KiCS – The Kiel Curry System

Bernd Braßel, Frank Huch

Christian-Albrechts-University of Kiel

WLP 2007 Würzburg, Germany October 6th, 2007

The KiCS Basic Facts

The KiCS-Compiler

- Source: Curry, a lazy functional logic language
- Target: Haskell, a lazy functional language
- Design Maxim: make functions fast and logic possible
- Research Interest: declarative manipulation of logic features
- Practical Applicability: libraries for net communication, dynamic web pages, data base access, graphical user interfaces
- Interesting Features: lots

The KiCS Design Principles

- make functions fast and logic possible
  Why?

The KiCS Design Principles

- make functions fast and logic possible
  Why?
- shocking but true: most Curry programs are purely functional
  Why??

The KiCS Design Principles

- make functions fast and logic possible
  Why?
- shocking but true: most Curry programs are purely functional
  Why??
- integration of functional and logic features not seamless in practice
  Why???

The KiCS Design Principles

- make functions fast and logic possible
  Why?
- shocking but true: most Curry programs are purely functional
  Why??
- integration of functional and logic features not seamless in practice
  Why???
- easy reasoning: functions lazy, logic strict

The KiCS Design Principles

- make functions fast and logic possible
  Why?
- shocking but true: most Curry programs are purely functional
  Why??
- integration of functional and logic features not seamless in practice
  Why???
- easy reasoning: functions lazy, logic strict

Is this necessarily so? We think not.

The KiCS Design Principles

- make functions fast and logic possible
  Why?
- shocking but true: most Curry programs are purely functional
  Why??
- integration of functional and logic features not seamless in practice
  Why???
- easy reasoning: functions lazy, logic strict

Is this necessarily so? We think not.
So what to do about it? Put more research on logic features.

The KiCS Design Principles

- make functions fast and logic possible
  Why?
- shocking but true: most Curry programs are purely functional
  Why??
- integration of functional and logic features not seamless in practice
  Why???
- easy reasoning: functions lazy, logic strict

Is this necessarily so? We think not.
So what to do about it? Put more research on logic features.
⇒ KiCS allows declarative manipulation of logic features

Logic Feature No. 1: Unification `(=:=) :: a -> a -> Success`

```
appL :: [a] -> [a] -> [a] -> Success
appL []     ys zs     | ys=:=zs = success
appL (x:xs) ys (z:zs) | x=:=z   = appL xs ys zs

appF :: [a] -> [a] -> [a]
appF []     ys = ys
appF (x:xs) ys = x : appF xs ys
```

Curry – a Functional Logic Language

Logic Feature No. 1: Unification `(=:=) :: a -> a -> Success`

```
appL :: [a] -> [a] -> [a] -> Success
appL []     ys zs     | ys=:=zs = success
appL (x:xs) ys (z:zs) | x=:=z   = appL xs ys zs

appF :: [a] -> [a] -> [a]
appF []     ys = ys
appF (x:xs) ys = x : appF xs ys
```

Logic Feature No. 2: Free Variables `(... where x free)`

```
main | appL [1,2,3] [4,5] x = x where x free
```

Equivalent formulation:

```
main = let x free in appL [1,2,3] [4,5] x &> x
```

Demo 1: Narrowing, Strict vs. Lazy

Curry – Concurrency and Residuation

Logic Feature No. 3: Residuation `($#),($##)::(a -> b) -> a -> b`
Meaning:

`(f $# e)`	suspends if e evaluates to free variable ground head normal form
`(f $## e)`	suspends if e evaluates to term containing free variable ground normal form

Logic Feature No. 4: Concurrency `(&)::Success->Success->Success`
Meaning:
`(e1 & e2)` on suspension switch evaluation between e1 and e2

Problems of Residuation

General Problem: Flaundering depending on order of evaluation

```
main = let x free
           y = id $# x
           z = x=:=True
        in ifAndOnlyIf y z
```

Specific Problem in Lazy Language:

- evaluation order far from trivial
- conflict with goals of software engineering
  you need to know how `ifAndOnlyIf` is defined
- both problems add up awkwardly
  try to debug suspensions with rigid library functions

Two complex mechanisms to control evaluation order is one too much.

Problems of Residuation

General Problem: Flaundering depending on order of evaluation

```
main = let x free
           y = id $# x
           z = x=:=True
       in ifAndOnlyIf z y
```

Specific Problem in Lazy Language:

- evaluation order far from trivial
- conflict with goals of software engineering
  you need to know how `ifAndOnlyIf` is defined
- both problems add up awkwardly
  try to debug suspensions with rigid library functions

Two complex mechanisms to control evaluation order is one too much.

Non-Determinism to the Rescue

Logic Feature No. 5: Non-Deterministic Choice `(?) :: a -> a -> a`
`(e1 ? e2)` can be reduced to `e1` or `e2`.

```
insert :: a -> [a] -> [a]
insert x ys = (x:ys) ? (head ys : insert x (tail ys))

permute :: [a] -> [a]
permute []     = []
permute (x:xs) = insert x (permute xs)
```

Non-Determinism to the Rescue

Logic Feature No. 5: Non-Deterministic Choice `(?) :: a -> a -> a`
`(e1 ? e2)` can be reduced to `e1` or `e2`.

```
insert :: a -> [a] -> [a]
insert x ys = (x:ys) ? (head ys : insert x (tail ys))

permute :: [a] -> [a]
permute []     = []
permute (x:xs) = insert x (permute xs)
```

Another example: generating values of certain type

```
type Generator a = () -> a

bool :: Generator Bool
bool _ = True ? False

list :: Generator a -> Generator [a]
list x _ = [] ? (x () : list x ())
```

Demo 2: permute, generator, or-mode, breadth first search

Eliminating Free Variables and Residuation

The Idea

- generators can replace free variables
- laziness can replace residuation

Is this feasable?

- How about Narrowing?

```
head (x:_) = x
ex1 = head xs =:= True where xs free
```

This example terminates with Narrowing. And without free variables?

Eliminating Free Variables and Residuation

The Idea

- generators can replace free variables
- laziness can replace residuation

Is this feasable?

- How about Narrowing?

```
head (x:_) = x
ex1 = head xs =:= True where xs free
```

  This example terminates with Narrowing. And without free variables?

- Generators need breadth first search. Does this work out?

```
matchBoolList [True] = success
ex2 = matchBoolList xs where xs free
```

  Will evaluating `main` with depth first search terminate?

Eliminating Residuation 2

- How about external types like integers?
  To get generator of a certain type a constructive definition is needed.

```
data Nat = O Nat | I Nat | IHi
data Int = Pos Nat | Zero | Neg Nat
ex3 = let x,y free in x+y=:=7 &> (x,y)
```

Eliminating Residuation 2

- How about external types like integers?
  To get generator of a certain type a constructive definition is needed.

```
data Nat = O Nat | I Nat | IHi
data Int = Pos Nat | Zero | Neg Nat
ex3 = let x,y free in x+y=:=7 &> (x,y)
```

- But sometimes the search space *is* too big. Suspension can prune it.

```
ex4 = let x,y free in (x+y=:=7 & x=:=3) &> (x,y)
```

  Will this program terminate?

Eliminating Residuation 2

- How about external types like integers?
  To get generator of a certain type a constructive definition is needed.

```
data Nat = O Nat | I Nat | IHi
data Int = Pos Nat | Zero | Neg Nat
ex3 = let x,y free in x+y=:=7 &> (x,y)
```

- But sometimes the search space *is* too big. Suspension can prune it.

```
ex4 = let x,y free in (x+y=:=7 & x=:=3) &> (x,y)
```

  Will this program terminate?

- Information propagation:

```
plusL :: Int -> Int -> Int -> Success
plusL x y z = x+y=:=z & z-x=:=y & z-y=:=x
ex5 = let x,y free in
        (plusL x y 7 & y=:=12313432) &> (x,y)
```

Considerations about Eliminating Residuation

The new approach is more general

- employs laziness only
- works for arbitrary Curry functions without further annotations

```
ex6 = let x,y free in
      (appF x y =:= [1,2,3] & x=:=[1]) &> (x,y)
```

ex6 has an infinite search space with residuation

- possibilities you cannot express with suspension

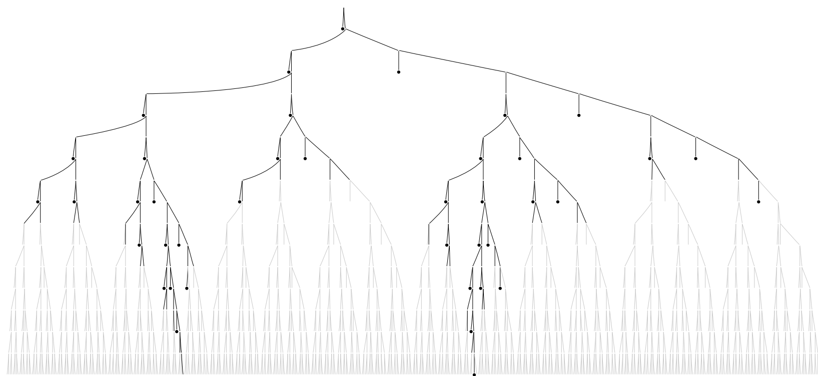
```
one, two :: [[Bool]] -> Success
one [x] = all' x
two [x,y] = all' x

all' :: [Bool] -> Success
all' (True:xs) = all' xs

ex7 = let x free in one x & two x
```

The last Logic Feature

Logic feature No. 7 Encapsulated Search

```
data SearchTree a = Fail | Value a | Or [SearchTree a]

searchTree :: a -> SearchTree a -- defined externally

dfs,bfs :: a -> [a]
dfs x = dfs' (searchTree x)
  where dfs' Fail      = []
        dfs' (Value v) = [v]
        dfs' (Or ts)   = concat (map dfs' ts)

bfs x = bfs' [searchTree x]
  where bfs' ts =  [ v | Value v <- ts]
                ++ bfs' (concat [ ts' | Or ts' <- ts])

ex8 = take 10 (dfs (list bool ()))
```

Arbitrary Search Possible: A Search Tree from a Test Tool



Make Functions Fast and Logic Possible

What makes logic attractive in KiCS

- user programmable search
- referential transparent and expressive encapsulation
- sharing across non-determinism

Sharing across non-determinism

Deterministic computations are only evaluated once, regardless of the state of the search.

Demo 3: the debugger

```
(+) :: Nat -> Nat -> Nat
O x + O y = O (x + y)
O x + I y = I (x + y)
O x + IHi = I x
I x + O y = I (x + y)
I x + I y = O (succ (x + y))
I x + IHi = O (succ x)
IHi + y   = succ y

ex10 = toNat 6 + toNat 2

toNat (Pos x) = x
```

Conclusions

The Kiel Curry System

- compiler from Curry to Haskell
- main research interest is declarative manipulation of logic features
- mode to use generators instead of free variables
- new approach to concurrency
- sharing across non-determinism

Future Work

- make use of haskell optimizations
- improve debugger
- further research on lazy logic features
- compare to information propagation techniques