Thrists: Dominoes of Data

Gabor Greif

gabor@mac.com

20 November, 2011

Abstract

We develop a novel list-like datastructure (which we name *Thrist*), that is able to capture the typing rule of function composition. Indeed, we show that when *Thrist* is parameterized with the function type constructor ($\rightarrow$) an interpretation function can be provided which completely emulates the classical function composition (.). Additionally we can perform pattern matching on our *Thrist* elements, thus obtaining the ability to do analysis on them. On the practical side we develop three new two-parameter GADTs for inclusion into *Thrist*s. The first, when accompanied by an appropriate interpreter, directly model the semantics of the Cat language, while the second caters for a new type of parser combinator libraries. In our third example we demonstrate the use of *Thrist*s in state machines. The *Thrist* approach of exposing the intermediate types where the elements are joined together shows its potential especially in the ability to stage the interpreter in a type-safe way and only allows for type correct transformations.

1. Structure of the Paper

In the following section we motivate our subsequent introduction of *Thrist* with well known examples. The third section is devoted to the development of three practial applications of *Thrist*s. Next we present some less useful but entertaining uses. Section five examines other work, while section six concludes, pointing out open problems and showing up possible future directions.

2. Introduction

We are well acquainted with data structures that are parameterized over data types, as they are a bread-and-butter tool for functional programmers. For example the *List* datatype could be introduced in the Haskell-like language Ωmega [17] with the following definition:

data *List* :: * ~> * **where**
  Nil :: *List a*
  Cons :: $a \rightarrow$ *List a* $\rightarrow$ *List a*

We deliberately avoid the traditional syntax of defining the *List* datatype, and use a *GADT-style* definition[1], because we want to build on this foundation later. It is enough to say that *List* does not impose any constraints on the contained datatype whatsoever:

```
prompt> Cons 42 (Cons 33 (Cons 5 Nil))
    Cons 42 (Cons 33 (Cons 5 Nil)) :: List Int
```

Departing from single-parameter datatypes we focus on two-parameter datatypes from now on. These are more interesting for our purposes, because the two parameters can be put in relation to each other. The most prominent member of this class of types is the arrow type:

```
prompt> :k (->)
    (->) :: *0 ~> *0 ~> *0
```

Fully saturated, $a \rightarrow b$ signifies the type of all functions that take elements of a to elements of b. Again, a and b can be any type, concrete ones will do just like universally qualified ones:

```
prompt> id
    <fn> : forall a.a -> a
prompt> ord
    <primfun ord> : Char -> Int
```

Now, functions are a bit more interesting than *List*s above, for they can be composed! Interestingly, composition (written as (.)) is again a function:

```
prompt> (.)
    <fn> : forall a b c.(a -> b) -> (c -> a) -> c -> b
```

This function signature has several interpretations, but the most common one tells us that the composition function (.) takes a function with type $a \rightarrow b$ as its first argument, then a function with type $c \rightarrow a$ as its second argument, and returns a function of type $c \rightarrow b$. Again, a, b and c can be arbitrarily specialized or be left universal. We can play with our new toy in the following way:

```
prompt> let locase = chr . (\x -> x + 32) . ord
```

[1]in a *generalized algebraic datatype* each data constructor is allowed to define a more specific type than the datatype itself

```
    locase
prompt> locase 'G'
    'g' : Char
```

The type of the composition function constrains the types of its arguments in a nontrivial way: the range of the second funtion must match with the domain of the first (universally qualified types will always match, while monotypes must be equal). Violating this rule gives a type error:

```
prompt> ord . ord
    In the expression: ord the result type: Char -> Int was not
what was expected: a -> Char
```

This crucial property of the composition function will guide our explorations in this paper.

2.1. Generalizing Function Composition

While function composition seems to be a cute artifact of mathematics, (nevertheless, all computable functions can be derived from it), this is no reason for us to stop at this point! First, we observe that function composition is a one-way street: once two functions are composed, they amalgamate beyond recognition. There is no way (inside our system) to take them apart again. This is very much resembling addition, where (23 + 19) gives 42 and this result has completely lost all memories of the fact how it was obtained. But can we create a datastructure that has all properties of function composition, without being amnesiac?

Indeed we can, and the rest of this section shall explain how.

Our crucial observation from the introduction was that the types at the ends of the function arrows must thread up, intuitively[1]:

$$a \to \boxed{b\ (.)\ b} \to c \equiv a \to c$$

We also observe that the intermediate types do not show up in the end result's type. Let's simulate these rules as a datatype:

data *Thrist* : : * ~> * ~> * **where**
  Nil : : *Thrist a a*
  Cons : : (*a*, *b*) → *Thrist b c* → *Thrist a c*

The alert reader will have noticed that we have defined a proper GADT, since e.g. the *Nil* constructor is only inhabiting the *diagonal* of the *two-dimensional* parameter space.

We can now duplicate the *feeling* of function composition: *Cons* (*'g'*, 103) \$ *Cons* (103, 71) \$ *Cons* (71, *'G'*) *Nil* Also we have obtained a datatype that is not amnesiac, i.e. it can be torn apart at any place by pattern matching, though we have to pay the price that the intermediate types are a bit hard to deal with (we shall revisit this issue later).

It appears that we have reached our goal, we can form list-like data where the intermediate types thread up, we are happy and name our new toy *Thrist*, a portmanteau of thread and list.

Our joy, however, quickly fades away when we compare our thrist with the composition function. How can some data of type *Thrist a b* be interpreted as an arrow type $a \to b$? If we fail to provide this embedding, we cannot consider thrists being a generalisation of function composition.

Not everything is lost, however. Close scrutiny reveals that our usage of (,)[2] is the culprit. If we could liberate ourselves from this premature decision, we could gain back our hope.

2.2. The Improved Thrist

We try all over again, this time abstracting away the pair into an additional parameter:

data *Thrist* : : (* ~> * ~> *) ~> * ~> * ~> * **where**
  Nil : : *Thrist p a a*
  Cons : : *p a b* → *Thrist p b c* → *Thrist p a c*
 deriving *List*(*l*)

All that remains of the pair is the bitter aftertaste and the letter *p* in the definition of *Thrist*.

The *deriving* clause above activates some special syntax for reading or displaying which we will peruse later. At the moment we are interested in the types only and it looks like we are getting closer now:

```
prompt> Cons chr (Cons (\x -> x + 32) (Cons ord Nil))
    ... : Thrist (->) Char Char
```

We have created an *arrow thrist*! Immediately we can retry our previous attempt:

```
prompt> Cons ('g', 103) (Cons (103, 71) (Cons (71, 'G') Nil))
    ... : Thrist (,) Char Char
```

The *pair thrist* that caused us some headache before!

We shall explore some other interesting but sometimes futile thrists later in the discussion. But now let's put the last missing piece in place to show that an arrow thrist is strictly more general than function composition. Here comes the *runArrThrist* function:

runArrThrist : : *Thrist* (→) *b c* → *b* → *c*
runArrThrist Nil b = *b*
runArrThrist (*Cons f r*) *a* = *runArrThrist r* (*f a*)

This function now confirms the vague intuition that *Nil* plays the rôle of the neutral element of the arrow thrist, just like the identity function plays the rôle of the neutral element in the monoid of composed functions. We shall encounter this recurring fact as we proceed to our examples' implementations.

[1] we reconcile the discrepancy between the function arrow's (→) direction and the customary composition $g \circ f$ by considering a modified *reverse composition*

[2] pronounced "pair"

The definition of *runArrThrist* on a *Cons* constructor deserves some attention. We have not yet explicitly mentioned it, but range of the function *f* is existentially qualified, because it does not appear in the final result of *Cons.* This implies a minor complication when pattern matching: the *head* component[1] of a *Cons* cannot be passed to non-local functions, because the existential type would escape.

Looking back at our progress so far, the analogy of thrists with the game of dominoes [23] could spring into our minds:

Dominoes are only laid out in a valid configuration, when the stones sharing a common edge possess the same number of dots next to this edge. Our *Nil* corresponds to an empty board, and *Cons* joins two stones with a common edge. A simple, but important difference to our thrists is that the dominoes' face numbers do not correspond to the contained *value*, but to the *type* of the contained value: we are playing dominoes at the type level.

3. Three Practical *Thrist*s

Now that we have defined the *Thrist* datatype and given a sufficiently generic interface to cover non-trivial cases, time has come to look for real-world applications. Specifically we shall describe a combinator library for creating ASTs of Cat, a statically typed stack-oriented language [2]. As a second application we shall draft a parser combinator library that admits various implementation strategies. We shall also sketch the use of thrists in transition arrows of state machines to obtain certain correctness guarantees, and finally show some curious examples that may even have some practical value.

3.1. Application one: the Cat Thrist

Like all stack-oriented languages, Cat employs a simple idiom of computation. A rich set of primitives are available for pushing values on a stack, permuting them and popping them off. Logical and arithmetic primitives consume portions of the top of the stack (TOS) and deposit results in their place. Procedures can be defined as a succession of primitive invocations and procedure calls. The semantics of procedure calls is defined as the insertion of the called procedure's contents to the point of the invocation.

Let's begin with the definition of the *Cat* datastucture, that will serve as the first parameter to *Thrist.* Clearly it should be parametrized with two types. Naturally we choose the first type parameter to describe the shape of the stack before and the second parameter after the *Cat* primitive has been executed.

```
data Cat :: * ~> * ~> * where
  Push :: a → Cat opaque (a, opaque)
  Pop :: Cat (a, opaque) opaque
  Dup :: Cat (a, opaque) (a, (a, opaque))
  Add :: Cat (Int, (Int, opaque)) (Int, opaque)
```

3.1.1. First Explorations

We shall extend our *Cat* with new primitives as the need arises, but for now we have enough to perform some experiments. We have chosen the tuple datatype to represent stack shapes, but we are free to pick any other sequence-like datatype that is able to record the type of each element. The *Cat* datatype is defined as a GADT, which will guarantee that only sematically sound programs can be expressed as a *Thrist Cat.* We can begin our explorations immediately:

```
prompt> Cons (Push 19) $ Cons (Push 23) $ Cons Add $ Cons Pop Nil
    Cons (Push 19) (Cons (Push 23) (Add (Cons Pop (Cons Add (Cons Pop Nil)))) :: Thrist Cat a a
```

The data we built up can be a representation of a Cat program that pushes 19 and then 23 on the stack, adds them, keeping only the result 42 on the stack, and then pops this result off. The inferred type tells us that there is no net change in the stack's shape.

3.1.2. Making Use of Ωmega's Features

We shall from now on make use of a feature of the Ωmega language to define custom syntax for datatypes. Our aim is to hide the *Thrist* constructors *Cons* and *Nil* behind a more intuitive façade. In the rest of this paper we shall write the above expression as

```
prompt> [Push 19, Push 23, Add, Pop]l
    [Push 19, Push 23, Add, Pop]l :: Thrist Cat a a
```

At this point it becomes obvious what we have merely hinted in Section 2.2: the *deriving* clause causes Ωmega's parser and printer to perform the conversion to the internal form whenever the list-like brackets [] followed by the letter "l" are encountered[2].

We can now continue using this terser syntax:

```
prompt> [Pop, Pop]l
    [Pop, Pop]l :: Thrist Cat (a, (b, c)) c
```

The inferred type reflects the function of this Cat fragment, namely starting out with a stack that has at least two elements pushed, we end up with those two values removed.

There are invalid Cat programs, for example addition of two characters:

```
prompt> [Push 'a', Dup, Add]l
    the result type: … was not what was expected: …
```

The GADT-based type inference fails, because *Add*

[1]or for symmetry reasons the *tail* too

[2]$[a, b]l$ is equivalent to *Cons a* (*Cons b Nil*), while semicolon separates off the tail: $[a; b]l \equiv$ *Cons a b*

expects two integers on the stack, but there are two *Char*s available instead.

3.1.3. Interpreter

Now it is time to build an interpreter for *Thrist Cat*, and thus define its big-step semantics:

```
interpret' :: Thrist Cat a b → a → b
interpret' []l st = st
interpret' [Push x; rest]l st = interpret' rest (x, st)
interpret' [Pop; rest]l (a, st) = interpret' rest st
interpret' [Dup; rest]l (a, st) = interpret' rest (a, a,
st)
interpret' [Add; rest]l (a, b, st) = interpret' rest (a + b,
st)
```

It works:

```
prompt> interpret' [Push 19, Push 23, Add]l ()
    (42, ()) :: (Int, ())
```

With this basic functionality in place, we get bolder and define a primitive with a side-effect:

```
data Cat :: * ~> * ~> * where
  Print :: Cat (a, opaque) opaque
  ...
```

To interpret the *Print* primitive we have to restructure our *interpret'* function to wrap the stack into the *IO* monad:

```
interpret' :: Thrist Cat a b → IO a → IO b
interpret' [Print; rest]l st = do
        (a, st') ← st
        putStr $ show a
        interpret' rest st'
  where monad ioM
```

In this function *ioM* is globally bound to a value of type *Monad IO* containing the monadic *return* and *bind* functions for the *do-notation*'s perusal[1]. In similar spirit we have to rewrite the other cases too:

```
interpret' [Pop; rest]l st = do
        (a, st') ← st
        interpret' rest $ return st'
  where monad ioM

interpret' [Push x; rest]l st = do
        interpret' rest $ return (x, st)
  where monad ioM

interpret' [Dup; rest]l st = do
        (a, st') ← st
        interpret' rest $ return (a, a, st')
  where monad ioM

interpret' [Add; rest]l st = do
        (a, b, st') ← st
        interpret' rest $ return (a + b, st')
  where monad ioM
```

Trying out this monadic interpreter gives us:

```
prompt> interpret' [Push 21, Dup, Add, Print]l (returnIO ())
    Executing IO action
    42
        () :: IO ()
```

It is a reasonable restriction to Cat programs that they can be started with any stack shape and they finish with the same stack unchanged. We can ensure this property by writing a top-level interpreter function for Cat programs using *rank-2 polymorphism*:

```
interpret :: (forall a. Thrist Cat a a) → IO ()
interpret program = interpret' program $
returnIO ()
```

Obviously this *interpret* function is called for its side-effects.

3.1.4. Extending the Primitives

Above we have defined an arithmetic primitive in Cat, namely *Add*. While possible, it is not desired to define all (which is potentially a lot) primitives this way, with their own typing rules, and own clause in the interpreter. Also, this approach does preclude a very useful notion, called *partial application*. In this example, *Add* must always be applied to two elements on the stack.

What we are looking for is a more-or-less generic approach to define logical and arithmetic operators in Cat, say, using the *Prim* (+) to frob arithmetic addition from the underlying Ωmega implementation.

Encountering First Problems

We could introduce *Prim* thus:

```
...
  Prim :: (a → b) → Cat (a, opaque) (b, opaque)
  ...
```

While this approach can surely be made to work with unary functions, it is not immediately seen how binary[2] operators can be formalized in the type-safe way mandated by Ωmega. The expectation is that an n-ary primitive would consume the top n items from the stack and produce one item as the result.

We have to reformulate our typing rule to deal with the case that the type parameter b is in turn a function arrow. Since Ωmega allows us to decompose the arrow's structure using a *type function*, we try:

```
Prim :: (a → b) → Cat {blowUpBy (a → b)
opaque} ({result b}, opaque)
  ...
```

blowUpBy[3] creates the expected stack shape needed for fully saturating the primitive, while *result* determines the rightmost type in the function's arrow type.

[1] Ωmega's (current) lack of *type classes* necessitates the explicit passing of *Monad* :: (* ~> *) ~> * values (dictionaries)

[2] or arbitrary arity functions for that matter

[3] see the definition of *blowUpBy* and *result* in the Appendix B

With these definitions we can observe the correct type inference of our *Add* substitute *Prim* (+):

```
prompt> Prim (+)
    Prim <fn> :: Cat (Int, Int, a) (Int, a)
```

More Problems while Interpreting

Unfortunately we have not mastered everything yet. We remember that the semantics of our *Cat* combinators is defined by the interpretation function. So we are obliged to extend *interpret'*. We can try thus:

```
interpret' [Prim f; rest]l st = do
        (a, st') ← st
        interpret' rest $ return (f a, st')
  where monad ioM
```

But this can only work for *unary* primitives, and since the primitive is allowed to possess higher arity it cannot typecheck. Clearly, a solution is needed that is able to distinguish between different-arity primitives at runtime. Since we are not inclined to introduce a new family of *Cat* alternatives, *Prim1*, *Prim2*, etc., we have to equip *Prim* with a second argument.

Using Representation Types to Describe Arities

The solution is to attach an object of *representation type* to every *Prim* combinator, to aid continued interpretation in the multi-arity case.

To this end we need a description of what types are being passed in stack slots. This description must be a value so that it can be pattern matched at runtime and it has to provide a constructor for all tractable datatypes.

```
data Tractable :: * ~> * where
  IntT :: Tractable Int
  BoolT :: Tractable Bool
  CharT :: Tractable Char
  PairT :: Tractable a → Tractable b → Tractable (a, b)
  ListT :: Tractable a → Tractable [a]
  ArrT :: Tractable a → Tractable b → Tractable (a → b)
```

provides a way to describe some data types that are built into Ωmega. Its first three constructors apply to basic datatypes, while the rest encodes rules, how compound datatypes can be represented, given tractable ones.

We can proceed by employing this descriptive facility into our *Prim* constructor:

```
Prim :: Tractable b → (a → b) →
            Cat {blowUpBy (a → b) opaque}
                ({result b}, opaque)
  ...
```

The first argument to *Prim* is called a *representation*[1], becase it records the structure of the function's range's type[2].

We can reproduce our previous *Add* primitive now:

```
prompt> Prim (ArrT IntT IntT) (+)
    Prim (ArrT IntT IntT) (+) <fn> :: Cat (Int, Int, a) (Int, a)
```

Interpreting Primitives

We finally have all ingredients together to embark on putting down the *interpret'* case on *Prim*. The key idea is here to pattern match on the representation value in order to incrementally saturate the Ωmega function present in the primitive:

```
interpret' [Prim (ArrT x y) f; rest]l st = do
        (a, st') ← st
        interpret' [Prim y (f a); rest]l (return st)
  where monad ioM
```

We see that the value popped off the stack is partially applied to the function and a new primitive is created with the right type representation and prepended to the thrist being interpreted. The rest of the possible represented types are implemented along the lines of our earlier attempt that only covered unary primitives:

```
interpret' [Prim IntT f; rest]l st = do
        (a, st') ← st
        interpret' rest $ return (f a, st')
  where monad ioM
```

Unfortunately this code for *IntT* must be duplicated for all alternatives in *Tractable* and cannot be wildcarded.

At this point the interpreter is essentially done, and missing pieces can be filled in easily.

3.1.5. Staging the Interpreter

A well-known technique to turn an interpreter into a compiler is staging [18]. The compile function takes a *Thrist Cat* into a function of the metalanguage, that when executed causes the same effect as the interpretation of the program itself. Naturally, the compiled program is expected to run faster, since the interpretative overhead is already removed. We demonstrate the technique for one *Cat* primitive only[3].

```
compile :: Thrist Cat a b → Code (IO a → IO b)
compile [Print; rest]l = ⟦ λ st → do
        (a, st') ← st
        putStr $ show a
        $(compile rest) $ return st'
    where monad ioM ⟧
```

[1]technically these types are called *singleton types* and constitute a reflection of the structure of the type-level objects into the value world.

[2]It suffices to describe the range, because the domain's type is easily handled without a representation.

[3]for a complete implementation consult the Appendix B

The code pretty much resembles the interpreter, with the main differences being the ⟦ ... ⟧ *code brackets* and the *escaping syntax* \$(...). The former allows for constructing values of type *Code a* (as seen in the function's type annotation), and the latter is needed for splicing code into a hole in the code. Essentially the whole Cat program gets compiled, which in turn can be executed by means of the *run* function by supplying an initial stack.

3.1.6. Optimization

The fact that Cat programs are represented as data in the metalanguage that is amenable to analysis by pattern matching, we can write an optimization function that performs several code optimizations on a program, such as head and tail merging of conditionals, value folding, inlining etc. Because the *Cat* thrist does not admit wrongly typed Cat programs and the optimization function takes *Thrist Cat* to *Thrist Cat*, all optimizations must be type preserving.

3.1.7. Generalization

The language Cat is intended as an intermediate language produced by front-end compilers and consumed by back-ends that target stack based virtual machines like JVM [8] and CIL [3]. It is advisable to generalize *Cat* in a way that *Pop* gets a count parameter that tells how many elements are to be popped of the stack. Also instead of *Swap* it would be beneficial to have a *Permute* primitive that subsumes all variants of stack shuffling operations, allowing us to get rid of *Swap* and friends. All these parametrized primitives would have one problem in common, namely that the stack shape would vary depending on the value of the parameter(s), requiring dependent types to define them. Fortunately Ωmega provides a device that is approaching the power of dependent types, namely *singleton types* and *type-level functions*. Here is a sketch of *PopN*:

```
PopN :: Nat' (S n) → Cat {blow (S n) s} s
```

It uses the type-level function *blow* to add the necessary number of universal type variables to the initial stack's shape:

```
blow :: Nat ~> *0 ~> *0
{blow Z s} = s
{blow (S n) s} = (t, {blow n s})
```

The interpreter can be written thus:

```
interpret' [PopN (S n); rest]l st → do
        (_, st') ← st
        case n of
        0v → interpret' rest st'
        _ → interpret' [PopN n; rest]l st'
```

Here we can rediscover the recursion pattern that helped us interpreting N-ary primitives.

3.2. Application two: a GADT-based Parser Thrist

Traditional monadic parser combinator libraries (like Parsec [9]) suffer from the same problem like the composition operator: they compose easily but cannot be dissected and analysed, nor translated to other representations. We proceed similarly to the *Thrist* (→) and *Thrist Cat* to create a parsing combinator library that is representation agnostic, i.e. built up parsers can be interpreted/compiled and analysed in any reasonable way.

3.2.1. Envisioning Parsing

But first let's be clear about what we aim at. We demonstrate the process of parsing by the example of a lexer with semantic evaluation. Our tokens are the various literal numerals like they occur in the C programming language:

```
0xCafeBabe 0XE0UL 123456L
```

These are the steps we wish to proceed on the second token as the running example:

0) the token as read from character stream:

   `0XE0UL`

1) we match the `0X` prefix:

   $\boxed{\texttt{0X}}\texttt{E0UL}$

2) we expect zero or more hexadecimal characters:

   $\boxed{\texttt{0X}}\boxed{\texttt{E0}}\texttt{UL}$

3) we check for a non-empty hex string and fold it to a decimal integer:

   $\boxed{\texttt{0X}}\boxed{\boxed{\texttt{E0}}}\texttt{UL}$

4) we look for an optional signedness hint:

   $\boxed{\texttt{0X}}\boxed{\boxed{\texttt{E0}}}\boxed{\texttt{U}}\texttt{L}$

5) we look for an optional storage size specifier:

   $\boxed{\texttt{0X}}\boxed{\boxed{\texttt{E0}}}\boxed{\texttt{U}}\boxed{\texttt{L}}$

6) we encapsulate the distilled information into a token datatype:

   $\boxed{\boxed{\texttt{0X}}\boxed{\boxed{\texttt{E0}}}\boxed{\texttt{U}}\boxed{\texttt{L}}}$.

We can imagine that types play a great role in this process, the small boxes tend to stand for *Char* or [*Char*], while the rounded boxes represent some *computed* information. We can also observe that several steps *partition* the input string into a prefix being scrutinized and the as of yet unanalysed rest.

3.2.2. Realization

To be able to compose these envisioned operations we define the GADT *Parse*:

```
data Parse :: * ~> * ~> * where
  Atom :: Char → Parse Char Char
  Sure :: (a → b) → Parse a b
  Try :: (a → Maybe b) → Parse a b
  Rep1 :: Parse a b → Parse [a] ([b], [a])
  Rep :: Parse [a] (b, [a]) → Parse [a] ([b], [a])
  Group :: [Parse a b] → Parse [a] ([b], [a])
  Par :: Parse a b → Parse c d → Parse (a, c) (b, d)
  Wrap :: Thrist Parse a b → Parse a b
```

The datatype *Parse a b* represents a parser that consumes data of type *a* and if a match is found it produces a value of type *b*. The types mentioned in the constructors already give us a strong indication about their intended meaning:

- (*Atom 'X'*) matches only the capital *'X'* character
- (*Sure ord*) alway matches, consuming a *Char* and returning its *Int* ASCII value
- (*Try hexdigit*) matches only if a character is a hexadecimal one and returns its hex value, fails otherwise
- (*Rep1* (*Atom 'X'*)) matches as many capital *'X'*s as possible and returns a pair consisting the matched and unconsumed portions
- (*Rep part*) similarly tries to match as many *part*s in the input's prefix as possible
- (*Group* [*Atom 'a'*, *Atom 'b'*]) just matches a prefix *"ab"* in the input, returning it in a pair along with the unconsumed portion, or fails otherwise
- (*Par fst snd*) runs *fst* and *snd* on the two components of the pair, failing if any of them fails
- (*Wrap parseThr*) simply allows to treat a thrist as a *Parse.*

In the above descriptions we only suggest a possible semantics, because *Parse* does not mandate it in any way. So when we talk about "returns something" then this is just an intention, an implementation may choose to employ an other strategy (such as *continuation passing*). It is also completely unspecified what *failure* and *success* actually entail.

Proceeding with our running example, we expect the end result of token parsing to produce a value of the following *Token* data type:

```
data Token =
  Number Int Bool Bool
  | ...
```

This amounts to our parser consuming a string and producing *Token*s (along with the unconsumed rest) to be assigned the type *Thrist Parse* [*Char*] ([*Token*], [*Char*]).

3.2.3. Using the Combinators

How can we use our combinators to describe the parsing steps 0) to 6) from the Section 3.2.1?

- First, we use *Group* [*Atom '0'*, *Atom 'X'*] to match the prefix of the string *"0XE0UL"* and produce (*"0X"*, *"E0UL"*) : : ([*Char*], [*Char*]).
- Then we can discard the prefix because at this point we know that we have to do a base 16 conversion later. We employ *Sure snd* to do this, obtaining *"E0UL"* : : [*Char*].
- Then *Rep1* (*Try hexdigit*) will split off two more characters, converting them to decimal values on the way. So far we've got ([14, 0], *"UL"*) : : ([*Int*], [*Char*]).
- Since we cannot discard either component, we have to proceed in parallel with

  ```
  Par (Wrap [Try nonEmpty, Sure foldHex]l)
      (Wrap [Rep1 $ Atom 'U',
           Par (Sure id)
              (Rep1 $ Atom 'L')]l)
  ```

  where the first component produces 224, and the second goes on by splitting off any *'U'* followed by any *'L'*, producing a triple (*"U"*, (*"L"*, *""*)). At this stage we have (224, (*"U"*, (*"L"*, *""*))) : : (*Int*, ([*Char*], ([*Char*], [*Char*]))).
- Finally, we feed this into *Try numberToken* that verifies the correct usage of 'U's and 'L's, and creates a pair (*Number* 224 **True True**, *""*) : : (*Token*, [*Char*]) consisting of the parsed token and the rest of the input.

We have not yet given the defininitions of the *hexdigit*, *foldHex*, *nonEmpty* and *numberToken* functions. These follow next and should be rather obvious.

```
hexdigit = λ c → if ord '0' ≤ ord c && ord c ≤
ord '9'
                 then Just (ord c - ord '0') else
              if ord 'a' ≤ ord c && ord c ≤ ord
'f'
                 then Just (ord c - ord 'W') else
              if ord 'A' ≤ ord c && ord c ≤ ord
'F'
                 then Just (ord c - ord '7') else
Nothing

foldHex = foldl (λ acc x → (acc * 16) + x) 0
```

```
nonEmpty (all@(_:_)) = Just all
nonEmpty _ = Nothing

numberToken :: (Int, (String, (String, String))) →
Maybe (Token, String)
numberToken (i, ("", ("", rest))) = Just (Number
i False False, rest)
numberToken (i, ("U", ("", rest))) = Just (Number
i True False, rest)
numberToken (i, ("", ("L", rest))) = Just (Number
i False True, rest)
numberToken (i, ("U", ("L", rest))) = Just
(Number i True True, rest)
numberToken _ = Nothing
```

Putting all together we can write:

```
signedSized = Wrap [Rep1 (Atom 'U'), Par (Sure
id) (Rep1 (Atom 'L'))]l

hexToken =
  [ Group [Atom '0', Atom 'X']
  , Sure snd
  , Rep1 (Try hexdigit)
  , Par (Wrap [Try nonEmpty, Sure foldHex]l)
signedSized
  , Try numberToken ]l

tokens = Rep (Wrap hexToken)
```

With some squinting we can summarize the principles as:

– The interesting prefix is split off the rest,

– if it is semantically important we condense it to a more appropriate form,

– resorting to parallel processing if both components of an input pair are relevant.

As the execution of the thrist proceeds, incrementally more of the token are analysed, condensed and converted.

3.2.4. Defining the Semantics by Interpretation

We provide the *parse* function for the *Rep* constructor as an example:

```
parse :: Thrist Parse a b → a → Maybe b

parse [Rep p; r]l as = parse r (parseRep [p]l as)
  where
    parseRep :: Thrist Parse [a] (b, [a]) →
                [a] → ([b], [a])
    parseRep _ [] = ([], [])
    parseRep p as = case parse p as of
                Nothing → ([], as)
                Just (b, as') → (b:bs, rest)
                  where (bs, rest) = parseRep p as'
```

This code is rather unsurprising: to obtain a list of *p*s we have to consume as many parsed constituents from the input as possible and accumulate them in the result along with the remaining input.

For the rest of the implementation we defer to Appendix C which contains a working example for parsing hexadecimal tokens.

3.2.5. Compilation

In a fashion similar to the *Thrist Cat* we can compile our parser combinators to a more efficient algorithm by removing the interpretative overhead. We have complete freedom in the selection of implementation methodology: we can target the object language directly or plumb monadic- resp. arrow-based parser combinators.

3.2.6. Analysis

We can run various analyses on our parsers, to ensure that the grammar is unambiguous, for example. A popular transformation would be the detection of left-recursion and its transformation to a right- recursive form.

3.2.7. Outlook

Many interesting other combinators can be defined, my repository contains also

```
Seq :: Parse [a] (b, [a]) → Parse [a] (c, [a]) → Parse [a]
((b, c), [a]) -- parse front first then second
 Seq1 :: Parse a b → Parse a c → Parse [a] ((b, c), [a])
-- same, but with single-elem first and second
 ButNot1 :: Parse a b → Parse a b → Parse a b      -
- match first and expect second to fail
 UpTo :: Parse [a] (b, [a]) → Parse [a] (c, [a]) → Parse
[a] ((b, c), [a]) -- scan for c then match b
```

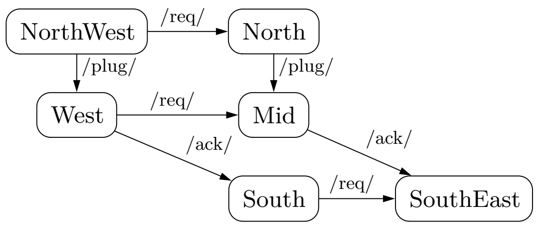
etc. I think working together with a parsing expert could result in a minimal set of combinators that allow parsing a great variety of grammars and optimization and compilation methods that make the parsing process *fast*.

3.3. Application three: Actions on State Machine Transitions

One popular use of metaprogramming tools is generating code for legacy systems, especially when the high-level language's whole feature set is too heavy for the application, such as in embedded systems with significant resource limitations. We wish a system where very strong guarantees are maintained in the model by resorting to typeful data representation and nevertheless preserve the ability to convert our model into a lower-level representation that is feasible for the target system.

3.3.1. The Transition System

Figure 1 depicts a fragment of a state machine intended to handle *request/acknowledge* type handshake with an identical remote instance of itself. The two in-



```
northWest :: State GateClosed SecondaryEnabled (H Idle Idle')
north :: State GateClosed SecondaryEnabled (H Idle Idle')
west :: State GateClosed SecondaryEnabled (H Idle NeedAck)
mid :: State GateClosed SecondaryBlocked (H Idle NeedAck)
south :: State GateClosed SecondaryBlocked (H Idle Idle')
southEast :: State GateOpen SecondaryBlocked (H Idle Idle')
```

Figure 1. State machine fragment

stances are communicating solely by message passing and originally both inhabit the *NorthWest* state. The intention is that the two machines control the ends of a unidirectional communication channel that is *protected*, i.e. two wires exist, carrying a data stream A on the first wire[1], and when the first wire does not carry data, the second wire can be utilized to carry a lower-priority data stream B. The typical use-case is the premium customer paying for A and the bulk customer paying significantly less for B. There is an extra twist however: it is forbidden for the data stream A to reach the sink of data stream B, even for very short time periods. This invariant calls for the request/acknowledge handshake, and the transient states on the way to the *SouthEast* state mirror this protocol.

3.3.2. Typed States

We intend to equip the states with an orthogonal system of properties (which we call *facets*), and hope to assemble transition arrows from a toolbox of elementary actions which deal with a minimal number of facets each. To formally introduce the facets, we employ another Ωmega feature, namely *user-defined kinds*, whose alternatives can be used to parameterize the *State* datatype:

```
kind Gate = GateClosed | GateOpen
kind Secondary = SecondaryEnabled |
SecondaryBlocked
kind HandshakeIn = Idle | Requested
kind HandshakeOut = Idle' | NeedAck
kind Handshake = H HandshakeIn HandshakeOut
```

```
data State :: Gate ~> Secondary ~> Handshake ~>
* where
  State :: [Transition g s h] → State g s h
```

Figure 1 includes the states along with the facets and the labeling of the transition lists the actions needed e.g. to ensure the working of the handshaking protocol. For now we leave the definition of *Transition* open, we shall supply it in due time.

[1]the second wire can either carry the same data as A (to increase redundancy), or the two wires could jointly carry stream A (to increase bandwidth)

3.3.3. Triggers as Obligations

The state machine holds a distinguished state at any time. Triggers acting on the machine are used to initiate a change of the current state. Additionally, triggers can impose certain constraints that need to be dealt with during the actual transition. To encode triggers we introduce[2]:

```
data Obligation :: (Gate, Secondary, Handshake)
~> (Gate, Secondary, Handshake) ~> * where
  TriggerExclusive :: Obligation (a, b, H Idle c) (a,
b, H Requested c)
  TriggerAck :: Obligation (a, b, H c NeedAck) (a,
b, H c Idle')
  TriggerPlugged :: Obligation (GateClosed, b, H
Idle c) (GateClosed, b, H Idle c)
  ...
```

From the type signatures we can deduce that e.g. the *TriggerAck* clears the *NeedAck* facet on the current state. On the other hand if the obligation raises some facet that corresponds to none of the possible states, then the subsequent action is in charge to clear this facet. To wit *TriggerExclusive* above does raise *Requested*, but no state in Figure 1 has such a facet listed. To obtain a transition system that passes the type checker some action must turn the *Requested* back to *Idle*. We'll talk about transition actions next.

3.3.4. Action Thrists

Actions associated with state transitions can be arbitrarily complex, so instead of putting down a monolythic action for each transition arrow, we wish to modularize our actions. To this end we break down monolithic blocks into *elementary actions* that also carry facets, so the typing constraints can be easily enforced. We choose to implement the elementary actions as a GADT:

```
data Action :: (Gate, Secondary, Handshake) ~>
(Gate, Secondary, Handshake) ~> * where
  RequestExclusive :: Action (a, b, H c Idle') (a, b,
H c NeedAck)
  AckExclusive :: Action (a, b, H Requested c) (a,
b, H Idle c)
```

[2]the code is not syntactically correct (Ωmega has no pairs at the kind level), but demonstrates our point

```
  OpenGate :: Action (GateClosed,
SecondaryBlocked, c) (GateOpen,
SecondaryBlocked, c)
  BlockSecondary :: Action (a, SecondaryEnabled,
c) (a, SecondaryBlocked, c)
  ...
```

Given this definition of elementary actions we can apply our thrist framework and form thrists from these. Then state transitions would correspond to three ingredients:

1 they originate from a state,

2 select a possible transition arrow given a trigger,

3 and finally execute a *Thrist Action* to arrive at the target state.

We define:

```
data Transition :: Gate ~> Secondary ~>
Handshake ~> * where
  From :: State g' s' h' →
          Obligation (g', s', h') (g", s", h") →
          Thrist Action (g", s", h") (g, s, h) →
          Transition g s h
```

We have fulfilled our earlier promise, and now we can define *States*. For instance the *south* state could be defined thus:

```
south = State [From west TriggerAck
[BlockSecondary]l]
```

At last we have succeeded in providing a very compact representation of state machines. From the above expression we can see that only one transition arrow arrives at *south*, how it is triggered and which action follows. The fact that the type checker allows this expression is an additional assurance of correctness.

So what do we gain with such a declarative description of the state machine at hand?

– First, given a fine-grained encoding of the protocol's and business logic's encoding in the state facets, the type system will take care of checking that all necessary elementary actions are mentioned in the transition's action thrist. This can be an important assurance in case of great statemachines with a complicated semantic model. Especially for communicating state machines, as in our example, the administrative states arising as a result of the handshaking protocol can be daunting. Getting the transition actions right between all these states is an error prone task and help from the type system highly welcome.

– Second, the possibility to pattern-match on the actions that form transitions it is possible to enumerate all possible admissible configurations of the communicating state machines and use proof techniques to ensure that the principal invariant of the system is met.

4. Exotic Uses

We have already seen *Thrist* (,) and *Thrist* (→) in the introduction. But several more common two-parameter datatypes exist that *Thrist* can be parametrized with, e.g. *Either*[1], *Equal* and so on. In this section we analyse the formal requirements for inclusion into the *Thrist* framework, constructing adapters as needed, and suggest possible uses.

4.1. Equal Thrist

In Ωmega the *Equal* datatype has two parameters, and is used to track type equality internally. The *Curry-Howard correspondence* is employed to prove propositions encoded in the types of functions, and the resulting *Equal* types can be introduced by *theorem declarations* into the type-checker's rewrite engine. Typical and useful instantiations of *Equal* arise in connection with type functions, e.g. *Equal {plus a b} {plus b a}* which is a manifestation of the *plus* type function's commutativity.

When we assert *lemma* :: *Thrist Equal a a* and *lemma* is a thrist of length greater or equal to 2, then the following function *trans* can be applied to get a proof object by transitivity:

```
trans :: Thrist Equal a b → Equal a b
trans []l = Eq
trans [eq; rest]l = case (eq, trans rest) of
                      (Eq, Eq) → Eq
```

Since the elementary proof objects that are lined up in the thrist already have their types aligned where they meet, it is easy to contract the whole thing to a single witness.

One could go a step farther and consider a (⇒)[2] relation as a binary datatype, with constructors that encode the implication. Then *circ* :: *Thrist* (⇒) *a a* could encode circular implications, equating all intermediate types (propositions).

With this speculative note we leave the realm of curiosity and turn our attention to interesting and useful abstractions that are already part of the functional programmer's weaponry.

4.2. Connection to Arrows and Monads

In the Haskell world, *arrows* [7] also originate from the attempt to generalize function composition. It is helpful to give a comparison of the *Arrow type class* in Haskell and our *Thrist* definition. It is fairly easy to see how the Haskell type class *Arrow* can be expressed as a thrist. Here is a stripped-down definition of *Arrow* to its essence:

```
class Arrow a where
  arr :: (b → c) → a b c
  (>>>) :: a b c → a c d → a b d
  first :: a b c → a (b, d) (c, d)
```

[1]in Ωmega named as (+)

[2]imaginary Ωmega operator

Name	Associative?	Identity?	Total?	Invertible?	Commutative?
Monoid	yes	yes	yes	no	no
Semigroup	yes	no	yes	no	no
Category	yes	yes	no	no	no
Groupoid	yes	yes	no	yes	no
Group	yes	yes	yes	yes	no

Figure 2. Mathematical structures with binary inner operation that is associative

The first method (*Arrow*) is taking an ordinary function into the arrow, the second (») can be regarded as composition and the third (*first*) defines the interaction between arrows and pairs. As we have already stated in the introduction (2.2), *Cons* already provides the rôle of (»). To cover the remaining two, all we have to do is to define a three-parameter datatype with two data constructors *Arr* and *First*:

```
data Arrow' :: (* → * → *) → * → * → * where
  Arr :: (b → c) → Arrow' a b c
  First :: Arrow' a b c → Arrow' a (b, d) (c, d)
```

To obtain the arrow *behaviour*, we have to accompany the *Thrist Arrow'* with an interpretation function that guarantees the arrow laws [7]. Fortunately such a semantics can be canonically defined in terms of the enclosed arrow's methods. Please consult Appendix D for a slightly different, but complete implementation.

As we can see now, thrists (along with an appropriate semantics) do generalize arrows, on the other hand they are not always easily fitted in an arrow. It is the *arr* method that is problematic to provide. Thrists just serve as a container and do not carry a semantics per se, while *Arrow* instances mandate a function argument for the method *arr*.

Monads can also be regarded as a specialization to arrows, so we expect that *Thrist* (*Monad' T*) can be canonically derived. Here *Monad' T a b* represents the type of a monadic actions either starting afresh and resulting a monadic value *T b* or representing a Kleisli arrow *a* → *T b*.

We can define the adapter *Monad'* to have three parameters, the first fixing the *Monad* and the last two to accomodate for the *Thrist* interface:

```
data Monad' :: (* ~> *) ~> * ~> * ~> * where
  Feed :: m b → Monad' m a b
  Digest :: (a → m b) → Monad' m a b
```

Here *Feed* can be regarded as the moral equivalent of the monadic *return*, while *Digest* stands for the *bind* operation (»=).

Again, we have to provide a semantics, to make the embedding complete:

```
runM' :: Monad m → m a →
         Thrist (Monad' m) a b → m b
runM' _ seed []l = seed
runM' v _ [Feed m; rest]l = runM' v m rest
runM' (v@Monad _ bind _) seed [Digest f; rest]l
      = runM' v (bind seed f) rest
```

Some words of explanation are in order. For technical reasons *runM'* receives a monadic *seed* argument, this allows to implement the *Feed* and *Digest* alternatives by simple recursion. The presence of the first argument compensates for a current deficiency of Ωmega (namely the lack of type classes), and is needed for explicit passing of the *Monad* dictionary.

We will stop here, but not without mentioning that recent research has established a connection between dataflow programming and its elegant reformulation using *comonads* [20]. An adapter for comonads is just as straightforward to put down as *Monad'* above.

4.3. Connection to Categories

When looking at the Wikipedia page [25] that explains mathematical structures possessing a binary operation, we can make an attempt to gain some insight of how thrists can be fitted in a niche. Obviously, the *associativity* property holds, like for lists, it is impossible to distinguish thrists appended[1] as long the ordered sequence of elements is pointwise equal. For reference, the table of mathematical structures with an associative binary operation is reproduced in Figure 2. Clearly, the *Cons* operation mandates an order, so *commutativity* cannot hold. Also, not every datatype that can serve as a first parameter to *Thrist* has an inverse, even when disregarding any semantics and concentrating on the types of its constructors. So it remains to find out whether totality holds to settle on the exact concept. Can we always form a new thrist by appending two arbitrary thrists? The answer is clearly *no*, as we have already stated several times, the typing rule for the *Cons* operation may interfere. So, looking up the concept with these properties in Figure 2 brings us straight to *category* [22]. Looking at it from another angle we can imagine *directed graphs*, and thrists corresponding to any *path* between two *nodes*, respecting directionality. But this is nothing new, as directed graphs and categories are the same concept. There is still something missing, though. For a mathematical structure to be a category: each object must have a corresponding *identity morphism*. This is not in general fulfilled for an arbitrary datatype that is passed

[1]see Appendix A.2

to *Thrist*, but is it possible to obtain it for *Thrist p* for an admissible *p*?

Let's look at the problem from another angle first. The type-threaded nature of the *Thrist* datatype is the distinguishing feature of *free categories* [12] too. In the Appendix A.2 we supply an *appendThrist* function that can be regarded as the inner multiplication operation (concatenation) in the free category. So, a free category is a free monoid [24] over an arbitrary category C where C's available morphisms restrict concatenation, or simply put, words formed from objects of C with consecutive letters having an arrow in C between them. The *Thrist*'s typing rule very much tells the same story, and *Nil* : : *Thist p a a* is obviously the identity under *appendThrist.* So we have shown *Thrist p* to be a category, which happens to be a free category C* if the datatype *p* is isomorphic to C.

5. Other Work

David Roundy's *Darcs* [16] is a version control system which builds upon a *theory of patches* [21], defining the evolution of data repositories (and of documents therein) as a gradual application of individual changes. The recent adoption of GADTs into the system has led to a two-parameter *Patch* datatype and *FL*, *RL* GADTs which are used for sequencing these (among others). The definition of *FL* is identical to how our *Thrist* would be defined in Haskell! Being hidden in a utility module, *FL* entered the *Darcs* codebase in the year 2007 but its origins can be traced back to a brain-storming session at the *Haskell'05* workshop in Tallinn.

In his *Fun in the Afternoon* talk [13], Conor McBride proposed the free category *R** as generalization to the conventional list datatype and several others. At the heart of the matter, his *R* stands for the first parameter to our *Thrist*, while the "*" is borrowed from the mathematical notation for free categories, and thus essentially our *Thrist* type constructor together with the *appendThrist* operation. To demonstrate his point,

```
data List' : : * ~> * ~> * where
  Elem : : a → List' a a
```

is a similar adapter to the one in Section 4.2 and *Thrist List'* is the datatype McBride proposes as the new list type. In this special case we really obtain a free category, since *List'* defines identity morphisms.

Ever since the invention of Lisp the program-as-data idiom has been a treasure trove for functional programmers. Chuan-kai Lin's *Unimo* framework [10] is an attempt to describe monads operationally by interpreting a datastructure that describes the monad. By the fact that the interpreter is proven to satisfy the monadic laws, a guarantee is given that the monadic semantics is fulfilled, regardless what callback functions the monad's creator supplies. GADTs appear in his work only as the datatypes modelling the *effect basis* of monads, while they are not needed for the general case.

Another example of the rekindled interest in modelling side-effecting computations in a purely functional manner is given by the Haskell Workshop paper of Swierstra et al. [19], which demonstrates another use of the *free monad*, a data structure built up as an ADT. The result can be examined intensionally by pattern matching, in a similar fashion like our thrists can be taken apart by the semantics functions.

Chris Heunen and Bart Jacobs' work [6] examines the connection of arrows and monads and their category theoretical formalisation. They reveal the mathematical structure behind these constructs, giving a common generalization to them as *monoids.* This is a different result from ours, while separating structure from behaviour we arrive at thrists that are augmented with a semantics.

Arrows [7] appeared in the general mindset as a pragmatic approach to deal with a certain class of parsers that did not fit into the monadic framework [4]. While the analysis aspect (of the combined grammar) was one of the objectives, it happened at runtime on data propagated through the arrow. Our approach enables the analysis of the arrow itself.

Nilsson [14] provides a method for optimizing limited cases of arrow combinator libraries using GADTs. In the scope of his framework for functional reactive programming he is still bound to the limitations of the amesiac nature of function composition inside of the arrow, but seems to gain some noticeable gains in performance especially with microbenchmarks modelling the arrow laws.

6. Conclusion and Further Work

We have found a way to generalize function composition by separating its type structure from its semantics. The data structure we suggest is a GADT with two constructors strongly resembling classical lists, but with a side-condition that the types must be threaded. The semantics is provided by an interpretation function that can be provided separately for each first parameter of the *Thrist* type constructor.

We have provided three examples for the usefulness of the thrist data structure and demonstrated that the ability to take thrists apart and analyse is a very good arguments for their use. Also, all operations performed on thrists, such as subdivision, insertion and extension must be performed by algorithms that preserve the strong typing constraints that are imposed by the *Thrist* type constructor's typing rule. For this to work, it is crucial that *Thrist* is defined to be a generalized algebraic datatype (GADT).

We have further shown that thrists generalize arrows naturally and monads with a shallow adaptation layer. We have identified free categories to be the closest relative of our thrists in the mathematical realm.

Last, but not least, we could successfully exploit Ωmega's extensible syntax to present thrists in a uniform and aestetical way, well alike *Haskell*'s syntax for lists, with the same ease of pattern matching and construction.

Although the above provides sufficient evidence of the usefulness of thrists, there is still plenty to find out.

What other useful *p*s in *Thrist p a b* exist? Since the *a* and *b* parameters can encode propositions, the *Thrist* approach can convey the evolution of abstract properties, e.g. the adherence to the SSA form [1]. Modern compiler architectures[1] tend to favor this formalism for internal representation of imperative programs. Explicitly tracking *def-use* information paired with annotation preserving transformations might pave the way to certified compilers, but appears to be a challenging task.

A similar question is that of restricted monads. Is it possible to provide a more generic adapter than the one presented in Section 4.2, that encompasses restricted monads too?

LLVM [11] has a *getelementpointer* instruction, which allows to do offset calculations into deeply nested (C-style) data structures. The descent from the encompassing pointer, array or structure type to the desired structure member can be conveniently encoded as a thrist, where the underlying GADT provides constructors for dereferencing of pointers, picking of array elements and skipping over (resp. selecting) of structure members. The resulting thrist can then easily converted into the sequence of integers that is encoded in the *getelementpointer* instruction.

Parametricity is a powerful weapon in proving program properties. It seems a worthwile direction of research to find out which sensible algorithms on thrists exist on *Thrist p a b*, where p is left universal. At first glance maps and folds seem to break because of the typing rule, but studying the correspondig free categories might show the way. As these transformations are essential for a wide range of regular data structures, we should be able to use similar higher-order functions too.

While the syntax presented here is already quite palatable, it remains to be seen how thrists can be given syntactic sugar along the lines of the monadic “do” (or even arrow[2]) notation used in current Haskell implementations.

We intend to tackle these theoretical and practical questions as a follow-on.

7. Acknowledgements

My special thanks go to Tim Sheard, whose Ωmega system served as an excellent testbed for formulating the ideas expressed in this paper. The *HaL 2* workshop gave me the opportunity to discuss the connection between categories and thrists, thanks to Heinrich-Gregor Zirnstein, for encouragement and to Johan Jeuring for constructive criticism. Christopher Diggins corrected some of my views on the Cat language and provided valuable input for the presentation. Ganesh Sittampalam told the story of thrists in darcs and gave the final impulse to get this paper published.

Appendix A. Useful Functions on Thrists

Following functions are parametric in the thrist type, i.e. in the first type parameter to the thrist. Thus they are universal.

A.1. Extending

```
extendThrist :: forall (a :: *1) (b :: a ~> a ~> *0) (c :: a) (d :: a) (e :: a).
                Thrist b c d ->
                b d e ->
                Thrist b c e

extendThrist []l a = [a]l
extendThrist [b; r]l a = [b; extendThrist r a]l
```

A.2. Appending

```
appendThrist :: forall (a :: *1) (b :: a ~> a ~> *0) (c :: a) (d :: a) (e :: a).
                Thrist b c d ->
                Thrist b d e ->
                Thrist b c e

appendThrist []l a = a
appendThrist [b; r]l a = [b; appendThrist r a]l
```

A.3. Flattening

```
flattenThrist :: Thrist (Thrist k) a b -> Thrist k a b

flattenThrist []l = []l
flattenThrist [a; as]l = appendThrist a $ flattenThrist as
```

A.4. Instrumenting

It might be useful to wrap each thrist member with a special instrumentation, e.g. for tracing (tracepoints) or other ways of debugging.

Following function (using rank-2 polymorphism) accomplishes this.

```
intersperseThrist :: (forall (x :: *). k x x) -> Thrist k a b -> Thrist k a b

intersperseThrist i []l = [i]l
intersperseThrist i [a; as]l = [i, a; intersperseThrist i as]l
```

[1]GCC [5] and LLVM [11] being notable representants

[2]*arrow syntax* [15] is available in certain Haskell implementations, but not in Ωmega

Appendix B. A Complete Cat Example: *fak*

We need a type-level function for converting type arrows into matching configutations of the top-of-the-stack. This is done by *blowUpBy*. *range* on the other hand finds the result type of a function arrow.

```
blowUpBy :: * ~> * ~> *
{blowUpBy (a → b) s} = (a, {blowUpBy b s})
{blowUpBy Int s} = s
{blowUpBy Bool s} = s

range :: * ~> *
{range (c → d)} = {range d}
{range Int} = Int
{range Bool} = Bool
```

All types in the type universe that our (simplified) Cat program can operate on is witnessed by a value of *Tractable*, and *sameRep* states their equality between them.

```
data Tractable :: * ~> * where
  IntT :: Tractable Int
  BoolT :: Tractable Bool
  ArrT :: Tractable a → Tractable b → Tractable (a →
b)

sameRep :: Tractable a → Tractable b → Maybe
(Equal a b)
sameRep IntT IntT = Just Eq
sameRep BoolT BoolT = Just Eq
sameRep (ArrT a b) (ArrT c d) = let monad
maybeM in do
    Eq ← sameRep a c
    Eq ← sameRep b d
    return Eq
sameRep _ _ = Nothing
```

Now we can introduce the *Cat* datatype which fixes the language primitives and typing rules.

```
data Cat :: * ~> * ~> * where
  Push :: a → Cat opaque (a, opaque)
  Pop :: Cat (a, opaque) opaque
  Dup :: Cat (a, opaque) (a, (a, opaque))
  Prim :: Tractable b → (a → b) → Cat {blowUpBy (a
→ b) opaque} ({range b}, opaque)
  Block :: Thrist Cat b c → Cat opaque (Thrist Cat b
c, opaque)
  Fun :: Function b c → Cat opaque (Thrist Cat b c,
opaque)
  Apply :: Cat (Thrist Cat b c, b) c
  If :: Cat (Thrist Cat s t, Thrist Cat s t, Bool, s) t
  Print :: Cat (a, opaque) opaque
  Call :: (IO b → IO c) → Cat b c
```

To define the needed primitives that can be used in this appendix we propose a small set of some abbreviations:

```
intint = ArrT IntT IntT
intbool = ArrT IntT BoolT

flip f a b = f b a

minus = Prim intint $ flip (-)
times = Prim intint (*)
equals = Prim intbool (≡)
```

We are about to define callable functions. To this end we need a datatype to evidence the transformations on the shape of the stack as performed by the function. It will allow us to compare two functions. Since our example only contains a single function, we do not bother naming them. The identification is taken care of by an atom. *Sound*'s constructors cater for elementary changes in the stack's shape. They can be contained in a thrist to get a net effect.

```
data Sound :: * ~> * ~> * where
  Nop :: Atom opaque → Sound (Cat () ()) (Cat opaque
opaque)
  In :: Tractable new → Sound (Cat a b) (Cat (new, a)
b)
  Out :: Tractable new → Sound (Cat a b) (Cat a (new,
b))

data Function :: * ~> * ~> * where
  Function :: Thrist Sound (Cat () ()) (Cat a b)
              → Thrist Cat a b
              → Function a b

–– build the faculty function
fakFunc :: IO (Function (Int, op) (Int, op))
fakFunc = let monad ioM in do
      a ← freshAtom
      let loosen :: Function (Int, op) (Int, op) →
Function (Int, op') (Int, op')
          loosen = unsafeCast
      let fak = [Dup, Push 0, equals,
                 Block [Pop, Push 1]l,
                 Block [Dup, Push 1, minus, Fun $ loosen
f, Apply, times]l,
                 If]l
          f = Function [Nop a, In IntT, Out IntT]l fak
      return f

compile :: Thrist Cat a b → Code (IO a → IO b)
compile []l = ⟦ id ⟧

compile [Print; rest]l =
  ⟦ λ st → let monad ioM in do
        (a, st') ← st
        putStr $ show a
        $(compile rest) $ return st' ⟧

compile [Pop; rest]l =
  ⟦ λ st → let monad ioM in do
        (_, st') ← st
        $(compile rest) $ return st' ⟧

compile [Dup; rest]l =
  ⟦ λ st → let monad ioM in do
        (st'@(a, _)) ← st
        $(compile rest) $ return (a, st') ⟧
```

```
compile [Push a; rest]l =
  ⟦ λ st → let monad ioM in do
      st' ← st
      $(compile rest) $ return (a, st') ⟧

compile [Prim tr f; rest]l =
  ⟦ λ st → let monad ioM in do
      st' ← st
      let (a', st”) = deepApply f tr st'
      $(compile rest) $ return (a', st”) ⟧

compile [Block true, Block false, If; rest]l =
  ⟦ λ st → let monad ioM in do
      (cond, st') ← st
      let st” = if cond then $(compile true) (return
st') else $(compile false) (return st')
      $(compile rest) st” ⟧

compile [Call c; rest]l =
  ⟦ λ st → let monad ioM in do
      $(compile rest) $ c st ⟧

compile [Fun f, Apply; rest]l =
  ⟦ λ st → let monad ioM in do
      $(compile rest') $ f' st ⟧
  where f' = run $ compileFun f
      rest' = subst f f' rest

deepApply :: (f → g) → Tractable g → (f, {blowUpBy
g u}) → ({range g}, u)
deepApply f (arr@ArrT _ to) (a, st) = deepApply (f
a) to st
deepApply f IntT (a, st) = (f a, st)
deepApply f BoolT (a, st) = (f a, st)

compileFun :: Function f t → Code (IO f → IO t)
compileFun (f@(Function _ c)) = let compiled
= compile $ subst f (mimic run compiled) c in
compiled

subst :: Function f t → (IO f → IO t) → Thrist Cat
a b → Thrist Cat a b
subst _ _ []l = []l

subst (f@Function t _) c [(b@Fun (Function t' _)),
(a@Apply); rest]l =
    case sameSound t t' of
      Just Eq → [Call c; subst f c rest]l
      _ → [b, a; rest]l

subst f c [Block b; rest]l = [Block (subst f c b); subst f
c rest]l
subst f c [head; tail]l = [head; subst f c tail]l


sameSound2 :: Equal a a' → Thrist Sound a b →
Thrist Sound a' b' → Maybe (Equal b b')
sameSound2 Eq [Nop a; rest]l [Nop a'; rest']l = let
monad maybeM in do
    Eq ← same a a'
    Eq ← sameSound2 Eq rest rest'
    return Eq
sameSound2 Eq [In a; rest]l [In a'; rest']l = let monad
maybeM in do
    Eq ← sameRep a a'
    Eq ← sameSound2 Eq rest rest'
    return Eq
sameSound2 Eq [Out a; rest]l [Out a'; rest']l = let
monad maybeM in do
    Eq ← sameRep a a'
    Eq ← sameSound2 Eq rest rest'
    return Eq
sameSound2 Eq []l []l = Just Eq
sameSound2 _ _ _ = Nothing

sameSound = sameSound2 Eq
```

We can now compute a factorial using

```
usecase = let monad ioM in do
    fak ← fakFunc
    let thr = [Push 5, Fun fak, Apply, Print]l
    _ ← (run $ compile thr) (returnIO ())
    return ()
```

```
prompt> usecase
    Executing IO action
    120
```

Appendix C. Parsing C Literals

```
data Parse :: * ~> * ~> * where
  Atom :: Char → Parse Char Char
  Sure :: (a → b) → Parse a b
  Try :: (a → Maybe b) → Parse a b
  Rep1 :: Parse a b → Parse [a] ([b], [a])
  Rep :: Parse [a] (b, [a]) → Parse [a] ([b], [a])
  Group :: [Parse a b] → Parse [a] ([b], [a])
  Par :: Parse a b → Parse c d → Parse (a, c) (b, d)
  Wrap :: Thrist Parse a b → Parse a b


parse :: Thrist Parse a b → a → Maybe b

parse []l a = Just a

parse [Atom c; r]l a = if ord c ≡ ord a then parse r a
else Nothing

parse [Sure f; r]l a = parse r (f a)

parse [Try f; r]l a = do { b ← f a; parse r b } where
monad maybeM

parse [Rep1 p; r]l as = parse r (parseRep p as)
where
      parseRep :: Parse a b → [a] → ([b], [a])
      parseRep _ [] = ([], [])
      parseRep p (as@(a:ar)) = case parse [p]l a of
          Nothing → ([], as)
          Just b → (b:bs, rest)
            where (bs, rest) = parseRep p ar
```

```
parse [Rep p; r]l as = parse r (parseRep [p]l as)
where
    parseRep :: Thrist Parse [a] (b, [a]) → [a] → ([b],
[a])
    parseRep _ [] = ([], [])
    parseRep p as = case parse p as of
                        Nothing → ([], as)
                        Just (b, as') → (b:bs, rest)
                            where (bs, rest) = parseRep
p as'

parse [Group ps; r]l as = do { bs ← parseGroup ps as;
parse r bs } where
    monad maybeM
    parseGroup :: [Parse a b] → [a] → Maybe ([b],
[a])
    parseGroup [] rest = Just ([], rest) -- overlength
input
    parseGroup _ [] = Nothing              -- input
too short
    parseGroup (p:ps) (a:as) = do
                        b ← parse [p]l a
                        (bs, rest) ← parseGroup ps as
                        return (b:bs, rest)

parse [Wrap thr; r]l a = do { a' ← parse thr a; parse
r a' } where monad maybeM
```

Appendix D. Adapting Haskell's *Arrow* Class

We show how the two elementary *Arrow* operations (viz. *arr* and ⋙) are directly related to the two *Thrist* data constructors. This example is in Haskell as Ωmega does not support type classes.

```
module Embeddings where
import Prelude
import Control.Arrow
import Char

data Thrist :: (* → * → *) → * → * → * where
  Nil :: Thrist p a a
  Cons :: p a b → Thrist p b c → Thrist p a c

data Arrow' :: (* → * → *) → * → * → * where
  Arr :: Arrow a ⇒ a b c → Arrow' a b c
  First :: Arrow a ⇒ Arrow' a b c → Arrow' a (b, d)
(c, d)

recover :: Arrow a ⇒ Thrist (Arrow' a) b c → a b c
recover Nil = arr id
recover (Cons (Arr f) r) = f ⋙ recover r
recover (Cons (First a) r) = first (recover $ Cons a
Nil) ⋙ recover r
```

References

[1] Andrew W. Appel. SSA is Functional Programming. *SIGPLAN Notices* **33** (4), 17–20 (1998).

[2] Christopher Diggins. The Cat Programming Language. URL *http://www.cat-language.com/*.

[3] Wikipedia. Common Intermediate Language. URL *http://en.wikipedia.org/wiki/Common_Intermediate_Language*.

[4] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, Error-Correcting Combinator Parsers. In *Advanced Functional Programming, Second International School-Tutorial Text*, pages 184–207. Springer-Verlag, London, UK, 1996.

[5] Diego Novillo. Tree SSA - A New Optimization Infrastructure for GCC. In *GCC Developers Summit*, 2003. URL *http://people.redhat.com/dnovillo/Papers/tree-ssa-gccs03.pdf*.

[6] Chris Heunen and Bart Jacobs. Arrows, like Monads, are Monoids. *Electr. Notes Theor. Comput. Sci.* **158**, 219–236 (2006).

[7] John Hughes. Generalising Monads to Arrows. *Science of Computer Programming* **37**, 67–111 (May 2000). URL *http://www.cs.chalmers.se/~rjmh/Papers/arrows.ps*.

[8] Wikipedia. Java bytecode. URL *http://en.wikipedia.org/wiki/Java_bytecode*.

[9] D. J. P. Leijen, H. J. M. Meijer. Parsec: Direct style monadic parser combinators for the real world. Tech. Rep. UU-CS-2001-35 (2001), Department of Information and Computing Sciences, Utrecht University. URL *http://www.cs.uu.nl/research/techreps/repo/CS-2001/2001-35.pdf*.

[10] Chuan-kai Lin. Programming monads operationally with Unimo. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 274–285. ACM Press, New York, NY, USA, 2006.

[11] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[12] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag. Second Edition, 1998.

[13] Conor McBride. R* is the new [α]. In *Fun in the Afternoon*, Nov 2007. URL *http://sneezy.cs.nott.ac.uk/fun/nov-07/R-star.pdf*.

[14] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 54–65. ACM Press, New York, NY, USA, 2005.

[15] Ross Paterson. A New Notation for Arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, Sep 2001. URL *http://www.soi.city.ac.uk/~ross/papers/notation.html*.

[16] Wikipedia. Darcs. URL *http://en.wikipedia.org/wiki/Darcs*.

[17] Sheard, Tim. Programming in Omega. In *2nd Central European Functional Programming School*, June 2007. URL *http://web.cecs.pdx.edu/~sheard/*.

[18] Tim Sheard and Zine-el-abidine Benaissa and Emir Pasalic. DSL implementation using staging and monads. In *DSL'99: Proceedings of the 2nd conference on Conference on Domain-Specific Languages*, pages 81–94. USENIX Association, Berkeley, CA, USA, 1999. URL *http://www.usenix.org/events/dsl99/full_papers/sheard/sheard.pdf*.

[19] Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 25–36. ACM, New York, NY, USA, 2007.

[20] Tarmo Uustalu and Varmo Vene. The Essence of Dataflow Programming. In Zoltán Horváth, *Lecture Notes in Computer Science*, pages 135–167. Springer. CEFP, 2005.

[21] Wikibooks. Understanding darcs/Patch theory. URL *http://en.wikibooks.org/wiki/Understanding_darcs/Patch_theory*.

[22] Wikipedia. Category (mathematics). URL *http://en.wikipedia.org/wiki/Category_(mathematics)*.

[23] Wikipedia. Dominoes. URL *http://en.wikipedia.org/wiki/Dominoes*.

[24] Wikipedia. Free monoid. URL *http://en.wikipedia.org/wiki/Free_monoid*.

[25] Wikipedia. Groupoid. URL *http://en.wikipedia.org/wiki/Groupoid*.