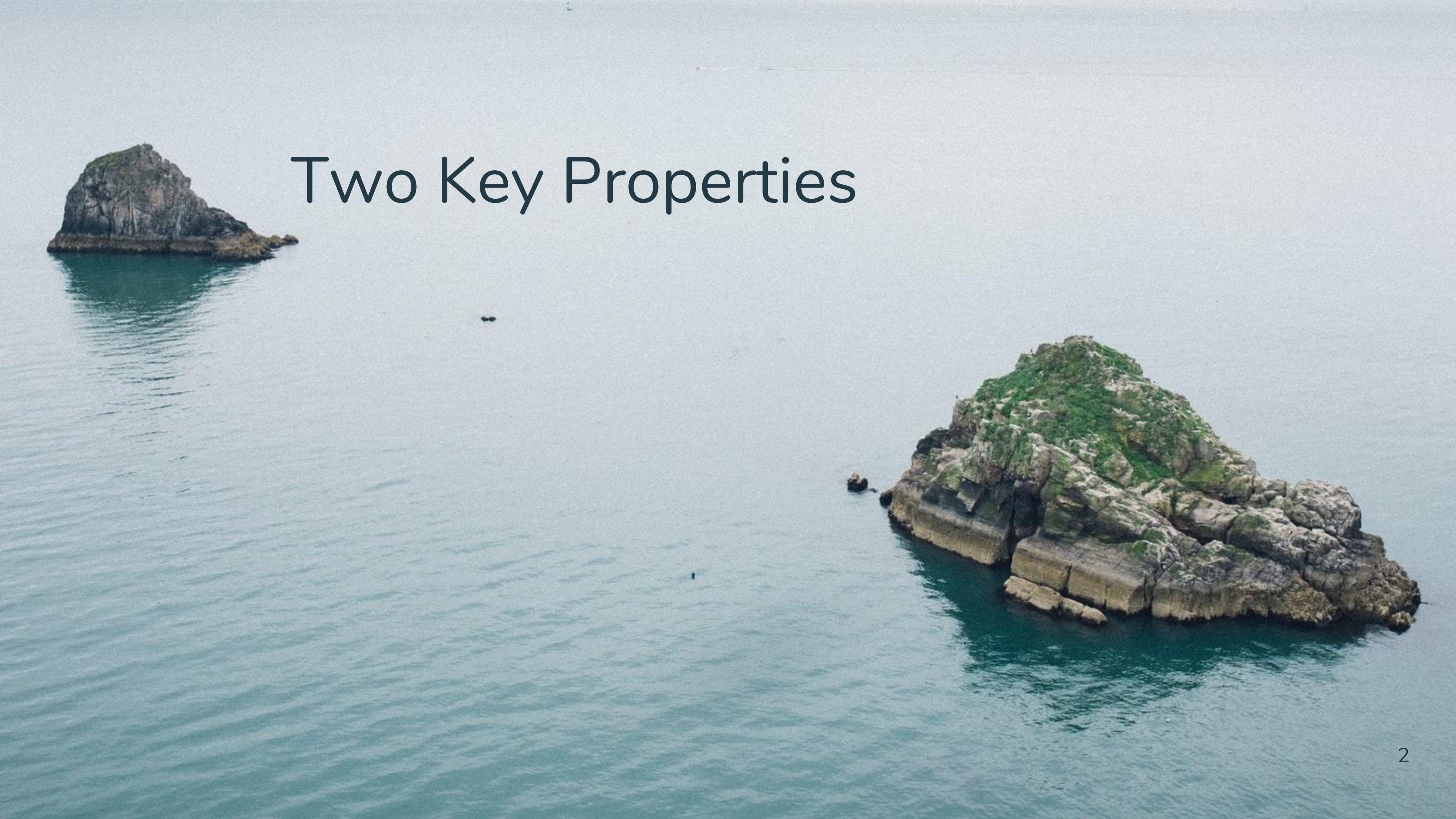
Cochis

Stable and Coherent Implicits

T. Schrijvers, B. Oliveira, P. Wadler and K. Marntirosian

Two Key Properties



Implicits

Coherence

If an expression evaluates to two results, then these are observationally equivalent.

Stability

Type Safety

Example:

Coherence ➜ Global uniqueness

Implicits

Coherence

If an expression evaluates to two results, then these are observationally equivalent.

Stability

Type Safety

Example:

```
{-# LANGUAGE FlexibleInstances,
             OverlappingInstances
#-}

class Trans a       where
  transform :: a -> a

instance Trans a    where
  transform x = x

instance Trans Int where
  transform x = x + 1
```

Implicits

Coherence

If an expression evaluates to two results, then these are observationally equivalent.

Stability

Type Safety

Example:

```haskell
{-# LANGUAGE FlexibleInstances,
             MultiParamTypeClasses,
             OverlappingInstances
#-}

class HasBool a b        where
  chooseBool :: a -> b -> Bool

instance HasBool Bool a where
  chooseBool a b = a

instance HasBool a Bool where
  chooseBool a b = b


       chooseBool True False ?
```

Implicits

Coherence
Stability
Instantiation of type variables should not affect resolution.
Type Safety

Example:

```
{-# LANGUAGE FlexibleInstances,
             IncoherentInstances
#-}

class Trans a      where
  trans :: a -> a

instance Trans a   where
  trans x = x

instance Trans Int where
  trans x = x + 1

bad :: a -> a
bad x = trans x
```

Implicits

Coherence
Stability
Type Safety
No well-typed program gets stuck.

```
trait A {
  implicit def id[a] : a => a
    = x ⇒ x
  def trans[a](x:a)
      (implicit f : a => a)
    = f(x)
}

object B extends A {
  implicit def succ : Int => Int
    = x => x + 1
  def universal[a](x:a) : a
    = trans[a](x)

  val v1 = universal[Int](3)
  // val v2 = trans[Int](3)
}
```

Cochis

Type Safe

Coherent

Stable

Syntax

Extension of
predicative System F

$$
\begin{array}{llll}
\text{Type Environments} & \Gamma & ::= & \epsilon \mid \Gamma, x : \rho \mid \Gamma, \alpha \mid \Gamma, ?\rho \rightsquigarrow x \\
\text{Types} & \rho & ::= & \alpha \mid \rho_1 \to \rho_2 \mid \forall \alpha . \rho \mid \rho_1 \Rightarrow \rho_2 \\
\text{Monotypes} & \sigma & ::= & \alpha \mid \sigma \to \sigma \\
\text{Expressions} & e & ::= & x \mid \lambda(x : \rho).e \mid e_1 \, e_2 \mid \Lambda \alpha . e \mid e \, \sigma \\
 & & \mid & ?\rho \mid \lambda_? \rho . e \mid e_1 \ \mathbf{with} \ e_2
\end{array}
$$

Overview

- User-defined rules
- Queries based on type

$$
\begin{array}{rcl}
\Gamma & ::= & \epsilon \mid \Gamma, x : \rho \mid \Gamma, \alpha \mid \Gamma, ?\rho \rightsquigarrow x \\
\rho & ::= & \alpha \mid \rho_1 \to \rho_2 \mid \forall \alpha . \rho \mid \rho_1 \Rightarrow \rho_2 \\
\sigma & ::= & \alpha \mid \sigma \to \sigma \\
e & ::= & x \mid \lambda(x : \rho).e \mid e_1\, e_2 \mid \Lambda \alpha . e \mid e\, \sigma \\
& \mid & ?\rho \mid \lambda_? \rho . e \mid e_1 \ \mathbf{with} \ e_2
\end{array}
$$

$$
\begin{array}{l}
\lambda_? Int \, . ?Int + 1 \\
\quad : Int \Rightarrow Int
\end{array}
$$

Overview

- User-defined rules
- Queries based on type
- Extension of implicit environment with rule application

$$
\begin{array}{lcl}
\Gamma & ::= & \epsilon \mid \Gamma, x : \rho \mid \Gamma, \alpha \mid \Gamma, ?\rho \rightsquigarrow x \\
\rho & ::= & \alpha \mid \rho_1 \rightarrow \rho_2 \mid \forall \alpha . \rho \mid \rho_1 \Rightarrow \rho_2 \\
\sigma & ::= & \alpha \mid \sigma \rightarrow \sigma \\
e & ::= & x \mid \lambda (x : \rho) . e \mid e_1 \, e_2 \mid \Lambda \alpha . e \mid e \, \sigma \\
 & \mid & ?\rho \mid \lambda_? \rho . e \mid e_1 \ \mathbf{with} \ e_2
\end{array}
$$

$$
\lambda_? \, Int \, . ?Int + 1 \\
\mathbf{with} \ 1
$$

$$
\begin{array}{lcl}
\Gamma & ::= & \epsilon \mid \Gamma, x : \rho \mid \Gamma, \alpha \mid \color{red}{\Gamma, ?\rho \rightsquigarrow x} \\
\rho & ::= & \alpha \mid \rho_1 \to \rho_2 \mid \forall \alpha . \rho \mid \color{red}{\rho_1 \Rightarrow \rho_2} \\
\sigma & ::= & \alpha \mid \sigma \to \sigma \\
e & ::= & x \mid \lambda (x : \rho) . e \mid e_1 \, e_2 \mid \Lambda \alpha . e \mid e \, \sigma \\
 & \mid & \color{red}{?\rho \mid \lambda_? \rho . e \mid e_1 \ \mathbf{with} \ e_2}
\end{array}
$$

Overview

- User-defined rules
- Queries based on type
- Extension of implicit environment with rule application

$$\mathbf{implicit}\ 1\ \mathbf{in}\ (?Int + 1)$$

$$
\begin{array}{rcl}
\Gamma & ::= & \epsilon \mid \Gamma, x : \rho \mid \Gamma, \alpha \mid \Gamma, ?\rho \rightsquigarrow x \\
\rho & ::= & \alpha \mid \rho_1 \to \rho_2 \mid \forall \alpha . \rho \mid \rho_1 \Rightarrow \rho_2 \\
\sigma & ::= & \alpha \mid \sigma \to \sigma \\
e & ::= & x \mid \lambda(x : \rho).e \mid e_1\, e_2 \mid \Lambda\alpha.e \mid e\,\sigma \\
& \mid & ?\rho \mid \lambda_?\rho.e \mid e_1 \textbf{ with } e_2
\end{array}
$$

Overview

- User-defined rules
- Queries based on type
- Extension of implicit environment with rule application
- Higher-order polymorphic rules
- Recursive resolution

$$
\begin{array}{l}
\textbf{implicit } 3 \textbf{ in implicit } True \textbf{ in} \\
\textbf{implicit } \Lambda\,\alpha\,.\,\lambda_?\,\alpha\,.\,(?\alpha,\,?\alpha) \textbf{ in} \\
?(Int \times Int) \times ?(Bool \times Bool)
\end{array}
$$

Overview

$$
\begin{array}{lcl}
\Gamma & ::= & \epsilon \mid \Gamma, x : \rho \mid \Gamma, \alpha \mid \Gamma, ?\rho \rightsquigarrow x \\
\rho & ::= & \alpha \mid \rho_1 \to \rho_2 \mid \forall \alpha . \rho \mid \rho_1 \Rightarrow \rho_2 \\
\sigma & ::= & \alpha \mid \sigma \to \sigma \\
e & ::= & x \mid \lambda(x : \rho).e \mid e_1\, e_2 \mid \Lambda \alpha . e \mid e\, \sigma \\
 & \mid & ?\rho \mid \lambda_? \rho . e \mid e_1 \ \mathbf{with}\ e_2
\end{array}
$$

- User-defined rules
- Polymorphic queries based on type
- Extension of implicit environment with rule application
- Higher-order polymorphic rules
- Recursive resolution

$$
\begin{array}{l}
\mathbf{implicit}\ 3\ \mathbf{in}\ \mathbf{implicit}\ True\ \mathbf{in} \\
\mathbf{implicit}\ \Lambda \alpha . \lambda_? \alpha . (?\alpha, ?\alpha)\ \mathbf{in} \\
?(\forall \beta . \beta \Rightarrow (\beta \times \beta))
\end{array}
$$

Overview

$$
\begin{array}{rcl}
\Gamma & ::= & \epsilon \mid \Gamma, x : \rho \mid \Gamma, \alpha \mid \Gamma, ?\rho \rightsquigarrow x \\
\rho & ::= & \alpha \mid \rho_1 \to \rho_2 \mid \forall \alpha . \rho \mid \rho_1 \Rightarrow \rho_2 \\
\sigma & ::= & \alpha \mid \sigma \to \sigma \\
e & ::= & x \mid \lambda(x : \rho).e \mid e_1\, e_2 \mid \Lambda \alpha . e \mid e\, \sigma \\
& \mid & ?\rho \mid \lambda_? \rho . e \mid e_1 \textbf{ with } e_2
\end{array}
$$

- User-defined rules
- Polymorphic queries based on type
- Extension of implicit environment with rule application
- Higher-order polymorphic rules
- Recursive resolution
- Lexical and local scoping

$\textbf{implicit } 1 \textbf{ in implicit } True \textbf{ in}$
$\textbf{implicit } (\lambda_? \, Bool \, . \, \textbf{if } ?Bool \textbf{ then } 0 \textbf{ else } 2) \textbf{ in}$
$?Int$

The resolution mechanism

Resolution

$\textsc{Cochis}$

Internal representation: System F

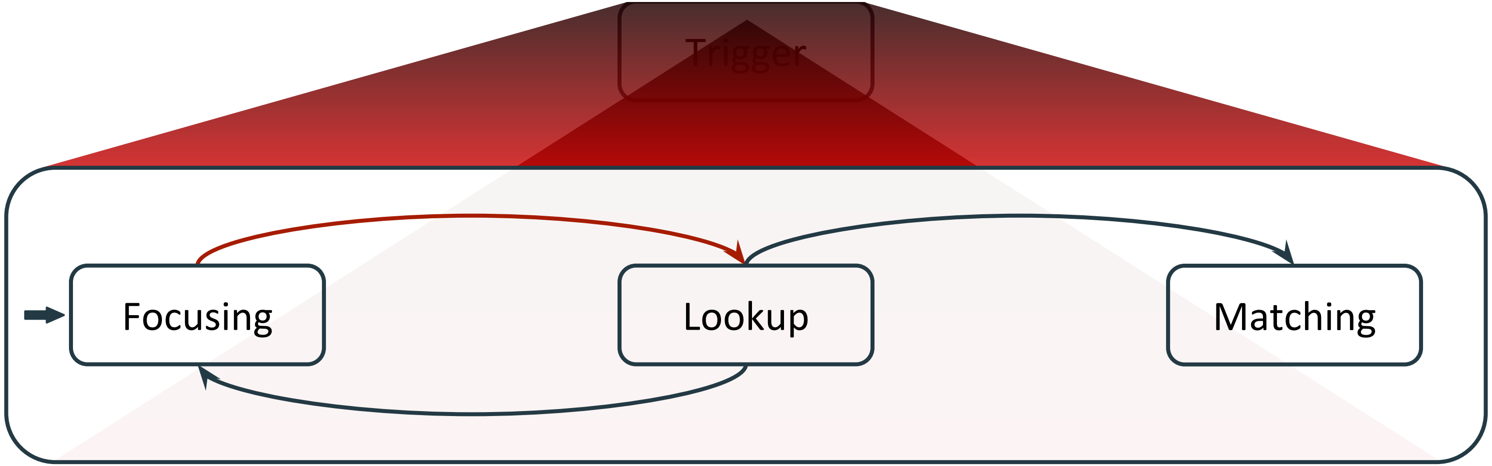
$$\Gamma \vdash e : \rho \rightsquigarrow E$$

Resolution:

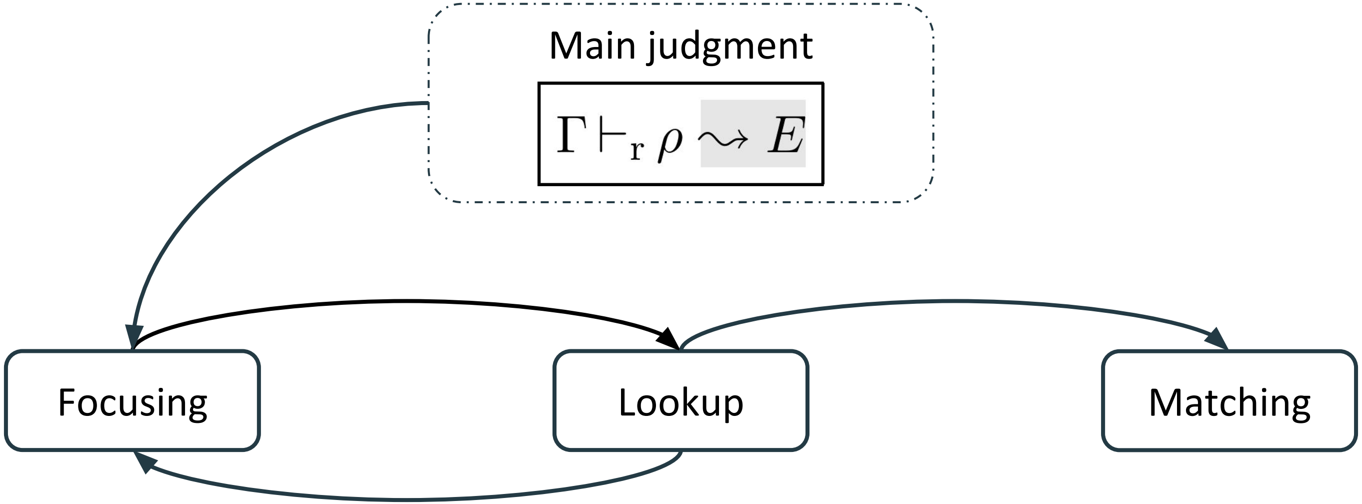
- Local scoping of rules
- Rules and queries are first-class entities
- Higher-order rules

$$(\textsc{Ty-Query})\ \frac{\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E \qquad \vdash_{\mathrm{unamb}} \rho}{\Gamma \vdash ?\rho : \rho \rightsquigarrow E}$$

Resolution



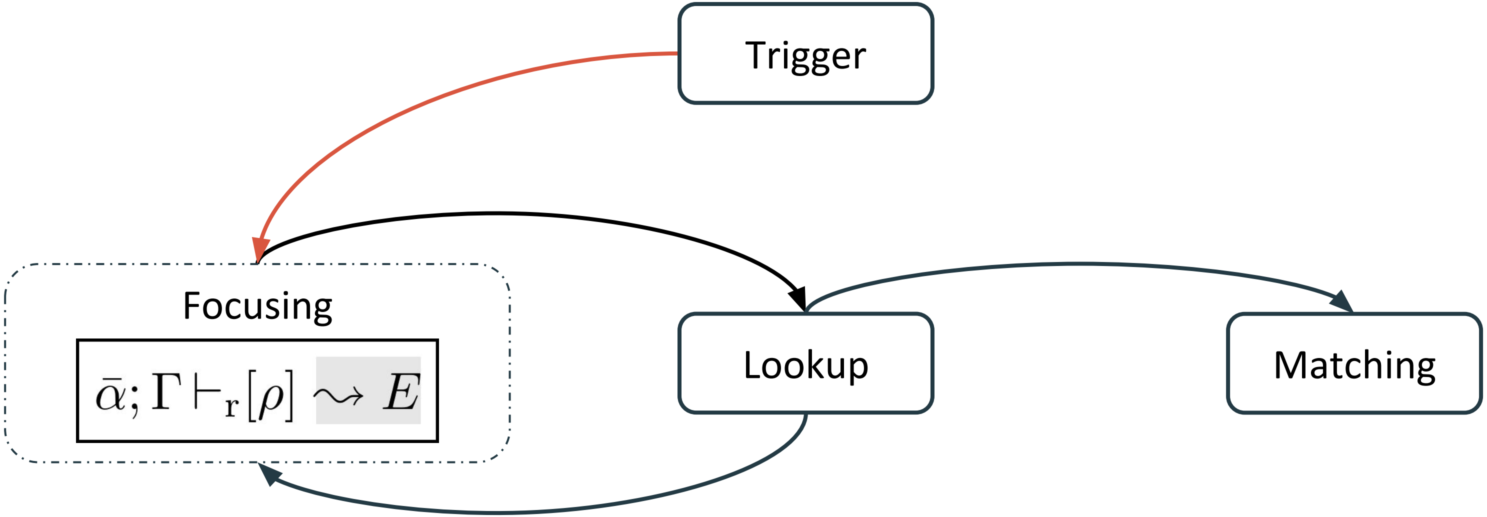
Resolution



Main judgment
$\Gamma \vdash_{\mathrm{r}} \rho \rightsquigarrow E$
Focusing
Lookup
Matching

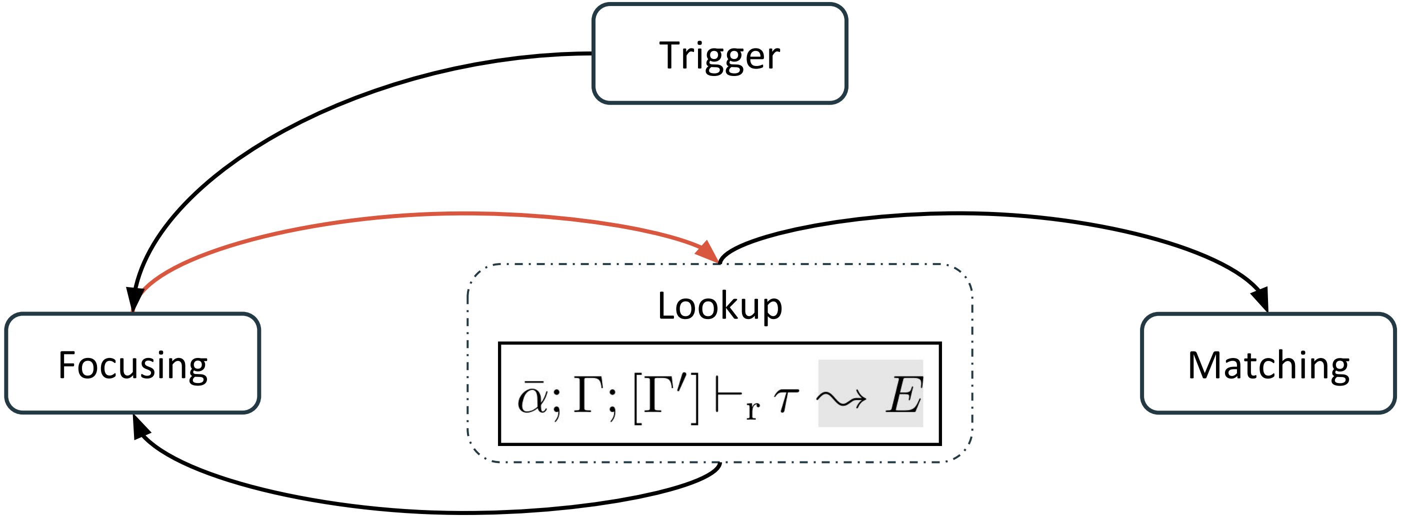
Resolution

$$
\begin{array}{rrl}
\text{Types} & \rho & ::= \forall \alpha . \rho \mid \rho_1 \Rightarrow \rho_2 \mid \tau \\
\text{Simple Types} & \tau & ::= \alpha \mid \rho_1 \to \rho_2
\end{array}
$$

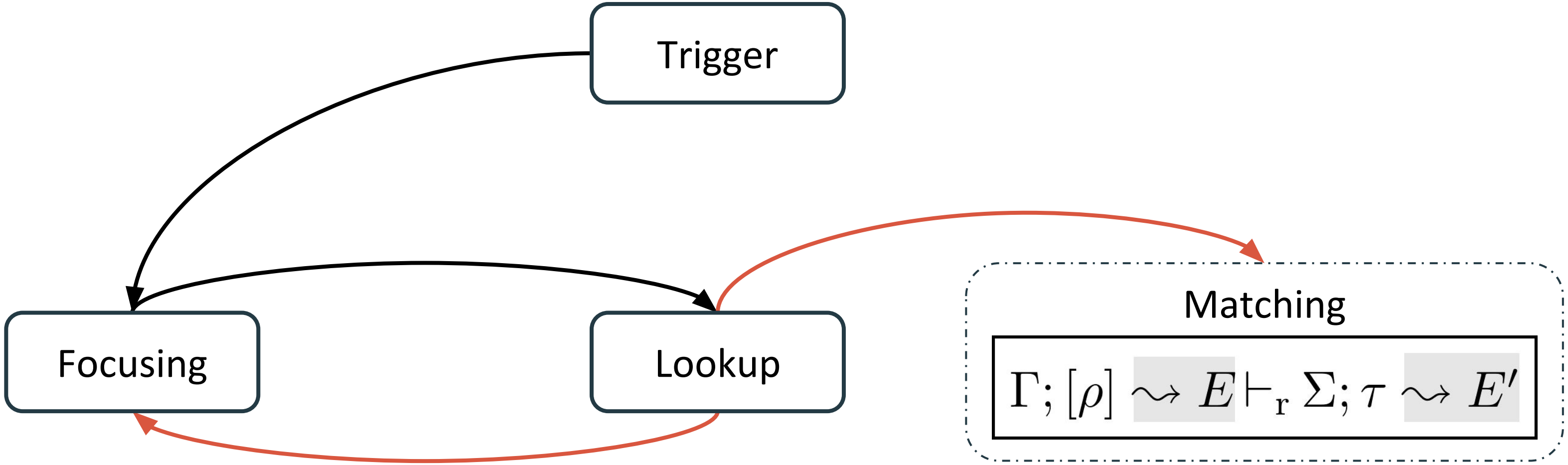


Resolution

$$\Gamma \ ::= \ \epsilon \mid \Gamma, x : \rho \mid \Gamma, \alpha \mid \Gamma, ?\rho \rightsquigarrow x$$

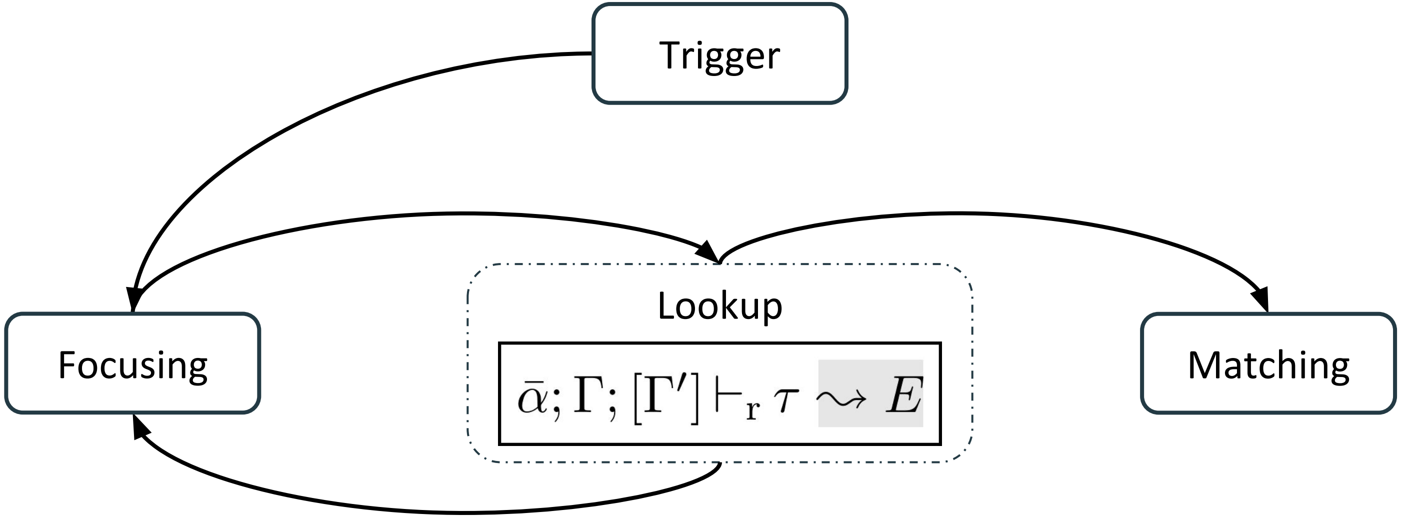


Resolution


Trigger
Focusing
Lookup
Matching
$\Gamma; [\rho] \rightsquigarrow E \vdash_{\mathrm{r}} \Sigma; \tau \rightsquigarrow E'$

Resolution

$\text{stable}(\bar{\alpha}; \Gamma; \rho \rightsquigarrow x; \tau)$



Under consideration for publication in J. Functional Programming 1

COCHIS: *Stable and Coherent Implicits*

TOM SCHRIJVERS
KU Leuven, Belgium
BRUNO C.D.S. OLIVEIRA
The University of Hong Kong
PHILIP WADLER
University of Edinburgh, UK
KOAR MARNTIROSIAN
KU Leuven, Belgium

Abstract

Implicit Progamming (IP) mechanisms infer values by type-directed resolution, making programs more compact and easier to read. Examples of IP mechanisms include Haskell's type classes, Scala's implicits, Agda's instance arguments, Coq's type classes, and Rust's traits. The design of IP mechanisms has led to heated debate: proponents of one school argue for the desirability of strong reasoning properties; while proponents of another school argue for the power and flexibility of local scoping or overlapping instances. The current state of affairs seems to indicate that the two goals are at odds with one another and cannot easily be reconciled.

This paper presents COCHIS, the Calculus Of CoHerent ImplicitS, an improved variant of the implicit calculus that offers flexibility while preserving two key properties: *coherence* and *stability of substitutions*. COCHIS supports polymorphism, local scoping, overlapping instances, first-class instances, and higher-order rules, while remaining type safe, coherent and stable under substitution.

We introduce a logical formulation of how to resolve implicits, which is simple but ambiguous and incoherent, and a second formulation, which is less simple but unambiguous, coherent and stable. Every resolution of the second formulation is also a resolution of the first, but not conversely. Parts of the second formulation bear a close resemblance to a standard technique for proof search called focussing.

1 Introduction

Programming language design is usually guided by two, often conflicting, goals: *flexibility* and *ease of reasoning*. Many programming languages aim at providing powerful, flexible language constructs that allow programmers to achieve reuse, and develop programs rapidly and concisely. Other programming languages aim at easy reasoning about programs, as well as at avoiding programming pitfalls. Often the two goals are at odds with each other, since highly flexible programming mechanisms make reasoning harder. Arguably the art of programming language design is to reconcile both goals.

A concrete case where this issue manifests itself is in the design of *Implicit Programming* (IP) mechanisms. Implicit programming denotes a class of language mechanisms, which infer values by using type information. Examples of IP mechanisms include Haskell's type

- Algorithmic resolution
  [Sound and complete]
- [Proof] Deterministic resolution
  ➜ Coherence
- [Proof] Stability of resolution

https://bitbucket.org/KlaraMar/cochiscoq/