June 13, 2018

Curry-Howard Correspondence through Type-based Program Analyses

Major Area Exam

Miroslav Gavrilov

Committee:

Ben Hardekopf

Tevfik Bultan

Chandra Krintz

Outline

History and Intuition

General Type-based Analyses

Resources and Protocols

feat. Linear Logics

Information Flow

feat. Predicate Logic

Bigger, better, faster, stronger…

feat. Refinements, Effects and Daft Punk

History and Intuition

General Type-based Analyses

Resources and Protocols
feat. Linear Logics

Information Flow
feat. Predicate Logic

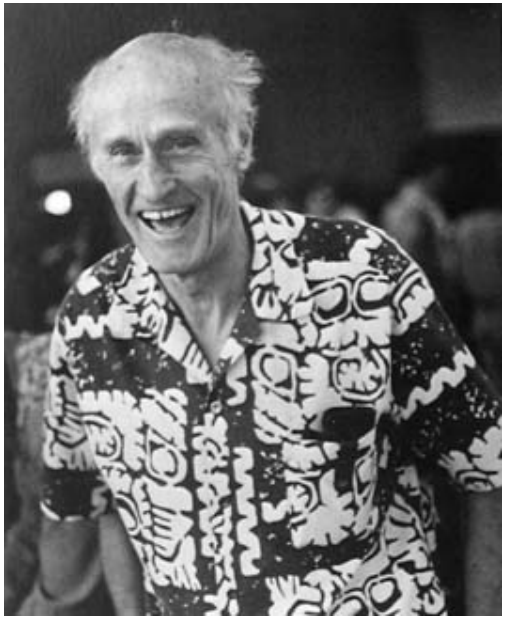
Bigger, better, faster, stronger…
feat. Refinements, Effects and Daft Punk

A Historical Perspective

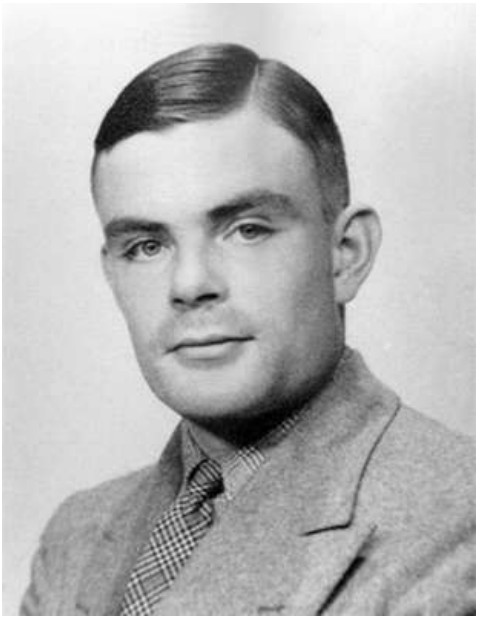
A definition of "*effectively calculable*"[1]:

Lambda Calculus[2]
Alonzo Church, 1930.

General Recursive Functions[3]
Stephen Kleene, 1936.



Turing Machines
Alan Turing, 1937.



[1] David Hilbert and Wilhelm Ackermann (1928). Grundzüge der theoretischen Logik. Springer-Verlag, ISBN 0-8218-2024-9.
[2] Church, Alonzo. An Unsolvable Problem of Elementary Number Theory. AJM 58, no. 2 (1936): 345-63. doi:10.2307/2371045.
[3] S. C. Kleene, General Recursive Functions of Natural Numbers. Matematische Annalen, Bd 112 (1935-6), S. 727-742

A Historical Perspective

A definition of "*effectively calculable*":

Alonzo Church

Lambda Calculus: a concise function notation

$$
\begin{array}{lll}
\mathbf{expr} & ::= \mathbf{var} & \textit{variable} \\
& \mid \lambda\ \mathbf{var}.\ \mathbf{expr} & \textit{abstraction} \\
& \mid \mathbf{expr}\ \mathbf{expr} & \textit{application}
\end{array}
$$

A Historical Perspective

A definition of “*effectively calculable*”:

Alonzo Church

Lambda Calculus: a concise function notation

$$\lambda f \ . \ \lambda x \ . \ f \, f \, f \, x$$

A Historical Perspective

A definition of "*effectively calculable*":

Alonzo Church

Lambda Calculus: a concise function notation

Added semantics of reduction:

$$(\lambda x \,.\, M[x]) \rightarrow (\lambda y \,.\, M[y]) \qquad \alpha\text{-conversion}$$

$$(\lambda x \,.\, M)\ E \rightarrow M[x := E] \qquad \beta\text{-reduction}$$

A Historical Perspective

An initial correspondence[4]: Russell's Paradox = Non-termination

Haskell Curry, 1934.

$$[\![R = \{X | X \notin X\}]\!]_{\text{Set}} = [\![R\ x = N(x\ x)]\!]_{\lambda}$$

$$\begin{aligned}
[\![(R\ R)]\!]_{\lambda} &= [\![N\ (R\ R)]\!]_{\lambda} \\
&= [\![N\ (N\ (R\ R))]\!]_{\lambda} \\
&= [\![N\ (N\ \cdots (R\ R))\cdots)]\!]_{\lambda}
\end{aligned}$$

[4] Curry, H. (1942). The Inconsistency of Certain Formal Logic. The Journal of Symbolic Logic, 7(3), 115-117. doi:10.2307/2269292

A Historical Perspective

A correspondence between functions and implications[5].

Haskell Curry, 1934.

$$A \to B \equiv A \supset B$$

(→): The type of every function corresponds to a provable proposition

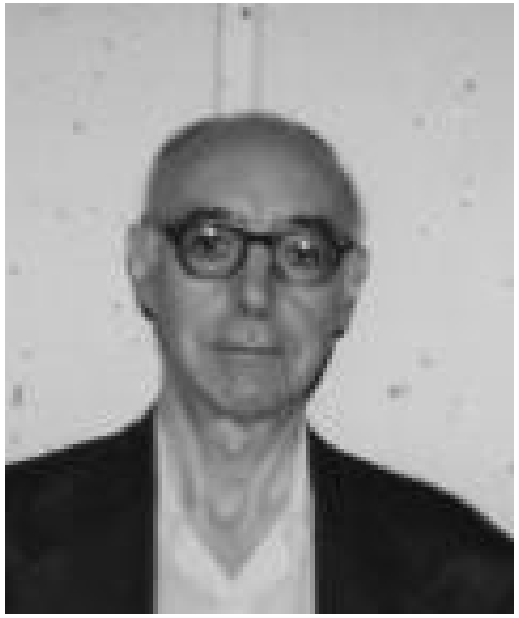
(←): For every provable proposition, there was a function with the corresponding type

[5] H. B. Curry. Functionality in combinatory logic. Proceedings of the National Academy of Science, 20:584–590, 1934.

A Historical Perspective

A correspondence between types and proposition[6].

William Alvin Howard, 1969.



$$A \times B \equiv A \ \wedge B$$
$$A + B \equiv A \ \vee B$$

Hinted at other correspondences between predicates and value-dependent types.

[6] W. A. Howard. The formulae-as-types notion of construction. In To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, pages 479–491. Academic Press, 1980.

A Historical Perspective

The Curry-Howard Correspondence: a link between programs and logic[7].

Intuitionistic Propositions as **Types**.

Proofs as **Programs.**

Simplification of Proofs as **Evaluation of Programs.**

[7] H. B. Curry and R. Feys. Combinatory Logic. North-Holland, 1958.

A Historical Perspective[8]

The Curry-Howard Correspondence: a link between programs and logic.

Classical Propositions as **Session Types**[9,10].

Proofs as **Processes.**

Cut Elimination as **Communication.**

[8] Philip Wadler. 2015. Propositions as types. Commun. ACM 58, 12 (November 2015), 75-84
[9] Philip Wadler. 2012. Propositions as sessions. In Proceedings of the 17th ACM SIGPLAN international conference on Functional programming (ICFP '12). ACM, New York, NY, USA, 273-286.
[10] Das, A., Hoffmann, J., & Pfenning, F. (2017). Work Analysis with Resource-Aware Session Types. CoRR, abs/1712.08310.

The Development of Type Theory

Logic blossoms in the 70s. Type systems follow.

Curry-Howard correspondence: Buy one, get one free.

Intuitionistic Propositional Logic: Simply Typed Lambda Calculus

Intuitionistic Predicate Logic: Calculus of Constructions[11]

Second-Order Logic: Polymorphic Type Systems[12,13]

Classical Logic: Parigot's λμ-Calculus[14]

Classical Linear Logic: Session Types

[11] Thierry Coquand and Gerard Huet. 1988. The calculus of constructions. Inf. Comput. 76, 2-3 (February 1988), 95-120.
[12] J.-Y. Girard. Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure, 1972. Université Paris VII, These D'Etat.
[13] J. C. Reynolds. Towards a theory of type structure. In Symposium on Programming, volume 19 of Lectures in Computer Science, pages 408–423, 1974.
[14] Michel Parigot. 1992. Lambda-My-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR '92), Andrei Voronkov (Ed.). Springer-Verlag, London, UK, UK, 190-201.

Type Systems Redux

Simply Typed Lambda Calculus

$$\begin{aligned} \mathbf{expr} ::= \ & \mathbf{var} \\ & \mid \lambda \mathbf{var} : \tau \ . \ \mathbf{expr} \\ & \mid \mathbf{expr} \ \ \mathbf{expr} \end{aligned}$$

Type definitions:

$$\tau ::= \alpha \mid \tau \to \tau$$

Typing rules:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} (1) \qquad \frac{\Gamma, x : \tau \vdash y : \sigma}{\Gamma \vdash (\lambda x : \tau \ . \ y) : (\tau \to \sigma)} (2)$$

$$\frac{\Gamma \vdash x : \tau \to \sigma \quad \Gamma \vdash y : \tau}{\Gamma \vdash x \ y : \sigma} (3)$$

Type Systems Redux

Most Typed Lambda Calculi are **strongly normalizing**:
every term can be written in a *normal form*, such that it doesn't β-reduce further

Strong normalization = All computations terminate[15]

Types:
infrastructure for building analyses and
lingua franca for analysis comparison[16].

[15] Girard, Jean-Yves (1972). "Interprétation fonctionnelle et Élimination des coupure de l'arithmétique d'ordre supérieur".
[16] Palsberg, Jens. "Type-based analysis and applications." PASTE (2001).

History and Intuition

General Type-based Analyses

Resources and Protocols

feat. Linear Logics

Information Flow

feat. Predicate Logic

Bigger, better, faster, stronger…

feat. Refinements, Effects and Daft Punk

Type-based Analyses

Type-checking is constraint solving.

Several methods can be utilized to base analyses on types:

Types as discriminators
Class Hierarchy Analysis, Rapid Type Analysis

Getting extra information from types
Redundant-load Elimination

Specialized type systems and type inference
Steensgaard pointer analysis

Types as Discriminators

Class Hierarchy Analysis[17]:
uses types as discriminators to do more precise method inlining

$$\begin{aligned}&\text{For virtual call } e.m(\cdots),\\ &\quad \forall C \in \text{SubtypesOf}(\text{StaticTypeOf}(e)),\\ &\quad \text{Ms} = \{M \mid \text{StaticLookup}_C(m) = M\}\end{aligned}$$

[17] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95), pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.

Types as Discriminators

Class Hierarchy Analysis:
uses types as discriminators to do precise method inlining

Rapid Type Analysis[18]:
traces all locations where *new()* is invoked and uses class-instantiation information to do class hierarchy analysis over *only* those classes

$$\begin{aligned}&\text{For virtual call } e.m(\cdots) \text{ in program } P,\\ &\quad \forall C \in \text{SubtypesOf}(\text{StaticTypeOf}(e)),\\ &\quad \text{Ms} = \{M \mid C \in \text{Instd}, \text{StaticLookup}_C(m) = M\}\\ &\quad\quad \text{where Instd} = \{C \mid \text{new } C(\cdots) \in P\}\end{aligned}$$

[18] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96), pages 324–341, San Jose, CA, 1996. SIGPLAN Notices 31(10)

Types as Discriminators

Class Hierarchy Analysis:
uses types as discriminators to do precise method inlining

Rapid Type Analysis:
traces all locations where *new()* is invoked and uses class-instantiation information to do class hierarchy analysis over *only* those classes

Weakness:
problems with libraries with source code missing during analysis-time, somewhat mitigated by library extractors (eg. Jax[19])

[19] Peter F. Sweeney and Frank Tip. Extracting librarybased object-oriented applications. In Proceedings of the Eighth International Symposium on the Foundations of Software Engineering (FSE-8), pages 98–107, November 2000

Redundant-Load Elimination

Redundant-Load Elimination[20]:
may benefit from alias information, due to being a mix of two optimizations (loop-invariant code motion and common-subexpression elimination)

Several type-based alias analyses that build off each other:

$$\begin{aligned}&\text{SubtypesOf}(\text{StaticType}(e_1)) \cap \\ &\quad \text{SubtypesOf}(\text{StaticType}(e_2)) = \emptyset \Leftrightarrow e_1 \not\cong e_2\end{aligned}$$

[20] Amer Diwan, Kathryn McKinley, and Eliot Moss. Type-based alias analysis. In Proceedings of PLDI'98, ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 106–117, 1998.

Redundant-Load Elimination

Redundant-Load Elimination:
may benefit from alias information, due to being a mix of two optimizations (loop-invariant code motion and common-subexpression elimination)

Several type-based alias analyses that build off each other:

$$\text{Given an expression } e, \text{ and fields } f \text{ and } g,$$
$$e.f \not\cong e.g \Leftrightarrow \text{StaticTypeDecl}(f) \neq \text{StaticTypeDecl}(g)$$

Redundant-Load Elimination

Redundant-Load Elimination:
may benefit from alias information, due to being a mix of two optimizations (loop-invariant code motion and common-subexpression elimination)

Several type-based alias analyses that build off each other:

$$\text{Given program } P,$$
$$e_1 \not\cong e_2 \Leftrightarrow \forall o_1 \in \text{StaticType}(e_1), o_2 \in \text{StaticType}(e_2)\ .$$
$$\{\ \text{Assign}(o_1, o_2), \text{Assign}(o_2, o_1)\ \} \cap P = \emptyset$$

Redundant-Load Elimination

Redundant-Load Elimination:
may benefit from alias information, due to being a mix of two optimizations (loop-invariant code motion and common-subexpression elimination)

Several type-based alias analyses that build off each other.

Weakness: in many cases, the separation of the optimizations and underlying execution environment is too great to be effective

Specialized Type Systems and Type Inference

Steensgaard Points-to Analyses[21,22]: Interprocedural flow-insensitive points-to analyses based on type inference using non-standard types.

Types describe a storage model: every type is a set of locations

$$\alpha ::= \tau \times \lambda$$
$$\tau ::= \bot \mid \mathbf{ref}(\alpha)$$
$$\lambda ::= \bot \mid \mathbf{lam}(\alpha_1 \cdots \alpha_n) \to (\alpha_{n+1} \cdots \alpha_{n+m})$$

[21] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96). ACM, New York, NY, USA, 32-41. DOI=http://dx.doi.org/10.1145/237721.237727

[22] Bjarne Steensgaard. 1996. Points-to Analysis by Type Inference of Programs with Structures and Unions. In Proceedings of the 6th International Conference on Compiler Construction (CC '96), Tibor Gyimothy (Ed.). Springer-Verlag, London, UK, UK, 136-150.

Specialized Type Systems and Type Inference

Steensgaard Points-to Analyses: Interprocedural flow-insensitive points-to analyses based on type inference using non-standard types.

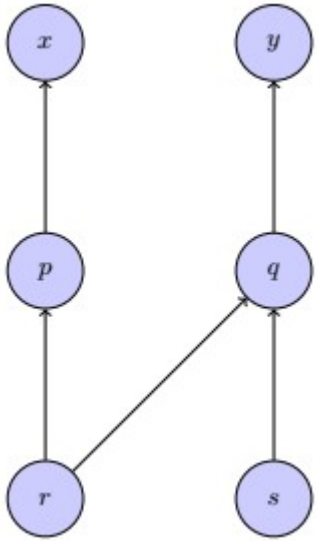
$$\frac{\begin{array}{c} A \vdash y \ : \ \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1 \cdots \alpha_n) \to (\alpha_{n+1} \cdots \alpha_{n+m})) \\ \forall i \in \{1 \cdots n\} \ . \ A \vdash f_i : \mathbf{ref}(\alpha_i) \\ \forall j \in \{1 \cdots m\} \ . \ A \vdash r_j : \mathbf{ref}(\alpha_{n+j}) \\ \forall s \in S \ . \ A \vdash \mathrm{welltyped}(s) \end{array}}{A \vdash \mathrm{welltyped}(x = \mathbf{fun}(f_1 \cdots f_n) \to (r_1 \cdots r_m))}$$

Specialized Type Systems and Type Inference

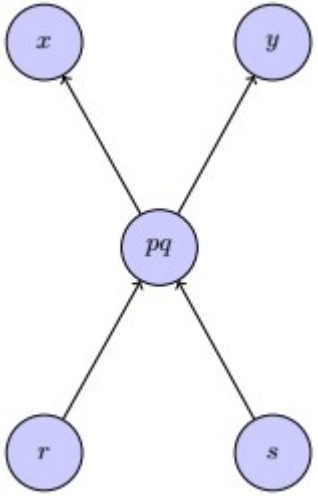
Steensgaard Points-to Analyses: Interprocedural flow-insensitive points-to analyses based on type inference using non-standard types.

Fast (union-find and simple type system) but often imprecise (joins multiple sets into one)

$$
\begin{array}{lll}
(1) & p & := \&x \\
(2) & r & := \&p \\
(3) & q & := \&y \\
(4) & s & := \&q \\
(5) & r & := s
\end{array}
$$



Andersen[23]



Steensgaard

[23] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen, May 1994. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z

Type-based Analyses in Review

Types are good bases for static analyses:
simple, efficient[24,25], correct[26], competitive

Type-based analysis construction:
Tools for writing analyses that enable focusing on the analysis itself and not the implementational details

Banshee[28] (successor to **BANE**[27])
compiles specifications to specialized resolution engines per analysis;
incremental analysis via backtracking;
both support mixed constraints;

[24] Flemming Nielson. The typed lambda-calculus with first-class processes. In Proceedings of PARLE, pages 357–373, April 1989.
[25] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Correct System Design, pages 114–136, 1999.
[26] Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17:348–375, 1978.
[27] Aiken, A., F¨ahndrich, M., Foster, J., Su, Z.: A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In: Proceedings of the Second International Workshop on Types in Compilation (TIC'98). (1998)
[28] John Kodumal and Alex Aiken. 2005. Banshee: a scalable constraint-based analysis toolkit. In Proceedings of the 12th international conference on Static Analysis (SAS'05), Chris Hankin and Igor Siveroni (Eds.). Springer-Verlag, Berlin, Heidelberg, 218-234. DOI=http://dx.doi.org/10.1007/11547662_16

Type-based Analyses in Review

Types are good bases for static analyses:
simple, efficient, correct, competitive

Type-based analysis construction:
Tools for writing analyses that enable focusing on the analysis itself and not the implementational details

Banshee (successor to **BANE**)
doesn't necessarily model the full type system, but rather just the constraint specification

```
specification steensgaard : STE =
  spec
    data location : set
    data T : term = ref of label * T
  end
```

History and Intuition

General Type-based Analyses

Resources and Protocols

feat. Linear Logics

Information Flow Analysis

feat. Predicate Logic

Bigger, better, faster, stronger…

feat. Refinements, Effects and Daft Punk

Linear Logic

Structural Rules:
Inference rules that operate on judgements directly, mimic meta-theoretical properties of logic

Weakening
sequents might be extended with additional members

Contraction
two equal members on the same side of the sequent may be reduced to one

Exchange
two members on the same side of the sequent may be swapped (permuted)

Linear Logic

Structural Rules:
Inference rules that operate on judgements directly,
mimic meta-theoretical properties of logic

Linear logic[29]: ~~Weakening~~, ~~Contraction~~, Exchange

Every assumption can be used *exactly* once.

Non-linear Proposition

$$A \to B, A \vdash A \times B$$

Linear Proposition

$$\langle A \multimap B \rangle, \langle A \rangle \not\vdash A \otimes B$$

Eating cake makes us full, a cake exists:
eating cake makes us full and cake exists.

[29] Girard, Jean-Yves. Linear logic, Theoretical Computer Science, Vol 50, no 1, pp. 1–102, 1987.

Linear Logic

Structural Rules:
Inference rules that operate on judgements directly, mimic meta-theoretical properties of logic

Linear logic: ~~Weakening~~, ~~Contraction~~, Exchange

Every assumption can be used *exactly* once.

Non-linear Proposition

$$A \to B, A \vdash A \times B$$

Linear Proposition

$$\langle A \multimap B \rangle, \langle A \rangle \not\vdash A \otimes B$$

Consuming one cake makes us full, one cake exists:
we can't both have our cake and eat it[30].

[30] Philip Wadler. 1993. A Taste of Linear Logic. In Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science (MFCS '93), Andrzej M. Borzyszkowski and Stefan Sokolowski (Eds.). Springer-Verlag, London, UK, UK, 185-210.

Linear Logic

Rust[31]:

The most well-known example of a linear type system in use.

```
let x = vec![1, 2, 3];
let y = x;                    // NOTE: x moved
println!("{:?}", x);          // ERROR: x gone
pritnln!("{:?}", y);          // OK
```

[31] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The rust language. In Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology (HILT '14). ACM, New York, NY, USA, 103-104

Linear Logic

Observers for linear types[32]:
introduces a third possibility for variables (beside *linear* or *non-linear*), *observers for linear variables* can be duplicated and read concurrently

Linear types as optimizations:
require no garbage collection or reference counting;

Reference Counting as a Computational Interpretation of Linear Logic[33]:
demonstrate a precise relationship between type correctness for a language based on linear logic and the correctness of reference-counting

[32] Odersky M. (1992) Observers for linear types. In: Krieg-Brückner B. (eds) ESOP '92. ESOP 1992. Lecture Notes in Computer Science, vol 582. Springer, Berlin, Heidelberg
[33] Chirimar, Jawahar & A. Gunter, Carl & G. Riecke, Jon. (2000). Reference Counting as a Computational Interpretation of Linear Logic. Journal of Functional Programming. 6. 10.1017/S0956796800001660.

Classical Linear Logic

Propositions as **Sessions**[9]

Session types:
communication-centric, conforming to protocols

$A ::=$	p	proposition
	$\mid p^{\perp}$	dual
	$\mid A \otimes B$	times; both
	$\mid A \oplus B$	plus; either
	$\mid A \& B$	amp; choose
	$\mid A \parr B$	par; in parallel
	$\mid !A$	of course!
	$\mid ?A$	why not?

$A ::=$	p	message
	$\mid p^{\perp}$	session dual
	$\mid A \otimes B$	output A then behave as B
	$\mid A \oplus B$	select from A and B
	$\mid A \& B$	offer choice between A and B
	$\mid A \parr B$	input A then behave as B
	$\mid !A$	server accept
	$\mid ?A$	client request

[9] Philip Wadler. 2012. Propositions as sessions. In Proceedings of the 17th ACM SIGPLAN international conference on Functional programming (ICFP '12). ACM, New York, NY, USA, 273-286.

Classical Linear Logic

Propositions as **Sessions**

Builds a linear session-typed lambda calculus (CP) on top of linear logic, and then a functional language with sessions (GV) on top of that.

The whole stack (linear types → CP → GV) is strongly normalizing.

Deadlock free due to Curry-Howard correspondence:
Russell's Paradox = Non-termination = Deadlock

Classical Linear Logic

Propositions as **Sessions**

Builds a linear session-typed lambda calculus (CP) on top of linear logic, and then a functional language with sessions (GV) on top of that.

The whole stack (linear types → CP → GV) is strongly normalizing.

Deadlock free due to Curry-Howard correspondence:
Russell's Paradox = Non-termination = Deadlock

Resource-awareness[10]:
annotate every binary operation with a communicational cost,
using the extra information to do type-based amortized resource analysis

$$A \overset{\alpha}{\multimap} B$$

[10] Das, A., Hoffmann, J., & Pfenning, F. (2017). Work Analysis with Resource-Aware Session Types. CoRR, abs/1712.08310.

History and Intuition

General Type-based Analyses

Resources and Protocols

feat. Linear Logics

Information Flow

feat. Predicative Logic

Bigger, better, faster, stronger...

feat. Refinements, Effects and Daft Punk

Linear Dependent Types

Static resource bound analysis, cont'd:

Lago et al[34] introduce a linear type-system which describes types dependent on resource-cost values

Dependent Types:

Curry-Howard correspondence dual of intuitionistic predicate logic[35]

$$\forall x \in A.B(x) \equiv \Pi_{x:A}\, B(x)$$
$$\exists x \in A.B(x) \equiv \Sigma_{x:A}\, B(x)$$

Types are now *indexed* by a value.

[34] Lago, U.D., Petit, B.: The Geometry of Types. In: (POPL'13) (2013)
[35] Martin-Löf, Per, 1971a, An intuitionistic theory of types, unpublished preprint.

Linear Dependent Types

Practical Examples for dependent types:
Some constraints we couldn’t write without dependent types

The type of a vector of strings of length x:

$$\Pi_{n:\mathbb{N}}\ \mathbf{Vec}(\mathbf{String}, x)$$

The type of the ≤-relation between two integers:

$$\Sigma_{n:\mathbb{N}}\ \lambda n\ .\ a + n = b$$

Information-Flow Type Systems

Type systems dependent only on special types[36]:
pass information-flow through control-flow,
either stop compilation due to value mismatch, or do runtime check

```
class C {
 final principal user = Runtime.getUser()

 void print(String{user:} s) {...}
 void printIfManager(String{Manager:} s)
  actsFor(user, Manager) {
   print(s);
  }
 }
}
```

Non-trivial example:
Public key-passing with runtime principals with a proof of soundness and noninterference[37]

[36] Anindya Banerjee and David A. Naumann. 2002. Secure Information Flow and Pointer Confinement in a Java-like Language. In Proceedings of the 15th IE
[37] A. Sabelfeld and A. C. Myers. 2006. Language-based information-flow security. IEEE J.Sel. A. Commun. 21, 1 (September 2006), 5-19.

Information-Flow Type Systems

Full dependence of information flow types[38]
instead of depending on special values (principals)

$$\mathrm{get\_passwd} \;:\; \Pi_{x:\mathbf{String}^{\perp}} \mathbf{String}^{\mathrm{usr}(x)}$$

$$\mathrm{get\_user\_cred} \;:\; \Sigma_{\mathrm{uid}:\mathbf{String}^{\perp} \times \mathrm{passwd}:\mathbf{String}^{\mathrm{usr}(\mathrm{uid})}} (\mathbf{String} \times \mathbf{String})^{\perp}$$

Constraint-solving at compile-time:
Full dependent types are basically compile-time programs

[38] Luísa Lourenço and Luís Caires. 2015. Dependent Information Flow Types. SIGPLAN Not. 50, 1 (January 2015), 317-328.

Interactive Access Control Policies

Dependent type languages are often interactive proof assistants:
use the proof assistant part as interactive access control check[39]

AGDA allows programmer calls to the theorem solver:

$$\begin{aligned}&\text{goal} = \text{MayRead}(\text{currentUser}, \text{File}"\text{secret.txt}")\\&\text{proof?} : \mathbf{Maybe}\ (\mathbf{Proof}\ \Gamma\ \text{goal})\\&\text{proof?} = \text{solve}\ (\text{proveToDepth}\ 15)\end{aligned}$$

$$\text{solve} \ : \ \forall\ \{A\}(s \ : \ \mathbf{Maybe}\ A) \rightarrow \{p \ : \ \text{Check}(\text{isSome}\ s)\} \rightarrow A$$

[39] Jamie Morgenstern and Daniel R. Licata. 2010. Security-typed programming within dependently typed programming. In Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10). ACM, New York, NY, USA, 169-180.

Interactive Access Control Policies

Dependent type languages are often interactive proof assistants:
use the proof assistant part as interactive access control check

AGDA allows programmer calls to the theorem solver:

Access control mechanism defined in library

$$\begin{aligned}
&\text{goal} = \text{MayRead}(\text{currentUser}, \text{File}"\text{secret.txt}") \\
&\text{proof?} : \textbf{Maybe}\ (\textbf{Proof}\ \Gamma\ \text{goal}) \\
&\text{proof?} = \text{solve}\ (\text{proveToDepth}\ 15)
\end{aligned}$$

$$\text{solve} : \forall\ \{A\}(s : \textbf{Maybe}\ A) \rightarrow \{p : \text{Check}(\text{isSome}\ s)\} \rightarrow A$$

Interactive Access Control Policies

Dependent type languages are often interactive proof assistants:
use the proof assistant part as interactive access control check

AGDA allows programmer calls to the theorem solver:

$$\begin{aligned}
&\text{goal} = \text{MayRead}(\text{currentUser}, \text{File}"\text{secret.txt}") \\
&\text{proof?} : \mathbf{Maybe}\ (\mathbf{Proof}\ \Gamma\ \text{goal}) \\
&\text{proof?} = \text{solve}\ (\text{proveToDepth}\ 15)
\end{aligned}$$

The environment Γ holds access control configurations against which we check

$$\text{solve} : \forall\ \{A\}(s : \mathbf{Maybe}\ A) \rightarrow \{p : \text{Check}(\text{isSome}\ s)\} \rightarrow A$$

Interactive Access Control Policies

Dependent type languages are often interactive proof assistants:
use the proof assistant part as interactive access control check

AGDA allows programmer calls to the theorem solver:

$$\begin{aligned}&\text{goal} = \text{MayRead}(\text{currentUser}, \text{File}"\text{secret.txt}")\\&\text{proof?} : \textbf{Maybe}\ (\textbf{Proof}\ \Gamma\ \text{goal})\\&\text{proof?} = \text{solve}\ (\text{proveToDepth}\ 15)\end{aligned}$$

Implicit type variable: starts theorem solver unification

$$\text{solve} : \forall\ \{A\}\ (s : \textbf{Maybe}\ A) \rightarrow \{p : \text{Check}(\text{isSome}\ s)\} \rightarrow A$$

Interactive Access Control Policies

Dependent type languages are often interactive proof assistants:
use the proof assistant part as interactive access control check

AGDA allows programmer calls to the theorem solver:

$$\begin{aligned}&\text{goal} = \text{MayRead}(\text{currentUser}, \text{File}"\text{secret.txt}")\\&\text{proof?} : \textbf{Maybe}\ (\textbf{Proof}\ \Gamma\ \text{goal})\\&\text{proof?} = \text{solve}\ (\text{proveToDepth}\ 15)\end{aligned}$$

Such a type exists that this predicate is valid

$$\text{solve} : \forall\ \{A\}(s : \textbf{Maybe}\ A) \rightarrow \{p : \text{Check}(\text{isSome}\ s)\} \rightarrow A$$

Interactive Access Control Policies

Dependent type languages are often interactive proof assistants:
use the proof assistant part as interactive access control check

AGDA allows programmer calls to the theorem solver:

$$\begin{aligned}&\text{goal} = \text{MayRead}(\text{currentUser}, \text{File}"\text{secret.txt}")\\&\text{proof?} : \textbf{Maybe}\ (\textbf{Proof}\ \Gamma\ \text{goal})\\&\text{proof?} = \text{solve}\ (\text{proveToDepth}\ 15)\end{aligned}$$

If not, type checking will fail and access control policy is upheld

$$\text{solve} : \forall\ \{A\}(s : \textbf{Maybe}\ A) \rightarrow \{p : \text{Check}(\text{isSome}\ s)\} \dashrightarrow A$$

Dependent Types in Review

We are not limited to one certain dependent type system.
Value-dependent types can be specialized or general.

New languages with selected-domain type-dependence
Permission-dependent types for secure information[40]

Existing languages with full dependent types
Introduce syntactic elements to support information-flow types[39,41]

Downsides
Require a theorem prover for solving higher-order constraints

Generally high-level of maintenance

[40] Chen, Hongxu & Tiu, Alwen & Xu, Zhiwu & Liu, Yang. (2017). A Permission-Dependent Type System for Secure Information Flow Analysis.
[41] Lindahl, Eric & Winter, Victor. (2008). Pattern Matching Information Flow using GADT. 3rd International Conference on Information Warfare and Security.

History and Intuition

General Type-based Analyses

Resources and Protocols

feat. Linear Logics

Information Flow

feat. Predicate Logic

Bigger, better, faster, stronger...

feat. Refinements, Effects and Daft Punk

Upgrading Type Systems

Liquid types[42]:

*L*ogically *Q*ualified *D*ata Types, combine Hindley[43]-Milner[44] type inference with predicate abstraction to infer dependent types

$$\text{average} :: [\text{Int}] \to \text{Int}$$
$$\text{average}\ xs = \text{sum}\ xs\ \mathbf{div}\ \text{length}\ xs$$

$$\text{average} :: \{l@[\text{Int}] \mid \text{length}\ l > 0\} \to \text{Int}$$
$$\text{average}\ xs = \text{sum}\ xs\ \mathbf{div}\ \text{length}\ xs$$

[42] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08). ACM, New York, NY, USA, 159-169

[43] Hindley, J. Roger (1969). "The Principal Type-Scheme of an Object in Combinatory Logic". Transactions of the American Mathematical Society. 146: 29–60

[44] Milner, Robin (1978). "A Theory of Type Polymorphism in Programming". Journal of Computer and System Science (JCSS). 17: 348–374.

Upgrading Type Systems

Liquid types:
*L*ogically *Q*ualified *D*ata Types, combine Hindley-Milner type inference with predicate abstraction to infer dependent types

Three step process:

1. Hindley-Milner type inference to get **templates** (dependent types with unknown base refinements)

2. Generate constraints to capture subtyping relations between refinements

3. Solve constraints to find, for each base refinement, the strongest conjunction of qualifiers to satisfy all constraints

Upgrading Type Systems

Liquid types:

*L*ogically *Q*ualified *D*ata Types, combine Hindley-Milner type inference with predicate abstraction to infer dependent types

Can be introduced into existing languages (LiquidHaskell, LiquidML, …)

Downsides:

System isn’t very expressive (refinements are usually meta-functions defined in the native language);

Error reporting is very abstract and requires intimate knowledge of the system to handle

A correspondence of sorts

An additional type-system for type refinements[45]:
Specialized type refinements, such that they present effectful computations

```
saveToFile(f : File, d : Data) : unit
{
  write(f, d);
  // Error: f not open.

  close(f);
  // Error: f not open.
}
```

[45] Yitzhak Mandelbaum, David Walker, and Robert Harper. 2003. An effective theory of type refinements. In Proceedings of the eighth ACM SIGPLAN international conference on Functional programming (ICFP '03). ACM, New York, NY, USA, 213-225

A correspondence of sorts

An additional type-system for type refinements:
Specialized type refinements, such that they present effectful computations

```
saveToFile(f : File, d : Data; closed(f))
  : (unit; closed(f))
{
  // closed(f)
  open(f);
  // open(f)
  write(f,d);
  // open(f)
  close(f);
  // closed(f)
}
```

initial conditions are given as input effects

A correspondence of sorts

An additional type-system for type refinements:
Specialized type refinements, such that they present effectful computations

```
saveToFile(f : File, d : Data; closed(f))
  : (unit; closed(f))
{
  // closed(f)
  open(f);
  // open(f)
  write(f,d);
  // open(f)
  close(f);
  // closed(f)
}
```

expected effects are asserted
at function exit

A correspondence of sorts

An additional type-system for type refinements:
Specialized type refinements, such that they present effectful computations

```
saveToFile(f : File, d : Data; closed(f))
  : (unit; closed(f))
{
  // closed(f)
  open(f);
  // open(f)
  write(f,d);
  // open(f)
  close(f);
  // closed(f)
}
```

Similar to Hoare logic (assume pre-condition/assert post-condition), but applicable to higher-order programs and with a decidable type-checking

A correspondence of side-effects

An additional type-system for type refinements:
Specialized type refinements, such that they present effectful computations

The base type system is implicitly dependent on refinements,
and open dependently-type programming is prohibited

A base of system/language-level refinements and refinement-typed functions
must exist for the system to be useful

Can we build a specialized type-effect system without dependent types?

Effect systems

Type-and-Effect systems:

A control flow analysis[46] built so that the type judgements include the effects directly

A set of special values, or union thereof

$$\rho ::= \{ \pi \} \mid \rho_1 \cup \rho_2 \mid \emptyset$$

$$\tau ::= \alpha \mid \tau_1 \xrightarrow{\rho} \tau_2$$

[46] Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel), Ernst-Rüdiger Olderog and Bernhard Steffen (Eds.). Springer-Verlag, Berlin, Heidelberg, 114-136.

Effect systems

Type-and-Effect systems:

A control flow analysis built so that the type judgements include the effects directly

$$\rho ::= \{ \pi \} \mid \rho_1 \cup \rho_2 \mid \emptyset$$

$$\tau ::= \alpha \mid \tau_1 \xrightarrow{\rho} \tau_2$$

A potentially effectful computation

Effect systems

Type-and-Effect systems:
A control flow analysis built so that the type judgements include the effects directly

$$\rho ::= \{ \pi \} \mid \rho_1 \cup \rho_2 \mid \emptyset$$
$$\tau ::= \alpha \mid \tau_1 \xrightarrow{\rho} \tau_2$$

Useful for expressing side-effects, binding type analysis, region structure and causal linear logic:
the last define communication protocols (corresponding to session types)

$$A \overset{\alpha}{\multimap} B$$

Very Specialized Effect systems

Music [h]as an effect[47]:
effects don’t have to be sets

$$play :: (\text{frequency} : \mathbf{Int}, \text{length} : \mathbf{Int})$$

[47] Mycroft A., Orchard D., Petricek T. (2016) Effect Systems Revisited—Control-Flow Algebra and Semantics. In: Probst C., Hankin C., Hansen R. (eds) Semantics, Logics, and Calculi. Lecture Notes in Computer Science, vol 9560. Springer, Cham

Very Specialized Effect systems

Music [h]as an effect:

```
let HarderBetterFasterStronger =
  play(A1, 1/4); play(A1, 1/4);
  twice {
    twice {i ⇒
      play(E1, 1/4); play(F#1, 1/4);
      play(A1, 1/4); play(F#1, 1/4);
      play(C#2, 1/4); play(H1, 1/4);
      if i == 1
        play(A1, 1/4); play(F#1, 1/4);
      else
        play(A0, 1/4); play(F#0, 1/4);
    }
  }
  play(A#0, 1/8); play(A#0, 1/8);
  play(F0, 1/8); play(F0, 1/8);
```

[48] Daft Punk, Harder, better, faster, stronger, Single 2001. https://musescore.com/user/7949346/scores/1817801

Very Specialized Effect systems

Music [h]as an effect:

Musical effects can compose

```
let HarderBetterFasterStronger =
  play(A1, 1/4); play(A1, 1/4);
  twice {
    twice {i ⇒
      play(E1, 1/4); play(F#1, 1/4);
      play(A1, 1/4); play(F#1, 1/4);
      play(C#2, 1/4); play(H1, 1/4);
      if i == 1
        play(A1, 1/4); play(F#1, 1/4);
      else
        play(A0, 1/4); play(F#0, 1/4);
    }
  }
  play(A#0, 1/8); play(A#0, 1/8);
  play(F0, 1/8); play(F0, 1/8);
```

Very Specialized Effect systems

Music [h]as an effect:

Musical effects can repeat

```
let HarderBetterFasterStronger =
  play(A1, 1/4); play(A1, 1/4);
  twice {
    twice {i ⇒
      play(E1, 1/4); play(F#1, 1/4);
      play(A1, 1/4); play(F#1, 1/4);
      play(C#2, 1/4); play(H1, 1/4);
      if i == 1
        play(A1, 1/4); play(F#1, 1/4);
      else
        play(A0, 1/4); play(F#0, 1/4);
    }
  }
  play(A#0, 1/8); play(A#0, 1/8);
  play(F0, 1/8); play(F0, 1/8);
```

Very Specialized Effect systems

Music [h]as an effect:

Musical effects can diverge

```
let HarderBetterFasterStronger =
  play(A1, 1/4); play(A1, 1/4);
  twice {
    twice {i ⇒
      play(E1, 1/4); play(F#1, 1/4);
      play(A1, 1/4); play(F#1, 1/4);
      play(C#2, 1/4); play(H1, 1/4);
      if i == 1
        play(A1, 1/4); play(F#1, 1/4);
      else
        play(A0, 1/4); play(F#0, 1/4);
    }
  }
  play(A#0, 1/8); play(A#0, 1/8);
  play(F0, 1/8); play(F0, 1/8);
```

Very Specialized Effect systems

Music [h]as an effect:

The requirements for musical effects seem like regular expressions.

$$(\textsc{play})\ \frac{}{\Gamma \vdash \mathsf{play}(N, l) : \mathsf{void},\ N} \qquad (\textsc{if})\ \frac{\Gamma \vdash e_0 : \mathsf{bool},\ \Phi_0 \quad \Gamma \vdash e_1 : \tau,\ \Phi_1 \quad \Gamma \vdash e_2 : \tau,\ \Phi_2}{\Gamma \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau,\ \Phi_0 \bullet (\Phi_1 + \Phi_2)}$$

$$(\textsc{seq})\ \frac{\Gamma \vdash e_1 : \tau_1,\ \Phi_1 \quad \Gamma \vdash e_2 : \tau_2,\ \Phi_2}{\Gamma \vdash e_1; e_2 : \tau_2,\ \Phi_1 \bullet \Phi_2} \qquad (\textsc{for})\ \frac{\Gamma \vdash e : \mathsf{void},\ \Phi}{\Gamma \vdash \mathsf{for}\ i = n_1\ \mathsf{to}\ n_2\ \mathsf{do}\ e : \mathsf{void},\ \Phi^*}$$

Very Specialized Effect systems

Music [h]as an effect:

The requirements for musical effects seem like regular expressions.

$$\textbf{let}\ \mathrm{Song}^{A_1 \bullet A_1 \bullet ((E_1 \bullet F\#_1 \bullet A_1 \bullet F\#_1 \bullet C\#_2 \bullet H_1)^* \bullet (A_1 \bullet F\#_1 + A_0 \bullet F\#_0))^* \bullet (A\#_0 \bullet A\#_0 \bullet F_0 \bullet F_0)} =$$

...expected, but...

$$\textbf{let}\ \mathrm{Song}^{A_1 \bullet A_1 \bullet (((E_1 \bullet F\#_1 \bullet A_1 \bullet F\#_1 \bullet C\#_2 \bullet H_1) \bullet (A_1 \bullet F\#_1 + A_0 \bullet F\#_0))^*)^* \bullet (A\#_0 \bullet A\#_0 \bullet F_0 \bullet F_0)} =$$

...found!

Very Specialized Effect systems

Music [h]as an effect:

The requirements for musical effects seem like regular expressions.

$$\textbf{let}\ \mathrm{Song} = A_1 \bullet A_1 \bullet ((E_1 \bullet F\#_1 \bullet A_1 \bullet F\#_1 \bullet C\#_2 \bullet H_1)^* \bullet (A_1 \bullet F\#_1 + A_0 \bullet F\#_0))^* \bullet (A\#_0 \bullet A\#_0 \bullet F_0 \bullet F_0) =$$

...expected, but...

$$\textbf{let}\ \mathrm{Song} = A_1 \bullet A_1 \bullet (((E_1 \bullet F\#_1 \bullet A_1 \bullet F\#_1 \bullet C\#_2 \bullet H_1) \bullet (A_1 \bullet F\#_1 + A_0 \bullet F\#_0))^*)^* \bullet (A\#_0 \bullet A\#_0 \bullet F_0 \bullet F_0) =$$

...found!

Refinement and Effects in Review

Baring implementational differences,
they can be quite similar in effect (no pun int'd)

Theory of refinements:
Includes the theory of effectful computations, and more

Coeffects[49]:
dual to effects – theory of context-dependent computations

Direct relation to Curry-Howard correspondence:
Visible somewhat more clearly through category theory[50]

[49] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. In Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (ICFP '14). ACM, New York, NY, USA, 123-135.
[50] Paul-André Melliès and Noam Zeilberger. 2015. Functors are Type Refinement Systems. In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15). ACM, New York, NY, USA, 3-16.

Thank You!

Proof that Russell’s Paradox is non-termination

Proof that Russell's Paradox is non-termination

An initial correspondence: Russell's Paradox = Non-termination

Haskell Curry, 1934.

$$\text{Is } R \in R \text{ if } R = \{X | X \notin X\}?$$

$$f_X(e) = \begin{cases} 1, \text{if } e \in X \\ 0, \text{otherwise} \end{cases}$$

$$\text{Given a set } X, \text{ if } x = f_X:$$
$$[\![(x\ x)]\!]_\lambda \Rightarrow [\![X \in X]\!]_{\text{Set}}$$

Proof that Russell's Paradox is non-termination

An initial correspondence: Russell's Paradox = Non-termination

Haskell Curry, 1934.

Given a set X, if $x = f_X$:

$$[\![(x\ x)]\!]_\lambda \Rightarrow [\![X \in X]\!]_{\mathrm{Set}}$$

If $(N\ e)_\lambda$ is negation, then:

$$[\![N(x\ x)]\!]_\lambda \Rightarrow [\![X \notin X]\!]_{\mathrm{Set}}$$

$$[\![R = \{X | X \notin X\}]\!]_{\mathrm{Set}} = [\![R\ x = N(x\ x)]\!]_\lambda$$

Proof that Russell's Paradox is non-termination

An initial correspondence: Russell's Paradox = Non-termination

Haskell Curry, 1934.

$$[\![R = \{X | X \notin X\}]\!]_{\mathrm{Set}} = [\![R\ x = N(x\ x)]\!]_{\lambda}$$

$$\begin{aligned}
[\![(R\ R)]\!]_{\lambda} &= [\![N\ (R\ R)]\!]_{\lambda} \\
&= [\![N\ (N\ (R\ R))]\!]_{\lambda} \\
&= [\![N\ (N\ \cdots (R\ R))\cdots)]\!]_{\lambda}
\end{aligned}$$

Proof that Russell's Paradox is non-termination

An initial correspondence: Russell's Paradox = Non-termination

Y-Combinator, 1933.
Haskell Curry

$$[\![R = \{X | X \notin X\}]\!]_{\mathrm{Set}} = [\![R\ x = N(x\ x)]\!]_{\lambda}$$

$$[\![(R\ R)]\!]_{\lambda} = [\![(\lambda x\ .\ N\ (x\ x))(\lambda x\ .\ N\ (x\ x))]\!]_{\lambda}$$

$$[\![(Y\ f)]\!]_{\lambda} = [\![(\lambda x\ .\ f\ (x\ x))]\!]_{\lambda}$$

$$[\![(R\ R)]\!]_{\lambda} = [\![(Y\ N)]\!]_{\lambda}$$

Further Examples of Correspondence

Further Examples of Correspondence

Intuitionistic Propositional Logic:
Simply Typed Lambda Calculus
Intuitionistic Higher-Order Predicate Logic:
Calculus of Constructions[11]
Second-Order Logic:
Polymorphic Type Systems[12,13]
Classical Logic:
Parigot's λμ-Calculus[14]
Classical Linear Logic:
Session Types
Classical First-Order Predicate Logic:
Lambda Calculus with **throw/catch** mechanisms[x]
Calculus of Inductive Constructions:
Coq

[x] Zorzi, Margherita & Aschieri, Federico. (2016). On Natural Deduction in Classical First-Order Logic: Curry-Howard Correspondence, Strong Normalization and Herbrand's Theorem. Theoretical Computer Science.

Papers I didn’t have time for

Papers I didn’t have time for

Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations.
In Proceedings ICFP ’97, International Conference on Functional Programming, ACM SIGPLAN Notices 32(8), pages 11–24, 1997.

Suresh Jagannathan, Andrew Wright, and Stephen Weeks. Type-directed flow analysis for typed intermediate languages.
In Proceedings of SAS’97, International Static Analysis Symposium. Springer-Verlag, 1997.

Christian Mossin. Exact flow analysis. In Proceedings of SAS’97, International Static Analysis Symposium, pages 250–264. Springer-Verlag (LNCS), 1997.

Jakob Rehof and Manule Fahndrich. Type-based flow analysis: From polymorphic subtyping to cfl-
reachability. In Proceedings of POPL’01, 28th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 54–66, 2001.

Franklyn Turbak, Allyn Dimock, Robert Muller, and J. B. Wells. Compiling with polymorphic and polyvariant flow types.
In ACM SIGPLAN Workshop on Types in Compilation, June 1997.

J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. Journal of Functional Programming.

Geoffrey Smith and Dennis Volpano. Secure information flow in multi-threaded imperative language.
In Proceedings of POPL’98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 355–364, 1998.

Zorzi, Margherita & Aschieri, Federico. (2016). On Natural Deduction in Classical First-Order Logic: Curry-Howard Correspondence, Strong Normalization and Herbrand’s Theorem. Theoretical Computer Science.