E :→ (E :→ E)

*SystemT> substWith (t 2) (t 3) (t 1)
((E :→ E) :→ E) :→ (E :→ E)
*SystemT> n it
12
```

and the beta-reduction `applyLambda` that inserts `(t 5)` in positions indicated by the path codes `(t 2)` and `(t 3)` in the tree `(t 8)`

```
*SystemT> t 5
E :→ ((E :→ E) :→ E)
*SystemT> t 8

(E :→ (E :→ E)) :→ E
*SystemT> applyLambda [2,3] 8 5
Just 871509787656907713528983453696

*SystemT> applyLambda [t 2,t 3] (t  8) (t 5)
Just (((E :→ ((E :→ E) :→ E)) :→ (E :→ E))
          :→ (E :→ ((E :→ E) :→ E)))
```

9. INSTANCES DERIVED FROM PARENTHESIS LANGUAGES

To hint towards the universality of the representation mechanisms described so far, we conclude by interpreting our type classes as parenthesis languages.

```
data Par = L|R deriving (Eq,Read,Show)
data Pars = Pars [Par] deriving (Eq,Read,Show)

instance PureTypes Pars where
  empty=Pars [L,R]
  arrow (Pars x) (Pars (L:xs)) =  Pars (L : x ++ xs)
  from = fst . from_to
  to = snd . from_to

  isEmpty (Pars [L,R]) = True
  isEmpty _ = False

from_to (Pars (c:cs)) | c==L =
  (Pars (L:fs),Pars (L:ts)) where
    (fs,ts)=parexpr cs

parexpr (c:cs) | c==L = parlist cs where
  parlist (c:cs) | c==R = ([R],cs)
  parlist (c:cs) = (c:fs++ts,cs2) where
    (fs,cs1)=parexpr (c:cs)
    (ts,cs2)=parlist cs1

par :: (PureTypes a)⇒a→Pars
par = view

instance PeanoArith Pars
instance FastArith Pars
instance Ordered Pars
instance Combinators Pars
instance LambdaTerms Pars
```

One can convert from this parenthesis language representation to natural numbers and back as follows:

```
*SystemT> par 2012
Pars [L,L,L,L,R,R,R,L,R,L,R,L,L,R,R,L,R,L,R,L,R,L,R,R]
*SystemT> n it
2012
```

10. RELATED WORK

Gödel's System **T** [5] can be considered an important ancestor of modern type systems. We refer to [11, 3, 1] for salient revisitings of its formal properties, its connection to the logical foundations of mathematics and computer science, as well as an assessment of its historical impact.

The techniques described in this paper originate in the data transformation framework described in [14, 12, 13]. In contrast to the work described in these papers which can be seen as "an existence proof" that, given a bijection to N, arithmetic computations can be performed with objects like System **T** types, in this paper we show it constructively. Moreover, we lift our conceptual framework to a polymorphic axiomatization which turns out to have as "twin" *interpretations* natural numbers and System **T** types.

An emulation of Peano and conventional binary arithmetic operations in Prolog, is given in [8]. Arithmetic computations with types expressed as `C++` templates are described in [7] and in online articles by Oleg Kiselyov using Haskell's type inference mechanism. However, the mechanism advocated there is basically the same as [8] and does not involve structural recursion on tree types like in this paper.

In [15] a similar type class based specification of arithmetic operations is given in terms of an abstraction of bijective base-2 arithmetics as bitstacks with two successor functions `o` and `i` corresponding to $\lambda x.2x+1$ and $\lambda x.2x+2$. This paper shares the same methodology as [15] by viewing type classes as an analogue to axiom systems in which a set of generic constructors and destructors are used. However, by contrast to [15] section 15, where a binary tree instance similar to `data T` is shown implementing the `s` and `p` arithmetic operations, this paper uses a type class abstraction of the System **T** types to express computations directly. By using the free magma of rooted binary trees as the underlying mathematical abstraction, this paper also provides novel encodings for combinators and lambda expressions.

11. CONCLUSION

The results described in this paper have been made possible by extending the techniques introduced in [14, 12, 13, 15] that allow observing the internal working of recursive definitions through isomorphisms transporting operations between fundamental data types.

Interpreting our free binary trees as a type language provides a key building block for a unified representation of types and data.

Encodings for parenthesis languages as well as for Turing-equivalent combinator calculi and lambda-terms highlight the possibility of a unified theory of data types and computations along the lines of [16].

As a possible practical application, arithmetic and ordering, expressed generically in terms of a type class abstracting away operations on binary trees have been implemented with complexity bounds similar to ordinary arithmetic. The arithmetic operations using instance `T`, as described in section 5, scale up to represent "towers of exponents" that overflow computer memory when implemented with conventional bitstring arithmetic.

Acknowledgments

We are grateful to Philip Wadler for valuable suggestions on the use of QuickCheck-able assertions for validating our Haskell code and for pointing out problems with our data representations that lead to the idea of using the free magma of binary trees as the key building block for our type classes.

We thank NSF (research grant 1018172) for support.

12. REFERENCES

[1] S. Alves, M. Fernández, M. Florido, and I. Mackie. Gödel's system T revisited. *Theoretical Computer Science*, 2009.

[2] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[3] G. Dowek. Gödel's system T as a precursor of modern type theory. 2006. manuscript, http://www.lix.polytechnique.fr/ dowek/Philo/-godel.pdf.

[4] J. Fokker. The systematic construction of a one-combinator basis for lambda-terms. *Formal Aspects of Computing*, 4:776–780, 1992.

[5] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12(280-287):12, 1958.

[6] R. Kaye and T. L. Wong. On Interpretations of Arithmetic and Set Theory. *Notre Dame J. Formal Logic Volume*, 48(4):497–510, 2007.

[7] O. Kiselyov. Type arithmetics: Computation based on the theory of types. *CoRR*, cs.CL/0104010, 2001.

[8] O. Kiselyov, W. E. Byrd, D. P. Friedman, and C.-c. Shan. Pure, declarative, and constructive arithmetic relations (declarative pearl). In *FLOPS*, pages 64–80, 2008.

[9] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, New York, NY, USA, 1998.

[10] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[11] W. W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[12] P. Tarau. A Groupoid of Isomorphic Data Transformations. In J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, editors, *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference MKM 2009* , pages 170–185, Grand Bend, Canada, July 2009. Springer, LNAI 5625.

[13] P. Tarau. An Embedded Declarative Data Transformation Language. In *Proceedings of 11th International ACM SIGPLAN Symposium PPDP 2009*, pages 171–182, Coimbra, Portugal, Sept. 2009. ACM.

[14] P. Tarau. Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell. In *Proceedings of ACM SAC'09*, pages 1898–1903, Honolulu, Hawaii, Mar. 2009. ACM.

[15] P. Tarau. Declarative modeling of finite mathematics. In *PPDP '10: Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 131–142, New York, NY, USA, 2010. ACM.

[16] P. Tarau. "Everything Is Everything" Revisited: Shapeshifting Data Types with Isomorphisms and Hylomorphisms. *Complex Systems*, (18), 2010.

[17] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.