Pre-proceedings of the Workshop on Coalgebra, Horn Clause Logic Programming and Types (CoALP-Ty'16)

Edinburgh,UK

November 28–29, 2016

Contents

Organising Committee

Workshop Chairs

Ekaterina Komendantskaya, Heriot-Watt University, UK

John Power, University of Bath, UK

Publicity Chair

Frantiek Farka, University of Dundee, UK and University of St Andrews, UK

Programme Committee

Ki Yung Ahn, Nanyang Technological University, Singapore

Davide Ancona, University of Genoa, Italy

Filippo Bonchi, CNRS, ENS de Lyon, France

Iavor Diatchki, Galois, Inc, USA

Peng Fu, Heriot-Watt University, Edinburgh, UK

Neil Ghani, University of Strathclyde, UK

Patricia Johann, Appalachian State University, USA

Ekaterina Komendantskaya, Heriot-Watt University, Edinburgh, UK

Clemens Kupke, University of Strathclyde, UK

J. Garrett Morris, University of Edinburgh, UK

Fredrik Nordvall Forsberg, University of Strathclyde, UK

John Power, University of Bath, UK

Claudio Russo, Microsoft Research Cambridge, UK

Martin Schmidt, DHBW Stuttgart and Osnabrck University , Germany

Stephan Schulz, DHBW Stuttgart, Germany

Aaron Stump, The University of Iowa, USA

Niki Vazou, University of California, San Diego, USA

Joe Wells, Heriot-Watt University, Edinburgh, UK

Fabio Zanasi, Radboud University of Nijmegen, The Netherlands

Foreword

Welcome to the Workshop on Coalgebra, Horn Clause Logic Programming and Types and to Edinburgh! The workshop marks the end of the EPSRC Grant *Coalgebraic Logic Programming for Type Inference*, by K. Komendantskaya, Heriot-Watt University and J. Power, University of Bath.

The workshop consist of two parts:

1. *Semantics:* Lawvere theories and Coalgebra in Logic and Functional Programming
2. *Programming languages:* Horn Clause Logic for Type Inference in Functional Languages and Beyond

Over recent years, there has been considerable research on the semantics of operational aspects of logic programming. The underlying mathematics has often involved coalgebra on a presheaf category or a variant of a presheaf category, with index given by the Lawvere theory generated by a signature. That has much in common with many years of research on the semantics of functional programming.

The combination of coalgebra with Lawvere theories has already led to applied development, and there seems good reason to hope that that will continue, with application to logic and functional programming both separately and in combination with each other. So we would like to spend a few days bringing researchers in the area together to compare the techniques they are developing and the problems they are addressing.

Our semantic investigations led to analysis of theorem proving, and that was reciprocated, with theorem proving further influencing our semantics. Theorem proving in turn led us to study type inference, leading to the central applied focus of our work, thus the second topic of the workshop.

Extended abstracts of all accepted contributions appear in this book and on the workshop webpage. We have six invited and seven regular talks, with topics ranging from Categorical Logic, Semantics, Type Theory, Proof Theory, Programming Languages and Type Inference.

We are very fortunate to have six excellent invited speakers in

- John Power from university of Bath with a talk on *Categorical Semantics of Logic Programming: Laxness and Saturation*,
- Steven Ramsay and Luke Ong from Oxford University with a talk on *Refinement Types and Higher-Order Constrained Horn Clauses*,
- Tarmo Uustalu from Tallinn University with a talk on *Comodels and Interaction*,
- Claudio Russo from Microsoft Research Cambridge with a talk on *Type Classes for the Masses*,
- Davide Ancona from University of Genova with a talk on *Abstract Compilation for Type Analysis of Object-Oriented Languages*, and
- Ki Yung Ahn from Nanyang Technological University with a talk on *Relational Specification of Type Systems Using Logic Programming*

We are grateful to them for agreeing to talk at the workshop.

To further facilitate the discussion between the audience and the invited speakers, the invited talks will be followed by a 45 minute panel session "Questions and Answers with Invited Speakers".

Many thanks to all who helped to organise the event! In particular we are grateful to Morag Jones, June Maxwell, Iain McCrone and Christine McBride for technical and administrative support. Thanks to International Center for Mathematical Sciences for providing an excellent venue for the workshop. We are grateful to EPSRC (grants EP/K031864/1-2, EP/K028243/1) for supporting this research and this workshop. Finally, we are very grateful to the Programme Committee, and, in particular, to all those who supported the workshop by submitting abstracts and attending the meeting. We hope that you all enjoy the workshop.

Katya Komendantskaya and František Farka,
Edinburgh, 28 November 2016

Category Theoretic Semantics for Logic Programming: Laxness and Saturation

John Power
Department of Computer Science
University of Bath
Bath, BA2 7AY, UK*
A.J.Power@bath.ac.uk

1 Summary

Recent research on category theoretic semantics of logic programming has focused on two ideas: lax semantics [3] and saturated semantics [1]. Until now, the two ideas have been presented as alternatives, but that competition is illusory, the two ideas being two views of a single, elegant body of theory, reflecting different but complementary aspects of logic programming.

Given a set of atoms At, one can identify a variable-free logic program P built over At with a $P_f P_f$-coalgebra structure on At, where P_f is the finite powerset functor on Set: each atom is the head of finitely many clauses in P, and the body of each clause contains finitely many atoms. If $C(P_f P_f)$ is the cofree comonad on $P_f P_f$, then, given a logic program P qua $P_f P_f$-coalgebra, the corresponding $C(P_f P_f)$-coalgebra structure characterises the and-or derivation trees generated by P.

Extending this to arbitrary programs, given a signature Σ of function symbols, let $\mathscr{L}_\Sigma$ denote the Lawvere theory generated by Σ, and, given a logic program P with function symbols in Σ, consider the functor category $[\mathscr{L}_\Sigma^{op}, Set]$, extending the set At of atoms in a variable-free logic program to the functor from $\mathscr{L}_\Sigma^{op}$ to Set sending a natural number n to the set $At(n)$ of atomic formulae with at most n variables generated by the function symbols in Σ and the predicate symbols in P. We would like to model P by a $[\mathscr{L}_\Sigma^{op}, P_f P_f]$-coalgebra $p : At \longrightarrow P_f P_f At$ that, at n, takes an atomic formula $A(x_1, \ldots, x_n)$ with at most n variables, considers all substitutions of clauses in P into clauses with variables among $x_1, \ldots, x_n$ whose head agrees with $A(x_1, \ldots, x_n)$, and gives the set of sets of atomic formulae in antecedents. However, that does not work for two reasons. The first may be illustrated as follows.

Example 1 *ListNat (for lists of natural numbers) denotes the logic program*

1. $\mathtt{nat(0)} \leftarrow$
2. $\mathtt{nat(s(x))} \leftarrow \mathtt{nat(x)}$
3. $\mathtt{list(nil)} \leftarrow$
4. $\mathtt{list(cons(x,y))} \leftarrow \mathtt{nat(x)}, \mathtt{list(y)}$

ListNat has nullary function symbols 0 *and* nil. *So there is a map in* $\mathscr{L}_\Sigma$ *of the form* $0 \to 1$ *that models the symbol* 0. *Naturality of* $p : At \longrightarrow P_f P_f At$ *in* $[\mathscr{L}_\Sigma^{op}, Set]$ *would yield commutativity of the diagram*

$$\begin{array}{ccc} At(1) & \xrightarrow{p_1} & P_f P_f At(1) \\ {\scriptstyle At(0)}\downarrow & & \downarrow{\scriptstyle P_f P_f At(0)} \\ At(0) & \xrightarrow{p_0} & P_f P_f At(0) \end{array}$$

*John Power would like to acknowledge the support of EPSRC grant EP/K028243/1 and Royal Society grant IE151369. No data was generated in the course of this project.

Submitted to:
CoALPTy 2016

But consider $\mathtt{nat(x)} \in At(1)$*: there is no clause of the form* $\mathtt{nat(x)} \leftarrow$ *in ListNat, so commutativity of the diagram would imply that there cannot be a clause in ListNat of the form* $\mathtt{nat(0)} \leftarrow$ *either, but in fact there is one. Thus p is not a map in the functor category* $[\mathscr{L}_\Sigma^{op}, Set]$.

Lax semantics addresses this by relaxing the naturality condition on p to a subset condition, so that, given, for instance, a map in $\mathscr{L}_\Sigma$ of the form $f : 0 \to 1$, the diagram need not commute, but rather the composite via $P_f P_f At(1)$ need only yield a subset of that via $At(0)$. In contrast, saturation semantics works as follows. Regarding $ob(\mathscr{L}_\Sigma)$, equally $ob(\mathscr{L}_\Sigma)^{op}$, as a discrete category with inclusion functor $I : ob(\mathscr{L}_\Sigma) \longrightarrow \mathscr{L}_\Sigma$, the functor

$$[I, Set] : [\mathscr{L}_\Sigma^{op}, Set] \longrightarrow [ob(\mathscr{L}_\Sigma)^{op}, Set]$$

that sends $H : \mathscr{L}_\Sigma^{op} \longrightarrow Set$ to the composite $HI : ob(\mathscr{L}_\Sigma)^{op} \longrightarrow Set$ has a right adjoint R, given by right Kan extension. So the data for $p : At \longrightarrow P_f P_f At$ may be seen as a map in $[ob(\mathscr{L}_\Sigma)^{op}, Set]$, which, by the adjointness, corresponds to a map $\bar{p} : At \longrightarrow R(P_f P_f AtI)$ in $[\mathscr{L}_\Sigma^{op}, Set]$, yielding saturation semantics. In this talk, we show that the two approaches can elegantly be unified, the relationship corresponding to the relationship between theorem proving and proof search in logic programming.

The second problem mentioned above is about *existential* variables, which we now illustrate.

Example 2 *GC (for graph connectivity) denotes the logic program*
1. $\mathtt{connected(x,x)} \leftarrow$
2. $\mathtt{connected(x,y)} \leftarrow \mathtt{edge(x,z)}, \mathtt{connected(z,y)}$

There is a variable z in the tail of the second clause of GC that does not appear in its head. Such a variable is called an existential variable, the presence of which challenges the algorithmic significance of lax semantics. In describing the putative coalgebra $p : At \longrightarrow P_f P_f At$ just before Example 1, we referred to *all* substitutions of clauses in P into clauses with variables among $x_1, \ldots, x_n$ whose head agrees with $A(x_1, \ldots, x_n)$. If there are no existential variables, that amounts to term-matching, which is algorithmically efficient; but if existential variables do appear, the mere presence of a unary function symbol generates an infinity of such substitutions, creating algorithmic difficulty, which, when first introducing lax semantics, we, also Bonchi and Zanasi, avoided modelling by replacing the outer instance of P_f by P_c, thus allowing for countably many choices. Such infiniteness militates against algorithmic efficiency, and we resolve it by refining the functor $P_f P_f$ while retaining finiteness.

This talk is based upon the paper [2].

References

[1] Filippo Bonchi & Fabio Zanasi (2015): *Bialgebraic Semantics for Logic Programming*. *Logical Methods in Computer Science* 11(1).

[2] Ekaterina Komendantskaya & John Power (2016): *Logic programming: laxness and saturation*. *Submitted, CoRR* abs/1608.07708. Available at `http://arxiv.org/abs/1608.07708`.

[3] Ekaterina Komendantskaya, John Power & Martin Schmidt (2016): *Coalgebraic logic programming: from Semantics to Implementation*. *J. Log. Comput.* 26(2), pp. 745 – 783.

Coalgebraic Logic Programming: Implementation and Optimization

Martin Schmidt
Institute of Cognitive Science, University of Osnabrück, Germany
martisch@uos.de

Logic programming (LP) is a programming language based on first-order Horn clause logic that uses SLD-resolution as a semi-decision procedure. Finite SLD-computations are inductively sound and complete with respect to least Herbrand models of logic programs. Dually, the corecursive approach to SLD-resolution views infinite SLD-computations as successively approximating infinite terms contained in programs' greatest complete Herbrand models. State-of-the-art algorithms implementing corecursion in LP such as CoLP [1, 4] are based on loop detection. However, such algorithms support inference of logical entailment only for rational terms, and they do not account for the important property of productivity in infinite SLD-computations. Structural resolution and productivity checks as proposed in [2, 3] provide an alternative to SLD-resolution and define an effective semi-decidable notion of productivity appropriate to LP.

In this work we present the first unified implementation of a prover for Coalgebraic Logic Programming (CoALP) that uses structural resolution as inference mechanism and implements global productivity checks. While the previous papers provide a detailed account of the theory and proof of properties, they do not outline all the decisions and optimizations needed to engineer a practical implementation. We will describe and analyze a compact implementation of CoALP implemented as a meta interpreter on top of a traditional Prolog system. Using Prolog as implementation language allows for a compact prover since primitive building blocks such as unification and other tooling for term handling are already inbuilt. It also allows deferring the proofs of existing utility and inductive predicates such as *print* and *member* to the underlying logic programming system. Other technologies that CoALP can inherit freely from Prolog for improved performance are, for example, term indexing and term sharing.

One of the high level algorithmic decisions incorporated is to not directly build trees such as outlined in the previous papers but to only construct the branches of the success trees one by one in a depth first manner. This provides for a more space efficient procedure for computing derivations and implementing checks for productivity. While this restricts the order of expansion of subgoals some other schemes such as always expanding inductive goals first can be emulated by permanently reordering body goals in clauses during loading of an input program.

A prerequisite for the prover to use structural resolution is that the program should be deemed guarded against uncontrolled infinite loops and therefore was proven productive. Since a fundamental property of guardedness checks is that further instantiation of a guarded loop cannot make it unguarded another algorithmic decision was to check a logic program statically once during loading of a program and avoid any overhead of runtime checks. The performance of these static checks can further be improved by only checking those predicates that do exhibit loops in the call graph of the program as well as reusing information about already verified guarded paths that were checked earlier. Furthermore, com-

Submitted to:
coalpty 2016

putation of guarding contexts that will be checked for fix points during runtime can also be improved by pre-computing some work such as the gathering of subterms of a clause head.

The CoALP input program syntax is chosen such that it can be read in by any other Prolog system and is compatible with CoLP. Since unmarked predicates default to being considered inductive, coinductive predicates have to be explicitly marked by adding a directive of the form *:- coinductive name/arity*. The following code is an example program that can be interpreted by the CoALP prover:

```
:- coninductive from/2.
from(X,scons(X,Y)) :- from(s(X),Y).
```

Purely inductive programs without any coinductive definitions can therefore be queried with a normal Prolog system as well as CoALP with the same semantics. This output shows a CoALP session to load a program and execute the query `from(0,X)`:

```
?- coalp("logic/from.logic").
?: from(0,X).
   Goal: from(0,scons(0,scons(s(0),scons(s(s(0)),scons(s(s(s(0))),_4020)))))
   Hypothesis: from(0,_3010) to from(s(s(s(0))),scons(s(s(s(0))),_4020)) with
    guarding context [(scons(_4398,_4400),[1])] for predicate from/2
   X = scons(0, scons(s(0), scons(s(s(0)), scons(s(s(s(0))), _4020)))).
```

While CoALP can deal with programs such as the *from* example that would run indefinitely in other coalgebraic systems it is not always terminating due to possible infinite derivation chains. A simple example is the program consisting of the coinductive predicate:

```
q(s(X)):-q(X),q(X).
```

Some of problematic predicates such as q can still be gracefully handled in finite time by reusing cached proofs. However handling other infinite derivation chains in general is an open topic for future research.

References

[1] G. Gupta et al. (2007): *Coinductive Logic Programming and Its Applications*. In: *ICLP*, pp. 27–44.

[2] P. Johann, E. Komendantskaya & M. Schmidt (2016): *A Productivity Checker for Logic Programming*. In: *Proceedings, LOPSTR*.

[3] E. Komendantskaya & P. Johann (2015): *Structural Resolution: a Framework for Coinductive Proof Search and Proof Construction in Horn Clause Logic*. Submitted.

[4] L. Simon et al. (2007): *Co-Logic Programming: Extending Logic Programming with Coinduction*. In: *ICALP*, pp. 472–483.

Refinement Types and Higher-Order Constrained Horn Clauses

C.-H. Luke Ong
University of Oxford
lo@cs.ox.ac.uk

Steven J. Ramsay
University of Oxford
stesay@cs.ox.ac.uk

Constrained Horn clauses have recently emerged as a very promising candidate for a logical basis for automatic program verification (especially symbolic model checking). There is evidence to suggest that many first-order verification problems which, at their heart, require finding relations satisfying some given conditions, can be posed as questions of solving systems of Horn clauses, modulo constraints expressed in some suitable background theory [1, 2]. Consequently, researchers have developed several highly efficient solvers for first-order constrained Horn clauses.

We are interested in verifying higher-order programs. Here too it seems very worthwhile to look for a unifying, logical basis for automated program verification. Furthermore, it is natural to frame the verification problem as a search for relations that meet appropriate criteria, and for these criteria to be expressed as constrained Horn clauses. However, one major difference with the foregoing work is that the relations will, in general, be of higher type. For example, consider the following higher-order program (shown on the left):

$\mathsf{let}\, add\, x\, y = x + y$	$\forall xyz.\, z = x + y \Rightarrow Add\, x\, y\, z$
$\mathsf{let\ rec}\, iter_0\, f\, n =$	$\forall fnr.\, n \leq 0 \wedge r = 0 \Rightarrow Iter_0\, f\, n\, r$
$\quad \mathsf{if}\, n \leq 0\, \mathsf{then}\, 0\, \mathsf{else}\, f\, n\, (iter_0\, f\, (n-1))$	$\forall fnr.\, (\exists p.\, n > 0 \wedge Iter_0\, f\, (n-1)\, p \wedge f\, n\, p\, r) \Rightarrow Iter_0\, f\, n\, r$
$\mathsf{in}\ \lambda n.\, \mathsf{assert}\, (n \leq iter_0\, add\, n)$	$\forall nr.\, Iter_0\, Add\, n\, r \Rightarrow n \leq r$

The term $iter_0\, add\, n$ is the sum of the integers from 1 to n in case n is non-negative and is 0 otherwise. The program is *safe* just in case the assertion is never violated, i.e. the summation is not smaller than n. To verify safety, we must find an invariant that implies the required property. For our purposes, an invariant will be an over-approximation of the input-output graphs of the functions defined in the program.

The existence of such an invariant is specified by the set of higher-order constrained Horn clauses on the right in the unknowns $Iter_0 : (\mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{bool}) \to \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$ and $Add : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$. The first, a definite clause, constrains Add to be an over-approximation of the addition function. The second and third, both definite clauses, constrain $Iter_0$ to be an over-approximation of the iteration combinator. The last, a goal clause, constrains these over-approximations to be small enough that they exclude the possibility of relating an input n to a smaller output r. A model of these clauses in the theory of integer linear arithmetic, assigning (the characteristic function of) relations to $Iter_0$ and Add, is exactly such an invariant. Hence, such an assignment constitutes a witness to safety of the program. One such model is the following (expressed as terms of higher-order logic):

$$Add \mapsto \lambda xyz.\, z = x + y$$
$$Iter_0 \mapsto \lambda f\, nr.\, (\forall xy.\, f\, x\, y\, z \Rightarrow 0 \leq x \Rightarrow y \leq z) \Rightarrow n \leq r$$

The aim of our work is to automate the search for such models. We take inspiration from an already successful method for higher-order program verification, namely refinement type inference [4, 3, 5, 6].

Submitted to:
CoALPTy 2016

Refinement types are a way to specify invariants for higher-order programs. Their power comes from the ability to lift rich first-order theories over data to higher types using subtyping and the dependent product. We have developed a system for assigning refinement types to higher-order constrained Horn clauses. The idea is that every valid type assignment Γ to the higher-order unknowns is a syntactic representation of a model of the definite clauses. For example, the model discussed previously can be represented by the type assignment Γ_I:

$$\begin{array}{l} \mathit{Add}\colon x{:}\mathsf{int} \to y{:}\mathsf{int} \to z{:}\mathsf{int} \to \mathsf{bool}\langle z = x + y\rangle \\ \mathit{Iter}_0\colon (x{:}\mathsf{int} \to y{:}\mathsf{int} \to z{:}\mathsf{int} \to \mathsf{bool}\langle 0 \leq x \Rightarrow y < z\rangle) \to n{:}\mathsf{int} \to r{:}\mathsf{int} \to \mathsf{bool}\langle n \leq r\rangle \end{array}$$

The key to the enterprise is the refinement type for Booleans, $\mathsf{bool}\langle\varphi\rangle$, which is parametrised by a *first-order constraint* formula φ. The meaning of this type under a valuation α of its free first-order variables is the set of truth values $\{\mathsf{false}, [\![\varphi]\!](\alpha)\}$ determined by evaluation of φ. The dependent product and integer types are interpreted standardly, so that the first type above can be read as the set of all ternary relations on integers x, y and z that are false whenever z is not $x + y$.

Soundness of the type system allows one to conclude that certain first-order constraint formulas can be used to approximate higher-order Horn formulas. Given goal formula s, from the derivability the judgement $\Gamma \vdash s : \mathsf{bool}\langle\varphi\rangle$ it follows that $s \Rightarrow \varphi$ in the model (represented by) Γ. Consequently, if Γ is a valid model of the definite clauses and s is the negation of the goal formula, then $\Gamma \vdash s : \mathsf{bool}\langle\mathsf{false}\rangle$ implies that the model validates the desired property. In particular, since Γ_I is a valid type assignment for *Add* and Iter_0 and since $\Gamma_I \vdash \exists nr.\, \mathit{Iter}_0\, \mathit{Add}\, n\, r \wedge n > r : \mathsf{bool}\langle\mathsf{false}\rangle$ is derivable, it follows that the constrained Horn clause problem above is solvable and hence the program is safe.

By phrasing the higher-order constrained Horn clause problem in this way, machinery already developed for refinement type inference can be adapted to automate the search for models (i.e. type assignments). This machinery actually reduces refinement type assignment to the problem of solving sets of *first-order* constrained Horn clauses. Consequently, in total, our work can be seen as a method for obtaining solutions of higher-order constrained Horn clause problems by solving first-order constrained Horn clause problems. An implementation of the method yields promising results.

References

[1] Tewodros A. Beyene, Corneliu Popeea & Andrey Rybalchenko (2013): *Solving Existentially Quantified Horn Clauses*. In: *Computer Aided Verification (CAV 2013)*, pp. 869–882.

[2] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan & Andrey Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In: *Fields of Logic and Computation*, pp. 24–51.

[3] Naoki Kobayashi, Ryosuke Sato & Hiroshi Unno (2011): *Predicate abstraction and CEGAR for higher-order model checking*. In: *Programming Languages Design and Implementation, PLDI'11*, ACM, pp. 222–233.

[4] Patrick Maxim Rondon, Ming Kawaguchi & Ranjit Jhala (2008): *Liquid types*. In: *Programming Language Design and Implementation (PLDI 2008)*, pp. 159–169.

[5] Niki Vazou, Alexander Bakst & Ranjit Jhala (2015): *Bounded refinement types*. In: *International Conference on Functional Programming (ICFP 2015)*, pp. 48–61.

[6] He Zhu & Suresh Jagannathan (2013): *Compositional and Lightweight Dependent Type Inference for ML*. In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pp. 295–314.

Interaction Morphisms [Extended Abstract]

Shin-ya Katsumata
RIMS, Kyoto University, Japan
sinya@kurims.kyoto-u.ac.jp

Tarmo Uustalu
Institute of Cybernetics at TUT, Estonia
tarmo@cs.ioc.ee

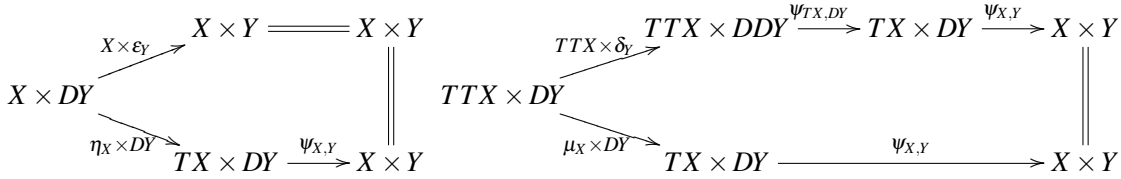
1 Introduction

When can effectful computations for a given notion of effect be run statefully, i.e., when is there a monad morphism from a given monad T on **Set** to the state monad for some state set? In previous work [4], we showed that, if T arises from a Lawvere theory L, then such monad morphisms are in a bijection with coalgebras of D where D is the comonad determined by L. We worked out a number of examples.

We have since found that, on a deeper level, this work used some properties of what we here call interaction morphisms.

2 Interaction Morphisms: Definition and Some Facts

Interaction morphisms Given a category $\mathscr{C}$ with finite products (or more generally a monoidal category). An *interaction morphism* of a monad $T = (T, \eta, \mu)$ and comonad $D = (D, \varepsilon, \delta)$ on $\mathscr{C}$ is a natural transformation ψ with components $\psi_{X,Y} : TX \times DY \to X \times Y$ satisfying



Similarly to monads, comonads, triples of a monad, comonad and an interaction morphisms are the same as monoids in a suitable monoidal category. An object in this category is a triple of two endofunctors F, G together with a functor-functor interaction morphism, i.e., a natural transformation ϕ with components $\phi_{X,Y} : FX \times GY \to X \times Y$.

An archetypical example of an interaction morphism is given by $TX = S \Rightarrow X$, $DY = S \times Y$ for some fixed S and $\psi_{X,Y} : (S \Rightarrow X) \times (S \times Y) \to X \times Y$ the canonical map. Another is given by $TX = S \Rightarrow (S \times X)$, $DY = S \times (S \Rightarrow Y)$ and $\psi_{X,Y} : (S \Rightarrow (S \times X)) \times (S \times (S \Rightarrow Y)) \to X \times Y$ the canonical map.

There are a number of constructions for obtaining interaction morphisms. For example, an interaction morphism ϕ of two functors F and G induces an interaction morphism of the free monad on F and the cofree monad on G (assuming these exist). This is the free monad-comonad interaction morphism on the given functor-functor interaction morphism.

Submitted to:
CoALP-Ty'16

Interaction morphisms vs runners A *runner* of a monad T is an object Y with a natural transformation θ with components $\theta_X : TX \times Y \to X \times Y$ satisfying

$$\begin{array}{ccc} X \times Y & \xrightarrow{\theta_X} & X \times Y \\ {\scriptstyle \eta_X \times Y}\downarrow & & \| \\ TX \times Y & = & X \times Y \end{array} \qquad \begin{array}{ccccc} TTX \times Y & \xrightarrow{\theta_{TX}} & TX \times Y & \xrightarrow{\theta_X} & X \times Y \\ {\scriptstyle \mu_X \times Y}\downarrow & & & & \| \\ TX \times Y & & \xrightarrow{\qquad\qquad\theta_X\qquad\qquad} & & X \times Y \end{array}$$

More concisely, a runner of a monad T is an object Y together with a monad morphism from T to the state monad for Y.

Interaction morphisms of T and D are in a bijection with carrier-preserving functors between the categories of D-coalgebras and T-runners.

Interaction morphisms as monad morphisms Given an endofunctor D on $\mathscr{C}$, then in the presence of enough structure on $\mathscr{C}$, it induces another endofunctor $\ulcorner D \urcorner$ on $\mathscr{C}$ by $\ulcorner D \urcorner X = \int_Y DY \Rightarrow X \times Y$

If D is a comonad, then $\ulcorner D \urcorner$ is a monad. This follows from the observation that $\ulcorner - \urcorner : [\mathscr{C},\mathscr{C}]^{\mathrm{op}} \to [\mathscr{C},\mathscr{C}]$ is a lax monoidal functor, so sends monoids in $[\mathscr{C},\mathscr{C}]^{\mathrm{op}}$ to monoids in $[\mathscr{C},\mathscr{C}]$.

Interaction morphisms of T and D are in a bijection with monad morphisms between T and $\ulcorner D \urcorner$.

3 Related Work, Future Work

This work is related by its motivations and approach to the comodels-based operational semantics works of Plotkin and Power [3] and Abou-Saleh and Pattinson [2, 1], however here we can circumvent Lawvere theories.

We see connections to session typing (the classical two-party case). This a direction we would like to pursue seriously in the future.

Acknowledgements Uustalu's research has been supported by the Estonian Science Foundation grant no. 9475 and the Estonian Ministry of Education and Research institutional research grant no. IUT33-13.

References

[1] F. Abou-Saleh (2013): *A Coalgebraic Semantics for Imperative Programming Languages*. PhD thesis, Imperial College. Available at `http://hdl.handle.net/10044/1/13693`

[2] F. Abou-Saleh & D. Pattinson (2013): *Comodels and effects in mathematical operational semantics*. In F. Pfenning, ed.: *Proc. of 16th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2013*, *Lect. Notes in Comput. Sci.* 7794, Springer, pp. 129–144. doi:10.1007/978-3-642-37075-5_9

[3] G. D. Plotkin & J. Power (2008): *Tensors of comodels and models for operational semantics*. In A. Bauer & M. Mislove, eds.: *Proc. of 24th Conf. on Mathematical Foundations of Programming Semantics, MFPS XXIV*,*Electron. Notes Theor. Comput. Sci.* 218. Elsevier, pp. 295–311. doi:10.1016/j.entcs.2008.10.018

[4] T. Uustalu (2015): *Stateful runners for effectful computations*. In D. Ghica, ed.: *Proc. of 31st Conf. on Mathematical Foundations of Programming Semantics, MFPS XXXI*, *Electron. Notes in Theor. Comput. Sci.* 319, Elsevier, pp. 403–421. doi:10.1016/j.entcs.2015.12.024

Proofs by Resolution and Existential Variables

František Farka
University of St Andrews
Scotland, UK, and
University of Dundee
Scotland, UK
ff32@st-andrews.ac.uk

There has been a renewed interest in Horn Clause Logic in the filed of program verification. Bjørner *et al.*[1] pointed out that Horn Clauses give a general framework for first order proving. They put emphasis on expressive power of and computational advantages arising from Horn Clause Logic. Meantime, Ong and Ramsay have argued [5] for type-based approach to model checking. Recently, Fu and Komendantskaya [3] studied operational semantics of resolution in Horn-clause logic and provided type-theoretic interpretation of logic programming (LP). Such semantics could bring the two aforementioned approaches together. However, that approach has limitations. Fu and Komendantskaya focused on universal fragment of Horn-clause logic and they did not treat existential variables. Additionally, in their recent work [4] on semantics of LP, Komendantskaya and Power observed that *term-matching* resolution in LP can be understood as theorem proving in LP whereas *unification* captures problem solving aspect of LP.

Following our previous work [2], we are interested in type theoretic formulation of the operational semantics and we strengthen the above claim: In fact, term matching steps in resolution reflect proof steps acting on universally quantified variables whereas unification steps correspond to proof steps acting on existentially quantified variables — the problem solving aspect here corresponds to search for proof witnesses for existential variables.

We work in a proof-relevant language. A *signature* comprises a set of function symbols $\mathscr{F}$, including constant symbols, a set of predicate symbols $\mathscr{P}$ and a set of proof-term symbols $\mathscr{K}$. Also, we assume there is a countably infinite set of variables *Var*. A *substitution*, an application of substitution, a *unifier* and a *matcher* are defined entirely standard.

Definition 1 (Syntax)**.**

Terms	$Ter ::=$	$Var \mid \mathscr{F}(Ter,\ldots,Ter)$	*Proof terms*	$PT ::=$	$\mathscr{K}\ PT \ldots PT \mid ind(Var.Term, PT)$
Atomic flae	$At ::=$	$\mathscr{P}(Ter,\ldots,Ter)$	*Programs*	$Prog ::=$	$\mathscr{K} : HC,\ldots,\mathscr{K} : HC$
Horn flae	$HC ::=$	$At \leftarrow At,\ldots,At$	*Goals*	$G ::=$	$\forall Var,\ldots,Var \exists Var,\ldots,Var.At$

Our language differs from the standard language of LP in two ways: Horn formulae in our programs are annotated with proof-term constant symbols, and goals are explicitly quantified by universal and existential quantifiers. We refer to a clause in a program by the symbol that annotates it. Moreover, we denote a sequence of distinct variables, by $\overline{x}$, and a sequence of variables occurring in an atomic formula A by $\mathrm{var}(A)$. Substitution application, set intersection and set difference are extended to variable sequences as is standard.

The explicit quantification of goals helps us to separate matching and unification steps in our proofs. First, let us introduce proof steps by term matching:

Definition 2 (TM-Resolution)**.**

$$\kappa : A \leftarrow B_1,\ldots,B_n \in P \quad \frac{P \vdash e_1 : \forall \overline{w_1} \exists \overline{z_1}.\sigma B_1 \qquad \ldots \qquad P \vdash e_n : \forall \overline{w_n} \exists \overline{z_n}.\sigma B_n}{P \vdash \kappa\ e_1\ \ldots\ e_n : \forall \overline{x} \exists \overline{y}.\sigma A}$$

Submitted to:
CoALP-Ty'16

where $\overline{w_i} = \overline{x} \cap \mathrm{var}(\sigma B_i)$ *and* $\overline{z_i} = \mathrm{var}(\sigma B_i) \setminus w_i$.

The above inference rule distributes quantified variables of the goal $\forall \overline{x} \exists \overline{y}.\sigma A$ to goals $\forall \overline{w_1} \exists \overline{z_1}.B_1$ to $\forall \overline{w_n} \exists \overline{z_n}.B_n$ in the premise. Universally quantified variables that are relevant to a goal B_i are quantified universally, existential variables are quantified existentially and any (existential) variable that does not occur in the head of clause κ is quantified existentially. This is in concordance with the above referred results of Komendantskaya and Power [4]. Consider the following example.

Example 1. *Let* $P_{ListNat}$*: be the following program:*

$$\begin{array}{llll} \kappa_1: & \mathrm{nat}(\mathrm{zero}) & \leftarrow & \\ \kappa_2: & \mathrm{nat}(\mathrm{s}(x)) & \leftarrow \mathrm{nat}(x) & \\ \kappa_3: & \mathrm{list}(\mathrm{nil})) & \leftarrow & \\ \kappa_4: & \mathrm{list}(\mathrm{cons}(x,y))) & \leftarrow \mathrm{nat}(x), \mathrm{list}(y) & \end{array}$$

A ground goal $\mathrm{list}(\mathrm{cons}(\mathrm{zero}, \mathrm{nil}))$ can be resolved using only the TM-Resolution rule:

$$\frac{\overline{P \vdash \kappa_1 : \mathrm{nat}(\mathrm{zero})} \qquad \overline{P \vdash \kappa_3 : \mathrm{list}(\mathrm{nil})}}{P \vdash \kappa_4\ \kappa_1\ \kappa_3 : \mathrm{list}(\mathrm{cons}(\mathrm{zero}, \mathrm{nil}))}$$

However, for other goals full unification is necessary. The Unif-resolution is defined as follows.

Definition 3 (Unif-Resolution)**.**

$$\kappa : A \leftarrow B_1, \ldots, B_n \in P \ \frac{P \vdash e_1 : \forall \overline{w_1} \exists \overline{z_1}.\sigma B_1 \qquad \ldots \qquad P \vdash e_n : \forall \overline{x}.\sigma B_n}{P \vdash \mathit{ind}(\overline{y}.\sigma \overline{y}, \kappa\ e_1, \ldots, e_n) : \forall \overline{x} \exists \overline{y}.A'}$$

if $\sigma \restriction_{\overline{y}} A' = \sigma \restriction_{\overline{y}} A$*. Moreover,* $w_i = \overline{x} \cap \mathrm{var}(\sigma B_i)$ *and* $\overline{z_i} = \mathrm{var}(\sigma B_i) \setminus w_i$.

We use $\sigma \restriction \overline{y}$ to denote a restriction of a substitution σ to variables $\overline{y}$. The notation $\overline{y}.\sigma\overline{y}$ indicates binding of variables in $\overline{y}$ given by substitution σ. Using both TM-resolution and Unif-resolution rules allows us to prove *e.g.* $\exists x, y.\, \mathrm{list}(\mathrm{cons}(x,y))$:

$$\frac{\overline{P \vdash \kappa_1 : \mathrm{nat}(\mathrm{zero})} \qquad \overline{P \vdash \kappa_3 : \mathrm{list}(\mathrm{nil})}}{P \vdash \mathit{ind}(x.\mathit{zero}, y.\mathit{nil}, \kappa_4\ \kappa_1\ \kappa_3) : \exists x, y.\, \mathrm{list}(\mathrm{cons}(x,y))}$$

In the paper, we are going to show soundness of the above calculus w.r.t. first order dependent type theory and completeness for proof terms in first-order ground normal form. Furthermore, we discuss relation of our calculus to *Structural resolution* [3].

References

[1] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan & Andrey Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, *Lecture Notes in Computer Science* 9300, Springer, pp. 24–51.

[2] František Farka, Ekaterina Komendantskaya, Kevin Hammond & Peng Fu (2016): *Coinductive Soundness of Corecursive Type Class Resolution*. In: *Pre-proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*.

[3] Peng Fu & Ekaterina Komendantskaya (2017): *Operational Semantics of Resolution and Productivity in Horn Clause Logic*. *Accepted to Formal Aspects of Computing*.

[4] Ekaterina Komendantskaya & John Power (2016): *Logic programming: laxness and saturation*. *Submitted to Journal of Logic and Algebraic Methods in Programming*.

[5] C.-H. Luke Ong & Steven J. Ramsay (2011): *Verifying higher-order functional programs with pattern-matching algebraic data types*. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, ACM, pp. 587–598.

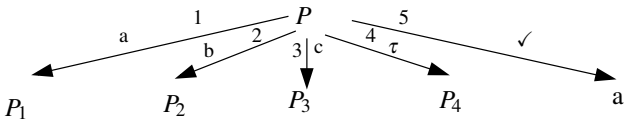
Defining Trace Semantics for CSP-Agda

Bashar Igried Anton Setzer
Swansea University, Department of Computer Science
Swansea,Wales, UK
bashar.igried@yahoo.com a.g.setzer@swansea.ac.uk

We have developed a library CSP-Agda for representing coinductively CSP processes, extended to monadic form, in the interactive theorem prover Agda. In this talk we add to CSP-Agda trace semantics of CSP. We show how to prove algebraic laws of CSP in Agda using this semantics.

1 Introduction

The process algebra CSP [3, 5] is a formal specification language which was developed in order to describe concurrent systems through their communications. It was developed by Hoare in 1978 [3]. There have been several successful approaches of combining functional programming with CSP, for instance [1], [2]. In [4] we introduced a library called CSP-Agda for representing processes in the dependently typed theorem prover and interactive programming language Agda. Processes in CSP-Agda are formed from an atomic one step operation, which defines a new process by determining the transitions it can make together with the processes we obtain when firing these transitions. Since processes can loop for ever, processes are defined coinductively from this one step operation. The transitions a process can make are labelled external choices, silent τ-transitions, and termination events $\checkmark$. We add a return value to termination events, and therefore extend CSP processes to monadic ones.



In CSP-Agda operators from CSP such as external/internal choice, parallel operators, and composition are in CSP-Agda defined operations. In this talk we extend CSP-Agda by adding (finite) trace semantics of CSP. We prove algebraic laws of CSP in Agda using this semantics.

2 Defining Trace Semantics in Agda

The traces of a process are the sequences of actions, a process can perform. Since the process in CSP are non-deterministic, a process can follow different traces during its execution. The trace semantics of a process is the set of its traces. In CSP-Agda the set of traces $traces(P)$ for a process P, which includes in case of termination the return values, is defined as follows (we omit a similar definition for Process):

```
data Tr+ {i : Size} {c : Choice } : (l : List Label) → Maybe (ChoiceSet c)
                                  → (P : Process+ i c) → Set where
  empty : {P : Process+ i c} → Tr+ [] nothing P
  extc  : {P : Process+ i c} → (l : List Label) → (tick : Maybe (ChoiceSet c))
        → (x : ChoiceSet (E P)) → Tr∞ l tick (PE P x) → Tr+ (Lab P x :: l) tick P
```

Submitted to:
CoALP-Ty'16

```
intc    : {P : Process+ i c} → (l : List Label) → (tick : Maybe (ChoiceSet c))
          → (x : ChoiceSet (I P)) → Tr∞ l tick (PI P x) → Tr+ l tick P
terc    : {P : Process+ i c} → (x : ChoiceSet (T P)) → Tr+ [] (just (PT P x)) P

record Tr∞ {i : Size} {c : Choice} (l : List Label) (tick : Maybe (ChoiceSet c))
           (P : Process∞ i c) : Set where
  coinductive
  field
    forcet : {j : Size< i} → Tr {j} l tick (forcep P)
```

In CSP, a process P refines a process Q, written $P \sqsubseteq+ Q$ if and only if any observable behaviour of Q is an observable behaviour of P, i.e. if $traces(Q) \subseteq traces(P)$:

```
_⊑+_ : {i : Size} {c : Choice} (P : Process+ i c) (Q : Process+ i c) → Set
_⊑+_ {i} {c} P Q = (l : List Label) → (m : Maybe (ChoiceSet c))
                   → Tr+ l m Q → Tr+ l m P
```

Two processes P, Q are equal w.r.t. trace semantics, written $P \equiv+ Q$, if they refine each other:

```
_≡+_ : {i : Size} → {c₀ : Choice} → (P Q : Process+ i c₀) → Set
P ≡+ Q = P ⊑+ Q × Q ⊑+ P
```

As an example we give the main case of the proof of symmetry of interleaving (fmap+ applies a function to the return values of a process, swap× swaps the two sides of a product):

```
S|||+ : {i : Size} {c₀ c₁ : Choice} (P : Process+ i c₀) (Q : Process+ i c₁)
        → (P |||++ Q) ⊑+ (fmap+ swap× (Q |||++ P))
S|||+ P Q .[] .nothing empty = empty
S|||+ P Q .(Lab Q x :: l) m (extc l .m (inj₁ x) q) = extc l m (inj₂ x) (S|||+∞ P (PE Q x) l m q)
S|||+ P Q .(Lab P x :: l) m (extc l .m (inj₂ x) q) = extc l m (inj₁ x) (S|||∞+ (PE P x) Q l m q)
S|||+ P Q l m (intc .l .m (inj₁ x) q)              = intc l m (inj₂ x) (S|||+∞ P (PI Q x) l m q)
S|||+ P Q l m (intc .l .m (inj₂ x) q)              = intc l m (inj₁ x) (S|||∞+ (PI P x) Q l m q)
S|||+ P Q .[] .(just (PT P x ,, PT Q y)) (terc (y ,, x)) = terc (x ,, y)
```

References

[1] Neil C. C. Brown (2008): *Communicating Haskell Processes: Composable Explicit Concurrency using Monads*. In: *The thirty-first Communicating Process Architectures Conference, CPA 2008*, pp. 67–83, doi:10.3233/978-1-58603-907-3-67. Available at `http://dx.doi.org/10.3233/978-1-58603-907-3-67`.

[2] Neil C. C. Brown (2009): *Automatically Generating CSP Models for Communicating Haskell Processes*. *ECEASST* 23. Available at `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/325`.

[3] C. A. R. Hoare (1978): *Communicating Sequential Processes*. *Commun. ACM* 21(8), pp. 666–677, doi:10.1145/359576.359585. Available at `http://doi.acm.org/10.1145/359576.359585`.

[4] Bashar Igried & Anton Setzer (2016): *Programming with Monadic CSP-style Processes in Dependent Type Theory*. In: *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, ACM, New York, NY, USA, pp. 28–38, doi:10.1145/2976022.2976032. Available at `http://doi.acm.org/10.1145/2976022.2976032`.

[5] A. W. Roscoe, C. A. R. Hoare & Richard Bird (1997): *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Models of Inductive-Coinductive Logic Programs

Henning Basold
Radboud University
CWI Amsterdam
henning@basold.eu

Ekaterina Komendantskaya
Heriot-Watt University
komendantskaya@gmail.com

Traditionally, logic programs have been used to describe relations between finite terms, a fact most prominently reflected in the notion of the least Herbrand model [7, 5]. A typical example is the following logic program, which generates the natural numbers.

$$\begin{aligned} 1: &\quad \mathrm{nat}(0) \longleftarrow \\ 2: &\quad \mathrm{nat}(\mathrm{s}(x)) \longleftarrow \mathrm{nat}(x) \end{aligned} \tag{1}$$

Later [1, 2, 3, 4, 6], also coinductive interpretations of logic programs have been proposed. Under a coinductive interpretation, the program given in (1) generates an extra element s^ω. The thus obtained interpretation of nat corresponds to the completion of the natural numbers with a point at infinity. Another example, which is non-trivial only under a coinductive interpretation, are streams over natural numbers:

$$3: \quad \mathrm{str}(\mathrm{cons}(x,y)) \longleftarrow \mathrm{nat}(x), \mathrm{str}(y) \tag{2}$$

Note, however, that a purely coinductive model also contains elements of the form $\mathrm{cons}(\mathrm{s}^\omega, t)$ for some (infinite) stream term t. To rule out such spurious terms we would have to interpret nat inductively and str coinductively, something that has not been studied so far.

In the following we will augment logic programs with a function par that assigns to each relation symbol its *parity*, which can be either μ or ν. The parity expresses whether a relation is supposed to be interpreted inductively or coinductively. For example, to obtain the intended interpretation of the clauses in (1) and (2), we would define $\mathrm{par}(\mathrm{nat}) = \mu$ and $\mathrm{par}(\mathrm{str}) = \nu$.

Another, perhaps more interesting, example is the sub-stream relation sub that relates streams s and t if all values of s appear in order in t. We can express the sub-stream relation as logic program by using a helper relation sub_μ. The relation sub_μ tries to match the head of s with a value in t. An inductive interpretation of sub_μ enforces then that the head of s must be found in t after finitely many steps.

$$\begin{aligned} &\mathrm{par}(\mathrm{sub}) = \nu \qquad \mathrm{par}(\mathrm{sub}_\mu) = \mu \\ 4: &\quad \mathrm{sub}(x,y) \longleftarrow \mathrm{sub}_\mu(x,y) \\ 5: &\quad \mathrm{sub}_\mu(\mathrm{cons}(n,x), \mathrm{cons}(n,y)) \longleftarrow \mathrm{nat}(n), \mathrm{sub}(x,y) \\ 6: &\quad \mathrm{sub}_\mu(x, \mathrm{cons}(n,y)) \longleftarrow \mathrm{nat}(n), \mathrm{sub}_\mu(x,y) \end{aligned} \tag{3}$$

It should be noted that the relation sub is interpreted as the full relation in a purely coinductive model because in such a model, the search of sub_μ does not have to terminate.

A first step towards understanding inductive-coinductive logic programs is understanding their denotational models. Here it is important that logic programs Φ are given by means of two signatures Σ_Φ and Δ_Φ. The signature Σ_Φ contains thereby the symbols that are used in the terms, like s and 0 in the examples above. On the other hand, Δ_Φ specifies the relation symbols used in Φ. We will denote the arity of a symbol $f \in \Sigma_\Phi$ by $\mathrm{ar}\, f$ and similarly for symbols in Δ_Φ. A *(term) model* $\mathscr{M}$ for Φ is then required

Submitted to:
CoALP-Ty 2016

to provide interpretations of the relation symbols in Δ_Φ as relations between possibly infinite terms over Σ_Φ. Moreover, for inductive relation symbols their interpretation in $\mathscr{M}$ needs to be forward closed under the clauses of Φ, whereas the interpretation of coinductive relations must be backwards closed.

Let us make this more precise. Suppose we are given signatures Σ and Δ, and a set V of variables. Let $\Sigma^*(V)$ be the set of terms over V, and Σ^∞ be the set of possibly infinite ground terms over Σ. A *formula* φ is given by $Q(\vec{t})$ for some $Q \in \Delta$ and a tuple $\vec{t} = (t_1, \ldots, t_{\mathrm{ar}\,Q})$ of terms in $\Sigma^*(V)$. We call a finite set of formulas a *sentence*. Finally, a *(Horn) clause* is a pair of a sentence S and a formula φ, denoted by $\varphi \longleftarrow S$. A *logic program* Φ consists of signatures Σ, Δ, a map $\mathrm{par}\colon \Delta \to \{\mu, \nu\}$ and a set of clauses.

Using this setup, we now characterise models for logic programs Φ. First, we associate to Φ a map $\widehat{\Phi}\colon \prod_{Q\in\Delta} \mathrm{Rel}_{\mathrm{ar}\,Q}(\Sigma_\Phi^\infty) \to \prod_{Q\in\Delta} \mathrm{Rel}_{\mathrm{ar}\,Q}(\Sigma_\Phi^\infty)$ by

$$\widehat{\Phi}(F)(Q) := \bigcup_{\substack{\phi \longleftarrow S \in \Phi \\ \phi = Q(\vec{t})}} \left\{ \vec{t}[\sigma] \;\middle|\; \sigma\colon V \to \Sigma_\Phi^\infty \text{ and } \forall P(\vec{s}) \in S.\ \vec{s}[\sigma] \in F(P) \right\},$$

where $\vec{t}[\sigma]$ denotes the substitution of σ into all terms in $\vec{t}$. Moreover, we define components

$$\widehat{\Phi}_\rho : \prod_{Q\in\Delta} \mathrm{Rel}_{\mathrm{ar}\,Q}(\Sigma^\infty) \to \prod_{Q\in\Delta_\rho} \mathrm{Rel}_{\mathrm{ar}\,Q}(\Sigma^\infty)$$

of $\widehat{\Phi}$ by restriction: $\widehat{\Phi}_\rho(Q) := \widehat{\Phi}(Q)|_{\Delta_\rho}$, where $\Delta_\rho := \mathrm{par}^{-1}(\rho)$. A Φ-*model* $\mathscr{M}$ is given by a map $\mathscr{M} \in \prod_{Q\in\Delta} \mathrm{Rel}_{\mathrm{ar}\,Q}(\Sigma^\infty)$, such that

$$\widehat{\Phi}_\mu(\mathscr{M}) \sqsubseteq \mathscr{M}_\mu \quad \text{and} \quad \mathscr{M}_\nu \sqsubseteq \widehat{\Phi}_\nu(\mathscr{M}),$$

where $\sqsubseteq$ is point-wise inclusion and $\mathscr{M}_\rho$ is the restriction $\mathscr{M}|_{\Delta_\rho}$. This encodes precisely the forward and backwards closure conditions.

This characterisation of models in terms of a monotone operator enables us to construct a fixed point model for Φ. This in turn gives us a universe of discourse for exploring other semantics and proof systems for mixed inductive-coinductive logic programs.

In the talk, we will discuss properties of the fixed point model. We will further discuss *standard models*, in which the interpretation of inductive relations is restricted. This allows us to prove weak completeness of the fixed point model with respect to standard models.

References

[1] M. A. Nait Abdallah (1984): *On the Interpretation of Infinite Computations in Logic Programming*. In: *ICALP*, *Lecture Notes in Computer Science* 172, Springer, pp. 358–370, doi:10.1007/3-540-13345-3_32.

[2] Mathieu Jaume (2000): *Logic Programming and Co-Inductive Definitions*. In: *CSL*, *Lecture Notes in Computer Science* 1862, Springer, pp. 343–355, doi:10.1007/3-540-44622-2_23.

[3] Mathieu Jaume (2002): *On Greatest Fixpoint Semantics of Logic Programming*. *J. Log. Comput.* 12(2), pp. 321–342, doi:10.1093/logcom/12.2.321.

[4] Ekaterina Komendantskaya & John Power (2011): *Coalgebraic Semantics for Derivations in Logic Programming*. In: *CALCO*, *LNCS* 6859, Springer, pp. 268–282, doi:10.1007/978-3-642-22944-2_19.

[5] John W. Lloyd (1987): *Foundations of Logic Programming, 2nd Edition*. Springer.

[6] Danny De Schreye & Stefaan Decorte (1994): *Termination of Logic Programs: The Never-Ending Story*. *J. Log. Program.* 19/20, pp. 199–260, doi:10.1016/0743-1066(94)90027-2.

[7] M. H. Van Emden & R. A. Kowalski (1976): *The Semantics of Predicate Logic As a Programming Language*. *J. ACM* 23(4), pp. 733–742, doi:10.1145/321978.321991.

Classes for the Masses

Claudio Russo
(Microsoft Research)

Matt Windsor
(University of York)

Don Syme
(Microsoft Research)

Rupert Horlick
(University of Cambridge)

James Clarke
(University of Cambridge)

Abstract Type classes are an immensely popular and productive feature of Haskell. They have since been adopted in, and adapted to, numerous other languages, including theorem provers. We show that type classes have a natural and efficient representation in .NET that paves the way for the extension of F#, C# and other .NET languages with type classes. Our encoding is type preserving and promises easy and safe cross-language inter-operation. We have extended the open source C# compiler and language service, Roslyn, with pervasive support for type classes and have prototyped a more minimalist design for F#.

(This talk was originally presented at the ACM SIGPLAN Workshop on ML, September 2016, Nara, Japan).

1. INTRODUCTION

Haskell's *type classes* [10, 11] are a powerful abstraction mechanism for describing generic algorithms applicable to types that have different representations but common interfaces. A *type class* is a predicate on types that specifies a set of required operations by their type signatures with optional, default code. A type may be declared to be an *instance* of a type class, and must supply an implementation for each of the class' operations. Type classes may be arranged hierarchically, permitting subsumption and inheritance.

Many modern languages have adopted features inspired by type classes, with different implementation techniques. Scala has *implicits*[9], implicit method arguments denoting dictionaries, that are inferred by the compiler but represented, at run-time, as additional heap-allocated arguments to methods (with commensurate overhead). C++ came very close to adopting *concepts*[7], a rather different extension of the template mechanism, directly inspired by Haskell's type classes but enforcing compile-time code specialization for performance. Rust has *traits*[3]. Swift has *protocols*[4].

Contribution We describe a simple encoding that allows us to add type classes to any .NET language, allowing interoperable definitions of type classes. Our encoding relies on the CLR's distinctive approach to representing and compiling generic code[8, 12]. Unlike, for example, the JVM, the CLR byte-code format is fully generic (all source level type information, including class and method type parameters, are represented in the metadata and virtual instruction set). Parameterized code is JIT-compiled to type passing code, with type parameters having run-time representations as (second-order) values. The JIT compiler uses the reified types to generate specialized memory representations (for instantiated generic types) and specialized (and thus more efficient) code for generic methods. For example, scalar types and compounds of scalars called structs have natural unboxed representations familiar to C(++) programmers; generic array manipulating code will manipulate array elements without boxing when instantiated at scalar types. This run-time specialization allows the JIT to avoid the uniform (i.e. lowest-common-denominator) representations adopted by many implementations of ML, Haskell, the JVM and most dynamic languages.

Haskell compilers typically compile type classes using the so-called *dictionary translation*. The translation, guided by source types, inserts evidence terms that justify type class constraints. The evidence terms are dictionaries (i.e. records) of functions that provide implementations (and thus proofs) for all of the constraint's methods. Although similar to object-oriented virtual method tables, dictionaries are not attached to objects, but passed separately as function arguments. Because type classes are resolved statically, aggressive in-lining can remove most, but not all, indirection through dictionary parameters. This leads to efficient code with fewer indirect calls and leaner representations of values than full-blown objects. Objects, in contrast, must lug their method-tables wherever they go.

Given the obvious similarity between type passing and dictionary passing, it is perhaps not surprising that type passing forms an excellent implementation technique for Haskell's dictionary passing. This talk will give on overview of the technique that we are applying to provide efficient, interoperable type class implementations to both C# and F#.

2. THE REPRESENTATION

This section sketches our representation of the Haskell'98 type classes on .NET by example. For each example, we give the Haskell code, underlying .NET code in vanilla C#, and proposed F# syntax. We use vanilla C# as a more readable proxy for .NET intermediate bytecode and metadata.

Type Classes A Haskell type class, for example:

```
class Eq a where
  (==) :: a -> a -> Bool
```

is naturally represented in C# as:

```
interface Eq<A> { bool Equal(A a, A b); }
```

For F#, we use a `Trait`-attributed interface declaration:

```
[<Trait>]
type Eq<'A> = (* an interface *)
  abstract equal: 'A -> 'A -> bool
```

Haskell Overloads Haskell's declaration of class `Eq a` implicitly declares its members as overloaded operations:

```
  (==) :: (Eq a) => a -> a -> Bool
```

Observe that the overloaded operation has a more general constrained type `(Eq a) =>...`.

This generic operation is captured in C# by the method:

```
static bool Equal<A,EqA>(A a, A b) EqA: struct, Eq<A>
            => default(EqA).Equal(a, b);
```

This method has not one, but *two*, type parameters. The first, `A`, is just the type parameter from the declaration. The

second, `EqA`, is a type parameter that is constrained to be a struct and is evidence for the constraint that `A` supports interface `Eq<A>`.

The use of the `struct` constraint on `EqA` is significant and subtle. Structs are stack-allocated so essentially free to create, especially when they contain no fields. Moreover, every struct type, including a type parameter `T` of kind struct, has a default (all-zero) value denoted by expression `default(T)`. Invoking a method on a default value of reference type would simply raise a null-reference exception because the receiver is `null`. However, methods on structs (including interface methods) can always be properly invoked by calling the method on the struct's default value.

Thus an operation over some class can be represented as a static generic method, parameterized by an additional dictionary type parameter (here `EqA`). Derived operations with type class constraints can be represented by generic methods with suitably constrained type parameters. Finally, Haskell dictionary *values* correspond to C# dictionary *types*.

For F#, we do not overload a top-level `Equal` binding but, instead, allow qualifed access to trait members (e.g. `Eq.equal`), as in:

```
let equal = Eq.equal (* defines overloaded equal *)
```

Instances A Haskell instance declaration is represented by the declaration of an empty (field-less) .NET struct that implements the associated type class (itself an interface). This gives us a cheap representation of Haskell instances.

For example, the Haskell instance declaration:

```
instance Eq Integer where
  x == y =  x `integerEq` y
```

can be represented by the C# structure:

```
struct EqInt : Eq<int>
{  public bool Equal(int a, int b) => a == b; }
```

For F#, we use a `Witness`-attributed struct declaration:

```
[<Witness>]
type EqInt = (* a struct *)
  interface Eq<int> with member equal a b = a = b
```

Note that the F# syntax, unlike Haskell, names the instance as in the C# representation. In Haskell, instances are anonymous but names are useful for explicit disambiguation and interoperation with languages that cannot always rely on type argument inference (such as C#).

Derived Instances This Haskell code defines a family of derived instances: given an equality type `a`, it defines equality over lists of `a`.

```
instance (Eq a) => Eq ([a]) where
    nil == nil  = true
   a:as == b:bs = (a == b) && (as == bs)
      _ == _    = false
```

We can represent such a Haskell parameterized instance as a *generic* struct:

```
struct EqList<A, EqA> : Eq<List<A>>
  where EqA : struct, Eq<A> {
    public bool Equal(List<A> a, List<A> b) =>
          (a.IsNull && b.IsNull)
      || (a.IsCons && b.IsCons
           && default(EqA).Equal(a.Head,b.Head)
           && Equal(a.Tail,b.Tail)); }
```

This struct implements the interface `Eq<List<A>>`, but only when instantiated with a suitable type argument and evidence for constraint `Eq<A>`. Notice that `EqList` has, once again, an additional evidence type parameter `EqA` for constraint `Eq<A>`. Instantiations of the generic struct `EqList<->`, in turn, construct evidence for `Eq<List<A>>`.

For F# we use a parameterized `Witness`-declaration:

```
[<Witness>]
type EqList<'A,'EqA when 'EqA :> Eq<'A>> = (* a struct *)
  interface Eq<'A list> with
    member equal a b = match a,b with
      | a::l,b::m -> Eq.equal a b && Eq.equal l m
      | [],[] -> true | _,_ -> false
```

Other features We do not have space to describe the representations of other features but suffice to say that we can encode [5]: type class operations that themselves have constrained types in their signatures (using interface methods that are generic); type class hierarchies using interface inheritance; default operations using shared static methods; instances requiring polymorphic recursion; instances as data (to constrained term constructors) and multi-parameter type classes. Moreover, choosing to provide named rather than anonymous instances would allow us to selectively support explicit as well as implicit evidence when preferable. We cannot support higher-kinded type classes (like `Monad`), because .NET lacks higher-kinded abstraction. First-order associated types are in reach. For C#, evidence inference is a mild generalization of type argument inference, with instantiations derived from the pervasive and locally assumed concept hierarchy. For F#, we have adapted Haskell's more elaborate techniques for propagating inferred type class constraints, by extension of F#'s existing constraint system.

Implementations We prioritized our efforts on designing and implementing type classes for C# in a fork[1] of Microsoft's open source Roslyn compiler[2], adopting a dedicated syntax loosely inspired by C++ concepts[7]. The F# design and implementation [6] was the result of a 3-day hackathon aiming for a minimal viable product (with suboptimal syntax). Performance results are promising - we anticipated .NET's code specialization to turn virtual calls to dictionary members into direct calls, but the JIT exceeded expectations and aggressively inlined those calls. The JIT failed to eliminate dictionary arguments that became dead after inlining; hoist dictionary allocations out of loops or do CSE on dictionary values. Fortunately, the latter two are suitable compiler optimizations.

[1] *Roslyn concepts fork, https://github.com/CaptainHayashi/roslyn.*

[2] *Roslyn https://github.com/dotnet/roslyn.*

[3] *Rust traits https://doc.rust-lang.org/book/traits.html.*

[4] *Swift https://swift.org.*

[5] *https://github.com/CaptainHayashi/roslyn/blob/master/concepts/docs/concepts.md.*

[6] *https://github.com/CaptainHayashi/visualfsharp/tree/hackathon-vs.*

[7] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in c++. OOPSLA '06, pages 291–310.

[8] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. PLDI '01, pages 1–12.

[9] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. OOPSLA '10, pages 341–360, 2010.

[10] S. Peyton Jones. *Haskell 98 language and libraries : the revised report.* Cambridge University Press, May 2003.

[11] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. POPL '89, pages 60–76.

[12] D. Yu, A. Kennedy, and D. Syme. Formalization of Generics for the .NET Common Language Runtime. POPL '04, pages 39–51.

Semantical Analysis of Type Classes

J. Garrett Morris

The University of Edinburgh

jgbm@acm.org

Background

Implicit polymorphism, as provided by Hindley-Milner type systems, provides a balance between the safety guarantees provided by strong typing and the convenience of generic programming. The type system is strong enough to guarantee that the evaluation of well-typed terms will not get stuck, while polymorphism and principal types allow programmers to reuse code and omit explicit type annotation. Type classes [5] play a similar role for overloading: they preserve strong typing (ruling out run-time failures from the use of overloaded symbols in undefined ways) without requiring that programmers explicitly disambiguate overloaded expressions. They are one of the defining features of the Haskell programming language: extensions of the core class system have been the subject of much research and vigorous debate, and uses of type classes range from simple overloading to maintaining complex type-encoded invariants. Type classes have been adopted in several other languages, including Isabelle [1] and Coq [4], and have inspired language features such as C++ concepts.

In their original presentation of type classes, Wadler and Blott give a translation from programs with class-based overloading to programs without overloading, guided by thetyping derivations in the original language. The role of the classes is implicit in the result of the translation, and thus in the meanings of programs. Semantic properties, either of class systems in general or of specific classes and their methods, can only be established indirectly (i.e., in terms of the translation).

Two Views of Type Classes

Consider an example of the use of type classes: the `elem` function, which determines whether a given value is an element of a list.

```
elem         :: Eq t ⇒ t → [t] → Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys
```

The class system plays two roles in this definition:

- In the type of `elem`, the predicate `Eq t` constrains the instantiation of type variable `t` to those types in the `Eq` class (i.e., those types for which the equality operator is defined).
- In the term `x == y`, the operator `==` is overloaded, with its implementation determined by the types of `x` and `y`.

These two roles are closely related: the witness that a type τ is in class `Eq` (i.e., that the predicate `Eq` τ holds) is the implementation of the `==` method. For example, in the dictionary-passing translation

Submitted to:
CoALP-Ty'16

proposed by Wadler and Blott [5], the definition above will be translated to one with an explicit parameter for the implementation of $==$. At each use of `elem`, the argument that supplies that implementation at a particular type τ is determined by the proof of $\texttt{Eq}\,\tau$. Despite this connection, most approaches to the semantics of languages with type classes focus only on the language after translation, at which point the role of the classes and their logical interpretation is only implicit. Semantic properties of either class systems in general or of specific classes and their methods can be established only indirectly (i.e., in terms of the translation).

Modeling Type Classes

We propose an alternative approach: we interpret type classes by their logical content, but without losing their term-level significance. To do so, we build on Kripke frame models [2], a standard approach to the model theory of intuitionistic and modal logics. A Kripke model includes a set of points, each of which represents a possible state of knowledge. To model type classes, we will consider points which include not only what is true (i.e., which predicates hold), but the witnesses of its truth (i.e., the implementations of the class methods). These models allow us to study both the meanings of classes in individual programs, and the properties of class systems as a whole. In particular, we identify four properties that characterize the relationship between a class system and its models. Model existence provides the foundation of our approach, guaranteeing that programs accepted by the compiler have models. Soundness and completeness relate the syntactic characterization of a class system to its semantics; they allow programmers to reason about programs without having to reason about the details of the class system implementation. Finally, monotonicity relates the models of individual modules to the models of a whole program, allowing programmers to reason about modules in isolation and assuring the coherence of the semantics of overloaded terms in different modules.

Our approach also allows us to relate class system features to standard logical connectives, by observing consequences of their modeling. For example, observing that instances actually introduce if-and-only-if constraints on their models justifies the context reduction algorithm used in Haskell '98 [3]. Observing that the modeling of functional dependencies is equivalent to modeling the refutation of classes of predicates, we can characterize an incompleteness in existing presentations of functional dependencies. Finally, we show that while we can view overlapping instances as introducing (biased) disjunction, the disjunction is degenerate because of the inexpressiveness of the remainder of the class system.

References

[1] F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In *TYPES 2006*, pages 160–174, Nottingham, UK, 2006. Springer.

[2] S. A. Kripke. Semantical analysis of modal logic I. Normal propositional calculi. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[3] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[4] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs 2008*, pages 278–293, Montreal, Canada, 2008. Springer.

[5] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76, Austin, Texas, USA, 1989. ACM.

Abstract compilation for type analysis of object-oriented languages

Davide Ancona
DIBRIS, Università di Genova, Italy
davide.ancona@unige.it

Abstract compilation is a technique for static type analysis which is based on coinductive Logic Programming, and consists in translating the code under analysis into a set of Horn clauses, and to solve queries w.r.t. the coinductive interpretation of the generated clauses.

Abstract compilation can be employed to perform precise type analysis of object-oriented languages, and support data polymorphism with recursive record and union types. To improve abstract compilation, two interesting research directions can be investigated: abstract compilation provides opportunities for reusing compilation technology, as the use of the Static Single Assignment intermediate representation, which allows more precise analysis; coinductive Constraint Logic Programming with a suitable notion of subtyping can further improve the accuracy of abstract compilation.

Introduction *Abstract compilation* [7, 5, 8] is a an approach to static type analysis aiming to reconcile compositional and global analysis through symbolic execution. Abstract compilation consists in translating the program under analysis in a more abstract type based representation expressed by a set of Horn clauses, and to solve an automatically generated goal w.r.t. the coinductive interpretation of such a translation.

Coinduction is required to properly deal with both recursive types, represented by cyclic terms, and mutually recursive user-defined functions/methods.

Abstract compilation allows modular analysis because several compilation schemes can be defined for the same language, each corresponding to a different kind of analysis, without changing the inference engine, which typically implements coinductive logic programming [12, 11, 6]. Compilation schemes based on union and record types have been studied to support parametric and data polymorphism (that is, polymorphic methods and fields) in object-oriented languages, and a smooth integration of nominal and structural types [7]; other proposed compilation schemes aim to detect uncaught exceptions [5], or to exploit advanced intermediate representations allowing more accurate analysis [8, 9].

Abstract compilation reconciles compositional and global analysis because on the one hand the source code of the program under analysis is compiled once and for all, hence preexisting source code needs not to be reinspected when new code is added, as happens in compositional analysis; on the other hand, context sensitive analysis is supported since different goals can be solved at different times for the same code fragment. Furthermore, with some limitations on analysis precision, compositional compilation can be achieved with more general non ground goals.

Another advantage of abstract compilation consists in its direct relationship with Prolog, and the natural support offered by this programming language for rapid prototyping of type analysis [1].

Intermediate code representations Recent compiler technology provides fast algorithms for translating source code into IR (Intermediate Representation), as, for instance, SSA (Static Single Assignment) form, particularly suitable for efficient static analysis.

Submitted to:
CALP-Ty 2016

Using SSA, and more advanced extensions such as SSI (Static Single Information), improves the precision of type analysis of local variables performed with abstract compilation [8, 9]; more recently, a high-level notion of SSA [3] has been introduced to simplify the compilation scheme used by abstract compilation.

To exploit SSA in abstract compilation, union types have to be introduced to be able to compile ϕ pseudo-functions that are introduced at program merge points during the translation into IR. In this way it is possible to infer different types for each occurrence of the same local variables.

SSI can further improve the analysis by refining the type of local variables in the branches of a program; to this aim, intersection and negations types are required.

Coinductive constraint logic programming with subtyping One of the limitations of abstract compilation based on coinductive Logic Programming is that in the presence of recursive methods the inference engine may fail to find a regular derivation even if the program under analysis is type safe.

To overcome this problem, one can exploit either structural resolution [10] to deal with non regular derivations, or subtyping constraints [5] to employ subsumption instead of simple term unification to make derivations regular. This latter solution allows exploitation of coinductive constraint logic programming with advanced forms of subtyping with union and intersection types [2, 4]; to this aim, predicates used to generate the logic program most curry variance annotations, and a sound decision procedure for satisfaction of subtyping constraints must be provided.

References

[1] K. Y. Ahn & A. Vezzosi (2016): *Executable Relational Specifications of Polymorphic Type Systems using Prolog*. In: *FLOPS 2016*, pp. 109–125.

[2] D. Ancona & A. Corradi (2014): *Sound and complete subtyping between coinductive types for object-oriented languages*. In: *ECOOP 2014*, pp. 282–307.

[3] D. Ancona & A. Corradi (2016): *A Formal Account of SSA in Java-like Languages*. In: *FTfJP'16*, pp. 2:1–2:8.

[4] D. Ancona & A. Corradi (2016): *Semantic subtyping for imperative object-oriented languages*. In: *OOPSLA 2016*, pp. 568–587.

[5] D. Ancona, A. Corradi, G. Lagorio & F. Damiani (2011): *Abstract compilation of object-oriented languages into coinductive CLP(X): can type inference meet verification?* In: *FoVeOOS 2010*, pp. 31–45.

[6] D. Ancona & A. Dovier (2015): *A Theoretical Perspective of Coinductive Logic Programming*. Fundamenta Informaticae 140(3-4), pp. 221–246.

[7] D. Ancona & G. Lagorio (2009): *Coinductive type systems for object-oriented languages*. In: *ECOOP 2009*, 5653, pp. 2–26.

[8] D. Ancona & G. Lagorio (2011): *Idealized coinductive type systems for imperative object-oriented programs*. RAIRO - Theoretical Informatics and Applications 45(1), pp. 3–33.

[9] D. Ancona & G. Lagorio (2012): *Static single information form for abstract compilation*. In: *IFIP TCS 2012*, pp. 10–27.

[10] E. Komendantskaya & P. Johann (2015): *Structural Resolution: a Framework for Coinductive Proof Search and Proof Construction in Horn Clause Logic*. CoRR abs/1511.07865.

[11] L. Simon, A. Bansal, A. Mallya & G. Gupta (2007): *Co-Logic Programming: Extending Logic Programming with Coinduction*. In: *ICALP 2007*, pp. 472–483.

[12] L. Simon, A. Mallya, A. Bansal & G. Gupta (2006): *Coinductive Logic Programming*. In: *ICLP 2006*, pp. 330–345.

Executable Specifications of Type Systems using Logic Programming

Ki Yung Ahn
School of Computer Science and Engineering
Nanyang Technological University, Singapore
kyagrd@gmail.com

Static types are considered indispensable in areas of software development where safety and performance are critical. On the other hand, there has been a strong preference for dynamic languages in certain domains (e.g., web development) where they traditionally valued programmer productivity over safety and performance. However, changes in the software landscape during the last two decades are demanding higher standards of safety and efficiency in web and mobile applications. As a result, it has become a recent trend trying to develop static type systems for dynamic languages, some of which are being supported by major corporations and the original language developer themselves, for instance, Flow type checker [3], TypeScript [6], and Typed Lua [5]. In other words, static type systems are being re-invented in the real-world software industry by the demands in practice.[1] Despite such demands, implementing type systems with rich modern features including type inference is still a challenge in practice (i.e., requires high development cost or man months) due to the lack of automated tools and systematic methods. In contrast, lexers and parsers have been automatically generated from declarative grammar specifications ever since lex and yacc were introduced in the 70's.

Applications of logic programming (LP) for type inference has been investigated long before this recent trend of re-inventing static type systems in "the real-world", at least from a decade ago (e.g., [7]). Some of the results from such investigations have been implemented in industry-strength compilers with advanced type systems such as the Glasgow Haskell Compiler [9]. However, we are still in lack of generic tools for deriving automated implementations of type systems from their declarative specifications, in contrast to various program generation tools (e.g., lex, yacc) widely available for automated implementations of syntax analysis from grammar specifications. In order to develop generic tools for automating type system implementations for modern programming languages, we take an approach summarized by the following steps:

1. Identify features of type systems that are commonly available (or frequently wishlisted) in modern programming languages.
2. Formulate specifications of pedagogical type systems for minimalistic languages that support subsets of those features using a widely available LP system such as Prolog.
3. Examine the advantages and the limitations of the LP system used for formulating the specifications of the type systems.
4. Design and develop an LP-based tool for type system specification, strengthening the advantages and overcoming the limitations.
5. Apply the tool to describe full features of the advanced type systems of modern programming languages and examine further advantages and shortcomings for improvements.

[1] These demands are, in fact, growing. For example, developers have been mentioning the need for more web programming languages [2] and such new languages are actually being developed and keep evolving on.

Submitted to:
SOS 2007

In our previous work [1], we have gone through several iterations of items 1, 2, and 3 over features of parametric polymorphism typically found in functional languages. In this work, we iterate through the same items 1, 2, and 3 over record types, more specifically, extensible records based on row polymorphism [4]. Because row polymorphism extends type-constructor polymorphism, we review the specification for type-constructor polymorphism. Then, we formulate a specification for a type system with extensible records. Next, we summarize how to represent order-irrelevant structures such as sets using membership constraints [8], especially in the context of applying such methods for record typing. A tutorial-style talk is to be presented in CoALP-Ty 2016 following through examples made available online at `http://kyagrd.github.io/tiper/`.

References

[1] Ki Yung Ahn & Andrea Vezzosi (2016): *Functional and Logic Programming: 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, chapter Executable Relational Specifications of Polymorphic Type Systems Using Prolog, p. 109125. Springer International Publishing, Cham, doi:10.1007/978-3-319-29604-3_8. Available at `http://dx.doi.org/10.1007/978-3-319-29604-3_8`.

[2] Gilad Bracha (2014): *Whither Web programming?* Keynote talk at QCon New York. Available at `https://qconnewyork.com/ny2014/keynote/whither-web-programming.html`.

[3] Facebook (2014): *Flow | a static type checker for javascript*. `http://flowtype.org/`. [Online: accessed 2016-05-05, first released in 2014].

[4] Benedict R. Gaster & Mark P. Jones (1996): *A Polymorphic Type System for Extensible Records and Variants*. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham.

[5] André Murbach Maidl, Fabio Mascarenhas & Roberto Ierusalimschy (2014): *Typed Lua: An Optional Type System for Lua*. In: *Proceedings of the Workshop on Dynamic Languages and Applications, Dyla 2014, Edinburgh, United Kingdom, June 9-11, 2014*, pp. 3:1–3:10.

[6] Microsoft (2012): *TypeScript - JavaScript that scales*. `http://www.typescriptlang.org/`.

[7] Martin Odersky, Martin Sulzmann & Martin Wehr (1999): *Type Inference with Constrained Types*. *Theor. Pract. Object Syst.* 5(1), pp. 35–55.

[8] Frieder Stolzenburg (1996): *Membership-Constraints and Complexity in Logic Programming with Sets*. In: *Frontiers in Combining Systems*, Kluwer Academic, pp. 285–302.

[9] Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers & Martin Sulzmann (2011): *Outsidein(x) Modular Type Inference with Local Assumptions*. *J. Funct. Program.* 21(4-5), pp. 333–412.

Structural Resolution for Abstract Compilation of Object-Oriented Languages

Luca Franceschini
Department of Informatics, Bioengineering, Robotics and Systems Engineering
University of Genoa
Italy
luca.franceschini@dibris.unige.it

Davide Ancona
davide.ancona@unige.it

Ekaterina Komendantskaya
Department of Computer Science
Heriot-Watt University
Edinburgh, Scotland
ek19@hw.ac.uk

We propose abstract compilation for precise static type analysis of object-oriented languages based on *coinductive logic programming*. Source code is translated to a logic program, then type-checking and inference problems amount to queries to be solved with respect to the resulting logic program. We exploit a coinductive semantics to deal with *infinite terms and proofs* produced by recursive types and methods. Thanks to the recent notion of *structural resolution* for coinductive logic programming, we are able to infer very precise type information, including a class of irrational recursive types causing non-termination for previously considered coinductive semantics.

Introduction

The use of dynamic languages is now widespread among developers, and the object-oriented community is no exception to this trend. The lack of type annotations and the dynamic nature of such languages is clearly a challenge for static type analysis.

The system we propose is based on *abstract compilation*: the source code is translated to a logic program, and after that type-checking and inference queries amount to queries (i.e., goals) against that logic program [2]. At this point different semantics give different results: the more powerful the resolution method is, the more programs we can successfully analyse. In the context of abstract compilation, a coinductive semantics is needed for type inference in presence of recursive types and methods. In this work we use *structural resolution* [6] (S-resolution, for short), a recently developed resolution method that advanced the state of the art for coinductive logic programming (under some assumptions).

Abstract compilation supports parametric polymorphism as well as data polymorphism [1]. Internally, structural types like union [4] and record types are used in order to analyse conditionals and objects in a precise way. Moreover, thanks to the type inference capabilities, the programmer is not required to write type annotations (though it is still possible).

Abstract compilation: a Simple Example

The following code is written in the object-oriented calculus similar to Featherweight Java [5], and it implements a very simple non-empty linked list:

```
class NEList extends List {
    Object el; List next;
    NEList(Object e, List n) {super(); el = e; next = n;}
    NEList add(Object e) {new NEList(e, this)} }
```

In the abstract compilation process each part of a program is translated to one or more Horn clauses. The `add` method, for instance, would generate the following clause[1]:

$$hasmeth(nelist, add, [E], nelist) \leftarrow typecomp(E, object) \wedge new(nelist, [E, nelist], nelist) \wedge override(list, add, [object], nelist)$$

[1] We use Prolog syntactic conventions: constants, function and predicate symbols start with a lowercase letter, variables start with an uppercase letter and lists are written as $[x_1, \ldots, x_n]$.

Submitted to:
CoALP-Ty'16 - Workshop on Coalgebra,
Horn Clause Logic Programming and Types

The clause can be understood as follows. For every type E the class `NEList` has a method `add` with two parameters of type E and `NEList`, respectively, returning an instance of `NEList`. This method is well-typed if and only if the following conditions hold: E is type-compatible with `Object`, the class `NEList` has a constructor accepting two arguments of type E and `NEList` respectively, and (in case of override) the new definition is compatible with the original one in the superclass.

Coinduction and Structural Resolution

Coinduction is needed for two reasons: first, we have to deal with *infinite terms* since recursive types are represented by cyclic terms in the logic program; second, type-checking of recursive methods and types requires *infinite proofs* as well. Suppose we add a method to the class `NEList` that builds a list with tail-recursion using an accumulating parameter (this time there are no type annotations):

```
buildList(n, l) {
    if (n ≤ 0) l
    else this.buildList(n−1, new NEList(n, l)) }
```

When the source method is recursive, the generated Horn clause is recursive as well:

$$\begin{aligned} \mathit{hasmeth}(\mathit{nelist}, \mathit{buildlist}, [\mathit{int}, L], L \vee R) \leftarrow\ & \mathit{new}(\mathit{nelist}, [\mathit{int}, L], R') \\ & \wedge\ \mathit{hasmeth}(\mathit{nelist}, \mathit{buildlist}, [\mathit{int}, R'], R) \wedge \cdots \end{aligned}$$

We use the *union type* $T_1 \vee T_2$ for conditional expressions since we do not know which path will be taken. The *new* atom is generated from the **`new`** expression while the *hasmeth* atom is derived from the recursive call (we ignore the other constraints for space reasons). At the logic programming level the recursion has no base case, thus a coinductive interpretation is needed. Past work proposed the use of *coinductive logic programming* [7], but it has an important limitation: it can only deal with *rational* [3] (i.e., *cyclic*) terms and proofs. In this case there cannot be any cyclic proof since, at each step, we will deal with different types (thus, terms) that will not unify with each other, and the resolution will not terminate.

We propose the use of *S-resolution* to overtake this restriction since it is not limited by rationality. In this case S-resolution will automatically formulate a suitable *coinductive hypothesis* and it will terminate in finite time. However, since the result type we are trying to infer is not rational (the infinite union of the types of all integer lists of any length) such term cannot be finitely represented. Instead, S-resolution will act as a *lazy* procedure, giving a *partial result* that can be subsequently unfolded as much as needed.

References

[1] Ole Agesen (1995): *The Cartesian Product Algorithm*, pp. 2–26. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/3-540-49538-X˙2.

[2] Davide Ancona & Giovanni Lagorio (2009): *Coinductive Type Systems for Object-Oriented Languages*, pp. 2–26. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-03013-0˙2.

[3] Bruno Courcelle (1983): *Fundamental properties of infinite trees*. *Theoretical Computer Science* 25(2), pp. 95 – 169, doi:http://dx.doi.org/10.1016/0304-3975(83)90059-2.

[4] Atsushi Igarashi & Hideshi Nagira (2006): *Union Types for Object-oriented Programming*. In: *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, ACM, New York, NY, USA, pp. 1435–1441, doi:10.1145/1141277.1141610.

[5] Atsushi Igarashi, Benjamin C. Pierce & Philip Wadler (2001): *Featherweight Java: A Minimal Core Calculus for Java and GJ*. *ACM Trans. Program. Lang. Syst.* 23(3), pp. 396–450, doi:10.1145/503502.503505.

[6] Ekaterina Komendantskaya & Patricia Johann (2015): *Structural Resolution: a Framework for Coinductive Proof Search and Proof Construction in Horn Clause Logic*. *CoRR* abs/1511.07865. Available at `http://arxiv.org/abs/1511.07865`.

[7] Luke Simon, Ajay Mallya, Ajay Bansal & Gopal Gupta (2006): *Coinductive Logic Programming*, pp. 330–345. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/11799573˙25.