

Thrists: Dominoes of Data

Gabor Greif

gabor@mac.com

16 July, 2007

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) we can provide an interpretation function which completely emulates the classical function composition (\cdot). 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 two new two-parameter GADTs. 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. The Thrist approach of exposing the intermediate types where the elements are joined together especially shows its potential 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 first section we motivate our later introduction of Thrist with well known examples. In the third section we develop three practical applications of Thrists. We also present some less useful but entertaining uses. Section four shows up possible future directions and summarizes open problems. Section five concludes.

2. Introduction

We are well acquainted with data structures that are parameterized over data types, they are the bread-and-butter tool of functional programmers. For example the List datatype could be introduced in Haskell with the following definition:

```
data List :: * -> * where
  Nil :: List a
  Cons :: a -> List a -> List a
```

We deliberately avoid the traditional syntax of defining the *List* datatype, and use a *GADT-style* definition¹, because we want to build on this fundament 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 to1> : Char -> Int
```

Now, functions are a bit more interesting than Lists above, they can be composed! Interestingly, composition (written as \cdot) 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 (\cdot) 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
      locase
prompt> locase 'G'
```

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

'g' : Char

The type of the composition function constrains the types of its arguments in a nontrivial way: the range of the second function must match with the domain of the first. (Universal types will always match, 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, (where nevertheless all computable functions can be derived from), 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 the 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 chapter shall explain how.

Our crucial observation from the introduction was that the types at the ends of the function arrows must thread up, intuitively¹:

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

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 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

¹we reconcile the conflict between the function arrow's (→) direction and the customary composition g ∘ f by considering a modified reverse composition

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 $a \rightarrow b$? 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
```

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

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, the *runArrowThrist* function:

```
runArrowThrist :: Thrist (->) b c -> c -> b
runArrowThrist Nil b = b
runArrowThrist (Cons f r) a = runThrist 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 implementations.

The definition of *runArrowThrist* on a Cons constructor deserves some attention. We have not yet explicitly mentioned it, but range of the function *f* is existentially

²pronounced "pair"

qualified, because it does not appear in the final result of *Cons*. This implies a minor complication when pattern matching: the *head* component¹ of a *Cons* cannot be passed to non-local functions, because the existential type would escape.

@Considerations [Question: is every thrist a monoid? <http://en.wikipedia.org/wiki/Monoid>]

Looking back at our progress so far, the analogy of thrists with the game of dominoes (<http://en.wikipedia.org/wiki/Dominoes>) could spring into our minds:

Dominoes are laid out in a valid configuration only, 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.

3. Three Practical Thrists

Now that we have defined the Thrist datatype and gave a sufficiently generic interface to cover nontrivial cases, time has come to look for real-world applications. Specifically we shall describe a combinator library for creating ASTs of the Cat language, a statically typed stack-oriented language (<http://www.cat-language.com/>), and a parser combinator library. We shall sketch the use of thrists in transition arrows of state machines and finally give some curious examples that may 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 datatype, 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)
```

¹or for symmetry reasons the *tail* too

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 semantically 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 netto change in the stack's shape.

3.1.2. Making Use of Ω mega's features

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

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

Ω mega's parser and printer perform the conversion to the internal form when the list-like brackets [] followed by the letter "I" are encountered.

We can now continue using this terser syntax:

```
prompt> [Pop, Pop]
[Pop, Pop] :: 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]
the result type: ... was not what was expected: ...
```

The GADT-based type inference fails, because *Add* expects two integers on the stack, but there are two *Chars* 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' [] st = st
interpret' [Push x; rest] st = interpret' rest (x, st)
interpret' [Pop; rest] (a, st) = interpret' rest st
interpret' [Dup; rest] (a, st) = interpret' rest (a, a, st)
interpret' [Add; rest] (a, b, st) = interpret' rest (a + b, st)

```

It works:

```

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

```

With this basic functionality in place, we get bolder and define a primitive with 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' :: Thirst Cat a b -> IO a -> IO b
interpret' [Print; rest] 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 Maybe* containing the monadic *return* and *bind* functions for the *do*-notation's perusal¹. In similar spirit we have to rewrite the other cases too:

```

interpret' [Pop; rest] st = do
  (a, st') <- st
  interpret' rest $ return st'
where monad ioM

```

```

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

```

```

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

```

```

interpret' [Add; rest] 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]
(returnIO ())

```

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

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. Thirst Cat a a) -> IO ()
interpret program = interpret' program $ returnIO ()

```

Obviously this *interpret* function is only called for 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 binary² operators can be formalized in the type-safe way mandated by Ωmega. The expectation is that a binary 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 type. With these definitions we can observe the correct type inference of our *Add* substitute *Prim (+)*:

²or arbitrary arity functions for that matter

³see the definition of *blowUpBy* and *result* in the Appendix

```
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: @TODO

Using Witnesses to Describe Arities

The solution is to attach a *witness* object 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 Ω mega. 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 *witness*¹, because it records the structure of the function's range's type².

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

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 witness.

is here to pattern match on the witness in order to incrementally saturate the Ω mega function present in the primitive: @TODO

3.1.5. Staging the Interpreter

A well-known technique to turn an interpreter into a compiler is staging. The *compile* function takes a Thirst 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 a selection of the Cat primitives only.

```
compile' :: Thirst Cat a b -> Code (IO a -> IO b)
compile' [Print; rest] l = [ | \lambda st -> do
  (a, st') <- st
  putStr $ show a
  $ compiledRest st'
  where monad ioM []
  where compiledRest = compile' rest
```

[TODO: Push, etc.]

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 thirst does not admit wrongly typed Cat programs and the optimization function takes Thirst Cat to Thirst 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 and CIL. 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 Ω mega provides a device that is approaching the power of dependent types, 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
```

$\{ \text{blow } Z s \} = s$
 $\{ \text{blow } (S n) s \} = (t, \{ \text{blow } n s \})$

The interpreter can be written thus:

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

3.2. Application two: a GADT-based Parse Thrist

Traditional monadic parser combinator libraries (like Parsec) suffer from the same problem like the composition operator: they compose easily but cannot be dissected and analysed, or translated to other representations. We proceed similarly to the *Thist* (\rightarrow) and *Thrist Cat* to create a parsing combinator library that is representation agnostic, i.e. can be interpreted or 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 language:

0xCafeBabe 0XE0UL 123456L

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

- 0) token as read from character stream:

0XE0UL

- 1) We match the 0X prefix:

0X|E0UL

- 2) We expect zero or more hexadecimal characters (we come to the explanation of why zero, later)

0X|E0|UL

- 3) We fold the hex string found to a decimal integer

0X|E0|UL

(at this step we would fail if the string had been empty)

- 4) We look for an optional signedness hint

0X|E0|U|L

- 5) We look for an optional storage size hint

0X|E0|U|L

- 6) We encapsulate the distilled information into a token datatype

0X|E0|U|L

3.2.2. Realization

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

```
data Parse :: * ~> * ~> * where
  Epsilon :: Parse [a] ([b], [a]) -- always
  match, return everything
  Atom :: Char → Parse Char Char -- exact
  match
  Sure :: (a → b) → Parse a b -- always match
  and convert
  Try :: (a → Maybe b) → Parse a b -- pipeline
  stops if no match
  Rep1 :: Parse a b → Parse [a] ([b], [a]) --
  consume as many as matches found, return rest
  Rep :: Parse [a] (b, [a]) → Parse [a] ([b], [a])
  ) -- consume as many as matches, return rest
  Group :: [Parse a b] → Parse [a] ([b], [a]) --
  - all must match, return rest
  CataPlus :: ([a] → b) → Parse ([a], c) (b, c) --
  - collapse one or more elements
  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 produces data of type *b*.

- (*Atom 'X'*) matches only the capital 'X' character
- *Epsilon* matches a zero-size prefix of a list of *as* and returns an empty list of *bs* along with the unconsumed rest of *as*
- (*Sure ord*) always 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
- (*Group [Atom 'a', Atom 'b']*) matches only a prefix "ab" in the input, returning it in a pair along with the unmatched portion or fails otherwise
- (*CataPlus foldDec*) consumes a pair of a list of digits and some other data, if the list is empty it

fails, otherwise it folds the sequence to a number, then returns the pair of this number and the other data, which remains unchanged

In the above descriptions we only suggest a possible semantics, the data of type Parse does not mandate it in any way. So when we talked about returns soandso then this is just an intention. The end result of parsing a token will amount to

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

So we expect that our parser that produces Tokens to have a 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) above?

- 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 we know we have to do base 16 conversion later, we can use `Sure snd` to do this obtaining `"E0UL" :: [Char]`.
- Then `Rep1 $ Try hexdigit` will split off two more characters, converting them to hex values on the way, we obtain `([14, 0], "UL") :: ([Int], [Char])`.
- We cannot discard either component, so we proceed in parallel with

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

where the first component produces 224, and the second will proceed by splitting off any 'U' and then splitting off 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.

Putting this all together we can write

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

hexToken = Wrap [Group [Atom '0', Atom 'X']
  , Sure snd
  , Rep1 $ Try hexdigit
  , Par (CataPlus foldHex) signedSized
```

```
, Try numberToken
]l
```

```
tokens = Rep hexToken
```

Of course there are some pitfalls here but the principle is clear:

- The interesting prefix is split off the rest,
- if the prefix is semantically important it gets condensed to a more appropriate form or else discarded,
- parallel processing is used if both components of an input pair are relevant.
- As the execution of the trist proceeds the 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'
```

3.2.5. Compilation

Similarly to the Thrist Cat we can compile our parser combinators to a more efficient algorithm by removing the interpretative overhead.

3.2.6. Analysis

We can run various analyses on our parsers, to ensure that the grammar is unambiguous, for example.

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 combi-

nators that allow parsing a great variety of grammars and optimization and compilation methods that make the parsing process *fast*.

4. Exotic Uses

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

4.1. Equal Thrust

In Ω mega 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 *trans* :: *Thrust Equal a a* and *trans* is a non-empty thrust, then *trans* can be interpreted as a transitivity proof: @TODO Modelling transitivity.

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

On a speculative note, one could come up with a proof algebra system that has types of the form *Equal* (*Equal a b*) (*Equal c d*) in which the outer *Equal* would signify equivalence of propositions.

Then

```
data Proof
  And :: ...
  Impl :: ...
  etc.
```

could serve as a way to formulate proofs

4.2. Connection to Arrows and Monads

Since arrows in the Haskell world also originated from the generalization of function composition it is helpful to give a comparison of the *Arrow type class* in Haskell and our *Thrust* datastructure. Obviously any instance of the Haskell type class *Arrow* can be expressed as a thrust. We have to accompany the *Thrust Arrow* with an interpretation function that is rather canonic, and will be given in the Appendix. This semantics obviously guarantees the arrow laws [hugh-

es19xx]. On the other hand thrusts are not always easily fitted in an arrow. It is the *arr* instance method that is problematic to provide. Thrusts just serve as a container and do not carry a semantics, while *arr* mandates a function argument for the method *arr*.

Nilsson [nilsson2005] embarks on optimization of a GADT-based *Arrow* library, but he only considers the identity function as a special case. He observes a substantial speedup in microbenchmarks when the optimizer is able to discover and short-circuit the identity arrow. In *ParseThrust* the identity can be written in several ways, e.g. as *Wrap Nil* or as *Sure id*. When interpreting, there is probably no significant win in using the former, when compiling to *Code* the underlying metaprogramming system in Ω mega presumably possesses all the information to recover the identity function from the Ω mega interpreter's internals. In case of translation to a different programming system every function inside of *Try* and *Sure* constructors would need auxiliary denotation anyway, and could be optimized either on export or inside of the external system itself.

Monads can also be regarded as a specialization to arrows, so we expect that *Thrust* (*Monad* ' *T*) can be canonically derived.

First we define the adapter to *Monad*, *Monad'* to have two parameters: @TODO

```
data Monad' :: ( * ~> * ) ~> * ~> * ~> * where
  Return :: m b -> Monad' m a b
  Bind :: ( t -> m u ) -> Monad' m t u
```

4.3. Connection to Categories

encoding free categories as Thrusts
 every Thrust gives rise to a category?
 RevThrust is the opposite category?
 commutative diagrams?
 limits?

5. Other Work

Chuan-kai Lin's *Unimo* framework [xxx] 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.

Chris Heunen and Bart Jacobs' work [xxx] on the connection of Arrows and Monads and their category theoretical formalisation is of relevance because it reveals the mathematical structure behind these constructs. Arrows appeared in the general mindset as a pragmatic approach to deal with a certain class of parsers [hughes2000] that did not fit into the monadic

¹in Ω mega named as (+)

²imaginary Ω mega operator

framework. Nilsson provides a method for optimizing limited cases of arrow combinator libraries using GADTs. He is still bound to the limitations of the amesiac nature of function composition inside the arrow framework for functional reactive programming, but seem to gain some noticeable gains in performance especially in microbenchmarks that are 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 *Thrust* type constructor.

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

We have further shown that thrusts generalize arrows naturally and monads with a shallow adaptation layer. Free categories can be modelled as thrusts and we conjecture that computationally relevant categories can be fitted into the *Thrust* framework.

Last, but not least, we could successfully exploit Ω mega's extensible syntax to present thrusts 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 thrusts, there is still plenty to find out.

what other useful *ps* in *Thrust p a b* exist? Since the *a* and *b* parameters can encode propositions, the *Thrust* approach can convey the evolution of abstract properties, e.g. the adherence to the SSA form [appel]. Modern compiler architectures¹ 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 apperars to be a challenging task.

is parametricity in the first parameter useful? I.e. can we formulate sensible functions on *Thrust p a b*, where *p* is left universal?

Given certain callback functions, can the *Thrust* be accompanied with a generic interpretation function, in the style of *Unimo*? Would it be possible to instrument

thrusts with a monadic semantics this way?

Folds on thrusts. Hard because types vary. Existentials? Folds are very important for lists, so we need them.

Syntactical questions: can "do" (monadic) or "proc" ... "do" (arrow²) syntax be used for more convenient construction of thrusts than with the $[a, b; r]$ notation?

More generalization, though not very important for demonstration, generalizing the *Thrust* type signature to accept the last two parameters from any level of the *sort hierarchy* seems useful. Parity in generality at least with the built-in *Equal* type seems desirable³, but even different levels for the second and third parameters are conceivable.

7. Acknowledgements

My special thanks go to Tim Sheard, whose Ω mega 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 thrusts, 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.

Appendix A. Useful Functions on Thrusts

extend, append, instrument (e.g. with tracepoints)

Appendix B. App2

¹GCC [gcc] and LLVM [llvm] being notable representants

²arrow syntax is available in certain Haskell implementatons, but not in Ω mega

³in Ω mega its type signature is $Equal :: level b. forall (a :: * (1 + b)). a \rightarrow a \rightarrow *0$