GHC Core and Linear Logic should be best friends
Haskell Implementors

Carter Tazio Schonwald & Joel Burget

created: 2017-08-16
presented 2017-09-19
last generated 2017-10-09, Mon at 11:12am

Introduction - Authors

Joel

Carter

Our opinions are our own. etc etc :)

What is context of our work?

(i) Our work is motivated by the need to design database and software systems that support modeling ownership and authorization correctly.

(ii) This could be called “Blockchain for finance/enterprise. . . done right”

(iii) And our particular approach is around developing a functional model of linear logic, applying the compiler engineering knowledge ghc and related systems embody, and our work is about building a programming language that prevents errors in ownership descriptions.

What we're here to share today

(i) Motivate linearity/ownership with some simple but nontrivial examples of ownership modeled in Haskell. We only need the idea of a linear pair for this.

(ii) Show how catchable pure exceptions break type safety/soundness of linearity.

(iii) Explain the 6 different type formers of classical linear logic in terms of their functional haskell analogues.

(iv) Show how some of these type formers tie into current work and design directions of ghc!

What *is* a Loan?

```haskell
import Control.Concurrent.STM
import Control.Concurrent.STM.TQueue

newtype InQ  a = InQ  { inQ  :: TQueue a }
newtype OutQ a = OutQ { outQ :: TQueue a }

type Repay = InQ BubbleGum
type Debt = OutQ BubbleGum

data Loan = Loan
  { principal :: Chocolate
  , repay     :: Repay }

-- So many offers. Do I want to take out this loan?
assessLoan :: Loan -> TVar (Set Loan) -> STM (Maybe Loan)
assessLoan loan wallet = undefined
```

What can I do with my bubble gum OutQueue's?

Alice and Bob both owe me some bubblegum. Rick (who's very risk averse) will happily pay me for the first 10 pieces of gum I receive, and his friend Morty (always taking risks) will also pay me for any gum after the first 10! In some circles, this sort of rearrangement is called a *Sequential Tranche*. Sounds like a great job for Concurrency!

So lets figure out how to recompose the OutQueues from Alice and Bob into new OutQueues which Rick and Morty can pay me for.

Concurrency Composes Ownership (1/2)

```
newDebt :: IO (Repay, Debt)
newDebt = fmap (InQ &&& OutQ) newTQueueIO

tranche :: Debt -> Debt -> IO (Debt, Debt)
tranche alice bob = do
    (mortyIn, morty) <- newDebt
    (rickIn, rick) <- newDebt
    _ <- forkIO (allocate alice bob mortyIn rickIn 10)
    return (morty, rick)

allocate :: Debt -> Debt -> Repay -> Repay -> Int -> IO ()
allocate alice bob morty rick remain = do
    taken <- atomically (divvy alice bob morty rick remain)
    allocate alice bob morty rick (remain - taken)
```

Concurrency Composes Ownership (2/2)

```haskell
divvy :: Debt -> Debt -> Repay -> Repay -> Int -> STM Int
divvy alice bob morty rick remain = do
    ma <- tryReadTQueue (outQ alice)
    mb <- tryReadTQueue (outQ bob)
    let available = catMaybes [ma, mb]
    let (safe, risky) = splitAt remain available
    mapM_ (writeTQueue (inQ morty)) safe
    mapM_ (writeTQueue (inQ rick)) risky
    return (length available)
```

linear typing is unsound in the presence of catchable exceptions implicit

```
module BadUncurry where
import Control.Exception

data Fst a = Only a deriving (Show, Typeable, Exception)

badFst :: (Typeable a, Show a) => a -> b -> c
badFst a = throw (Only a)

notOnly (Only a) = a

nonLinear :: (Show a, Typeable a) => (a, b) -> IO a
nonLinear pr = catch
  (return  (uncurry badFst pr))
  (return . notOnly)
```

Possible strategies to recover soundness?

Linear types, even just pairs . . . seem to be unsound in the face of implicit effects.

(i) Type and Effect systems!

(ii) restrict how pure exceptions can be caught.

(iii) only consider exceptions in an STM / transactional effects with rollback setting. (this is the semantics we're delving into for our work).

(iv) Perhaps some sort of checkpointing style roll back as seen

These strategies all seem like an ill fit for GHC today.

Back to business: Linear Logic for the Working Haskeller

The big 6 type operators in linear logic are:

Linear	Haskell	HS CPS ($\neg\neg$)	Name
$\neg X$	$\neg X$ *or* $X \to \forall a.a$	same	negation
$A \otimes B$	(A, B)		tensor or product
$A \oplus B$	$A + B$		sum or either
$A \& B$	?	$\neg(\neg A + \neg B)$	With or (External) Choice
$A \parr B$	?	$\neg(\neg A, \neg B)$	Par

Aside: If you start with a function type that has unboxed tuple arguments/results and sum types, you can encode all of these in the CPS style. See eg: types are calling conventions.

We dont always need linearity to get benefits from linear logic

By relating various control and abstraction representations to their linear logical siblings, a number of really lovely and potentially useful design clarifications arise!

What is & (Choice)

Its sort of a “right hand side of case expressions” that can act as a generalized join point representation.
Modeling the RHS of a case as a collection of functions may force needless closure allocation when in fact that all share the same environment! Choice (&) can be used to encapsulate a collection of continuations (join points) that work in the same scope!

Choice makes case expressions first class

at least for join points, we can go first class

```
case :: ( a + b , a & b ) -> forall r . r
```

What is Par?

consider the humble forkIO

```
forkIO :: IO () -> IO ThreadId
forkIOCPS :: IO ()->(ThreadId -> IO ())-> forall c . c
forkSimpleCPS ::
  (() -> forall c.c ,() -> forall c.c ) -> forall c . c
```

hrmmm ... forkSimpleCPS's type can be written $\neg(\neg(), \neg())$, which is a special case of $\neg(\neg a, \neg b)$. Par is a generalization of forkIO sans ThreadID!
And its resource aware!

Units, so many units

$\&$	empty case
$\oplus$	Void type
$\otimes$	() aka 1
$\parr$	close thread / finalize

GHC Core already has generalized Linear types

Demand analysis and the usage info on every binding *IS* a very interesting fancy linear type system! It just lives in a different place than the rest of the types... and are only used by the optimizer! It has ideas like ≥ 1 and ≤ 1 to name but the simplest two!
linearity / usage here roughly means “entered the closure”, which is stronger than some other notions that seem to be used. (the range of possible notions isn’t cleanly articulated anywhere I’ve seen)

Intuition and Relevance

Some programs dont need linearity, some do, and sometimes we can just erase unused computations, how might we model them?
Theres a lot of different strategies, many of which are on some lattice or rig, usually loosely inspired by the lattice induced by the powerset of $> 1, \equiv 1, < 1$.
Lots of new/ongoing work on that problem, no best answer yet.

Thanks!

questions?