

# A Parametric Model of $PILL_{\gamma}$ from the operational semantics of LILY.

Rasmus Lerchedahl Petersen

Programming, Logic and Semantics Group  
IT University of Copenhagen

March 6th 2006 / Lunch Talk

# Outline

- 1 Motivation
  - Parametricity

# Outline

- 1 Motivation
  - Parametricity
- 2 The Theory
  - The Language
  - The Logic
  - The Model

# Outline

- 1 Motivation
  - Parametricity
- 2 The Theory
  - The Language
  - The Logic
  - The Model
- 3 LILY as an LAPL-structure
  - LILY
  - A Counter Example

# Outline

- 1 Motivation
  - Parametricity
- 2 The Theory
  - The Language
  - The Logic
  - The Model
- 3 LILY as an LAPL-structure
  - LILY
  - A Counter Example
- 4 Results

# Outline

- 1 Motivation
  - Parametricity
- 2 The Theory
  - The Language
  - The Logic
  - The Model
- 3 LILY as an LAPL-structure
  - LILY
  - A Counter Example
- 4 Results

# Polymorphism

Strict typing discipline has proven to be an efficient tool for developing structured programs

## Limitations

However, distinguishing programs with different types, the typing discipline would consider the following two programs different

```
int - reverse : int - list → int - list
string - reverse : string - list → string - list
```

although they both simply reverse a list, and thus perform essentially the same operation.



# Polymorphism

## Solution

Polymorphism allows one to define a general reverse function

$$\text{reverse} : \forall \alpha. \alpha - \text{list} \rightarrow \alpha - \text{list}$$

which yields the needed functions upon instantiation:

$$\text{int} - \text{reverse} = \text{reverse}(\text{int})$$
$$\text{string} - \text{reverse} = \text{reverse}(\text{string})$$

`reverse` is then called a *polymorphic* function.

# Parametric Polymorphism

For a language with polymorphism we can consider parametricity.

## Intuition

Intuitively parametricity is the statement that

“Polymorphic functions behave the same on all type-instantiations”

For example, a function  $t$  of type  $\forall \alpha. \alpha \rightarrow \alpha$  can not instantiate to the identity on some types and to the successor function on the natural numbers. In fact the program  $t$  can only be the polymorphic identity function.



# Formalizing Parametricity

We use a formalization due to Reynolds called

## Relational Parametricity

Intuitively relational parametricity is the statement that

“Polymorphic functions respect all relations”

# Formalizing Parametricity

We use a formalization due to Reynolds called

## Relational Parametricity

Intuitively relational parametricity is the statement that

“Polymorphic functions respect all relations”

As an example consider the relation  $R$  from `letters` to `numbers` given by

$$a R 1 \quad b R 2 \quad c R 3 \dots$$

# Formalizing Parametricity

We use a formalization due to Reynolds called

## Relational Parametricity

Intuitively relational parametricity is the statement that

“Polymorphic functions respect all relations”

As an example consider the relation  $R$  from `letters` to `numbers` given by

$$a R 1 \quad b R 2 \quad c R 3 \dots$$

This may be lifted to a relation  $R$  – `list` from `letter – lists` to `number – lists` whereby

$$c, e, a, b, d \ R\text{-list} \ 3, 5, 1, 2, 4$$



# Formalizing Parametricity

If `reverse` is relational parametric we then require

```
reverse(letter - list)(c, e, a, b, d)
      R - list
reverse(number - list)(3, 5, 1, 2, 4)
```

# Formalizing Parametricity

If `reverse` is relational parametric we then require

$$\begin{array}{c} \text{reverse}(\text{letter} - \text{list})(c, e, a, b, d) \\ \quad \quad \quad R - \text{list} \\ \text{reverse}(\text{number} - \text{list})(3, 5, 1, 2, 4) \end{array}$$

which means, that if

$$\text{reverse}(\text{letter} - \text{list})(c, e, a, b, d) = (d, b, a, e, c)$$


# Formalizing Parametricity

If `reverse` is relational parametric we then require

$$\begin{array}{c} \text{reverse}(\text{letter} - \text{list})(c, e, a, b, d) \\ R - \text{list} \\ \text{reverse}(\text{number} - \text{list})(3, 5, 1, 2, 4) \end{array}$$

which means, that if

$$\text{reverse}(\text{letter} - \text{list})(c, e, a, b, d) = (d, b, a, e, c)$$

we can be sure that

$$\text{reverse}(\text{number} - \text{list})(3, 5, 1, 2, 4) = (4, 2, 1, 5, 3)$$


# Formalizing Parametricity

“In fact the program  $t$  can only be the polymorphic identity function.”

# Formalizing Parametricity

“In fact the program  $t$  can only be the polymorphic identity function.”

Recall  $t$  had type  $\forall\alpha.\alpha \rightarrow \alpha$ .

# Formalizing Parametricity

“In fact the program  $t$  can only be the polymorphic identity function.”

Recall  $t$  had type  $\forall\alpha.\alpha \rightarrow \alpha$ .

Let  $\sigma$  be a type and consider  $x : \sigma$ .

We wish to show  $t(\sigma)(x) = x$ .



# Formalizing Parametricity

“In fact the program  $t$  can only be the polymorphic identity function.”

Recall  $t$  had type  $\forall\alpha.\alpha \rightarrow \alpha$ .

Let  $\sigma$  be a type and consider  $x : \sigma$ .

We wish to show  $t(\sigma)(x) = x$ .

Define  $R$  from  $\sigma$  to  $\sigma$  by

$$a R b \iff a = x$$



# Formalizing Parametricity

“In fact the program  $t$  can only be the polymorphic identity function.”

Recall  $t$  had type  $\forall \alpha. \alpha \rightarrow \alpha$ .

Let  $\sigma$  be a type and consider  $x : \sigma$ .

We wish to show  $t(\sigma)(x) = x$ .

Define  $R$  from  $\sigma$  to  $\sigma$  by

$$a R b \iff a = x$$

Since  $x R x$ , we know that  $t(\sigma)(x) R t(\sigma)(x)$  which means that  $t(\sigma)(x) = x$ .



# Outline

- 1 Motivation
  - Parametricity
- 2 The Theory
  - The Language
  - The Logic
  - The Model
- 3 LILY as an LAPL-structure
  - LILY
  - A Counter Example
- 4 Results

# PILL<sub>γ</sub>

We study parametricity of functions written in PILL<sub>γ</sub>, which has the following types  $\tau$  and terms  $M$ :

$$\begin{aligned}
 \tau & ::= \alpha \mid \tau \multimap \tau \mid \forall \alpha. \tau \mid !\tau \mid I \mid \tau \otimes \tau \\
 M & ::= \mathbf{a} \mid \mathbf{x} \mid \lambda^\circ \mathbf{a} : \tau. M \mid M_1 M_2 \mid \Lambda \alpha. M \mid M \tau \mid \\
 & \quad Y \mid !M \mid \text{let } !\mathbf{x} \text{ be } M_1 \text{ in } M_2 \mid * \mid \\
 & \quad \text{let } * \text{ be } M_1 \text{ in } M_2 \mid M_1 \otimes M_2 \mid \\
 & \quad \text{let } \mathbf{a}_1 \otimes \mathbf{a}_1 \text{ be } M_1 \text{ in } M_2
 \end{aligned}$$

# PILL<sub>γ</sub> types

## Linearity

$\sigma \multimap \tau$  is to be thought of as linear functions from  $\sigma$  to  $\tau$ .



# PILL<sub>γ</sub> types

## Linearity

$\sigma \multimap \tau$  is to be thought of as linear functions from  $\sigma$  to  $\tau$ .

## Polymorphism

$\forall \alpha. \tau$  is polymorphic functions. An element of this type produces an element of type  $\tau[\sigma/\alpha]$  once supplied with a type  $\sigma$ .

# PILL<sub>γ</sub> types

## Linearity

$\sigma \multimap \tau$  is to be thought of as linear functions from  $\sigma$  to  $\tau$ .

## Polymorphism

$\forall \alpha. \tau$  is polymorphic functions. An element of this type produces an element of type  $\tau[\sigma/\alpha]$  once supplied with a type  $\sigma$ .

## Intuitionism

$! \tau$  can be thought of as “ $\tau$ -factories”. A single element of this type can produce any amount of elements of type  $\tau$ .

# PILL<sub>γ</sub> types

## Linearity

$\sigma \multimap \tau$  is to be thought of as linear functions from  $\sigma$  to  $\tau$ .

## Polymorphism

$\forall \alpha. \tau$  is polymorphic functions. An element of this type produces an element of type  $\tau[\sigma/\alpha]$  once supplied with a type  $\sigma$ .

## Intuitionism

$! \tau$  can be thought of as “ $\tau$ -factories”. A single element of this type can produce any amount of elements of type  $\tau$ .

## Emptiness

$/$  is “things that can be discarded”.



# Intuitionistic function space

Since an ordinary function can only use its argument a finite number of times (when denoted by a finite term), we can define ordinary function space  $\sigma \rightarrow \tau$  as

$$\sigma \rightarrow \tau = !\sigma \multimap \tau$$

Even though a function of type  $!\sigma \multimap \tau$  can only place exactly one order at its  $\sigma$  factory this order can be arbitrarily large (even empty).



# PILL<sub>γ</sub> terms

## Fixed points

$Y$  is a fixed point combinator it has type  $\forall\alpha.!(\alpha \multimap \alpha) \multimap \alpha$  or equivalently  $\forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha$ .

# PILL<sub>γ</sub> terms

## Fixed points

$Y$  is a fixed point combinator it has type  $\forall\alpha.!(\alpha \multimap \alpha) \multimap \alpha$  or equivalently  $\forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha$ .

## Factories

$!M$  is an  $M$ -factory.

# PILL<sub>γ</sub> terms

## Fixed points

$Y$  is a fixed point combinator it has type  $\forall\alpha.!(\alpha \multimap \alpha) \multimap \alpha$  or equivalently  $\forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha$ .

## Factories

$!M$  is an  $M$ -factory.

## Nothing

$*$  is a “thing that can be discarded”.

# Explosion

Since a factory can be ordered to produce zero elements, it can in effect be discarded. The term

$$\text{discard} = \lambda^\circ x \ !:\sigma. \text{let } !y \text{ be } x \text{ in } * : \textit{sigma} \multimap I$$

formalizes this in that for any  $\Xi \mid \Gamma; \Delta \vdash M \ !:\sigma$

$$\Xi \mid \Gamma; \Delta \vdash \text{discard } M : I$$



# Outline

- 1 Motivation
  - Parametricity
- 2 The Theory
  - The Language
  - **The Logic**
  - The Model
- 3 LILY as an LAPL-structure
  - LILY
  - A Counter Example
- 4 Results

## LAPL

Propositions in the logic exist in contexts of free type variables, free variables of  $PILL_{\gamma}$  and free relational variables. The free relational variables may be relations or admissible relations. Propositions are written as

$$\vec{\alpha} \mid \vec{x} : \vec{\sigma} \mid \vec{R} : \text{Rel}(\vec{\sigma}, \vec{\sigma}'), \vec{S} : \text{AdmRel}(\vec{\tau}, \vec{\tau}') \vdash \phi : \text{Prop.}$$



# Admissible Relations

Allowing all relations in the notion of relational parametricity leads to problems. Thus we have axiomatized the properties of “a set of suitable relations”. We call this a notion of admissible relations.

# Admissible Relations

Allowing all relations in the notion of relational parametricity leads to problems. Thus we have axiomatized the properties of “a set of suitable relations”. We call this a notion of admissible relations.

## The Admissible Relations

- contain all equality relations
- contain all graph relations
- are closed under reindexing

# Outline

- 1 Motivation
  - Parametricity
- 2 **The Theory**
  - The Language
  - The Logic
  - **The Model**
- 3 LILY as an LAPL-structure
  - LILY
  - A Counter Example
- 4 Results

# Stage I: A DILL-model

## *Type*

It is well known that a model of the simply typed lambda calculus is a cartesian closed category.

## Stage I: A DILL-model

*LinType*

*Type*

It is well known that a model of the simply typed lambda calculus is a cartesian closed category.

In the same way a model of the linear lambda calculus is a symmetric monoidal closed category.



## Stage I: A DILL-model

$$\text{LinType} \begin{array}{c} \xrightarrow{\quad} \\ \perp \\ \xleftarrow{\quad} \end{array} \text{Type}$$

It is well known that a model of the simply typed lambda calculus is a cartesian closed category.

In the same way a model of the linear lambda calculus is a symmetric monoidal closed category.

The correspondence,  $\sigma \rightarrow \tau = !\sigma \multimap \tau$ , between the two is so strong that it gives rise to an adjunction.



## Stage I: A DILL-model

$$\text{LinType} \begin{array}{c} \xrightarrow{\quad} \\ \perp \\ \xleftarrow{\quad} \end{array} \text{Type}$$

It is well known that a model of the simply typed lambda calculus is a cartesian closed category.

In the same way a model of the linear lambda calculus is a symmetric monoidal closed category.

The correspondence,  $\sigma \rightarrow \tau = !\sigma \multimap \tau$ , between the two is so strong that it gives rise to an adjunction.

In fact the intuitionistic part is the (cartesian closure of the) Kleisli category for the monad induced by !.

## Stage II: A PILL-model

Polymorphism is added in the standard way:

## Stage II: A PILL-model

Polymorphism is added in the standard way:

$$\Gamma; \Delta \vdash M : \sigma \mapsto \Xi \mid \Gamma; \Delta \vdash M : \sigma$$



## Stage II: A PILL-model

Polymorphism is added in the standard way:

$$\Gamma; \Delta \vdash M : \sigma \mapsto \Xi \mid \Gamma; \Delta \vdash M : \sigma$$

$\Xi = \alpha_1, \dots, \alpha_n$  denote the free type variables. If we substitute in types for those, we get an ordinary DILL typing.

## Stage II: A PILL-model

Polymorphism is added in the standard way:

$$\Gamma; \Delta \vdash M : \sigma \mapsto \Xi \mid \Gamma; \Delta \vdash M : \sigma$$

$\Xi = \alpha_1, \dots, \alpha_n$  denote the free type variables. If we substitute in types for those, we get an ordinary DILL typing. In this way we can see  $\Xi \mid \Gamma; \Delta \vdash M : \sigma$  as a function

$$\Omega^n \rightarrow \text{DILL}$$

where  $\Omega$  denotes “all types”.



## Stage II: A PILL-model

This suggest indexed categories (indexed by  $\Omega^n$ ), but we like to formulate is as the equivalent fibrations:

## Stage II: A PILL-model

This suggest indexed categories (indexed by  $\Omega^n$ ), but we like to formulate is as the equivalent fibrations:

*LinType*  $\overset{\quad}{\underset{\perp}{\rightleftarrows}}$  *Type*

$\mapsto$

*LinType*  $\overset{\quad}{\underset{\perp}{\rightleftarrows}}$  *Type*  
 $\searrow$   
 $\swarrow$   
*Kind*



## Stage II: A PILL-model

This suggest indexed categories (indexed by  $\Omega^n$ ), but we like to formulate is as the equivalent fibrations:

$$\text{LinType} \begin{array}{c} \xrightarrow{\quad} \\ \perp \\ \xleftarrow{\quad} \end{array} \text{Type}$$

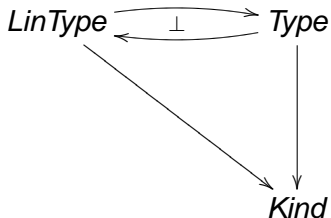
$\mapsto$

$$\begin{array}{ccc} \text{LinType} & \begin{array}{c} \xrightarrow{\quad} \\ \perp \\ \xleftarrow{\quad} \end{array} & \text{Type} \\ & \searrow & \swarrow \\ & \text{Kind} & \end{array}$$

If we then require an element of type  $\forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha$  to exist and behave like a fixed point combinator, we have enough structure to model  $\text{PILL}_\gamma$ .



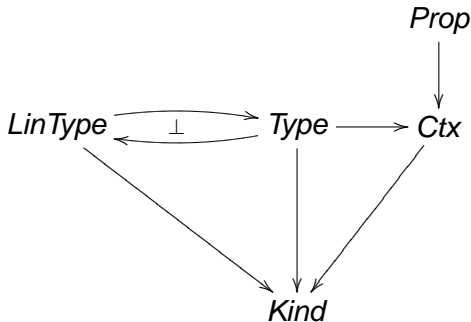
## Stage III: An LAPL-structure



In order to also model the logic, we introduce. . .



## Stage III: An LAPL-structure



In order to also model the logic, we introduce. . . a logic fibration.

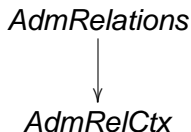


## Stage III: An LAPL-structure

$$Type \times_{Kind} Type \longrightarrow Ctx$$

If there is a functor  $Type \times_{Kind} Type \rightarrow Ctx$  giving for each type the set of relations on this type, and it has a subfunctor that enjoys the required closure properties of admissible relations, we can model admissible relations.

## Stage III: An LAPL-structure



If there is a functor  $\textit{Type} \times_{\textit{Kind}} \textit{Type} \rightarrow \textit{Ctx}$  giving for each type the set of relations on this type, and it has a subfunctor that enjoys the required closure properties of admissible relations, we can model admissible relations.

From this structure we may construct yet a PILL-model of admissible relations.

## Stage III: An LAPL-structure

$$\begin{array}{ccc}
 \text{LinType} & \longrightarrow & \text{AdmRelations} \\
 \downarrow J & & \downarrow \\
 \text{Kind} & \xrightarrow{J} & \text{AdmRelCtx}
 \end{array}$$

If there is a functor  $\text{Type} \times_{\text{Kind}} \text{Type} \rightarrow \text{Ctx}$  giving for each type the set of relations on this type, and it has a subfunctor that enjoys the required closure properties of admissible relations, we can model admissible relations.

From this structure we may construct yet a PILL-model of admissible relations. If there is a fibred functor,  $J$ , giving each type a relational interpretation, we can model LAPL. Hence we call this an LAPL-structure.

## Stage III: An LAPL-structure

The LAPL structure is said to be parametric, if identity extension, as formulated in LAPL, holds in the model.

### Identity extension

$$\sigma[eq_{\vec{\alpha}}] \equiv eq_{\sigma(\vec{\alpha})}$$

# Outline

- 1 Motivation
  - Parametricity
- 2 The Theory
  - The Language
  - The Logic
  - The Model
- 3 LILY as an LAPL-structure**
  - LILY**
  - A Counter Example
- 4 Results

# PILL $\gamma$ with an operational semantics

Bierman, Pitts and Russo have defined an operational semantics for (a language equivalent to) PILL $\gamma$ .

# PILL<sub>γ</sub> with an operational semantics

Bierman, Pitts and Russo have defined an operational semantics for (a language equivalent to) PILL<sub>γ</sub>.

Thus one might write an interpreter (or even a compiler) and run PILL<sub>γ</sub> programs.

## PILL<sub>γ</sub> with an operational semantics

Bierman, Pitts and Russo have defined an operational semantics for (a language equivalent to) PILL<sub>γ</sub>.

Thus one might write an interpreter (or even a compiler) and run PILL<sub>γ</sub> programs.

This allows us to equate terms based on their operational behavior rather than provability in the logic (as we did previously).



## Ongoing work

Building an LAPL-structure from equivalence classes of LILY-terms would prove the consequences of parametricity for LILY.

## Ongoing work

Building an LAPL-structure from equivalence classes of LILY-terms would prove the consequences of parametricity for LILY.

So that we did. . .

# The $PILL_{\gamma}$ model

As  $LILY$  and  $PILL_{\gamma}$  are equivalent, we may see our new  $PILL_{\gamma}$  model as a quotient of the syntactic model.

# The $PILL_{\gamma}$ model

As  $LILY$  and  $PILL_{\gamma}$  are equivalent, we may see our new  $PILL_{\gamma}$  model as a quotient of the syntactic model.

Thus we just have to prove, that the equality theory of  $PILL_{\gamma}$  can be lifted to equivalence classes (Carsten did this).



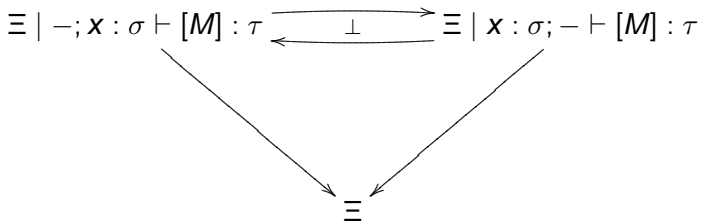
# The $PILL_{\gamma}$ model

As  $LILY$  and  $PILL_{\gamma}$  are equivalent, we may see our new  $PILL_{\gamma}$  model as a quotient of the syntactic model.

Thus we just have to prove, that the equality theory of  $PILL_{\gamma}$  can be lifted to equivalence classes (Carsten did this).

And that all functors still are welldefined and all adjunctions still hold (I did this).

# The $PILL_{\gamma}$ model



## Logic: Ctx and Prop

To model the logic, we use the category of sets:

- Types are modeled as sets.
- Propositions on types are modeled as subsets.
- Admissible relations are modelled as  $\top\top$ -closed relations.
- $J$  is given by the  $\Delta$  relation.

# Logic: Types as sets

Each type  $\alpha_1, \dots, \alpha_n \vdash \sigma : \text{Type}$  gives rise to a function  $\text{Typ}^n \rightarrow \text{Typ}$  from closed types to closed types.

# Logic: Types as sets

Each type  $\alpha_1, \dots, \alpha_n \vdash \sigma : \text{Type}$  gives rise to a function  $\text{Typ}^n \rightarrow \text{Typ}$  from closed types to closed types.

We model the closed type  $- \vdash \tau : \text{Type}$  by the set of closed terms of type  $\tau$ ,  $S(\tau) = \{M \mid -; - \vdash m : \tau\}$ .

## Logic: Types as sets

Each type  $\alpha_1, \dots, \alpha_n \vdash \sigma : \text{Type}$  gives rise to a function  $\text{Typ}^n \rightarrow \text{Typ}$  from closed types to closed types.

We model the closed type  $- \vdash \tau : \text{Type}$  by the set of closed terms of type  $\tau$ ,  $\mathcal{S}(\tau) = \{M \mid -; - \vdash m : \tau\}$ .

Thus  $\sigma$  is modelled as the function

$$(\tau_1, \dots, \tau_n) \mapsto \mathcal{S}(\sigma[\tau_1, \dots, \tau_n / \alpha_1, \dots, \alpha_n])$$



# Logic: Propositions as subsets

A proposition is modelled as a pointwise subset.

# Logic: Propositions as subsets

A proposition is modelled as a pointwise subset.

Thus a proposition on  $\sigma$  is modelled as a function  $\rho : \text{Typ}^n \rightarrow \text{Set}$  such that for any  $(\tau_1, \dots, \tau_n) \in \text{Typ}^n$

$$\rho(\tau_1, \dots, \tau_n) \subseteq \mathcal{S}(\sigma[\tau_1, \dots, \tau_n / \alpha_1, \dots, \alpha_n])$$



# Outline

- 1 Motivation
  - Parametricity
- 2 The Theory
  - The Language
  - The Logic
  - The Model
- 3 **LILY as an LAPL-structure**
  - LILY
  - **A Counter Example**
- 4 Results

# Parametric Counters

On the blackboard. . .

# Parametric Counters

On the blackboard. . .

(This is real math. . .)

# Results

A concrete LAPL-structure has been built out of the operational semantics of LILY. Through this we have established:

- The LAPL logic can be used to reason about LILY terms.
- Encoding of recursive types is correct for LILY.
- We have reasoning principles for recursive types in LILY (Møgelberg).

## Future Work

- Extending the calculus with other effect, pointers. . .
- Rasmus Møgelberg is interpreting FPC in an LAPL-structure
- Carsten Varming might look at interpreting PCF
- Representation in Twelf?

# The End

Thanks for listening :)