Lambda Calculus
Type Theory
and
Natural Language

\[ \lambda \text{Tt.n} \]

Edited by
Maribel Fernández
Chris Fox
Shalom Lappin

King’s College London
Strand, London, U.K.
8th–9th December 2003
Workshop on Lambda Calculus, Type Theory, and Natural Language

King’s College London, Strand, London, U.K., 8th–9th December 2003

Organizers:
- Maribel Fernández (maribel@dcs.kcl.ac.uk)
- Chris Fox (foxcj@essex.ac.uk)
- Shalom Lappin (lappin@dcs.kcl.ac.uk)

Contributors:
- David Clark King’s College London, U.K.
- Robin Cooper Göteborg University, Sweden
- Maribel Fernández King’s College London, U.K.
- Chris Fox University of Essex, U.K.
- Dov Gabbay King’s College London, U.K.
- Jamie Gabbay INRIA Futurs, France
- Simon Gay University of Glasgow, U.K.
- Jonathan Ginzburg King’s College London, U.K.
- Chris Hankin Imperial College, U.K.
- Shalom Lappin King’s College London, U.K.
- Ian Mackie King’s College London, U.K.
- François-Regis Sinot École Polytechnique, France
- Ray Turner University of Essex, U.K.

Key Note Speaker:
- Jan van Eijck CWI, Netherlands

Sponsors:
- Department of Computer Science
  - King’s College London
  - Strand
  - London WC2R 2LS
  - United Kingdom
  - http://www.dcs.kcl.ac.uk
- Department of Computer Science
  - University of Essex
  - Wivenhoe Park
  - Colchester CO4 3SQ
  - United Kingdom
  - http://cswww.essex.ac.uk
Programme

Monday, 8th December.

9:30  Coffee — Welcome
10:00 Opening
10:15 Chris Hankin.
      Lambda calculus and static analysis.
11:00 Coffee Break
11:30 Ian Mackie.
      Reduction in the lambda calculus.
12:15 Francois-Regis Sinot.
      N-ary director strings: Efficient representations of variables in terms.
13:00 Lunch
14:00 Jan van Eijck.
      Relations, Types and Scoping.
15:00 Chris Fox and Shalom Lappin.
      Underspecified Representations for Natural Language using Curry-typed \( \lambda \)-Calculus.
15:45 Coffee Break
16:15 Robin Cooper.
      Records and record types in semantic theory.
17:00 Jonathan Ginzburg.
      Abstraction and Ontology.
19:00 Dinner

Tuesday, 9th December.

9:30  Coffee
10:00 Maribel Fernandez.
      Rewriting Frameworks and Types.
10:45 David Clark.
      Measuring interference in PCF.
11:30 Coffee Break
12:00 Jamie Gabbay and Dov Gabbay.
      The restart rule and evaluation.
12:45 Lunch
14:00 Ray Turner.
      Polymorphism in Specifications.
14:45 Simon Gay.
      Session Types: Specifying Structured Communication.
15:30 Closing
Preface

The Workshop on Lambda Calculus, Type Theory, and Natural Language (LCTTNL) is designed to bring together researchers interested in functional programming, type theory, and the application of the lambda calculus to the analysis of natural language. The communities that have developed around each of these areas share many formal and computational interests. Unfortunately, they have not had much contact with each other. We hope that LCTTNL will provide a forum in which people working on the lambda calculus from a variety of distinct perspectives will share their research and come to appreciate new domains of application. If this encounter is successful, it can provide the basis for future cooperation and ongoing meetings of this kind.

We would like to thank the Computer Science Departments of King’s College London and the University of Essex for sponsorship and financial support. We are grateful to the speakers for their participation, and to our keynote speaker, Jan van Eijck, for agreeing to present his lecture.

Maribel Fernández
Chris Fox
Shalom Lappin

Contents

Relations, Types and Scoping 1
Jan van Eijck

Measuring Interference in PCF 2
David Clark, Sebastian Hunt and Pasquale Malacaria

Rewriting Frameworks and Types 4
Maribel Fernández

Restart as a computational rule 7
Murdoch Gabbay, Michael Gabbay and Dov Gabbay

Session Types: Specifying Structured Communication 9
Simon Gay

\(\lambda\text{-calculus and Program Analysis} 11
Chris Hankin and Herbert Wiklicky

Reduction in the lambda calculus 14
Ian Mackie

n-ary Director Strings 16
François-Régis Sinot

Underspecified Representations for Natural Language using Curry-typed \(\lambda\text{-Calculus} 17
Chris Fox and Shalom Lappin

Abstraction and Ontology 20
Jonathan Ginzburg

Polymorphism in Specification 24
Ray Turner

Records and record types in semantic theory 25
Robin Cooper
Many arguments for flexible type assignment to syntactic categories have to do with the need to account for the various scopings resulting from the interaction of quantified DPs with other quantified DPs or with intensional or negated verb contexts.

We will define a type for arbitrary arity relations in polymorphic type theory. In terms of this, we develop the Boolean algebra of relations as far as needed for natural language semantics. The type for relations is flexible: it can do duty for the whole family of types $t, e \rightarrow t, e \rightarrow e \rightarrow t$, and so on.

If we use this flexible type for the interpretations of verbs, we can perform a ‘flexible lift’ on DP interpretations, so that DPs get interpreted as operations on verb meanings. This leads to an elegant implementation of Keenan’s proposal for polyadic quantification in natural language (‘Beyond the Frege Boundary’). It turns out that scope reorderings are particularly easy to implement in this framework.
Measuring Interference in PCF (abstract)

D. Clark, S. Hunt and P. Malacaria

In recent work we have developed a security analysis for a simple imperative (while) language which has measured the leakage of secrets from confidential variables to a low security attacker via observation of the low security outputs from the execution of a program [1]. The analysis uses information theory to quantify the leakage and takes an information theoretic view: the quantity of information supplied to the program is taken to be the upper limit on the amount of information induced by the probability distribution on inputs (or entropy) as given by Shannon [4]. This analysis is based not on the syntax of the program per se but on a use definition graph (UDG) which is derived from the flow chart of the program and relates each use of a variable in an expression to its previous definition. The analysis uses this graph to trace the flow of information through the program. In the course of the analysis it is necessary to associate random variables with program variables and expressions. The analysis proceeds as a forward analysis, propagating bounds on the entropic measure of confidential information associated with the value at a program point of each variable or expression.

We have speculated that this approach could be applied to a functional language not too distant from the typed lambda calculus but containing delta rules for arithmetic, such as an enriched PCF. This would rely on using a version of the control flow analysis (CFA) based on game semantics for PCF as developed by Malacaria and Hankin [3]. To date we have yet to define a suitable notion of random variable for this CFA. This difficulty has caused us to examine anew the relationship between program variables, program points, and random variables [2] out of which has come the basis for this talk.

In this talk we examine another possible approach to quantifying interference between variables in a PCF program. The approach is quite general and could possibly be applied to any program whose semantics can be given as a function. We establish a relationship between program variables, random variables and equivalence relations and show how this relationship allows us to set an upper bound on the interference between variables at the beginning and the end of the program. We illustrate the power of the approach by means of examples. A feature of this approach is the separation between the qualitative and quantitative aspects of the analysis. The key notion is that constraints on the observation of outputs induce an equivalence relation on inputs (which may be interpreted as a random variable). This equivalence relation is the “coarsest” which is consistent with the constraints given the semantics of the program. An upper bound on the entropy of the random variable may be found by discovering any “finer” equivalence class on inputs. A way of doing this consistent with the program semantics is via an abstract interpretation.

References


Rewriting Frameworks and Types

Maribel Fernández
King’s College London

Rewriting systems are usually defined by specifying a set of terms, and a set of rewrite rules that are used to “reduce” terms. This simple idea is very powerful, and has had deep influence in the development of computational models, programming and specification languages, theorem provers and proof assistants. Term rewriting systems can be used to define, amongst others:

- first-order equational theories: terms are built out of first-order functions and variables and rewrite rules define an equality relation. We obtain in this case the standard (first-order) term rewriting systems, sometimes called algebraic systems.

- a theory of functions: Church’s λ-calculus was designed as a notation for functions, using λ-abstraction, application and variables. The main rewrite rule is β-reduction:

  \[(β) \quad (\lambda x.t)s \rightarrow t\{x \mapsto s\}\]

  The right hand side of this rule includes a substitution, which is not part of the syntax of λ-terms: it is a meta-operation defined modulo α-conversion since λ is a binder. This is an example of “higher-order” rule.

- higher-order equational theories, using first- and higher-order functions and variables: we obtain in this case higher-order algebraic systems.

- a generalization of the previous cases, where terms are defined using functions, variables and binders. These are usually called higher-order rewriting systems, and can be divided in two classes:

  - Algebraic λ-calculi: the syntax of the λ-calculus is enriched with algebraic functions, and the β-reduction rule is combined with algebraic rewrite rules. This kind of rewrite system is used as a computational model for functional programming languages and proof assistants.

  - First-order rewriting extended with a general notion of binder: Examples of systems in this class are Klop’s Combinatory Reduction Systems (CRS), Khasidashvili’s Expression Reduction Systems, Nipkow’s Higher-Order Rewrite Systems, and the ρ-calculus of Cirstea and Kirchner.

From the computational point of view all the systems mentioned above are equally powerful (they are all Turing complete), but from the point of view of their expressive power there are clear differences. The systems in the last class are the most expressive. For instance, it is easy to specify the λ-calculus, object-calculi (e.g. Abadi and Cardelli’s ζ-calculus), and process calculi (e.g. the π-calculus), using Combinatory Reduction Systems (see [7]).

Although rewriting can be defined on untyped terms, for most of the formalisms mentioned above some notion of typing has been defined. Types can be used to define semantic
interpretations for terms and rewrite rules, as specifications of programs defined by rewrite rules, as a link between computation and logic (as in the typed versions of the \( \lambda \)-calculus), etc. When combining \( \lambda \)-calculus with algebraic rewriting, type disciplines provide an environment in which rewrite rules and \( \beta \)-reduction can interact without loss of their useful properties [4, 6, 3].

Type disciplines come in two main flavours: explicitly typed systems (à la Church), and type inference systems (à la Curry) which do not require type annotations. An important feature of type systems is a notion of principal type, that is, a type from which all the other types of a term can be derived, or more generally, a notion of principal typing. The latter means that any typing judgement for a term can be obtained from the principal one, that is, not only the type but also the basis (containing the type assumptions for the free variables) is obtained. Unlike principal types, principal typings provide support for separate compilation, incremental type inference, and for accurate type error messages.

In the context of the \( \lambda \)-calculus type inference disciplines have been extensively studied, and type inference is now standard in functional languages (e.g. ML and Haskell). ML’s type system is polymorphic and has principal types, but does not have principal typings. System F provides a much more general notion of polymorphism, but lacks principal typings and type inference is undecidable in general (although it is decidable for some subsystems, in particular if we consider types of rank 2). The intersection type discipline [5] is an extension of Curry’s system that consists of allowing more than one type for variables and terms, considering the type constructor \( \cap \) in addition to the arrow. Intersection type systems have principal typings (but are also undecidable above rank 2), and can be used to characterize normalization of \( \lambda \)-terms. Adding intersection types to System F, Margaria and Zacchi obtained a very powerful type system for the \( \lambda \)-calculus, which has principal typings.

Type inference for algebraic term rewriting systems or more general higher-order rewriting systems has attracted less attention. In some cases, it is possible to translate a term rewriting system into the \( \lambda \)-calculus and use the type systems mentioned above. However, this imposes strong restrictions in the rewrite rules. For example, adding the rewrite rules defining surjective pairing to the \( \lambda \)-calculus gives a system in which the Church-Rosser property no longer holds, which implies that surjective pairing is not definable in the \( \lambda \)-calculus.

We have defined in [2] intersection type systems for Curried term rewriting systems (CTRS), which are first-order systems with application used in the language Clean. Typeable terms in CTRS need not even be head-normalizable (in contrast with the situation in the \( \lambda \)-calculus), since we can write rules such as \( t \to t \) for a typeable term \( t \). Inspired by [6], we defined a recursive scheme for rewrite rules such that typed systems are terminating, and their combination with (typed) \( \lambda \)-calculus is also terminating. We extended Margaria and Zacchi’s polymorphic intersection system to algebraic rewrite systems (combining \( \lambda \)-calculus and CTRS) in [1], and proved that the rank 2 combination of System F and the intersection system is decidable and still has the principal typing property in this context.

For more general rewriting systems, such as Combinatory Reduction Systems, type disciplines have not been investigated. Since CRS combine term rewriting and a notion of bound variables like in the \( \lambda \)-calculus, it is natural to ask whether there is a natural notion of type system for CRS which generalizes the results above. In this talk we give three alternative definitions: we start with a system based on simple types, then consider intersection types, and finally define a rank 2 system for CRS, combining polymorphism and intersection types.
References


Abstract

We consider a novel Natural Deduction rule Restart. We show that Intuitionistic Logic plus Restart gives Classical Logic. Unlike other logical extensions with the same effect Restart is a computational extension, in the sense that it does not compromise proof-normalisation and the extension automatically transforms the $\lambda$-calculus associated to Intuitionistic Logic to a novel one associated to Classical Logic. We investigate its relation to the $\lambda$-calculus.

We consider different Restart rules, each of which automatically generates a $\lambda$-calculus. Thus Restart is a way of going from logic to logic and take the proof theory with us.

We consider the geometric significance of the Restart rule: Classical Logic is sound with respect to Kripke Structures with just one world, different Restart rules correspond to this and other less drastic geometric restrictions on the Kripke Structures. We show how Restart is an intermediate step between geometric properties of Kripke Structures and corresponding $\lambda$-calculi.

1 The restart rule

The Natural Deduction restart rule is as follows:

\[
\begin{array}{c}
(\text{Restart}) \\
A \\
\hline
B \\
\end{array}
\]

That is, from any $A$ we may deduce $B$.

There is a side-condition of course: a proof that uses restart is valid if below every occurrence of restart from $A$ to $B$, there is an occurrence of $A$. Otherwise, the proof is not valid.

Thus for example $\Gamma \vdash B$ is not a valid proof. However, the following proof of Pierce’s Law in Intuitionistic Logic (IL) with Restart (ILR) is valid, as we can easily verify:

\[
\frac{[A] \quad \frac{(A \rightarrow B) \rightarrow A \quad \frac{(A \rightarrow B) \rightarrow A \rightarrow I}{A\uparrow}}{(A \rightarrow B) \rightarrow A \rightarrow E}}{A \rightarrow (A \rightarrow B) \rightarrow A \rightarrow I}
\]

We easily check this proof with its use of (Restart) is valid: the use at $\uparrow$ is justified at $\hfill$.

We immediately observe that IL plus Pierce’s law has the same deductions as Classical Logic (CL). This is one half of a theorem:

**Theorem:** Intuitionistic Logic plus (Restart) has the same deductions as Classical Logic.
There are many other ways of making a logic classical; we have already mentioned Pierce’s Law, others are Double Negation Elimination $\neg \neg A \rightarrow A$, or Excluded Middle $A \vee \neg A$.

(Restart) has interesting properties from the point of view of the Curry-Howard Correspondence. We shall see how adding the rule to a Natural Deduction system preserves proof-normalisation, and without major structural changes such as making the logic multiple conclusion. Thus (Restart) is a general method for making a programming language ‘classical’ without having to start from scratch.

$\lambda$-calculi for Classical Logic exist, notably Parigot’s $\lambda\mu$-calculus [4]. We shall extract the $\lambda$-calculus from ILR and consider the two system’s relation (very close). We shall then consider generalisations of (Restart) which give different intermediate logics between IL and CL, while always maintaining the rigorously structural nature of the extension.

(Restart), as the reader no doubt already expects from its name and its relation to existing work, is a form of exception. Different restart rules give different exceptions. We shall relate the soundness of restart rules to geometric properties of the Kripke models for particular logics, making a connection between the geometry of a logic and its exception principles.

While for certain intermediate logics $\lambda$-calculi have already been created and studied, Restart lets us consider systems and their interrelationships in a systematic fashion, and potentially transfer results between them.

Restart dates back to work by Dov Gabbay [2, 1] in the framework of goal directed deductions. (As a goal-directed rule (Restart) is read bottom-up and the side-condition is that at a goal $B$ we can ‘restart’ at any previous goal $A$ we have considered.) Michael Gabbay treated the same rule top-down in Prawitz natural deduction [3].

References


Session Types: Specifying Structured Communication

Simon Gay

Session types [5, 6, 9] enable patterns of structured communication, such as client-server protocols, to be specified type-theoretically. For example, a server which expects repeatedly to receive an integer value and respond with a boolean value implements a protocol expressed by the recursive session type $S = ?[\text{int}] . ![\text{bool}] . S$, while a matching client must respect the dual type $\overline{S} = ![\text{int}] . ?[\text{bool}] . \overline{S}$. More precisely, these are the types of the server's side and the client's side of the communication channel which links them.

Branching and termination can also be specified. A server offering a choice of mathematical operations might use a channel of the following type:

$$S = \& \langle \text{add} : ?[\text{int}] . ?[\text{int}] . ![\text{int}] . ![\text{int}] . ![\text{int}] . ![\text{bool}] . S, \text{quit} : \text{end} \rangle$$

In this case the type of the client's side of the channel indicates that a choice is made, and specifies the form of the subsequent communication:

$$\overline{S} = \& \langle \text{add} : ![\text{int}] . ![\text{int}] . ![\text{int}] . ![\text{int}] . ![\text{int}] . ![\text{bool}] . \overline{S}, \text{quit} : \text{end} \rangle$$

Session types were originally formulated for the pi calculus [7, 8], a language for defining communicating concurrent processes, but can also be defined in other computational settings [3]. If protocols are specified by means of session types then client or server implementations can be verified by static typechecking. This remains true if channels themselves can be transmitted from process to process, and this supports delegation of sessions from one process to another.

A theory of subtyping for session types [1, 2] supports a definition of compatibility between communicating components, whereby instead of requiring their communication capabilities to match exactly, it is sufficient that every message is understood by its recipient. For example, if the mathematical server is extended by the addition of a new service:

$$T = \& \langle \ldots, \text{mult} : ?[\text{int}] . ?[\text{int}] . ![\text{int}] . ![\text{int}] . T \rangle$$

then the new server type $T$ is compatible with the original client type $\overline{S}$, even though they do not match exactly.

For further flexibility, polymorphism can be introduced. A type system which has been developed in detail [4] combines polymorphism with subtyping in a notion of bounded polymorphism. For example, assuming that we have a hierarchy of numeric types, the mathematical server might be described by

$$S = \& \langle \text{add}(X <: \text{float}) : ?[X] . ?[X] . ![X] . ![X] . ![X] . ![\text{bool}] . S, \text{eq}(X <: \text{int}) : ?[X] . ?[X] . ![X] . ![\text{bool}] . S, \text{quit} : \text{end} \rangle$$

where $X$ is a type parameter which must be instantiated, respecting the specified upper bound, when a service is chosen.
References


1 Classical Abstract Interpretation

We start by sketching the classical approach to semantics-based program analysis: *abstract interpretation*. The *semantics* of a program $p$ identifies some set $V$ of values and specifies how the program transforms one value $v_1$ to another $v_2$: $p \vdash v_1 \rightarrow v_2$. In a similar way, a *program analysis* identifies the set $L$ of properties and specifies how a program $p$ transforms one property $l_1$ to another $l_2$: $p \vdash l_1 \triangleright l_2$. Every program analysis should be correct with respect to the semantics. For first-order program analyses this is established by directly relating properties to values using a *correctness relation*: $R : V \times L \rightarrow \{ \text{true, false} \}$. The most common scenario in abstract interpretation is when both $V$ and $L$ are complete lattices. The correctness relation is then often achieved via a *Galois connection*: $(V, \alpha, \gamma, L)$ is a Galois connection between the complete lattices $(V, \sqsubseteq)$ and $(L, \sqsubseteq)$ if and only if $\alpha : V \rightarrow L$ and $\gamma : L \rightarrow V$ are monotone functions that satisfy:

$$\gamma \circ \alpha \trianglerighteq \lambda v. v \text{ and } \alpha \circ \gamma \trianglerighteq \lambda l. l$$

Having defined a suitable “set” of properties we then define suitable interpretations of program operations $(\alpha \circ F \circ \gamma)$ and construct efficient implementations (widenings).

2 Strictness Analysis

Strictness analysis aims to answer for some function $f$: Does $f \bot = \bot$? An affirmative answer would mean that arguments can be passed by value rather than using (the more costly) lazy evaluation. We will restrict ourselves to a first order functional language with integers as the only data type.

We can construct a Galois connection $(P_H(Z), \alpha, \gamma, \text{Two})$ where $P_H$ is the *Hoare Powerdomain* construction and $\text{Two}$ is $\{0, 1\}$ ordered by $0 \sqsubseteq 1$. The elements of the Hoare Powerdomain in this case are just down-closed sets ordered by subset inclusion.

We define:

$$\alpha(Z) = \begin{cases} 0 & \text{if } Z = \bot \\ 1 & \text{otherwise} \end{cases}, \quad \gamma(S) = \begin{cases} \{\bot\} & \text{if } S = 0 \\ Z & \text{if } S = 1 \end{cases}$$

We can construct the induced operations that correspond to the operations in the language:

<table>
<thead>
<tr>
<th>Concrete operation</th>
<th>Induced operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>1</td>
</tr>
<tr>
<td>if $x$ then $y$ else $z$</td>
<td>$x \sqcap (y \sqcup z)$</td>
</tr>
<tr>
<td>$x \text{ op } y$</td>
<td>$x \sqcap y$</td>
</tr>
</tbody>
</table>
Thus the abstract interpretation of $(\lambda \; x. \; \text{if} \; x = 0 \; \text{then} \; 15 \; \text{else} \; 42)$ is $(\lambda \; x. (x \cap 1) \cap (1 \cup 1)) \equiv \lambda x. x$. Since $(\lambda \; x. \; x) \; 0 = 0$ this tells us that our original function is strict. We now extend this approach to a probabilistic λ-calculus with the usual terms and $e_1 \oplus p \; e_2$ (indicating a probabilistic choice, the right hand summand being chosen with probability it p).

3 Linear Representations

The semantics of terms in our extended calculus can be represented by a probabilistic reduction graph where edges are labelled with probabilities. We associate to each quantitative relation $R \subseteq X \times W \times X$ a matrix, i.e. a linear operator $M_R$ on $\mathcal{V}(X)$ defined by:

$$(M_R)_{ij} = \begin{cases} w & \text{iff } \Sigma_{R(x,w',y)} w' = w \\ 0 & \text{otherwise} \end{cases}$$

For probabilistic relations this gives a Stochastic matrix.

4 Probabilistic Abstract Interpretation

Given two probabilistic domains, $\mathcal{C}$ and $\mathcal{D}$, a probabilistic abstract interpretation is a pair of linear maps, $\mathbf{A} : \mathcal{C} \rightarrow \mathcal{D}$ and $\mathbf{G} : \mathcal{D} \rightarrow \mathcal{C}$, between the concrete domain $\mathcal{C}$ and the abstract domain $\mathcal{D}$, such that $\mathbf{G}$ is the Moore-Penrose pseudo-inverse of $\mathbf{A}$, and vice versa. Let $\mathcal{C}$ and $\mathcal{D}$ be two Hilbert spaces and $\mathbf{A} : \mathcal{C} \rightarrow \mathcal{D}$ a bounded linear map between them. A bounded linear map $\mathbf{A}^\dagger = \mathbf{G} : \mathcal{D} \rightarrow \mathcal{C}$ is the Moore-Penrose pseudo-inverse of $\mathbf{A}$ iff

$$\mathbf{A} \circ \mathbf{G} = \mathbf{P}_A \quad \text{and} \quad \mathbf{G} \circ \mathbf{A} = \mathbf{P}_G$$

where $\mathbf{P}_A$ and $\mathbf{P}_G$ denote orthogonal projections onto the ranges of $\mathbf{A}$ and $\mathbf{G}$.

Alternatively, if $\mathbf{A}$ is Moore-Penrose invertible, its Moore-Penrose pseudoinverse, $\mathbf{A}^\dagger$ satisfies the following: $\mathbf{A}^\dagger \mathbf{A} = \mathbf{A}$, $\mathbf{A}^\dagger \mathbf{A} \mathbf{A}^\dagger = \mathbf{A}^\dagger$, $(\mathbf{A}^\dagger)^* = \mathbf{A} \mathbf{A}^\dagger$, $(\mathbf{A}^\dagger \mathbf{A})^* = \mathbf{A}^\dagger \mathbf{A}$. It is instructive to compare these equations with the classical setting. For example, if $(\alpha, \gamma)$ is a Galois insertion: $\alpha \circ \gamma \circ \alpha = \alpha$ and $\gamma \circ \alpha \circ \gamma = \gamma$.

A simple method to construct a probabilistic abstract interpretation is as follows: Given a linear operator $\Phi$ on some vector space $\mathcal{V}$ expressing the probabilistic semantics of a concrete system, and a linear abstraction function $\mathbf{A} : \mathcal{V} \rightarrow \mathcal{W}$ from the concrete domain into an abstract domain $\mathcal{W}$, we compute the (unique) Moore-Penrose pseudo-inverse $\mathbf{G} = \mathbf{A}^\dagger$ of $\mathbf{A}$. The abstract semantics can then be defined as the linear operator on the abstract domain $\mathcal{W}$:

$$\Psi = \mathbf{A} \circ \Phi \circ \mathbf{G}.$$ 

5 Probabilistic Strictness Analysis

Consider the term $(\lambda x.0 \oplus_\frac{1}{2} x)(\bot \oplus_\frac{1}{2} 42)$. By considering its reduction graph, we can construct the following enumeration of terms: $(\lambda x.0 \oplus_\frac{1}{2} x)(\bot \oplus_\frac{1}{2} 42)$, $(\lambda x.0 \oplus_\frac{1}{2} x)\bot$, $(\lambda x.0 \oplus_\frac{1}{2} x)42$, 0,
The semantics of the term can be represented by the following operator:

\[ A = \begin{pmatrix}
0 & \frac{2}{3} & \frac{1}{2} & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{3} & \frac{1}{2} & 0 \\
0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix} \]

A suitable abstraction classifies terms as undefined, don’t know or defined. This abstraction is achieved by a classification operator – a stochastic matrix with a 1 in each row. A suitable classification matrix for this example is

\[ K = \begin{pmatrix}
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 0 & 1
\end{pmatrix} \]

which has Moore-Penrose pseudoinverse \[ K^+ = \begin{pmatrix}
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2}
\end{pmatrix} \]. The abstract semantics of our original program is

\[ K^+A = \begin{pmatrix}
1 & 0 & 0 \\
\frac{1}{6} & \frac{1}{3} & \frac{1}{2}
\end{pmatrix} \]. The middle row and column represent the don’t know value. Iterating this abstract operator causes the probability of a transition from don’t know to don’t know to decrease rapidly. Achieving a defined outcome becomes more and more likely. This result could be used to support the decision to speculatively evaluate the argument.
Reduction in the lambda calculus

Ian Mackie

Department of Computer Science
King’s College London

1 Introduction

Lamping [4] presented one of the first algorithms for implementing Lévy’s notion [5] of optimal reduction in the λ-calculus. This algorithm, with the help of linear logic [2], was tidied up and lead to the well-known algorithm of Gonthier, Abadi and Lévy [3]. It soon became apparent that this algorithm contained several inefficiencies (specifically accumulation of certain nodes in the graph rewriting formalism, or bookkeeping). Asperti and others [1] devised BOHM (Bologna Optimal Higher-Order Machine) to overcome some of these, and stood until now as not only the most efficient (in terms of graph rewriting steps) implementation of optimal reduction, but also the most efficient implementation of the λ-calculus. Although BOHM remains the standard benchmark for optimal reduction, it is complex, and generally not well understood or used outside a small community.

A parallel thread of work takes interaction nets (a graph rewriting framework) as a starting point. The key observation here is that the previously mentioned implementations of the λ-calculus, a β-reduction is just another graph rewrite. The aim therefore is to find an alternative solution to optimal reduction where we aim to find the minimum number of rewrite steps (β included). This is therefore a more pragmatic approach to a notion of optimal reduction.

Historically, this notion of “practical” optimality began in [6], based on an interaction net encoding of linear logic (due to Abramsky). Although this first λ-evaluator based on interaction nets performed fewer interactions (rewrite steps) for specific λ-terms than Lamping’s algorithm, it was never a match for BOHM. A further attempt, YALE [7], provided a substantial improvement which could systematically perform better than Lamping’s algorithm, and approximate BOHM on specific classes of terms. Nevertheless, when the need for optimality kicks in, YALE is a very poor second best.

A question therefore remained: is there an efficient interaction net implementation of the λ-calculus which does less work than BOHM? The purpose of the present paper is to answer this question in the positive. Specifically, we give a new λ-evaluator, which we call the King’s College Lambda Evaluator (KCLE), which has the following features:

- Evaluation to full normal form, even for open terms.
- Although not optimal in the sense of Lévy (KCLE performs more β-reduction steps), the overall number of graph rewrite steps is smaller, offering asymptotically better performance.
- It is based on interaction nets, and thus can take advantage of the numerous results and implementations — specifically parallel — where almost linear speedup has been achieved.
References


Many fields in computer science make use of a notion of variable and in particular of free or bound variable. Such fields include rewriting, $\lambda$-calculus, natural language processing, etc. In particular, the representation of sets of free variables is crucial for the efficiency of the implementation of the process of substitution. For instance, to avoid useless substitutions, we may want to define substitution with conditional rewrite rules similar to:

$$f(t_1, \ldots, t_i, \ldots, t_n)\{x \leftarrow u\} \rightarrow f(t_1, \ldots, t_i\{x \leftarrow u\}, \ldots, t_n)$$

if $x \in \text{fv}(t_i)$ and $x \notin \text{fv}(t_j)$ for $j \neq i$

However, it is not satisfactory to implement these tests in the naive way. We thus provide an alternative representation of free variables (director maps) which has the following advantages:

- it allows to direct substitutions efficiently only to where they are needed;
- the representation can be preprocessed efficiently;
- it can be maintained efficiently by the reduction;
- it is a natural alternative to the usual representation in the sense that, instead of the set of free variables for each subterm, we have information about the subterms which contain each free variable.

Moreover, we give an algebraic characterization of director maps and define several notions of rewrite systems on terms with director maps. These systems have to respect certain conditions with respect to director maps in order to be valid, and we make these conditions explicit.

We then show how the work done in [1] for pure $\lambda$-calculus fits in that formalism. We thus provide a generalization of the director strings of [1].

As an example of application, we show how (type-erased) terms of the Calculus of Inductive Constructions also fit in that formalism.

References

Underspecified Representations for Natural Language using Curry-typed $\lambda$-Calculus

Chris Fox  
University of Essex  
foxcj@essex.ac.uk

Shalom Lappin  
King’s College London  
lappin@dcs.kcl.ac.uk

1 Underspecification

Underspecification has been proposed as a means of dealing with scoping problems in natural language (Reyle, 1993; Bos, 1995; Bos et al., 1996; Blackburn & Bos, 2003; Copestake, Flickinger, & Sag, 1997).

This problem is exemplified by “Every woman loves a man”, where the scoping of the quantifiers is ambiguous: it can be represented as either $\forall x (\text{woman}(x) \rightarrow \exists y (\text{man}(y) \land \text{loves}(x, y)))$, or $\exists y (\text{man}(y) \land \forall x (\text{woman}(x) \land \text{loves}(x, y)))$. The issue of ambiguous scope also occurs with negation, modifier expressions and conjunctive phrases (e.g. “the red ball and balloon”). In general, we need to be able to produce all felicitous scopings, or produce an underspecified representation that can be “expanded out” to any such reading.

Existing treatments of the scoping problem and underspecification require additional machinery to be added to the semantic representation, for example, a store in Cooper storage (Cooper, 1983), or some means of keeping track of indices in Montague’s approach (Montague, 1974), and more generally a notion of labels and place-holders, together with machinery for manipulating them, in the case of other theories of underspecification (e.g. Bos (1995)).

Property Theory with Curry Typing (PTCT) is a first-order logic with Curry typing (Fox, Lappin, & Pollard, 2002a, 2002b). It has been developed with the explicit purpose of producing a first-order theory with the expressiveness required by natural language. As PTCT incorporates the $\lambda$-calculus, which can model computable functions, we are able to produce underspecified representations directly within the semantic language, and so we do not require any meta-theoretic machinery.

2 Property Theory with Curry Typing (PTCT)

PTCT consists of the untyped $\lambda$-calculus, together with a language of types, and a first-order language of wffs, which takes the role of a meta-theory. Rules in the language of wffs allow the $\lambda$-calculus to be Curry-typed.

One of the types is that of “proposition.” There are rules in the language of wffs that characterise what it is to be a proposition, and there is a truth predicate that allows us to consider the truth conditions of well formed propositions.

PTCT allows us to capture the distinction between extensional equivalence (given by the language of wffs) and intensional identity (given by term equality in the $\lambda$-calculus).

The types include general function space types, polymorphic types (implicit) and separation types. To remain first-order, all types are term representable. This requires that there
be constraints on the nature of wffs that can appear in separation types. To this end, a term representable fragment of wffs is defined. The polymorphic types allow us to consider types for conjunction and disjunction that allow them to combine arbitrary categories (II X ⇒ X ⇒ X). They may also be used for type general predicates. Separation types allow us to produce a type theoretic analysis of anaphora and ellipsis (Fox & Lappin, 2003a, 2003).

3 Underspecification in PTCT

We build upon an existing proposal (van Eijck, 2003). Simplifying, and abstracting from the details, we have a function \texttt{perms} (Campbell, To Appear) that takes a \(k\)-ary product (rather than a list) of quantifiers (representing the noun phrases) together with a \(k\)-place relation (representing the verb) and produces a \(k!\)-ary product, consisting of the propositions that result from all permutations of the quantifiers (in parallel with the appropriate permutations of the argument positions of the verb). This \(k!\)-ary product is an underspecified representation of all scopings of the sentence. A particular reading can be produced by projecting an element from the output of the permutation function.

As an alternative, we can define a function \texttt{permute}_{k}^{i} (1 \leq i \leq k!) corresponding to the computation of the \(i\)th permutation of \(k\) quantifiers (and argument positions of the verb). An underspecified representation would then be a \(\lambda\)-abstract awaiting an appropriate value of \(i\) in order to form a proposition representing the \(i\)th reading of the sentence.

We could consider imposing constraints on the permutation operator which would exclude certain permutations if we wanted to encode empirically motivated scope constraints. Most theories of underspecified semantic representations do not permit the formulation of scoping constraints in such a straightforward manner (Ebert, 2003).

4 Polymorphism

The polymorphism of the theory allows us to consider quantifier scoping, and underspecification, when the quantifiers are of different types. This could arise in the case where a verb takes an argument that is an individual, and an argument that is a proposition, such as \texttt{believe} for example.

As PTCT stands, there is an issue concerning from where the appropriate types come, as in both cases the relevant typing is given by the determiner (“a”, “every”), which is normally taken to be fixed. One possibility would be to adopt a version of \textit{explicit} polymorphism, so that the type of the determiner could itself be determined by the type of the noun, for example.

5 Conclusions

PTCT was devised to produce a weak but expressive theory for natural language. Initially it has been applied to the problem of intensionality, anaphora and ellipsis. It seems to provide a good vehicle for exploring underspecified representations, by adapting van Eijck’s proposed treatment for quantifier scoping to PTCT.
The proposal raises fundamental questions, such as which form of polymorphism is most appropriate for natural language semantics, and whether we are obliged to extend the type system of PTCT.

References


1 Introduction

λ-abstracts are commonly used to model the meaning of interrogatives in NLP and AI work (for a recent example see Larsson, 2002). Such a modelling was quite popular among NL formal semanticists in the late 1970s and early 1980s (see e.g. Hausser, 1983), but is, generally, avoided in contemporary work, given intrinsic problems discussed in Groenendijk & Stokhof, 1997. Ginzburg and Sag 2000 argued that a theory of questions as *propositional abstracts* has a number of desirable consequences and, in particular, possesses significant advantages over the standard view of questions as encoders of exhaustive answerhood conditions. Moreover, they showed how in a situation theoretic ontology, modelled using tools from non-well-founded set theory (see Seligman & Moss, 1997), the problems for an abstract-based view could be overcome. Nonetheless, Ginzburg and Sag’s account, involves use of an *ad hoc* theory of λ-abstraction. The question arises—can the benefits of this account be preserved using standard notions of abstraction? In this paper, I will consider this question and offer a guarded affirmative response, using a version of Martin-Löf Type Theory (MLTT) (see e.g. Betarte & Tasistro, 1998; Coquand, Pollack, & Takeyama, 2003; Cooper, 2003).

2 Questions as (Propositional) Abstracts: pros and cons

The original motivation for identifying the meaning of (wh-)interrogatives with λ-abstracts comes from the phenomena of *short answers*—the nature of the question asked strongly influences the semantic type, and to some extent, the form of the short answer:

<table>
<thead>
<tr>
<th>Question</th>
<th>Short Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. A: Who left? B: Mo/No students./A friend of Jo’s.</td>
<td></td>
</tr>
<tr>
<td>b. A: When did Bo leave? B: Yesterday./At two.</td>
<td></td>
</tr>
<tr>
<td>c. A: Why did Maire cross the road? B: Because she thought no cars were passing.</td>
<td></td>
</tr>
</tbody>
</table>

1 A variant on this strategy is to use open sentences. This avoids some of the problems a simple-minded view of questions as abstracts runs into, but acquires even more serious problems along the way. See Ginzburg & Sag, 2000 for discussion.
d. A: Who relies on whom (in this department)?
   B: Bo on Mo./Some of my friends on each of her friends.

On a *questions as abstracts* view, we get the following basic denotations:

\( (2) \)

\[ a. \text{Who left} \rightarrow \lambda x.\text{leave}(x) \]

\[ b. \text{When did Bo leave} \rightarrow \lambda t.\text{At}(t, \text{leave}(b)) \]

\[ c. \text{Why did Maire cross the road} \rightarrow \lambda c.\text{Cause}(c, \text{cross}(m, r)) \]

\[ d. \text{Who relies on whom} \rightarrow \lambda x, y.\text{rely}(x, y). \]

Within many versions of typed \( \lambda \)-calculus, such as Montague’s Intensional Logic (Montague, 1974), this leads to several problematic consequences. These include:

- Type multiplicity of interrogatives (e.g. the types of (2a) and (2d) are unavoidably distinct). This yields direct difficulties in defining Boolean operations on interrogatives.

- Propositional abstracts are akin to properties, the denotata of (intransitive) verbs, adjectives, and common nouns, and so we are required to identify questions with the denotata of verbs, common nouns, and adjectives. This is an unpalatable ontological consequence.

- What to do with polar interrogatives, which also exhibit a phenomenon akin to short answers?

\( (3) \)

\[ A: \text{Did Bo attend the meeting?} \]

\[ B: \text{Yes./Maybe./Probably/No.} \]

### 3 Questions in a Situational Universe with Abstract Entities

Ginzburg and Sag 2000 provide a model theory for situation theory, based on the non-well-founded set theory-based framework developed in Seligman & Moss, 1997. Within this ontological setting, a notion of abstraction is presented whereby the universe is closed under abstraction over sets of entities. This allows the definition of a uniform type for interrogatives. Polar interrogatives are accommodated by setting up a notion of abstraction in which \( \eta \)-reduction fails and allowing for abstraction over empty sets. Moreover, given that the ontology distinguishes various kinds of ‘informational entities’ (including states-of-affairs (SOAs) and propositions), one can also avoid in a principled way the identification of questions with properties.

Additional benefits that follow in this setting include:

- **Answerhood:** Abstracts can be used to *underspecify* answerhood. This is important given that NL requires a variety of answerhood notions, not merely exhaustive answerhood.
Grain: in this setting one can distinguish the denotations of positive and negative polar questions, as in (4), while deriving the identity of their answerhood relations. This linguistically desirable consequence contrasts with approaches which characterize questions in terms of exhaustive answerhood conditions (e.g. Groenendijk & Stokhof, 1997), where the denotations of such questions are necessarily identical:

(4) a. Did Bo leave?
   b. Didn’t Bo leave?

- Which polyadicity: a straightforward account of the polyadic phenomena exhibited by which-interrogatives.

4 Redoing Questions in MLTT

The account of Ginzburg & Sag, 2000 relies on a non-standard (read: ad hoc) notion of abstraction. MLTT (e.g. as developed in the references above) offers a promising framework for recasting this account. I will show how the MLTT notion of a family over a type simulates simultaneous (including vacuous) abstraction with restrictions such that all the (arguable) benefits enumerated in the previous section can be maintained. The main conceptual difficulty is ontological: although one can code in MLTT the SOA/proposition distinction, it does not live there very naturally.

5 Future applications

Something like the MLTT theory of abstraction is required in order to formulate meaning coercion operations exploited in an account of partial understanding in dialogue. Such operations take as input a linguistic sign and yield as output a clarification question (or a context for asking such a question). Initial work on this problem was carried out in Cooper & Ginzburg, 2002.

References


Cooper, R. 2003. Records and dialogue.. Göteborg Universitet Ms.


Polymorphism in Specification

Ray Turner

University of Essex

turnr@essex.ac.uk

By a logical specification languages (Z, VDM, B, PVS) we mean one based upon the Predicate Calculus in the sense that the language in which the specifications are expressed is some version or dialect of PC. These differ from the simple single sorted versions of PC in that they distinguish between different types of data where the types of these theories are usually presented in an inductive fashion: there are basic types together with a battery of type constructors such as products, sets and recursive types.

There has been some attention given to the logical foundations of these languages. However, most if not all this work has been aimed at specific languages; there has been little work on the logical foundations of the subject as a whole. Subsequently, we have few mathematical studies of the relationships between different styles of specification or indeed different specification languages. Furthermore, the interaction of computability and specification has not been the subject of much study. Much the same can be said for type inference systems for specification languages. Finally there has been very little work done of the metamathematical properties of specification languages as logical theories where the theories become objects of study in their own right. The overall objective of this research is to address these issues and to put logical specification and specification languages on a firmer mathematical footing and thereby open up the area for more systematic mathematical investigation.

In this paper we shall concentrate on the mathematical underpinnings of polymorphism in specification — both for relational specifications, which are typified by Z, and functional specifications, which form a central feature of VDMSL. For both we shall investigate two different styles of polymorphism i.e. Implicit and Explicit varieties. For example, explicit relational specifications take the form

\[ R[X_1, \ldots, X_n] = [x_1 : T_1, \ldots, x_n : T_n] \phi \]

I.e. the new relation takes type arguments. In contrast, implicit specifications do not take type arguments i.e.

\[ R = [x_1 : T_1, \ldots, x_n : T_n] \phi \]

Where \( T_1, \ldots, T_n \) are type terms, \( X_1, \ldots, X_n \) type variables and \( \phi \) is a wff. This distinction parallels the different notions of polymorphism in the Lambda calculus and functional programming. This investigation will be carried out within the confines of a core theory of types which forms the heart of most logical specification languages. Particular attention will be given to the conditions under which such specifications result in conservative additions to the base theory.
Records and record types in semantic theory*

Robin Cooper
Göteborg University

I will explore possibilities for formulating linguistic semantics in terms of records and record types of the kind used in recent developments of Martin-Löf type theory (Betarte, 1998, Betarte and Tasistro, 1998, Coquand, Pollock and Takeyama, 2003, Tasistro, 1997). I will suggest that they give us the tools to develop a theory which includes aspects of Montague semantics, using the lambda calculus\(^1\), Discourse Representation Theory (DRT)\(^2\), situation semantics\(^3\) and Head-Driven Phrase Structure Grammar (HPSG)\(^4\) in a single theory. I will also argue that formulating these theories in terms of record types may provide us not only with a unified approach but also with certain improvements over the individual theories.

The ingredients from Martin-Löf type theory that I will use, in addition to records and record types can be summarised as:

- dependent types (including dependent function types)
- the notion of “propositions” as types of proofs
- the first class status of types as objects

The approach I am advocating has the following features:

- compositionality using the \(\lambda\)-calculus (from Montague semantics)
- dynamic binding (from DRT)
- a treatment of intensionality including perception complements and intensional verbs like \textit{seek} (from situation semantics)
- a treatment of context dependence including resource situations (from situation semantics)

---

*This work was supported by Vetenskapsrådet project number 2002-4879 Records, types and computational dialogue semantics, \url{http://www.ling.gu.se/cooper/records/}. I am grateful to Thierry Coquand, Jonathan Ginzburg, Fritz Hamm, Staffan Larsson, Uwe Mönich, Yiannis Moschovakis, Bengt Nordström and Aarne Ranta for helpful comments.

\(^1\)Montague (1974) is the classic reference.
\(^2\)Kamp and Reyle (1993), van Eijck and Kamp (1997) and much other literature.
\(^3\)Barwise and Perry (1983) and subsequent literature.
\(^4\)For a recent account see Ginzburg and Sag (2000).
• a sign-based approach to the relation between syntax and semantics (from HPSG)

Some of these features have been combined in other approaches, but to my knowledge they have not all been combined before.

References


