

Higher-Ranked Exception Types

WORK-IN-PROGRESS

Ruud Koot

Department of Information and Computing Sciences
Utrecht University

June 12, 2014

Motivation

- ▶ Types should not lie; we would like to have *checked exceptions* in Haskell:

$$\text{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \text{ throws } e$$

- ▶ What should be the correct value of e ?

Motivation

Assigning accurate exception types is complicated by:

Higher-order functions Exceptions raised by higher-order functions depend on the exceptions raised by functional arguments.

$$\text{map} :: (\alpha \rightarrow \beta \text{ throws } e_1) \rightarrow [\alpha] \rightarrow [\beta] \text{ throws } (e_1 \cup e_2)$$

Non-strict evaluation Exceptions are embedded inside values.

$$\begin{aligned} \text{map} :: & (\alpha \text{ throws } e_1 \rightarrow \beta) \text{ throws } e_2 \\ & \rightarrow [\alpha \text{ throws } e_3] \text{ throws } e_4 \rightarrow [\beta \text{ throws } e_5] \text{ throws } e_6 \end{aligned}$$

Motivation

- ▶ Instead of τ **throws** e , write τ^e for a type τ that can evaluate to \perp_χ for some $\chi \in e$.
- ▶ The fully annotated exception type for *map* would be:

$$\begin{aligned} \text{map} &:: (\alpha^{e_1} \rightarrow \beta^{(e_1 \cup e_2)})^{e_3} \rightarrow [\alpha^{e_1}]^{e_4} \rightarrow [\beta^{(e_1 \cup e_2 \cup e_3)}]^{e_4} \\ \text{map} &= \lambda f. \lambda x s. \mathbf{case} \ x s \ \mathbf{of} \\ &\quad [] \quad \quad \mapsto [] \\ &\quad (y : y s) \mapsto f \ y : \text{map} \ f \ y s \end{aligned}$$

Motivation

- ▶ Instead of τ **throws** e , write τ^e for a type τ that can evaluate to \perp_χ for some $\chi \in e$.
- ▶ The fully annotated exception type for *map* would be:

$$\begin{aligned} \text{map} &:: (\alpha^{e_1} \rightarrow \beta^{(e_1 \cup e_2)})^{e_3} \rightarrow [\alpha^{e_1}]^{e_4} \rightarrow [\beta^{(e_1 \cup e_2 \cup e_3)}]^{e_4} \\ \text{map} &= \lambda f. \lambda xs. \mathbf{case} \ xs \ \mathbf{of} \\ &\quad [] \quad \quad \mapsto [] \\ &\quad (y : ys) \mapsto f \ y : \text{map} \ f \ ys \end{aligned}$$

- ▶ If you want to be pedantic:

$$\begin{aligned} \text{map} &:: \forall \alpha \ \beta \ e_1 \ e_2 \ e_3 \ e_4. \\ &\quad ((\alpha^{e_1} \rightarrow \beta^{(e_1 \cup e_2)})^{e_3} \rightarrow ([\alpha^{e_1}]^{e_4} \rightarrow [\beta^{(e_1 \cup e_2 \cup e_3)}]^{e_4})^\emptyset)^\emptyset \end{aligned}$$

Motivation

- ▶ Instead of τ **throws** e , write τ^e for a type τ that can evaluate to \perp_χ for some $\chi \in e$.
- ▶ The fully annotated exception type for *map* would be:

$$\begin{aligned} \text{map} &:: (\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_1 \cup e_2)}) \rightarrow [\alpha^{e_1}]^{e_4} \rightarrow [\beta^{(e_1 \cup e_2 \cup e_3)}]^{e_4} \\ \text{map} &= \lambda f. \lambda xs. \mathbf{case} \ xs \ \mathbf{of} \\ &\quad [] \quad \quad \mapsto [] \\ &\quad (y : ys) \mapsto f \ y : \text{map} \ f \ ys \end{aligned}$$

- ▶ If you want to be pedantic:

$$\begin{aligned} \text{map} &:: \forall \alpha \ \beta \ e_1 \ e_2 \ e_3 \ e_4. \\ &(\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_1 \cup e_2)}) \xrightarrow{\emptyset} [\alpha^{e_1}]^{e_4} \xrightarrow{\emptyset} [\beta^{(e_1 \cup e_2 \cup e_3)}]^{e_4} \end{aligned}$$

Motivation

- ▶ The exception type

$$\text{map} :: (\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_1 \cup e_2)}) \rightarrow [\alpha^{e_1}]^{e_4} \rightarrow [\beta^{(e_1 \cup e_2 \cup e_3)}]^{e_4}$$

is not as accurate as we would like.

- ▶ Consider the instantiations:

$$\text{map } id \quad :: [\alpha^{e_1}]^{e_4} \rightarrow [\alpha^{e_1}]^{e_4}$$

$$\text{map } (\text{const } \perp_{\mathbf{E}}) :: [\alpha^{e_1}]^{e_4} \rightarrow [\beta^{(e_1 \cup \{\mathbf{E}\})}]^{e_4}$$

- ▶ A more appropriate type for $\text{map } (\text{const } \perp_{\mathbf{E}})$ would be

$$\text{map } (\text{const } \perp_{\mathbf{E}}) :: [\alpha^{e_1}]^{e_4} \rightarrow [\beta^{\{\mathbf{E}\}}]^{e_4}$$

as it cannot propagate exceptional elements inside the input list to the output list.

Motivation

- ▶ The problem is that we have already committed the first argument of *map* to be of type

$$\alpha^{e_1} \xrightarrow{e_3} \beta^{(e_1 \cup e_2)},$$

i.e. it propagates exceptional values from the its input to the output while possibly adding additional exceptional values.

- ▶ This is a worst-case scenario: it is sound but inaccurate.

Motivation

- ▶ The solution is to move from Hindley–Milner to F_ω , introducing *higher-ranked types* and *type operators*.
 - ▶ Recall that System F_ω replicates the *simply typed λ -calculus* on the type level.
- ▶ This gives us the expressiveness to state the exception type of *map* as:

$$\begin{aligned} & \forall e_2 \, e_3. (\forall e_1. \alpha^{e_1} \xrightarrow{e_3} \beta^{(e_2 \, e_1)}) \\ & \rightarrow (\forall e_4 \, e_5. [\alpha^{e_4}]^{e_5} \rightarrow [\beta^{(e_2 \, e_4 \cup e_3)}]^{e_5}) \end{aligned}$$

- ▶ Note that e_2 is an *exception operator* of kind $\text{EXN} \rightarrow \text{EXN}$.

Motivation

- ▶ Given the following functions:

$$\begin{aligned} \text{map} \quad &:: \forall e_2 \ e_3. (\forall e_1. \alpha^{e_1} \xrightarrow{e_3} \beta^{(e_2 \ e_1)}) \\ &\rightarrow (\forall e_4 \ e_5. [\alpha^{e_4}]^{e_5} \rightarrow [\beta^{(e_2 \ e_4 \cup e_3)}]^{e_5}) \end{aligned}$$

$$\text{id} \quad :: \forall e. \alpha^e \xrightarrow{\emptyset} \alpha^e$$

$$\text{const } \perp_{\mathbf{E}} :: \forall e. \alpha^e \xrightarrow{\emptyset} \beta^{\{\mathbf{E}\}}$$

- ▶ Applying *id* or *const* $\perp_{\mathbf{E}}$ to *map* will give rise the the instantiations $e_2 \mapsto \lambda e. e$, respectively $e_2 \mapsto \lambda e. \{\mathbf{E}\}$.
- ▶ This gives us the exception types:

$$\text{map id} \quad :: \forall e_4 \ e_5. [\alpha^{e_4}]^{e_5} \rightarrow [\alpha^{e_4}]^{e_5}$$

$$\text{map } (\text{const } \perp_{\mathbf{E}}) :: \forall e_4 \ e_5. [\alpha^{e_4}]^{e_5} \rightarrow [\beta^{\{\mathbf{E}\}}]^{e_5}$$

as desired.

Intermezzo: Simply-typed λ -calculus

Types

$$\begin{array}{lll} \tau \in \mathbf{Ty} & ::= & B \quad \text{(base type)} \\ & | & \tau_1 \rightarrow \tau_2 \quad \text{(function type)} \end{array}$$

Terms

$$\begin{array}{lll} t \in \mathbf{Tm} & ::= & x, y, \dots \quad \text{(variable)} \\ & | & \lambda x : \tau. t \quad \text{(abstraction)} \\ & | & t_1 t_2 \quad \text{(application)} \end{array}$$

Values

$$\begin{array}{lll} v \in \mathbf{Val} & ::= & x, y, \dots \quad \text{(free variable)} \\ & | & \lambda x : \tau. v \quad \text{(abstraction value)} \end{array}$$

Intermezzo: Simply-typed λ -calculus

Typing

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} [\text{T-VAR}] \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1. t : \tau_1 \rightarrow \tau_2} [\text{T-ABS}]$$
$$\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 \ t_2 : \tau_2} [\text{T-APP}]$$

Intermezzo: Simply-typed λ -calculus

Evaluation We perform *full β -reduction*, i.e. we also evaluate under binders.

$$\frac{t \longrightarrow t'}{\lambda x : \tau. t \longrightarrow \lambda x : \tau. t'} \text{ [E-ABS]}$$

$$\frac{t_1 \longrightarrow t'_1}{t_1 \ t_2 \longrightarrow t'_1 \ t_2} \text{ [E-APP}_1\text{]} \qquad \frac{t_2 \longrightarrow t'_2}{t_1 \ t_2 \longrightarrow t_1 \ t'_2} \text{ [E-APP}_2\text{]}$$

$$\frac{}{(\lambda x : \tau. t_1) \ t_2 \longrightarrow [t_2/x] \ t_1} \text{ [E-BETA]}$$

Intermezzo: Simply-typed λ -calculus

Theorem (Progress)

A term t is either a value v , or we can reduce $t \longrightarrow t'$.

Theorem (Preservation)

If $\Gamma \vdash t : \tau$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : \tau$.

Theorem (Confluence)

If $t \longrightarrow t_1$ and $t \longrightarrow t_2$, then exists a term t' such that $t_1 \longrightarrow^ t'$ and $t_2 \longrightarrow^* t'$.*

Theorem (Normalization)

For any term t we have that $t \longrightarrow^ v$ (in a finite number of steps).*

Corollary (Uniqueness of normal forms)

If $t \longrightarrow^ v_1$ and $t \longrightarrow^* v_2$, then $v_1 \equiv v_2$.*

Intermezzo: The lambda “cube”

- ▶ The simply-typed λ -calculus can be extended with *parametric polymorphism*, or *type operators*, or both.

$$\begin{array}{ccc} F & \longrightarrow & F_\omega \\ \uparrow & & \uparrow \\ \lambda & \longrightarrow & \lambda_\omega \end{array}$$

- | | |
|--|--|
| ▶ $id : B \rightarrow B$
$id = \lambda x : B. x$ | ▶ $id : \forall \alpha :: *. \alpha \rightarrow \alpha$
$id = \Lambda \alpha : *. \lambda x : \alpha. x$ |
| ▶ $Id :: * \Rightarrow *$
$Id = \lambda \alpha :: *. \alpha$
$id : B \rightarrow Id\ B$
$id = \lambda x : B. x$ | ▶ $Id :: * \Rightarrow *$
$Id = \lambda \alpha :: *. \alpha$
$id : \forall \alpha :: *. \alpha \rightarrow Id\ \alpha$
$id = \Lambda \alpha : *. \lambda x : \alpha. x$ |

- ▶ Omitted: the axis for dependent types.

Intermezzo: System F_ω

Types

$\tau \in \mathbf{Ty}$	$::=$	α	(type variable)
	$ $	$\tau_1 \rightarrow \tau_2$	(function type)
	$ $	$\forall \alpha :: \kappa. \tau$	(universal type)
	$ $	$\lambda \alpha :: \kappa. \tau$	(operator abstraction)
	$ $	$\tau_1 \tau_2$	(operator application)

Kinds

$\kappa \in \mathbf{Kind}$	$::=$	$*$	(base kind)
	$ $	$\kappa_1 \Rightarrow \kappa_2$	(operator kind)

Intermezzo: System F_ω

Terms

$t \in \mathbf{Tm}$	$::=$	x, y, \dots	(variable)
		$\lambda x : \tau. t$	(abstraction)
		$t_1 \ t_2$	(application)
		$\Lambda \alpha :: \kappa. t$	(type abstraction)
		$t \ \langle \tau \rangle$	(type application)

Values

$v \in \mathbf{Val}$	$::=$	x, y, \dots	(free variable)
		$\lambda x : \tau. v$	(abstraction value)

Technicalities

- ▶ Due to their syntactic weight, higher-ranked exception type only seem useful if they can be inferred automatically.
- ▶ Unlike for HM type inference is undecidable in F_ω .
- ▶ However, the exception types are annotations piggybacking on top of an underlying type system.
- ▶ Holdermans and Hage [HH10] showed type inference is decidable for a higher-ranked annotated type system with type operators performing control-flow analysis.

Technicalities

1. Perform Hindley–Milner type inference to reconstruct the underlying types.
2. Run a second inference pass to reconstruct the exception types.
 - 2.1 Collect a set of subtyping constraints.
 - 2.2 In case of a λ -abstraction $\lambda x : \tau.e$, we *complete* the type τ to an exception type.
 - 2.3 In case of an application we *match* the types of the formal and actual parameter.
3. Solve the generated subtyping constraints.

Technicalities: Completion

- The completion procedure adds as many quantifiers and type operators as possible to a type.

$$\frac{}{\overline{e_i :: \kappa_i} \vdash \mathbf{bool} : \widehat{\mathbf{bool}} \ \& \ e \ \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow_{\text{EXN}}} \text{[C-Bool]}$$

$$\frac{\overline{e_i :: \kappa_i} \vdash \tau : \widehat{\tau} \ \& \ \chi \triangleright \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash [\tau] : [\widehat{\tau} \ \mathbf{throws} \ \chi] \ \& \ e \ \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow_{\text{EXN}}, \overline{e_j :: \kappa_j}} \text{[C-List]}$$

$$\frac{\vdash \tau_1 : \widehat{\tau_1} \ \& \ \chi_1 \triangleright \overline{e_j :: \kappa_j} \quad \overline{e_i :: \kappa_i}, \overline{e_j :: \kappa_j} \vdash \tau_2 : \widehat{\tau_2} \ \& \ \chi_2 \triangleright \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash \tau_1 \rightarrow \tau_2 : \forall \overline{e_j :: \kappa_j}. (\widehat{\tau_1} \ \mathbf{throws} \ \chi_1 \rightarrow \widehat{\tau_2} \ \mathbf{throws} \ \chi_2) \ \& \ e \ \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow_{\text{EXN}}, \overline{e_k :: \kappa_k}} \text{[C-ARR]}$$

Figure : Type completion ($\Gamma \vdash \tau : \widehat{\tau} \ \& \ \chi \triangleright \Gamma'$)

Technicalities: Completion

- ▶ $\cdot \vdash \mathbf{bool} : \widehat{\mathbf{bool}} \ \& \ e_1 \triangleright e_1 :: \mathbf{EXN}$

Technicalities: Completion

- ▶ $\cdot \vdash \mathbf{bool} : \widehat{\mathbf{bool}} \ \& \ e_1 \triangleright e_1 :: \mathbf{EXN}$
- ▶ $e_1 :: \mathbf{EXN} \vdash \mathbf{bool} : \widehat{\mathbf{bool}} \ \& \ e_2 \ e_1 \triangleright e_2 :: \mathbf{EXN} \Rightarrow \mathbf{EXN}$

Technicalities: Completion

- ▶ **bool** \rightarrow **bool**
- ▶ $\forall e_1 :: \text{EXN}. \widehat{\text{bool}}^{e_1} \rightarrow \widehat{\text{bool}}^{(e_2 \ e_1)} \ \& \ e_3$
- ▶ $e_2 :: \text{EXN} \Rightarrow \text{EXN}, e_3 :: \text{EXN}$

Technicalities: Completion

► **bool** → **bool** → **bool**

►

$$\forall e_1 :: \text{EXN}. \widehat{\text{bool}}^{e_1} \rightarrow$$
$$(\forall e_4 :: \text{EXN}. \widehat{\text{bool}}^{e_4} \xrightarrow{e_2 \ e_1} \widehat{\text{bool}}^{(e_5 \ e_1 \ e_4)}) \ \& \ e_3$$

► $e_2 :: \text{EXN} \Rightarrow \text{EXN}, e_3 :: \text{EXN}, e_5 :: \text{EXN} \Rightarrow \text{EXN} \Rightarrow \text{EXN}$

Technicalities: Completion

► $(\mathbf{bool} \rightarrow \mathbf{bool}) \rightarrow \mathbf{bool}$

►

$\forall e_2 :: \text{EXN} \Rightarrow \text{EXN}. \forall e_3 :: \text{EXN}.$

$\left(\forall e_1 :: \text{EXN}. \widehat{\text{bool}}^{e_1} \xrightarrow{e_3} \widehat{\text{bool}}^{(e_2 \ e_1)} \right) \rightarrow \widehat{\text{bool}}^{(e_4 \ e_2 \ e_3)} \ \& \ e_5$

► $e_4 :: (\text{EXN} \Rightarrow \text{EXN}) \Rightarrow \text{EXN} \Rightarrow \text{EXN}, e_5 :: \text{EXN}$

Technicalities: Constraint solving

- ▶ Solving subtyping constraints can be done using a fixed-point iteration.
- ▶ To decide we have reached a fixed point we need an equality on types.
- ▶ But types are now a simply typed λ -calculus.

Technicalities: λ^U

Types

$\tau \in \mathbf{Ty}$	$::= \mathcal{P}$	(base type)
	$ \tau_1 \rightarrow \tau_2$	(function type)

Terms

$t \in \mathbf{Tm}$	$::= x, y, \dots$	(variable)
	$ \lambda x : \tau. t$	(abstraction)
	$ t_1 t_2$	(application)
	$ \emptyset$	(empty)
	$ \{c\}$	(singleton)
	$ t_1 \cup t_2$	(union)

Values Values v are terms of the form

$$\lambda x_1 : \tau_1. \dots \lambda x_i : \tau_i. \{c_1\} \cup (\dots \cup (\{c_j\} \cup (x_1 v_{11} \dots v_{1m} \cup (\dots \cup x_k v_{k1} \dots v_{kn}))))))$$

Technicalities: λ^\cup

$$(\lambda x : \tau. t_1) \ t_2 \longrightarrow [t_2/x] t_1 \quad (\beta\text{-reduction})$$

$$(t_1 \cup t_2) \ t_3 \longrightarrow t_1 \ t_3 \cup t_2 \ t_3$$

$$(\lambda x : \tau. t_1) \cup (\lambda x : \tau. t_2) \longrightarrow \lambda x : \tau. (t_1 \cup t_2) \quad (\text{congruences})$$

$$x \ t_1 \cdots t_n \cup x' \ t'_1 \cdots t'_n \longrightarrow x \ (t_1 \cup t'_1) \cdots (t_n \cup t'_n)$$

$$(t_1 \cup t_2) \cup t_3 \longrightarrow t_1 \cup (t_2 \cup t_3) \quad (\text{associativity})$$

$$\emptyset \cup t \longrightarrow t$$

$$t \cup \emptyset \longrightarrow t$$

$$x \cup x \longrightarrow x$$

$$x \cup (x \cup t) \longrightarrow x \cup t$$

$$\{c\} \cup \{c\} \longrightarrow \{c\}$$

$$\{c\} \cup (\{c\} \cup t) \longrightarrow \{c\} \cup t$$

$$x \ t_1 \cdots t_n \cup \{c\} \longrightarrow \{c\} \cup x \ t_1 \cdots t_n \quad (1)$$

$$x \ t_1 \cdots t_n \cup (\{c\} \cup t) \longrightarrow \{c\} \cup (x \ t_1 \cdots t_n \cup t) \quad (2)$$

$$x \ t_1 \cdots t_n \cup x' \ t'_1 \cdots t'_n \longrightarrow x' \ t'_1 \cdots t'_n \cup x \ t_1 \cdots t_n \quad \text{if } x' \prec x \quad (3)$$

$$x \ t_1 \cdots t_n \cup (x' \ t'_1 \cdots t'_n \cup t) \longrightarrow x' \ t'_1 \cdots t'_n \cup (x \ t_1 \cdots t_n \cup t) \quad \text{if } x' \prec x \quad (4)$$

$$\{c\} \cup \{c'\} \longrightarrow \{c'\} \cup \{c\} \quad \text{if } c' \prec c \quad (5)$$

$$\{c\} \cup (\{c'\} \cup t) \longrightarrow \{c'\} \cup (\{c\} \cup t) \quad \text{if } c' \prec c \quad (6)$$

Technicalities: λ^U

Conjecture

The reduction relation \longrightarrow preserves meaning.

Conjecture

The reduction relation \longrightarrow is strongly normalizing.

Conjecture

The reduction relation \longrightarrow is locally confluent.

Corollary

The reduction relation \longrightarrow is confluent.

Corollary

The λ^U -calculus has unique normal forms.

Corollary

Equality of λ^U -terms can be decided by normalization.

Problems

- ▶ Not sound w.r.t. *imprecise exception semantics*.
- ▶ Making it sound negates the precision gained by higher-ranked types.
- ▶ Need to move to a more powerful constraint language.
 - ▶ In previous work we used conditionals/implications and a somewhat ad hoc non-emptiness guard.
 - ▶ Now I want to look at *Boolean rings*, which look more well-behaved.
 - ▶ May make more sense to use *equational unification* instead of constraints.

Problems: Imprecise exception semantics

- In an optimizing compiler we want the following equality, called the *case-switching transformation*, to hold:

$$\begin{aligned} \forall e_i. & \text{if } e_1 \text{ then} \\ & \quad \text{if } e_2 \text{ then } e_3 \text{ else } e_4 \\ & \text{else} \\ & \quad \text{if } e_2 \text{ then } e_5 \text{ else } e_6 \equiv \text{if } e_2 \text{ then} \\ & \quad \quad \text{if } e_1 \text{ then } e_3 \text{ else } e_5 \\ & \quad \text{else} \\ & \quad \quad \text{if } e_1 \text{ then } e_4 \text{ else } e_6 \end{aligned}$$

- This does not hold if we have observable exceptions and track them precisely.
 - Counterexample: Take $e_1 = \perp_{E_1}$ and $e_2 = \perp_{E_2}$.
- Introduce some “imprecision”: If the guard can reduce to an exceptional value, then pretend both branches get executed.

Problems: Imprecise exception semantics

- ▶ In an optimizing compiler we want the following equality, called the *case-switching transformation*, to hold:

$$\begin{aligned} \forall e_i. & \text{if } e_1 \text{ then} \\ & \quad \text{if } e_2 \text{ then } e_3 \text{ else } e_4 \\ & \text{else} \\ & \quad \text{if } e_2 \text{ then } e_5 \text{ else } e_6 \equiv \text{if } e_2 \text{ then} \\ & \quad \quad \text{if } e_1 \text{ then } e_3 \text{ else } e_5 \\ & \quad \text{else} \\ & \quad \quad \text{if } e_1 \text{ then } e_4 \text{ else } e_6 \end{aligned}$$

- ▶ This does not hold if we have observable exceptions and track them precisely.
 - ▶ Counterexample: Take $e_1 = \perp_{E_1}$ and $e_2 = \perp_{E_2}$.
- ▶ Introduce some “imprecision”: **If** the guard can reduce to an exceptional value, **then** pretend both branches get executed.

References



Stefan Holdermans and Jurriaan Hage, *Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators*, Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (New York, NY, USA), ICFP '10, ACM, 2010, pp. 63–74.



Andrew J. Kennedy, *Type inference and equational theories*, Tech. Report LIX-RR-96-09, Laboratoire D'Informatique, École Polytechnique, 1996.