Bibliography

March 28, 2017

Contents

J. Bernardy, P. Jansson, M. Zalewski, *et al.*
Generic programming with C++ concepts and Haskell type classes - a comparison

R. S. Bird, J. Gibbons, S. Mehner, *et al.*
Understanding idiomatic traversals backwards and forwards

F. Bonchi and F. Zanasi
Bialgebraic Semantics for Logic Programming

A. Colmerauer
Equations and Inequations on Finite and Infinite Trees

P. Cousot and R. Cousot
Inductive Definitions, Semantics and Abstract Interpretation

K. Crary, R. Harper, and S. Puri
What is a Recursive Module?

D. Van Dalen
Intuitionistic Logic

L. Damas and R. Milner
Principal Type-Schemes for Functional Programs

N. A. Danielsson, J. Hughes, P. Jansson, *et al.*
Fast and loose reasoning is morally correct

E. DE ANGELIS, F. FIORAVANTI, A. PETTOROSSI, *et al.*
Proving correctness of imperative programs by linearizing constrained Horn clauses

D. De Schreye, V. Nys, and C. Nicholson
Analysing and Compiling Coroutines with Abstract Conjunctive Partial Deduction

D. Devriese and F. Piessens
On the bright side of type classes: instance arguments in Agda

A. Dijkstra, J. Fokker, and S. D. Swierstra
The Structure of the Essential Haskell Compiler, or Coping with Compiler Complexity

M. Falaschi, G. Levi, C. Palamidessi, *et al.*
Declarative Modeling of the Operational Behavior of Logic Languages

F. Farka, E. Komendantskaya, K. Hammond, *et al.*
Coinductive Soundness of Corecursive Type Class Resolution

K. Faxén
A static semantics for Haskell

P. Fu and E. Komendantskaya
A Type-Theoretic Approach to Resolution

P. Fu and E. Komendantskaya
Operational semantics of resolution and productivity in Horn clause logic

P. Fu, E. Komendantskaya, T. Schrijvers, *et al.*
Proof Relevant Corecursive Resolution

N. Ghani and P. Hancock
Containers, monads and induction recursion

J. Gibbons and G. Hutton
Proof Methods for Corecursive Programs

G. Gonthier, B. Ziliani, A. Nanevski, *et al.*
How to make ad hoc proof automation less ad hoc

G. Gupta, A. Bansal, R. Min, *et al.*
Coinductive Logic Programming and Its Applications

C. V. Hall, K. Hammond, S. L. P. Jones, *et al.*
Type Classes in Haskell

R. Hinze and S. L. P. Jones
Derivable Type Classes

R. Hinze and S. Peyton Jones
Derivable Type Classes

R. Hinze, N. Wu, and J. Gibbons
Unifying structured recursion schemes

J. S. Hodas and D. Miller
Logic Programming in a Fragment of Intuitionistic Linear Logic

P. Hudak, J. Hughes, S. L. P. Jones, *et al.*
A history of Haskell: being lazy with class

G. Huet and A. Saibi
Constructive Category Theory

R. Iemhoff
On The Admissible Rules of Intuitionistic Propositional Logic

J. Jaffar and P. J. Stuckey
Semantics of Infinite Tree Logic Programming

M. Jaskelioff and O. Rypacek
An Investigation of the Laws of Traversals

P. Johann and N. Ghani
A principled approach to programming with nested types in Haskell

R. van Kesteren, M. C.J. D. van Eekelen, and M. de Mol
Proof support for generic type classes

O. Kiselyov and H. Ishii
Freer monads, more extensible effects

P. Kokke and W. Swierstra
Auto in Agda - Programming Proof Search Using Reflection

E. Komendantskaya and J. Power
Coalgebraic Semantics for Derivations in Logic Programming

D. Kozen and A. Silva
Practical coinduction

R. Lämmel and S. L. P. Jones
Scrap your boilerplate with class: extensible generic functions

C. S. Lee, N. D. Jones, and A. M. Ben-Amram
The size-change principle for program termination

M. Lenisa, J. Power, and H. Watanabe
Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads

F. Lindblad and M. Benke
A Tool for Automated Theorem Proving in Agda

J. W. Lloyd
Foundations of Logic Programming, 2nd Edition

A. Löh and R. Hinze
Open data types and open functions

D. B. MacQueen, G. D. Plotkin, and R. Sethi
An Ideal Model for Recursive Polymorphic Types

C. McBride
Faking it: Simulating dependent types in Haskell

C. McBride and R. Paterson
Applicative programming with effects

D. Miller and G. Nadathur
Programming with Higher-Order Logic

D. Miller, G. Nadathur, F. Pfenning, *et al.*
Uniform Proofs as a Foundation for Logic Programming

R. Milner
A Theory of Type Polymorphism in Programming

R. Milner
Communication and concurrency

I. Moerdijk and E. Palmgren
Wellfounded trees in categories

J. H. Morris
Lambda-calculus models of programming languages

L. S. Moss and N. Danner
On the Foundations of Corecursion

U. Norell
Dependently typed programming in agda

B. C. d. S. Oliveira, T. Schrijvers, W. Choi, *et al.*
The implicit calculus: a new foundation for generic programming

L. C. Paulson and A. W. Smith
Logic Programming, Functional Programming, and Inductive Definitions

B. C. Pierce
Types and programming languages

A. M. Pitts
Parametric Polymorphism and Operational Equivalence

G. D. Plotkin and J. Power
Algebraic Operations and Generic Effects

J. Power and H. Watanabe
Combining a monad and a comonad

G. Rosu and D. Lucanu
Circular Coinduction: A Proof Theoretical Foundation

J. J.M. M. Rutten
Behavioural differential equations: A coinductive calculus of streams, automata, and power series

D. Sangiorgi
On the origins of bisimulation and coinduction

D. D. Schreye, V. Nys, and C. J. Nicholson
Analysing and Compiling Coroutines with Abstract Conjunctive Partial Deduction

T. Schrijvers, S. L. P. Jones, M. M. T. Chakravarty, *et al.*
Type checking with open type functions

L. Simon, A. Bansal, A. Mallya, *et al.*
Co-Logic Programming: Extending Logic Programming with Coinduction

A. Simpson and G. Plotkin
Complete axioms for categorical fixed-point operators

S. Staton
An Algebraic Presentation of Predicate Logic - (Extended Abstract)

P. J. Stuckey and M. Sulzmann
A theory of overloading

M. Sulzmann, G. J. Duck, S. L. P. Jones, *et al.*
Understanding functional dependencies via constraint handling rules

G. Sutcliffe
The TPTP problem library and associated infrastructure : the FOF and CNF Parts, v3.5.0

H. Thielemann
How to Refine Polynomial Functions

D. Vytiniotis, S. L. P. Jones, T. Schrijvers, *et al.*
OutsideIn(X) Modular type inference with local assumptions

P. Wadler
Theorems for Free!

P. Wadler and S. Blott
How to Make ad-hoc Polymorphism Less ad-hoc

N. Bjørner, A. Gurfinkel, K. McMillan, *et al.*
Horn Clause Solvers for Program Verification

H. Geuvers and R. Nederpelt
N.G. de Bruijn's contribution to the formalization of mathematics

R. Jhala, R. Majumdar, and A. Rybalchenko
HMC : Verifying Functional Programs

E. Komendantskaya and J. Power
Logic programming: laxness and saturation

E. Komendantskaya and J. Power
Logic programming: laxness and saturation

N. P. Mendler, P. Panangaden, P. J. Scott, *et al.*
A Logical View of Concurrent Constraint Programming

M. Odersky, M. Sulzmann, and M. Wehr
Type inference with constrained types

P. W. O'Hearn and R. D. Tennent
Parametricity and local variables

C.-H. L. Ong and S. J. Ramsay
Verifying higher-order functional programs with pattern-matching algebraic data types

F. Pfenning
Logic programming in the LF logical framework

F. Pfenning and C. Schürmann
System description: Twelf a meta-logical framework for deductive systems

V. Simonet and F. Pottier
A constraint-based approach to guarded algebraic data types

M. Abadi and M. P. Fiore

Syntactic Considerations on Recursive Types

Abstract We study recursive types from a syntactic perspective. In particular, we compare the formulations of recursive types that are used in programming languages and formal systems. Our main tool is a new syntactic explanation of type expressions as functors. We QkO introduce Q sample logic for programs with recursive types in which we carry out our proof

References

M. Abadi and M. P. Fiore, "Syntactic considerations on recursive types," in *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, IEEE Computer Society, 1996, pp. 242–252, ISBN: 0-8186-7463-6. DOI: 10.1109/LICS.1996.561324. [Online]. Available: http://dx.doi.org/10.1109/LICS.1996.561324

A. Abel and B. Pientka

Wellfounded recursion with copatterns: a unified approach to termination and productivity

Abstract We present a core programming language that supports writing well-founded structurally recursive functions using simultaneous pattern matching on contextual LF objects and contexts. The main technical tool is a coverage checking algorithm that also generates valid recursive calls. To establish consistency, we define a call-by-value small-step semantics and prove that every well-typed program terminates using a reducibility semantics. Based on the presented methodology we have implemented a totality checker as part of the programming and proof environment Beluga where it can be used to establish that a total Beluga program corresponds to a proof.

References

A. Abel and B. Pientka, "Wellfounded recursion with copatterns: A unified approach to termination and productivity," in *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, G. Morrisett and T. Uustalu, Eds., ACM, 2013, pp. 185–196, ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500591. [Online]. Available: http://doi.acm.org/10.1145/2500365.2500591

A. Abel, B. Pientka, D. Thibodeau, *et al.*

Copatterns: programming infinite structures by observations

Abstract Inductive datatypes provide mechanisms to define finite data such as finite lists and trees via constructors and allow programmers to analyze and manipulate finite data via pattern matching. In this paper, we develop a dual approach for working with infinite data structures such as streams. Infinite data inhabits coinductive datatypes which denote greatest fixpoints. Unlike finite data which is defined by constructors we define infinite data by observations. Dual to pattern matching, a tool for analyzing finite data, we develop the concept of copattern matching, which allows us to synthesize infinite data. This leads to a symmetric language design where pattern matching on finite and infinite data can be mixed. We present a core language for programming with infinite structures by observations together with its operational semantics based on (co)pattern matching and describe coverage of copatterns. Our language naturally supports both call-by-name and call-by-value interpretations and can be seamlessly integrated into existing languages like Haskell and ML. We prove type soundness for our language and sketch how copatterns open new directions for solving problems in the interaction of coinductive and dependent types

References

A. Abel, B. Pientka, D. Thibodeau, *et al.*, "Copatterns: Programming infinite structures by observations," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, R. Giacobazzi and R. Cousot, Eds., ACM, 2013, pp. 27–38, ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429075. [Online]. Available: http://doi.acm.org/10.1145/2429069.2429075

S. Abramsky and N. Tzevelekos

Introduction to Categories and Categorical Logic

Abstract The aim of these notes is to provide a succinct, accessible introduction to some of the basic ideas of category theory and categorical logic. The notes are based on a lecture course given at Oxford over the past few years. They contain numerous exercises, and hopefully will prove useful for self-study by those seeking a first introduction to the subject, with fairly minimal prerequisites. The coverage is by no means comprehensive, but should provide a good basis for further study; a guide to further reading is included. The main prerequisite is a basic familiarity with the elements of discrete mathematics: sets, relations and functions. An Appendix contains a summary of what we will need, and it may be useful to review this first. In addition, some prior exposure to abstract algebra - vector spaces and linear maps, or groups and group homomorphisms - would be helpful.

References

S. Abramsky and N. Tzevelekos, "Introduction to categories and categorical logic," *CoRR*, vol. abs/1102.1313, 2011. [Online]. Available: `http://arxiv.org/abs/1102.1313`

J. Adámek and V. Koubek

Least Fixed Point of a Functor

Abstract no abstract provided

References

J. Adámek and V. Koubek, "Least fixed point of a functor," *J. Comput. Syst. Sci.*, vol. 19, no. 2, pp. 163–178, 1979. DOI: 10.1016/0022-0000(79)90026-6. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(79)90026-6

J. Adámek and V. Koubek

On the Greatest Fixed Point of a Set Functor

Abstract The greatest fixed point of a set functor is proved to be (a) a metric completion and (b) a CPO-completion of finite iterations. For each (possibly infinitary) signature Σ the terminal Σ-coalgebra is thus proved to be the coalgebra of all Σ-labelled trees; this is the completion of the set of all such trees of finite depth. A set functor is presented which has a fixed point but does not have a greatest fixed point. A sufficient condition for the existence of a greatest fixed point is proved: the existence of two fixed points of successor cardinalities.

References

J. Adámek and V. Koubek, "On the greatest fixed point of a set functor," *Theor. Comput. Sci.*, vol. 150, no. 1, pp. 57–75, 1995. DOI: 10.1016/0304-3975(95)00011-K. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(95)00011-K

D. Ancona and A. Dovier

A Theoretical Perspective of Coinductive Logic Programming

Abstract In this paper we study the semantics of Coinductive Logic Programming and clarify its intrinsic computational limits, which prevent, in particular, the definition of a complete, computable, operational semantics. We propose a new operational semantics that allows a simple correctness result and the definition of a simple meta-interpreter. We compare, and prove the equivalence, with the operational semantics defined and used in other papers on this topic.

References

D. Ancona and A. Dovier, "A theoretical perspective of coinductive logic programming," *Fundam. Inform.*, vol. 140, no. 3-4, pp. 221–246, 2015. DOI: 10.3233/FI-2015-1252. [Online]. Available: http://dx.doi.org/10.3233/FI-2015-1252

D. Ancona and G. Lagorio

Idealized coinductive type systems for imperative object-oriented programs

Abstract In recent work we have proposed a novel approach to define idealized type systems for object-oriented languages, based on abstract compilation of programs into Horn formulas which are interpreted w.r.t. the coinductive (that is, the greatest) Herbrand model. In this paper we investigate how this approach can be applied also in the presence of imperative features. This is made possible by considering a natural translation of Static Single Assignment intermediate form programs into Horn formulas, where functions correspond to union types.

References

D. Ancona and G. Lagorio, "Idealized coinductive type systems for imperative object-oriented programs," *RAIRO - Theor. Inf. and Applic.*, vol. 45, no. 1, pp. 3–33, 2011. DOI: 10.1051/ita/2011009. [Online]. Available: http://dx.doi.org/10.1051/ita/2011009

E. De Angelis, F. Fioravanti, A. Pettorossi, *et al.*

Proving correctness of imperative programs by linearizing constrained Horn clauses

Abstract We present a method for verifying the correctness of imperative programs which is based on the automated transformation of their specifications. Given a program prog, we consider a partial correctness specification of the form $\{\phi\}$, prog $\{\psi\}$, where the assertions and are predicates defined by a set Spec of possibly recursive Horn clauses with linear arithmetic (LA) constraints in their premise (also called constrained Horn clauses). The verification method consists in constructing a set PC of constrained Horn clauses whose satisfiability implies that $\{\phi\}$, prog, $\{\psi\}$ is valid. We highlight some limitations of state-of-the-art constrained Horn clause solving methods, here called LA-solving methods, which prove the satisfiability of the clauses by looking for linear arithmetic interpretations of the predicates. In particular, we prove that there exist some specifications that cannot be proved valid by any of those LA-solving methods. These specifications require the proof of satisfiability of a set PC of constrained Horn clauses that contain nonlinear clauses (that is, clauses with more than one atom in their premise). Then, we present a transformation, called linearization, that converts PC into a set of linear clauses (that is, clauses with at most one atom in their premise). We show that several specifications that could not be proved valid by LA-solving methods, can be proved valid after linearization. We also present a strategy for performing linearization in an automatic way and we report on some experimental results obtained by using a preliminary implementation of our method.

References

E. De Angelis, F. Fioravanti, A. Pettorossi, *et al.*, "Proving correctness of imperative programs by linearizing constrained horn clauses," *TPLP*, vol. 15, no. 4-5, pp. 635–650, 2015. DOI: 10.1017/S1471068415000289. [Online]. Available: http://dx.doi.org/10.1017/S1471068415000289

R. Atkey

What is a Categorical Model of Arrows?

Abstract We investigate what the correct categorical formulation of Hughes Arrows should be. It has long been folklore that Arrows, a functional programming construct, and Freyd categories, a categorical notion due to Power, Robinson and Thielecke, are somehow equivalent. In this paper, we show that the situation is more subtle. By considering Arrows wholly within the base category we derive two alternative formulations of Freyd category that are equivalent to Arrowsenriched Freyd categories and indexed Freyd categories. By imposing a further condition, we characterise those indexed Freyd categories that are isomorphic to Freyd categories. The key differentiating point is the number of inputs available to a computation and the structure available on them, where structured input is modelled using comonads.

References

R. Atkey, "What is a categorical model of arrows?" *Electr. Notes Theor. Comput. Sci.*, vol. 229, no. 5, pp. 19–37, 2011. DOI: 10.1016/j.entcs.2011.02.014. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2011.02.014

author
Algebraic and Coalgebraic Methods in the Mathematics of Program Construction

Abstract Program construction is about turning specifications of computer software into implementations. Doing so in a way that guarantees correctness is an undertaking requiring deep understanding of the languages and tools being used, as well as of the application domain. Recent research aimed at improving the process of program construction exploits insights from abstract algebraic tools such as lattice theory, fixpoint calculus, universal algebra, category theory and allegory theory. This book provides an introduction to these mathematical theories and how they are applied to practical problems.

References

R. Backhouse, R. Crole, and J. Gibbons, Eds., *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2002, vol. 2297, ISBN: 3540436138. [Online]. Available: `http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/acmmpc-toc.pdf`

H. P. Barendregt

Functional Programming and Lambda Calculus

Abstract An abstract is not available

References

H. P. Barendregt, "Functional programming and lambda calculus," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 1990, pp. 321–363

H. Basold and H. H. Hansen

Well-definedness and observational equivalence for inductivecoinductive programs

Abstract **abstract**

References

H. Basold and H. H. Hansen, "Well-definedness and observational equivalence for inductivecoinductive programs," *Journal of Logic and Computation*, exw091, 2016, ISSN: 0955-792X. DOI: 10.1093/logcom/exv091. [Online]. Available: https://academic.oup.com/logcom/article-lookup/doi/10.1093/logcom/exv091

M. Bellia and G. Levi

The Relation between Logic and Functional Languages: A Survey

Abstract The paper considers different methods of integrating the functional and logic programming paradigms, starting with the identification of their semantic differences. The main methods to extend functional programs with logic features (i.e. unification) are then considered. These include narrowing, completion, SLD-resolution of equational formulas, and set abstraction. The different techniques are analyzed from several viewpoints, including the ability to support both paradigms, lazy evaluation, and concurrency.

References

M. Bellia and G. Levi, "The relation between logic and functional languages: A survey," *J. Log. Program.*, vol. 3, no. 3, pp. 217–236, 1986. DOI: 10.1016/0743-1066(86)90014-2. [Online]. Available: http://dx.doi.org/10.1016/0743-1066(86)90014-2

B. van den Berg and F. D. Marchi

Non-well-founded trees in categories

Abstract Non-well-founded trees are used in mathematics and computer science, for modelling non-well-founded sets, as well as non-terminating processes or infinite data structures. Categorically, they arise as final coalgebras for polynomial endofunctors, which we call M-types. We derive existence results for M-types in locally cartesian closed pretoposes with a natural numbers object, using their internal logic. These are then used to prove stability of such categories with M-types under various topos-theoretic constructions; namely, slicing, formation of coalgebras (for a cartesian comonad), and sheaves for an internal site.

References

B. van den Berg and F. D. Marchi, "Non-well-founded trees in categories," *Ann. Pure Appl. Logic*, vol. 146, no. 1, pp. 40–59, 2007. DOI: 10.1016/j.apal.2006.12.001. [Online]. Available: http://dx.doi.org/10.1016/j.apal.2006.12.001

J. Bernardy, P. Jansson, M. Zalewski, *et al.*

Generic programming with C++ concepts and Haskell type classes - a comparison

Abstract Earlier studies have introduced a list of high-level evaluation criteria to assess how well a language supports generic programming. Languages that meet all criteria include Haskell because of its type classes and C++ with the concept feature. We refine these criteria into a taxonomy that captures commonalities and differences between type classes in Haskell and concepts in C++ and discuss which differences are incidental and which ones are due to other language features. The taxonomy allows for an improved understanding of language support for generic programming, and the comparison is useful for the ongoing discussions among language designers and users of both languages.

References

J. Bernardy, P. Jansson, M. Zalewski, *et al.*, "Generic programming with C++ concepts and haskell type classes - a comparison," *J. Funct. Program.*, vol. 20, no. 3-4, pp. 271–302, 2010. DOI: 10.1017/S095679681000016X. [Online]. Available: http://dx.doi.org/10.1017/S095679681000016X

R. S. Bird, J. Gibbons, S. Mehner, *et al.*

Understanding idiomatic traversals backwards and forwards

Abstract We present new ways of reasoning about a particular class of effectful Haskell programs, namely those expressed as idiomatic traversals. Starting out with a specific problem about labelling and unlabelling binary trees, we extract a general inversion law, applicable to any monad, relating a traversal over the elements of an arbitrary traversable type to a traversal that goes in the opposite direction. This law can be invoked to show that, in a suitable sense, unlabelling is the inverse of labelling. The inversion law, as well as a number of other properties of idiomatic traversals, is a corollary of a more general theorem characterising traversable functors as finitary containers: an arbitrary traversable object can be decomposed uniquely into shape and contents, and traversal be understood in terms of those. Proof of the theorem involves the properties of traversal in a special idiom related to the free applicative functor.

References

R. S. Bird, J. Gibbons, S. Mehner, *et al.*, "Understanding idiomatic traversals backwards and forwards," in *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, C. Shan, Ed., ACM, 2013, pp. 25–36, ISBN: 978-1-4503-2383-3. DOI: 10.1145/2503778.2503781. [Online]. Available: http://doi.acm.org/10.1145/2503778.2503781

F. Bonchi and F. Zanasi

Bialgebraic Semantics for Logic Programming

Abstract Bialgebrae provide an abstract framework encompassing the semantics of different kinds of computational models. In this paper we propose a bialgebraic approach to the semantics of logic programming. Our methodology is to study logic programs as reactive systems and exploit abstract techniques developed in that setting. First we use saturation to model the operational semantics of logic programs as coalgebrae on presheaves. Then, we make explicit the underlying algebraic structure by using bialgebrae on presheaves. The resulting semantics turns out to be compositional with respect to conjunction and term substitution. Also, it encodes a parallel model of computation, whose soundness is guaranteed by a built-in notion of synchronisation between different threads.

References

F. Bonchi and F. Zanasi, "Bialgebraic semantics for logic programming," *Logical Methods in Computer Science*, vol. 11, no. 1, 2015. DOI: 10.2168/LMCS-11(1:14)2015. [Online]. Available: http://dx.doi.org/10.2168/LMCS-11(1:14)2015

A. Colmerauer

Equations and Inequations on Finite and Infinite Trees

Abstract No abstract available

References

A. Colmerauer, "Equations and inequations on finite and infinite trees," in *FGCS*, 1984, pp. 85–99

P. Cousot and R. Cousot

Inductive Definitions, Semantics and Abstract Interpretation

Abstract We introduce and illustrate a specification method combining rule-based inductive definitions, well-founded induction principles, fixed-point theory and abstract interpretation for general use in computer science. Finite as well as infinite objects can be specified, at various levels of details related by abstraction. General proof principles are applicable to prove properties of the specified objects. The specification method is illustrated by introducing $G\infty SOS$, a structured operational semantics generalizing Plotkin's [28] structured operational semantics (SOS) so as to describe the finite, as well as the infinite behaviors of programs in a uniform way and by constructively deriving inductive presentations of the other (relational, denotational, predicate transformers, ...) semantics from $G\infty SOS$ by abstract interpretation.

References

P. Cousot and R. Cousot, "Inductive definitions, semantics and abstract interpretation," in *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, R. Sethi, Ed., ACM Press, 1992, pp. 83–94, ISBN: 0-89791-453-8. DOI: 10.1145/143165.143184. [Online]. Available: http://doi.acm.org/10.1145/143165.143184

K. Crary, R. Harper, and S. Puri

What is a Recursive Module?

Abstract A hierarchical module system is an effective tool for structuring large programs. Strictly hierarchical module systems impose an acyclic ordering on import dependencies among program units. This can impede modular programming by forcing mutually-dependent components to be consolidated into a single module. Recently there have been several proposals for module systems that admit cyclic dependencies, but it is not clear how these proposals relate to one another, nor how one might integrate them into an expressive module system such as that of ML.To address this question we provide a type-theoretic analysis of the notion of a recursive module in the context of a "phase-distinction" formalism for higher-order module systems. We extend this calculus with a recursive module mechanism and a new form of signature, called a recursively dependent signature, to support the definition of recursive modules. These extensions are justified by an interpretation in terms of more primitive language constructs. This interpretation may also serve as a guide for implementation.

References

K. Crary, R. Harper, and S. Puri, "What is a recursive module?" In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, B. G. Ryder and B. G. Zorn, Eds., ACM, 1999, pp. 50–63, ISBN: 1-58113-094-5. DOI: 10.1145/301618.301641. [Online]. Available: http://doi.acm.org/10.1145/301618.301641

D. Van Dalen
Intuitionistic Logic

Abstract Among the logics that deal with the familiar connectives and quantifiers two stand out as having a solid philosophicalmathematical justification. On the one hand there is classical logic with its ontological basis and on the other hand intuitionistic logic with its epistemic motivation. The case for other logics is considerably weaker; although one may consider intermediate logics with more or less plausible principles from certain viewpoints none of them is accompanied by a comparably compelling philosophy. For this reason we have mostly paid attention to pure intuitionistic theories.

References

D. Van Dalen, "Intuitionistic logic," in *Handbook of Philosophical Logic: Volume III: Alternatives in Classical Logic*, D. Gabbay and F. Guenthner, Eds. Dordrecht: Springer Netherlands, 1986, pp. 225–339, ISBN: 978-94-009-5203-4. DOI: 10.1007/978-94-009-5203-4_4. [Online]. Available: http://dx.doi.org/10.1007/978-94-009-5203-4_4

L. Damas and R. Milner

Principal Type-Schemes for Functional Programs

Abstract No abstract available

References

L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, R. A. DeMillo, Ed., ACM Press, 1982, pp. 207–212, ISBN: 0-89791-065-6. DOI: 10.1145/582153.582176. [Online]. Available: http://doi.acm.org/10.1145/582153.582176

N. A. Danielsson, J. Hughes, P. Jansson, *et al.*

Fast and loose reasoning is morally correct

Abstract Functional programmers often reason about programs as if they were written in a total language, expecting the results to carry over to non-total (partial) languages. We justify such reasoning.Two languages are defined, one total and one partial, with identical syntax. The semantics of the partial language includes partial and infinite values, and all types are lifted, including the function spaces. A partial equivalence relation (PER) is then defined, the domain of which is the total subset of the partial language. For types not containing function spaces the PER relates equal values, and functions are related if they map related values to related values.It is proved that if two closed terms have the same semantics in the total language, then they have related semantics in the partial language. It is also shown that the PER gives rise to a bicartesian closed category which can be used to reason about values in the domain of the relation.

References

N. A. Danielsson, J. Hughes, P. Jansson, *et al.*, "Fast and loose reasoning is morally correct," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. G. Morrisett and S. L. P. Jones, Eds., ACM, 2006, pp. 206–217, ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111056. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111056

E. DE ANGELIS, F. FIORAVANTI,
A. PETTOROSSI, *et al.*

Proving correctness of imperative programs by linearizing constrained Horn clauses

Abstract We present a method for verifying the correctness of imperative programs which is based on the automated transformation of their specifications. Given a program prog , we consider a partial correctness specification of the form {}, prog {ψ}, where the assertions and ψ are predicates defined by a set Spec of possibly recursive Horn clauses with linear arithmetic (LA) constraints in their premise (also called constrained Horn clauses). The verification method consists in constructing a set PC of constrained Horn clauses whose satisfiability implies that {}, prog , {ψ} is valid. We highlight some limitations of state-of-the-art constrained Horn clause solving methods, here called LA-solving methods , which prove the satisfiability of the clauses by looking for linear arithmetic interpretations of the predicates. In particular, we prove that there exist some specifications that cannot be proved valid by any of those LA -solving methods. These specifications require the proof of satisfiability of a set PC of constrained Horn clauses that contain nonlinear clauses (that is, clauses with more than one atom in their premise). Then, we present a transformation, called linearization , that converts PC into a set of linear clauses (that is, clauses with at most one atom in their premise). We show that several specifications that could not be proved valid by LA -solving methods, can be proved valid after linearization. We also present a strategy for performing linearization in an automatic way and we report on some experimental results obtained by using a preliminary implementation of our method.

References

E. DE ANGELIS, F. FIORAVANTI, A. PETTOROSSI, *et al.*, "Proving correctness of imperative programs by linearizing constrained horn clauses," *Theory and Practice of Logic Programming*, vol. 15, no. 4-5, pp. 635–650, 2015, ISSN: 1471-0684. DOI: 10.1017/S1471068415000289. [Online]. Available: http://dx.doi.org/10.1017/S1471068415000289http://www.journals.cambridge.org/abstract{_}S1471068415000289

D. De Schreye, V. Nys, and C. Nicholson

Analysing and Compiling Coroutines with Abstract Conjunctive Partial Deduction

Abstract We provide an approach to formally analyze the computa- tional behavior of coroutines in Logic Programs and to compile these computations into new programs, not requiring any support for corou- tines. The problem was already studied near to 30 years ago, in an analy- sis and transformation technique called Compiling Control. However, this technique had a strong ad hoc flavor: the completeness of the analysis was not well understood and its symbolic evaluation was also very ad hoc. We show how Abstract Conjunctive Partial Deduction, introduced by Leuschel in 2004, provides an appropriate setting to redefine Com- piling Control. Leuschel's framework is more general than the original formulation, it is provably correct, and it can easily be applied for simple examples. We also show that the Abstract Conjunctive Partial Deduc- tion framework needs some further extension to be able to deal with more complex examples.

References

D. De Schreye, V. Nys, and C. Nicholson, "Analysing and compiling coroutines with abstract conjunctive partial deduction," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8981, 2015, pp. 21–38, ISBN: 9783319178219. DOI: 10.1007/978-3-319-17822-6_2. [Online]. Available: http://link.springer.com/10.1007/978-3-319-17822-6{_}2

D. Devriese and F. Piessens

On the bright side of type classes: instance arguments in Agda

Abstract We present instance arguments: an alternative to type classes and related features in the dependently typed, purely functional programming language/proof assistant Agda. They are a new, general type of function arguments, resolved from call-site scope in a type-directed way. The mechanism is inspired by both Scala's implicits and Agda's existing implicit arguments, but differs from both in important ways. Our mechanism is designed and implemented for Agda, but our design choices can be applied to other programming languages as well.

Like Scala's implicits, we do not provide a separate structure for type classes and their instances, but instead rely on Agda's standard dependently typed records, so that standard language mechanisms provide features that are missing or expensive in other proposals. Like Scala, we support the equivalent of local instances. Unlike Scala, functions taking our new arguments are first-class citizens and can be abstracted over and manipulated in standard ways. Compared to other proposals, we avoid the pitfall of introducing a separate type-level computational model through the instance search mechanism. All values in scope are automatically candidates for instance resolution. A final novelty of our approach is that existing Agda libraries using records gain the benefits of type classes without any modification.

We discuss our implementation in Agda (to be part of Agda 2.2.12) and we use monads as an example to show how it allows existing concepts in the Agda standard library to be used in a similar way as corresponding Haskell code using type classes. We also demonstrate and discuss equivalents and alternatives to some advanced type class-related patterns from the literature and some new patterns specific to our system.

References

D. Devriese and F. Piessens, "On the bright side of type classes: Instance arguments in agda," in *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, M. M. T. Chakravarty, Z. Hu, and O. Danvy, Eds., ACM, 2011, pp. 143–155, ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034796. [Online]. Available: http://doi.acm.org/10.1145/2034773.2034796

A. Dijkstra, J. Fokker, and S. D. Swierstra

The Structure of the Essential Haskell Compiler, or Coping with Compiler Complexity

Abstract In this paper we describe the structure of the Essential Haskell Compiler (EHC) and how we manage its complexity, despite its growth from essentials to a full Haskell compiler. Our approach splits both language and implementation into smaller, manageable steps, and uses specific tools to generate parts of the compiler from higher level descriptions.

References

A. Dijkstra, J. Fokker, and S. D. Swierstra, "The structure of the essential haskell compiler, or coping with compiler complexity," in *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, O. Chitil, Z. Horváth, and V. Zsók, Eds., ser. Lecture Notes in Computer Science, vol. 5083, Springer, 2007, pp. 57–74, ISBN: 978-3-540-85372-5. DOI: 10.1007/978-3-540-85373-2_4. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85373-2_4

M. Falaschi, G. Levi, C. Palamidessi, *et al.*

Declarative Modeling of the Operational Behavior of Logic Languages

Abstract The paper defines a new declarative semantics for logic programs, which is based on interpretations containing (possibly) non-ground atoms. Two different interpretations are introduced and the corresponding models are defined and compared. The classical results on the Herbrand model semantics of logic programs are shown to hold in the new models too (i.e. existence of a minimal model, fixpoint characterization, etc.). With the new models, we have a stronger soundness and completeness result for SLD-resolution. In particular, one of the two models allows the set of computed answer substitutions to be characterized precisely.

References

M. Falaschi, G. Levi, C. Palamidessi, *et al.*, "Declarative modeling of the operational behavior of logic languages," *Theor. Comput. Sci.*, vol. 69, no. 3, pp. 289–318, 1989. DOI: 10.1016/0304-3975(89)90070-4. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(89)90070-4

F. Farka, E. Komendantskaya, K. Hammond, *et al.*

Coinductive Soundness of Corecursive Type Class Resolution

Abstract Horn clauses and first-order resolution are commonly used for the implementation of type classes in Haskell. Recently, several core- cursive extensions to type class resolution have been proposed, with the common goal of allowing (co)recursive dictionary construction for those cases when resolution does not terminate. This paper shows, for the first time, that corecursive type class resolution and its recent extensions are coinductively sound with respect to the greatest Herbrand models of logic programs and that they are inductively unsound with respect to the least Herbrand models.

References

F. Farka, E. Komendantskaya, K. Hammond, *et al.*, "Coinductive soundness of corecursive type class resolution," 2016. arXiv: `1608.05233`. [Online]. Available: `http://arxiv.org/abs/1608.05233`

K. Faxén

A static semantics for Haskell

Abstract This paper gives a static semantics for Haskell 98, a non-strict purely functional programming language. The semantics formally specifies nearly all the details of the Haskell 98 type system, including the resolution of overloading, kind inference (including defaulting) and polymorphic recursion, the only major omission being a proper treatment of ambiguous overloading and its resolution. Overloading is translated into explicit dictionary passing, as in all current implementations of Haskell. The target language of this translation is a variant of the GirardReynolds polymorphic lambda calculus featuring higher order polymorphism and explicit type abstraction and application in the term language. Translated programs can thus still be type checked, although the implicit version of this system is impredicative. A surprising result of this formalization effort is that the monomorphism restriction, when rendered in a system of inference rules, compromises the principal type property.

References

K. Faxén, "A static semantics for haskell," *J. Funct. Program.*, vol. 12, no. 4&5, pp. 295–357, 2002. DOI: 10.1017/S0956796802004380. [Online]. Available: http://dx.doi.org/10.1017/S0956796802004380

P. Fu and E. Komendantskaya

A Type-Theoretic Approach to Resolution

Abstract Resolution lies at the foundation of both logic programming and type class context reduction in functional languages. Terminating derivations by resolution have well-defined inductive meaning, whereas some non-terminating derivations can be understood coinductively. Cycle detection is a popular method to capture a small subset of such derivations. We show that in fact cycle detection is a restricted form of coinductive proof, in which the atomic formula forming the cycle plays the role of coinductive hypothesis.

This paper introduces a heuristic method for obtaining richer coinductive hypotheses in the form of Horn formulas. Our approach subsumes cycle detection and gives coinductive meaning to a larger class of derivations. For this purpose we extend resolution with Horn formula resolvents and corecursive evidence generation. We illustrate our method on non-terminating type class resolution problems.

References

P. Fu and E. Komendantskaya, "A type-theoretic approach to resolution," in *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*, M. Falaschi, Ed., ser. Lecture Notes in Computer Science, vol. 9527, Springer, 2015, pp. 91–106, ISBN: 978-3-319-27435-5. DOI: 10.1007/978-3-319-27436-2_6. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-27436-2_6

P. Fu and E. Komendantskaya

Operational semantics of resolution and productivity in Horn clause logic

Abstract This paper presents a study of operational and type-theoretic properties of different resolution strategies in Horn clause logic. We distinguish four different kinds of resolution: resolution by unification (SLD-resolution), resolution by term-matching, the recently introduced structural resolution, and partial (or lazy) resolution. We express them all uniformly as abstract reduction systems, which allows us to undertake a thorough comparative analysis of their properties. To match this small-step semantics, we propose to take Howard's System H as a type-theoretic semantic counterpart. Using System H, we interpret Horn formulas as types, and a derivation for a given formula as the proof term inhabiting the type given by the formula. We prove soundness of these abstract reduction systems relative to System H, and we show completeness of SLD-resolution and structural resolution relative to System H. We identify conditions under which structural resolution is operationally equivalent to SLD-resolution. We show correspondence between term-matching resolution for Horn clause programs without existential variables and term rewriting.

References

P. Fu and E. Komendantskaya, "Operational semantics of resolution and productivity in horn clause logic," *Formal Aspects of Computing*, pp. 1–22, 2016, ISSN: 0934-5043. DOI: 10.1007/s00165-016-0403-1. arXiv: 1604.04114. [Online]. Available: http://arxiv.org/abs/1604.04114http://link.springer.com/10.1007/s00165-016-0403-1

P. Fu, E. Komendantskaya, T. Schrijvers, *et al.*

Proof Relevant Corecursive Resolution

Abstract Resolution lies at the foundation of both logic programming and type class context reduction in functional languages. Terminating derivations by resolution have well-defined inductive meaning, whereas some non-terminating derivations can be understood coinductively. Cycle detection is a popular method to capture a small subset of such derivations. We show that in fact cycle detection is a restricted form of coinductive proof, in which the atomic formula forming the cycle plays the rle of coinductive hypothesis.

This paper introduces a heuristic method for obtaining richer coinductive hypotheses in the form of Horn formulas. Our approach subsumes cycle detection and gives coinductive meaning to a larger class of derivations. For this purpose we extend resolution with Horn formula resolvents and corecursive evidence generation. We illustrate our method on non-terminating type class resolution problems.

References

P. Fu, E. Komendantskaya, T. Schrijvers, *et al.*, "Proof relevant corecursive resolution," in *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, O. Kiselyov and A. King, Eds., ser. Lecture Notes in Computer Science, vol. 9613, Springer, 2016, pp. 126–143, ISBN: 978-3-319-29603-6. DOI: 10.1007/978-3-319-29604-3_9. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-29604-3_9

N. Ghani and P. Hancock

Containers, monads and induction recursion

Abstract Induction recursion offers the possibility of a clean, simple and yet powerful meta-language for the type system of a dependently typed programming language. At its crux, induction recursion allows us to define a universe, that is a set U of codes and a decoding function $T : U \to D$ which assigns to every code $u : U$, a value T, u of some type D, e.g. the large type Set of small types or sets. The name induction recursion refers to the build-up of codes in U using inductive clauses, simultaneously with the definition of the function T, by structural recursion on codes.

Our contribution is to (i) bring out explicitly algebraic structure which is less visible in the original type-theoretic presentation in particular showing how containers and monads play a pivotal role within induction recursion; and (ii) use these structures to present a clean and high level definition of induction recursion suitable for use in functional programming.

References

N. Ghani and P. Hancock, "Containers, monads and induction recursion," *Mathematical Structures in Computer Science*, vol. 26, no. 1, pp. 89–113, 2016. DOI: 10.1017/S0960129514000127. [Online]. Available: http://dx.doi.org/10.1017/S0960129514000127

J. Gibbons and G. Hutton

Proof Methods for Corecursive Programs

Abstract Recursion is a well-known and powerful programming technique, with a wide variety of applications. The dual technique of corecursion is less well-known, but is increasingly proving to be just as useful. This article is a tutorial on the four main methods for proving properties of corecursive programs: fixpoint induction, the approximation (or take) lemma, coinduction, and fusion.

References

J. Gibbons and G. Hutton, "Proof methods for corecursive programs," *Fundam. Inform.*, vol. 66, no. 4, pp. 353–366, 2005. [Online]. Available: `http://content.iospress.com/articles/fundamenta-informaticae/fi66-4-03`

G. Gonthier, B. Ziliani, A. Nanevski, *et al.*

How to make ad hoc proof automation less ad hoc

Abstract Most interactive theorem provers provide support for some form of user-customizable proof automation. In a number of popular systems, such as Coq and Isabelle, this automation is achieved primarily through tactics, which are programmed in a separate language from that of the prover's base logic. While tactics are clearly useful in practice, they can be difficult to maintain and compose because, unlike lemmas, their behavior cannot be specified within the expressive type system of the prover itself.

We propose a novel approach to proof automation in Coq that allows the user to specify the behavior of custom automated routines in terms of Coq's own type system. Our approach involves a sophisticated application of Coq's canonical structures, which generalize Haskell type classes and facilitate a flexible style of dependently-typed logic programming. Specifically, just as Haskell type classes are used to infer the canonical implementation of an overloaded term at a given type, canonical structures can be used to infer the canonical proof of an overloaded lemma for a given instantiation of its parameters. We present a series of design patterns for canonical structure programming that enable one to carefully and predictably coax Coq's type inference engine into triggering the execution of user-supplied algorithms during unification, and we illustrate these patterns through several realistic examples drawn from Hoare Type Theory. We assume no prior knowledge of Coq and describe the relevant aspects of Coq type inference from first principles.

References

G. Gonthier, B. Ziliani, A. Nanevski, *et al.*, "How to make ad hoc proof automation less ad hoc," *J. Funct. Program.*, vol. 23, no. 4, pp. 357–401, 2013. DOI: 10.1017/S0956796813000051. [Online]. Available: http://dx.doi.org/10.1017/S0956796813000051

G. Gupta, A. Bansal, R. Min, *et al.*

Coinductive Logic Programming and Its Applications

Abstract Coinduction has recently been introduced as a powerful technique for reasoning about unfounded sets, unbounded structures, and interactive computations. Where induction corresponds to least fixed point semantics, coinduction corresponds to greatest fixed point semantics. In this paper we discuss the introduction of coinduction into logic programming. We discuss applications of coinductive logic programming to verification and model checking, lazy evaluation, concurrent logic programming and non-monotonic reasoning.

References

G. Gupta, A. Bansal, R. Min, *et al.*, "Coinductive logic programming and its applications," in *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, V. Dahl and I. Niemelä, Eds., ser. Lecture Notes in Computer Science, vol. 4670, Springer, 2007, pp. 27–44, ISBN: 978-3-540-74608-9. DOI: 10.1007/978-3-540-74610-2_4. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74610-2_4

C. V. Hall, K. Hammond, S. L. P. Jones, *et al.*
Type Classes in Haskell

Abstract This article defines a set of type inference rules for resolving overloading introduced by type classes, as used in the functional programming language Haskell. Programs including type classes are transformed into ones which may be typed by standard Hindley-Milner inference rules. In contrast to other work on type classes, the rules presented here relate directly to Haskell programs. An innovative aspect of this work is the use of second-order lambda calculus to record type information in the transformed program.

References

C. V. Hall, K. Hammond, S. L. P. Jones, *et al.*, "Type classes in haskell," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 2, pp. 109–138, 1996. DOI: 10.1145/227699.227700. [Online]. Available: http://doi.acm.org/10.1145/227699.227700

R. Hinze and S. L. P. Jones

Derivable Type Classes

Abstract Generic programming allows you to write a function once, and use it many times at different types. A lot of good foundational work on generic programming has been done. The goal of this paper is to propose a practical way of supporting generic programming within the Haskell language, without radically changing the language or its type system. The key idea is to present generic programming as a richer language in which to write default method definitions in a class declaration.

On the way, we came across a separate issue, concerning type-class overloading where higher kinds are involved. We propose a simple type-class system extension to allow the programmer to write richer contexts than is currently possible.

References

R. Hinze and S. L. P. Jones, "Derivable type classes," *Electr. Notes Theor. Comput. Sci.*, vol. 41, no. 1, pp. 5–35, 2000. DOI: 10.1016/S1571-0661(05)80542-0. [Online]. Available: http://dx.doi.org/10.1016/S1571-0661(05)80542-0

R. Hinze and S. Peyton Jones

Derivable Type Classes

Abstract Generic programming allows you to write a function once, and use it many times at different types. A lot of good foundational work on generic programming has been done. The goal of this paper is to propose a practical way of supporting generic programming within the Haskell language, without radically changing the language or its type system. The key idea is to present generic programming as a richer language in which to write default method definitions in a class declaration. {\par} On the way, we came across a separate issue, concerning type-class overloading where higher kinds are involved. We propose a simple type-class system extension to allow the programmer to write richer contexts than is currently possible.

References

R. Hinze and S. Peyton Jones, "Derivable type classes," *Electronic Notes in Theoretical Computer Science*, vol. 41, no. 1, pp. 5–35, 2001, ISSN: 15710661. DOI: 10.1016/S1571-0661(05)80542-0. [Online]. Available: http://dx.doi.org/10.1016/S1571-0661(05)80542-0http://linkinghub.elsevier.com/retrieve/pii/S1571066105805420

R. Hinze, N. Wu, and J. Gibbons

Unifying structured recursion schemes

Abstract Folds over inductive datatypes are well understood and widely used. In their plain form, they are quite restricted; but many disparate generalisations have been proposed that enjoy similar calculational benefits. There have also been attempts to unify the various generalisations: two prominent such unifications are the 'recursion schemes from comonads' of Uustalu, Vene and Pardo, and our own 'adjoint folds'. Until now, these two unified schemes have appeared incompatible. We show that this appearance is illusory: in fact, adjoint folds subsume recursion schemes from comonads. The proof of this claim involves standard constructions in category theory that are nevertheless not well known in functional programming: Eilenberg-Moore categories and bialgebras.

References

R. Hinze, N. Wu, and J. Gibbons, "Unifying structured recursion schemes," *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 209–220, 2013. [Online]. Available: `http://dl.acm.org/citation.cfm?doid=2544174.2500578`

J. S. Hodas and D. Miller

Logic Programming in a Fragment of Intuitionistic Linear Logic

Abstract When logic programming is based on the proof theory of intuitionistic logic, it is natural to allow implications in goals and in the bodies of clauses. Attempting to prove a goal of the form $D \subset G$ from the context (set of formulas) Γ leads to an attempt to prove the goal G in the extended context $\Gamma \cup \{D\}$. Thus contexts, represented as the left-hand side of intuitionistic sequents, grow as stacks during the bottom-up search for a cut-free proof. While such an intuitionistic notion of context provides for elegant specifications of many computations, contexts can be made more expressive and flexible if they are based on linear logic. After presenting two equivalent formulations of a fragment of linear logic, we show that the fragment has a goal-directed interpretation, thereby partially justifying calling it a logic program-ming language. Logic programs based on the intuitionistic theory of hereditary Harrop formulas can be modularly embedded into this linear logic setting. Programming examples taken from theorem proving, natural language parsing, and data base programming are presented: each example requires a linear, rather than intuitionistic, notion of context to be modeled adequately. An interpreter for this logic programming language must address the problem of splitting contexts; that is, in the attempt to prove a multiplicative conjunction (tensor), say $G_1 \otimes G_2$, from the context Δ the latter must be split into disjoint contexts Δ_1 and Δ_2 for which G_1 follows from Δ_1 and G_2 follows from Δ_2. Since there is an exponential number of such splits, it is important to delay the choice of a split as much as possible. A mechanism for the lazy splitting of contexts is presented based on viewing proof search as a process that takes a context, consumes part of it, and returns the rest (to be consumed elsewhere). In addition, we use collections of Kripke interpretations indexed by a commutative monoid to provide models for this logic programming language and show that logic programs admit canonical models.

References

J. S. Hodas and D. Miller, "Logic programming in a fragment of intuitionistic linear logic," *Inf. Comput.*, vol. 110, no. 2, pp. 327–365, 1994. DOI: 10.1006/inco.1994.1036. [Online]. Available: http://dx.doi.org/10.1006/inco.1994.1036

P. Hudak, J. Hughes, S. L. P. Jones, *et al.*

A history of Haskell: being lazy with class

Abstract This paper describes the history of Haskell, including its genesis and principles, technical contributions, implementations and tools, and applications and impact.

References

P. Hudak, J. Hughes, S. L. P. Jones, *et al.*, "A history of haskell: Being lazy with class," in *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, B. G. Ryder and B. Hailpern, Eds., ACM, 2007, pp. 1–55. DOI: 10.1145/1238844.1238856. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238856

G. Huet and A. Saibi

Constructive Category Theory

Abstract no abstract provided

References

G. Huet and A. Saibi, "Constructive category theory," in *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg*, Goteberg, 1995, p. 27

R. Iemhoff

On The Admissible Rules of Intuitionistic Propositional Logic

Abstract We present a basis for the admissible rules of intuitionistic propositional logic. Thereby a conjecture by de Jongh and Visser is proved. We also present a proof system for the admissible rules, and give semantic criteria for admissibility.

References

R. Iemhoff, "On the admissible rules of intuitionistic propositional logic," *J. Symb. Log.*, vol. 66, no. 1, pp. 281–294, 2001. DOI: 10.2307/2694922. [Online]. Available: http://dx.doi.org/10.2307/2694922

J. Jaffar and P. J. Stuckey

Semantics of Infinite Tree Logic Programming

Abstract We address the problem of declarative and operational semantics for logic programming in the domain of infinite trees. We consider logic programming semantics based on the now familiar function T_P which maps from and into interpretations of the program P. The main point of departure of our work from the literature is that we include unequations in our treatment. Specifically, we prove that the intuitive notions of success and finite failure, defined in terms of T_P, exactly correspond to the operational semantics. The corresponding proofs in the case where no unequations are considered are relatively straightforward mainly because the function T_P has a closure property with respect to a suitable metric space of infinite trees. When unequations are considered, however, the function loses this property and consequently the proofs become more complex. The key to our treatment is a result about images of T_P; we show that these sets have a property analogous to closure. Finally, we also prove certain results pertaining to infinite derivations. These concern the greatest fixpoint of T_P and the concept of completed logic programs and negation-as-failure.

References

J. Jaffar and P. J. Stuckey, "Semantics of infinite tree logic programming," *Theor. Comput. Sci.*, vol. 46, no. 3, pp. 141–158, 1986. DOI: 10.1016/0304-3975(86)90027-7. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(86)90027-7

M. Jaskelioff and O. Rypacek

An Investigation of the Laws of Traversals

Abstract Traversals of data structures are ubiquitous in programming. Consequently, it is important to be able to characterise those structures that are traversable and understand their algebraic properties. Traversable functors have been characterised by McBride and Paterson as those equipped with a distributive law over arbitrary applicative functors; however, laws that fully capture the intuition behind traversals are missing. This article is an attempt to remedy this situation by proposing laws for characterising traversals that capture the intuition behind them. To support our claims, we prove that finitary containers are traversable in our sense and argue that elements in a traversable structure are visited exactly once.

References

M. Jaskelioff and O. Rypacek, "An investigation of the laws of traversals," in *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012.*, J. Chapman and P. B. Levy, Eds., ser. EPTCS, vol. 76, 2012, pp. 40–49. DOI: 10.4204/EPTCS.76.5. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.76.5

P. Johann and N. Ghani

A principled approach to programming with nested types in Haskell

Abstract Initial algebra semantics is one of the cornerstones of the theory of modern functional programming languages. For each inductive data type, it provides a Church encoding for that type, a build combinator which constructs data of that type, a fold combinator which encapsulates structured recursion over data of that type, and a fold/build rule which optimises modular programs by eliminating from them data constructed using the buildcombinator, and immediately consumed using the foldcombinator, for that type. It has long been thought that initial algebra semantics is not expressive enough to provide a similar foundation for programming with nested types in Haskell. Specifically, the standard folds derived from initial algebra semantics have been considered too weak to capture commonly occurring patterns of recursion over data of nested types in Haskell, and no build combinators or fold/build rules have until now been defined for nested types. This paper shows that standard folds are, in fact, sufficiently expressive for programming with nested types in Haskell. It also defines buildcombinators and fold/build fusion rules for nested types. It thus shows how initial algebra semantics provides a principled, expressive, and elegant foundation for programming with nested types in Haskell.

References

P. Johann and N. Ghani, "A principled approach to programming with nested types in haskell," *Higher-Order and Symbolic Computation*, vol. 22, no. 2, pp. 155–189, 2009. DOI: 10.1007/s10990-009-9047-7. [Online]. Available: http://dx.doi.org/10.1007/s10990-009-9047-7

R. van Kesteren, M. C.J. D. van Eekelen, and M. de Mol

Proof support for generic type classes

Abstract We present a proof rule and an effective tactic for proving properties about HASKELL type classes by proving them for the available instance definitions. This is not straightforward, because instance definitions may depend on each other. The proof assistant ISABELLE handles this problem for single parameter type classes by structural induction on types. However, this does not suffice for an effective tactic for more complex forms of overloading. We solve this using an induction scheme derived from the instance definitions. The tactic based on this rule is implemented in the proof assistant SPARKLE.

References

R. van Kesteren, M. C.J. D. van Eekelen, and M. de Mol, "Proof support for generic type classes," in *Revised Selected Papers from the Fifth Symposium on Trends in Functional Programming, TFP 2004, München, Germany, 25-26 November 2004.*, H. Loidl, Ed., ser. Trends in Functional Programming, vol. 5, Intellect, 2004, pp. 1–16, ISBN: 1-84150-144-1

O. Kiselyov and H. Ishii

Freer monads, more extensible effects

Abstract MetaOCaml is a superset of OCaml extending it with the data type for program code and operations for constructing and executing such typed code values. It has been used for compiling domain-specific languages and automating tedious and error-prone specializations of high-performance computational kernels. By statically ensuring that the generated code compiles and letting us quickly run it, MetaOCaml makes writing generators less daunting and more productive. The current BER MetaOCaml is a complete re-implementation of the original MetaOCaml by Taha, Calcagno and collaborators. Besides the new organization, new algorithms, new code, BER MetaOCaml adds a scope extrusion check superseding environment classifiers. Attempting to build code values with unbound or mistakenly bound variables (liable to occur due to mutation or other effects) is now caught early, raising an exception with good diagnostics. The guarantee that the generated code always compiles becomes unconditional, no matter what effects were used in generating the code. We describe BER MetaOCaml stressing the design decisions that made the new code modular and maintainable. We explain the implementation of the scope extrusion check.

References

O. Kiselyov and H. Ishii, "Freer monads, more extensible effects," in *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, B. Lippmeier, Ed., ACM, 2015, pp. 94–105, ISBN: 978-1-4503-3808-0. DOI: 10.1145/2804302.2804319. [Online]. Available: http://doi.acm.org/10.1145/2804302.2804319

P. Kokke and W. Swierstra

Auto in Agda - Programming Proof Search Using Reflection

Abstract As proofs in type theory become increasingly complex, there is a growing need to provide better proof automation. This paper shows how to implement a Prolog-style resolution procedure in the dependently typed programming language Agda. Connecting this resolution procedure to Agdas reflection mechanism provides a first-class proof search tactic for first-order Agda terms. As a result, writing proof automation tactics need not be different from writing any other program.

References

P. Kokke and W. Swierstra, "Auto in agda - programming proof search using reflection," in *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, R. Hinze and J. Voigtländer, Eds., ser. Lecture Notes in Computer Science, vol. 9129, Springer, 2015, pp. 276–301, ISBN: 978-3-319-19796-8. DOI: 10.1007/978-3-319-19797-5_14. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-19797-5_14

E. Komendantskaya and J. Power

Coalgebraic Semantics for Derivations in Logic Programming

Abstract As proofs in type theory become increasingly complex, there is a growing need to provide better proof automation. This paper shows how to implement a Prolog-style resolution procedure in the dependently typed programming language Agda. Connecting this resolution procedure to Agdas reflection mechanism provides a first-class proof search tactic for first-order Agda terms. As a result, writing proof automation tactics need not be different from writing any other program.

References

E. Komendantskaya and J. Power, "Coalgebraic semantics for derivations in logic programming," in *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings*, A. Corradini, B. Klin, and C. Cîrstea, Eds., ser. Lecture Notes in Computer Science, vol. 6859, Springer, 2011, pp. 268–282, ISBN: 978-3-642-22943-5. DOI: 10.1007/978-3-642-22944-2_19. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22944-2_19

D. Kozen and A. Silva

Practical coinduction

Abstract Induction is a well-established proof principle that is taught in most undergraduate programs in mathematics and computer science. In computer science, it is used primarily to reason about inductively defined datatypes such as finite lists, finite trees and the natural numbers. Coinduction is the dual principle that can be used to reason about coinductive datatypes such as infinite streams or trees, but it is not as widespread or as well understood. In this paper, we illustrate through several examples the use of coinduction in informal mathematical arguments. Our aim is to promote the principle as a useful tool for the working mathematician and to bring it to a level of familiarity on par with induction. We show that coinduction is not only about bisimilarity and equality of behaviors, but also applicable to a variety of functions and relations defined on coinductive datatypes.

References

D. Kozen and A. Silva, "Practical coinduction," *Mathematical Structures in Computer Science*, 121, Feb. 2016. DOI: 10.1017/S0960129515000493

R. Lämmel and S. L. P. Jones

Scrap your boilerplate with class: extensible generic functions

Abstract The 'Scrap your boilerplate' approach to generic programming allows the programmer to write generic functions that can traverse arbitrary data structures, and yet have type-specific cases. However, the original approach required all the type-specific cases to be supplied at once, when the recursive knot of generic function definition is tied. Hence, generic functions were closed. In contrast, Haskell's type classes support open, or extensible, functions that can be extended with new type-specific cases as new data types are defined. In this paper, we extend the 'Scrap your boilerplate' approach to support this open style. On the way, we demonstrate the desirability of abstraction over type classes, and the usefulness of recursive dictionarie.

References

R. Lämmel and S. L. P. Jones, "Scrap your boilerplate with class: Extensible generic functions," in *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, O. Danvy and B. C. Pierce, Eds., ACM, 2005, pp. 204–215, ISBN: 1-59593-064-7. DOI: 10.1145/1086365.1086391. [Online]. Available: http://doi.acm.org/10.1145/1086365.1086391

C. S. Lee, N. D. Jones, and
A. M. Ben-Amram

The size-change principle for program termination

Abstract The "size-change termination" principle for a first-order functional language with well-founded data is: a program terminates on all inputs if every infinite call sequence (following program control flow) would cause an infinite descent in some data values.

Size-change analysis is based only on local approximations to parameter size changes derivable from program syntax. The set of infinite call sequences that follow program flow and can be recognized as causing infinite descent is an ω-regular set, representable by a Büchi automaton. Algorithms for such automata can be used to decide size-change termination. We also give a direct algorithm operating on "size-change graphs" (without the passage to automata).

Compared to other results in the literature, termination analysis based on the size-change principle is surprisingly simple and general: lexical orders (also called lexicographic orders), indirect function calls and permuted arguments (descent that is not in-situ) are all handled automatically and without special treatment, with no need for manually supplied argument orders, or theorem-proving methods not certain to terminate at analysis time.

We establish the problem's intrinsic complexity. This turns out to be surprisingly high, complete for PSPACE, in spite of the simplicity of the principle. PSPACE hardness is proved by a reduction from Boolean program termination. An ineresting consequence: the same hardness result applies to many other analyses found in the termination and quasitermination literature.

References

C. S. Lee, N. D. Jones, and A. M. Ben-Amram, "The size-change principle for program termination," in *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, C. Hankin and D. Schmidt, Eds., ACM, 2001, pp. 81–92, ISBN: 1-58113-336-7. DOI: 10.1145/360204.360210. [Online]. Available: http://doi.acm.org/10.1145/360204.360210

M. Lenisa, J. Power, and H. Watanabe

Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads

Abstract We generalise the notion of a distributive law between a monad and a comonad to consider weakened structures such as pointed or co-pointed endofunctors, or endofunctors. We investigate Eilenberg-Moore and Kleisli constructions for each of these possibilities. Then we consider two applications of these weakened notions of distributivity in detail. We characterise Turi and Plotkin's model of GSOS as a distributive law of a monad over a co-pointed endofunctor, and we analyse generalised coiteration and coalgebraic coinduction up-to in terms of a distributive law of the underlying pointed endofunctor of a monad over an endofunctor.

References

M. Lenisa, J. Power, and H. Watanabe, "Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads," *Electr. Notes Theor. Comput. Sci.*, vol. 33, pp. 230–260, 2000. DOI: 10.1016/S1571-0661(05)80350-0. [Online]. Available: http://dx.doi.org/10.1016/S1571-0661(05)80350-0

F. Lindblad and M. Benke

A Tool for Automated Theorem Proving in Agda

Abstract We present a tool for automated theorem proving in Agda, an implementation of Martin-Löf's intuitionistic type theory. The tool is intended to facilitate interactive proving by relieving the user from filling in simple but tedious parts of a proof. The proof search is conducted directly in type theory and produces proof terms. Any proof term is verified by the Agda type-checker, which ensures soundness of the tool. Some effort has been spent on trying to produce human readable results, which allows the user to examine the generated proofs. We have tested the tool on examples mainly in the area of (functional) program verification. Most examples we have considered contain induction, and some contain generalisation. The contribution of this work outside the Agda community is to extend the experience of automated proof for intuitionistic type theory.

References

F. Lindblad and M. Benke, "A tool for automated theorem proving in agda," in *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, J. Filliâtre, C. Paulin-Mohring, and B. Werner, Eds., ser. Lecture Notes in Computer Science, vol. 3839, Springer, 2004, pp. 154–169, ISBN: 3-540-31428-8. DOI: 10.1007/11617990_10. [Online]. Available: http://dx.doi.org/10.1007/11617990_10

J. W. Lloyd

Foundations of Logic Programming, 2nd Edition

Abstract In the two and a half years since the frrst edition of this book was published, the field of logic programming has grown rapidly. Consequently, it seemed advisable to try to expand the subject matter covered in the first edition. The new material in the second edition has a strong database flavour, which reflects my own research interests over the last three years. However, despite the fact that the second edition has about 70% more material than the first edition, many worthwhile topic!! are still missing. I can only plead that the field is now too big to expect one author to cover everything. In the second edition, I discuss a larger class of programs than that discussed in the first edition. Related to this, I have also taken the opportunity to try to improve some of the earlier terminology. Firstly, I introduce "program statements", which are formulas of the form A+-W, where the head A is an atom and the body W is an arbitrary formula. A "program" is a finite set of program statements. There are various restrictions of this class. "Normal" programs are ones where the body of each program statement is a conjunction of literals. (The terminology "general", used in the first edition, is obviously now inappropriate).

References

J. W. Lloyd, *Foundations of Logic Programming, 2nd Edition.* Springer, 1987, ISBN: 3-540-18199-7

A. Löh and R. Hinze

Open data types and open functions

Abstract The problem of supporting the modular extensibility of both data and functions in one programming language at the same time is known as the expression problem. Functional languages traditionally make it easy to add new functions, but extending data (adding new data constructors) requires modifying existing code. We present a semantically and syntactically lightweight variant of open data types and open functions as a solution to the expression problem in the Haskell language. Constructors of open data types and equations of open functions may appear scattered throughout a program with several modules. The intended semantics is as follows: the program should behave as if the data types and functions were closed, defined in one place. The order of function equations is determined by best-fit pattern matching, where a specific pattern takes precedence over an unspecific one. We show that our solution is applicable to the expression problem, generic programming, and exceptions. We sketch two implementations: a direct implementation of the semantics, and a scheme based on mutually recursive modules that permits separate compilation.

References

A. Löh and R. Hinze, "Open data types and open functions," in *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, A. Bossi and M. J. Maher, Eds., ACM, 2006, pp. 133–144, ISBN: 1-59593-388-3. DOI: 10.1145/1140335.1140352. [Online]. Available: http://doi.acm.org/10.1145/1140335.1140352

D. B. MacQueen, G. D. Plotkin, and R. Sethi

An Ideal Model for Recursive Polymorphic Types

Abstract An anbstract is available

References

D. B. MacQueen, G. D. Plotkin, and R. Sethi, "An ideal model for recursive polymorphic types," *Information and Control*, vol. 71, no. 1/2, pp. 95–130, 1986. DOI: 10.1016/S0019-9958(86)80019-5. [Online]. Available: http://dx.doi.org/10.1016/S0019-9958(86)80019-5

C. McBride

Faking it: Simulating dependent types in Haskell

Abstract Dependent types reflect the fact that validity of data is often a relative notion by allowing prior data to affect the types of subsequent data. Not only does this make for a precise type system, but also a highly generic one: both the type and the program for each instance of a family of operations can be computed from the data which codes for that instance. Recent experimental extensions to the Haskell type class mechanism give us strong tools to relativize types to other types. We may simulate some aspects of dependent typing by making counterfeit type-level copies of data, with type constructors simulating data constructors and type classes simulating datatypes. This paper gives examples of the technique and discusses its potential.

References

C. McBride, "Faking it: Simulating dependent types in haskell," *J. Funct. Program.*, vol. 12, no. 4&5, pp. 375–392, 2002. DOI: 10.1017/S0956796802004355. [Online]. Available: http://dx.doi.org/10.1017/S0956796802004355

C. McBride and R. Paterson

Applicative programming with effects

Abstract In this article, we introduce Applicative functors an abstract characterisation of an applicative style of effectful programming, weaker than Monads and hence more widespread. Indeed, it is the ubiquity of this programming pattern that drew us to the abstraction. We retrace our steps in this article, introducing the applicative pattern by diverse examples, then abstracting it to define the Applicative type class and introducing a bracket notation that interprets the normal application syntax in the idiom of an Applicative functor. Furthermore, we develop the properties of applicative functors and the generic operations they support. We close by identifying the categorical structure of applicative functors and examining their relationship both with Monads and with Arrow.

References

C. McBride and R. Paterson, "Applicative programming with effects," *J. Funct. Program.*, vol. 18, no. 1, pp. 1–13, 2008. DOI: 10.1017/S0956796807006326. [Online]. Available: http://dx.doi.org/10.1017/S0956796807006326

D. Miller and G. Nadathur
Programming with Higher-Order Logic

Abstract Formal systems that describe computations over syntactic structures occur frequently in computer science. Logic programming provides a natural framework for encoding and animating such systems. However, these systems often embody variable binding, a notion that must be treated carefully at a computational level. This book aims to show that a programming language based on a simply typed version of higher-order logic provides an elegant, declarative means for providing such a treatment. Three broad topics are covered in pursuit of this goal. First, a proof-theoretic framework that supports a general view of logic programming is identified. Second, an actual language called λProlog is developed by applying this view to higher-order logic. Finally, a methodology for programming with specifications is exposed by showing how several computations over formal objects such as logical formulas, functional programs, and λ-terms and π-calculus expressions can be encoded in λProlog.

References

D. Miller and G. Nadathur, *Programming with Higher-Order Logic*. Cambridge University Press, 2012, ISBN: 978-0-521-87940-8. [Online]. Available: `http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/programming-higher-order-logic?format=HB`

D. Miller, G. Nadathur, F. Pfenning, *et al.* **Uniform Proofs as a Foundation for Logic Programming**

Abstract A proof-theoretic characterization of logical languages that form suitable bases for Prolog-like programming languages is provided. This characterization is based on the principle that the declarative meaning of a logic program, provided by provability in a logical system, should coincide with its operational meaning, provided by interpreting logical connectives as simple and fixed search instructions. The operational semantics is formalized by the identification of a class of cut-free sequent proofs called uniform proofs. A uniform proof is one that can be found by a goal-directed search that respects the interpretation of the logical connectives as search instructions. The concept of a uniform proof is used to define the notion of an abstract logic programming language, and it is shown that first-order and higher-order Horn clauses with classical provability are examples of such a language. Horn clauses are then generalized to hereditary Harrop formulas and it is shown that first-order and higher-order versions of this new class of formulas are also abstract logic programming languages if the inference rules are those of either intuitionistic or minimal logic. The programming language significance of the various generalizations to first-order Horn clauses is briefly discussed.

References

D. Miller, G. Nadathur, F. Pfenning, *et al.*, "Uniform proofs as a foundation for logic programming," *Ann. Pure Appl. Logic*, vol. 51, no. 1-2, pp. 125–157, 1991. DOI: 10.1016/0168-0072(91)90068-W. [Online]. Available: http://dx.doi.org/10.1016/0168-0072(91)90068-W

R. Milner

A Theory of Type Polymorphism in Programming

Abstract The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types, entails defining procedures which work well on objects of a wide variety. We present a formal type discipline for such polymorphic procedures in the context of a simple programming language, and a compile time type-checking algorithm W which enforces the discipline. A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot go wrong and a Syntactic Soundness Theorem states that if W accepts a program then it is well typed. We also discuss extending these results to richer languages; a type-checking algorithm based on W is in fact already implemented and working, for the metalanguage ML in the Edinburgh LCF system.

References

R. Milner, "A theory of type polymorphism in programming," *J. Comput. Syst. Sci.*, vol. 17, no. 3, pp. 348–375, 1978. DOI: 10.1016/0022-0000(78)90014-4. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(78)90014-4

R. Milner
Communication and concurrency

Abstract Communication and Concurrency develops a general calculus of concurrent programming from first principles. The book provides an understanding of concurrency through a very small number of primitive ideas and illustrates how these ideas apply to hardware and software, to specification and implementation. The material is organised to form the basis of a practical course.

References

R. Milner, *Communication and concurrency*, ser. PHI Series in computer science. Prentice Hall, 1989, ISBN: 978-0-13-115007-2

I. Moerdijk and E. Palmgren

Wellfounded trees in categories

Abstract In this paper we present and study a categorical formulation of the W-types of Martin-Lf. These are essentially free term algebras where the operations may have finite or infinite arity. It is shown that W-types are preserved under the construction of sheaves and Artin gluing. In the proofs we avoid using impredicative or nonconstructive principles.

References

I. Moerdijk and E. Palmgren, "Wellfounded trees in categories," *Ann. Pure Appl. Logic*, vol. 104, no. 1-3, pp. 189–218, 2000. DOI: 10.1016/S0168-0072(00)00012-9. [Online]. Available: http://dx.doi.org/10.1016/S0168-0072(00)00012-9

J. H. Morris

Lambda-calculus models of programming languages

Abstract No absract available.

References

J. H. Morris, "Lambda-calculus models of programming languages," PhD thesis, Massachusetts Institute of Technology, 1968, p. 134. DOI: 1721.1/64850

L. S. Moss and N. Danner

On the Foundations of Corecursion

Abstract We consider foundational questions related to the definition of functions by corecursion. This method is especially suited to functions into the greatest fixed point of some monotone operator, and it is most applicable in the context of non-wellfounded sets. We review the work on the Special Final Coalgebra Theorem of Aczel [1] and the Corecursion Theorem of Barwise and Moss [4]. We offer a condition weaker than Aczel's condition of uniformity on maps, and then we prove a result relating the operators satisfying the new condition to the smooth operators of [4].

References

L. S. Moss and N. Danner, "On the foundations of corecursion," *Logic Journal of the IGPL*, vol. 5, no. 2, pp. 231–257, 1997. DOI: 10.1093/jigpal/5.2.231. [Online]. Available: http://dx.doi.org/10.1093/jigpal/5.2.231

U. Norell

Dependently typed programming in agda

Abstract Dependently typed languages have for a long time been used to describe proofs about programs. Traditionally, dependent types are used mostly for stating and proving the properties of the programs and not in defining the programs themselves. An impressive example is the certified compiler by Leroy (2006) implemented and proved correct in Coq (Bertot and Castéran 2004). Recently there has been an increased interest in dependently typed programming, where the aim is to write programs that use the dependent type system to a much higher degree. In this way a lot of the properties that were previously proved separately can be integrated in the type of the program, in many cases adding little or no complexity to the definition of the program. New languages, such as Epigram (McBride and McKinna 2004), are being designed, and existing languages are being extended with new features to accomodate these ideas, for instance the work on dependently typed programming in Coq by Sozeau (2007). This talk gives an overview of the Agda programming language (Norell 2007), whose main focus is on dependently typed programming. Agda provides a rich set of inductive types with a powerful mechanism for pattern matching, allowing dependently typed programs to be written with minimal fuss. To read about programming in Agda, see the lecture notes from the Advanced Functional Programming summer school (Norell 2008) and the work by Oury and Swierstra (2008). In the talk a number of examples of interesting dependently typed programs chosen from the domain of programming language implementation are presented as they are implemented in Agda.

References

U. Norell, "Dependently typed programming in agda," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5832 LNCS, 2009, pp. 230–266, ISBN: 3642046517. DOI: 10.1007/978-3-642-04652-0_5

B. C. d. S. Oliveira, T. Schrijvers, W. Choi, *et al.*

The implicit calculus: a new foundation for generic programming

Abstract Generic programming (GP) is an increasingly important trend in programming languages. Well-known GP mechanisms, such as type classes and the C++0x concepts proposal, usually combine two features: 1) a special type of interfaces; and 2) implicit instantiation of implementations of those interfaces.

Scala implicits are a GP language mechanism, inspired by type classes, that break with the tradition of coupling implicit instantiation with a special type of interface. Instead, implicits provide only implicit instantiation, which is generalized to work for any types. This turns out to be quite powerful and useful to address many limitations that show up in other GP mechanisms.

This paper synthesizes the key ideas of implicits formally in a minimal and general core calculus called the implicit calculus ($\lambda \Rightarrow$), and it shows how to build source languages supporting implicit instantiation on top of it. A novelty of the calculus is its support for partial resolution and higher-order rules (a feature that has been proposed before, but was never formalized or implemented). Ultimately, the implicit calculus provides a formal model of implicits, which can be used by language designers to study and inform implementations of similar mechanisms in their own languages.

References

B. C. d. S. Oliveira, T. Schrijvers, W. Choi, *et al.*, "The implicit calculus: A new foundation for generic programming," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds., ACM, 2012, pp. 35–44, ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254070. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254070

L. C. Paulson and A. W. Smith

Logic Programming, Functional Programming, and Inductive Definitions

Abstract An attempt at unifying logic and functional programming is reported. As a starting point, we take the view that "logic programs" are not about logic but constitute inductive definitions of sets and relations. A skeletal language design based on these considerations is sketched and a prototype implementation discussed.

References

L. C. Paulson and A. W. Smith, "Logic programming, functional programming, and inductive definitions," *CoRR*, vol. cs.LO/9301109, 1993. [Online]. Available: `http://arxiv.org/abs/cs.LO/9301109`

B. C. Pierce
Types and programming languages

Abstract A type system is a syntactic method for enforcing levels of abstraction in programs. The study of type systems–and of programming languages from a type-theoretic perspective–has important applications in software engineering, language design, high-performance compilers, and security.

This text provides a comprehensive introduction both to type systems in computer science and to the basic theory of programming languages. The approach is pragmatic and operational; each new concept is motivated by programming examples and the more theoretical sections are driven by the needs of implementations. Each chapter is accompanied by numerous exercises and solutions, as well as a running implementation. Dependencies between chapters are explicitly identified, allowing readers to choose a variety of paths through the material.

The core topics include the untyped lambda-calculus, simple type systems, type reconstruction, universal and existential polymorphism, subtyping, bounded quantification, recursive types, kinds, and type operators. Extended case studies develop a variety of approaches to modeling the features of object-oriented languages.

References

B. C. Pierce, *Types and programming languages*. MIT Press, 2002, ISBN: 978-0-262-16209-8

A. M. Pitts

Parametric Polymorphism and Operational Equivalence

Abstract Studies of the mathematical properties of impredicatively polymorphic types have for the most part focused on the polymorphic lambda calculus of Girard-Reynolds, which is a calculus of total polymorphic functions. This paper considers polymorphic types from a functional programming perspective, where the partialness arising from the presence of fixpoint recursion complicates the nature of potentially infinite (lazy) datatypes. An operationally-based approach to Reynolds' notion of relational parametricity is developed for an extension of Plotkin's PCF with types and lazy lists. The resulting logical relation is shown to be a useful tool for proving properties of polymorphic types up to a notion of operational equivalence based on Morris-style contextual equivalence.

References

A. M. Pitts, "Parametric polymorphism and operational equivalence," *Electr. Notes Theor. Comput. Sci.*, vol. 10, pp. 2–27, 1997. DOI: 10.1016/S1571-0661(05)80685-1. [Online]. Available: http://dx.doi.org/10.1016/S1571-0661(05)80685-1

G. D. Plotkin and J. Power

Algebraic Operations and Generic Effects

Abstract Given a complete and cocomplete symmetric monoidal closed category V and a symmetric monoidal V-category C with cotensors and a strong V-monad T on C, we investigate axioms under which an ObC-indexed family of operations of the form $\alpha_x : (Tx)^v \to (Tx)^w$ provides semantics for algebraic operations on the computational λ-calculus. We recall a definition for which we have elsewhere given adequacy results, and we show that an enrichment of it is equivalent to a range of other possible natural definitions of algebraic operation. In particular, we define the notion of generic effect and show that to give a generic effect is equivalent to giving an algebraic operation. We further show how the usual monadic semantics of the computational λ-calculus extends uniformly to incorporate generic effects. We outline examples and non-examples and we show that our definition also enriches one for call-by-name languages with effects.

References

G. D. Plotkin and J. Power, "Algebraic operations and generic effects," *Applied Categorical Structures*, vol. 11, no. 1, pp. 69–94, 2003. DOI: 10.1023/A:1023064908962. [Online]. Available: http://dx.doi.org/10.1023/A:1023064908962

J. Power and H. Watanabe

Combining a monad and a comonad

Abstract We give a systematic treatment of distributivity for a monad and a comonad as arises in giving category theoretic accounts of operational and denotational semantics, and in giving an intensional denotational semantics. We do this axiomatically, in terms of a monad and a comonad in a 2-category, giving accounts of the EilenbergMoore and Kleisli constructions. We analyse the eight possible relationships, deducing that two pairs are isomorphic, but that the other pairs are all distinct. We develop those 2-categorical definitions necessary to support this analysis.

References

J. Power and H. Watanabe, "Combining a monad and a comonad," *Theor. Comput. Sci.*, vol. 280, no. 1-2, pp. 137–162, 2002. DOI: 10.1016/S0304-3975(01)00024-X. [Online]. Available: http://dx.doi.org/10.1016/S0304-3975(01)00024-X

G. Rosu and D. Lucanu

Circular Coinduction: A Proof Theoretical Foundation

Abstract Several algorithmic variants of circular coinduction have been proposed and implemented during the last decade, but a proof theoretical foundation of circular coinduction in its full generality is still missing. This paper gives a three-rule proof system that can be used to formally derive circular coinductive proofs. This three-rule system is proved behaviorally sound and is exemplified by proving several properties of infinite streams. Algorithmic variants of circular coinduction now become heuristics to search for proof derivations using the three rules.

References

G. Rosu and D. Lucanu, "Circular coinduction: A proof theoretical foundation," in *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings*, A. Kurz, M. Lenisa, and A. Tarlecki, Eds., ser. Lecture Notes in Computer Science, vol. 5728, Springer, 2009, pp. 127–144, ISBN: 978-3-642-03740-5. DOI: 10.1007/978-3-642-03741-2_10. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03741-2_10

J. J.M. M. Rutten

Behavioural differential equations: A coinductive calculus of streams, automata, and power series

Abstract We present a theory of streams (infinite sequences), automata and languages, and formal power series, in terms of the notions of homomorphism and bisimulation, which are the cornerstones of the theory of (universal) coalgebra. This coalgebraic perspective leads to a unified theory, in which the observation that each of the aforementioned sets carries a so-called final automaton structure, plays a central role. Finality forms the basis for both definitions and proofs by coinduction, the coalgebraic counterpart of induction. Coinductive definitions take the shape of what we have called behavioural differential equations, after Brzozowski's notion of input derivative. A calculus is developed for coinductive reasoning about all of the aforementioned structures, closely resembling calculus from classical analysis.

References

J. J.M. M. Rutten, "Behavioural differential equations: a coinductive calculus of streams, automata, and power series," *Theoretical Computer Science*, vol. 308, no. 1-3, pp. 1–53, 2003, ISSN: 03043975. DOI: 10.1016/S0304-3975(02)00895-2

D. Sangiorgi

On the origins of bisimulation and coinduction

Abstract The origins of bisimulation and bisimilarity are examined, in the three fields where they have been independently discovered: Computer Science, Philosophical Logic (precisely, Modal Logic), Set Theory.

Bisimulation and bisimilarity are coinductive notions, and as such are intimately related to fixed points, in particular greatest fixed points. Therefore also the appearance of coinduction and fixed points is discussed, though in this case only within Computer Science. The paper ends with some historical remarks on the main fixed-point theorems (such as Knaster-Tarski) that underpin the fixed-point theory presented.

References

D. Sangiorgi, "On the origins of bisimulation and coinduction," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 4, 2009. DOI: 10.1145/1516507.1516510. [Online]. Available: http://doi.acm.org/10.1145/1516507.1516510

D. D. Schreye, V. Nys, and C. J. Nicholson

Analysing and Compiling Coroutines with Abstract Conjunctive Partial Deduction

Abstract We provide an approach to formally analyze the computational behavior of coroutines in Logic Programs and to compile these computations into new programs, not requiring any support for coroutines. The problem was already studied near to 30 years ago, in an analysis and transformation technique called Compiling Control. However, this technique had a strong ad hoc flavor: the completeness of the analysis was not well understood and its symbolic evaluation was also very ad hoc. We show how Abstract Conjunctive Partial Deduction, introduced by Leuschel in 2004, provides an appropriate setting to redefine Compiling Control. Leuschels framework is more general than the original formulation, it is provably correct, and it can easily be applied for simple examples. We also show that the Abstract Conjunctive Partial Deduction framework needs some further extension to be able to deal with more complex examples.

References

D. D. Schreye, V. Nys, and C. J. Nicholson, "Analysing and compiling coroutines with abstract conjunctive partial deduction," in *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers*, M. Proietti and H. Seki, Eds., ser. Lecture Notes in Computer Science, vol. 8981, Springer, 2014, pp. 21–38, ISBN: 978-3-319-17821-9. DOI: 10.1007/978-3-319-17822-6_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-17822-6_2

T. Schrijvers, S. L. P. Jones,
M. M. T. Chakravarty, *et al.*

Type checking with open type functions

Abstract We report on an extension of Haskell with open type-level functions and equality constraints that unifies earlier work on GADTs, functional dependencies, and associated types. The contribution of the paper is that we identify and characterise the key technical challenge of entailment checking; and we give a novel, decidable, sound, and complete algorithm to solve it, together with some practically-important variants. Our system is implemented in GHC, and is already in active use.

References

T. Schrijvers, S. L. P. Jones, M. M. T. Chakravarty, *et al.*, "Type checking with open type functions," in *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, J. Hook and P. Thiemann, Eds., ACM, 2008, pp. 51–62, ISBN: 978-1-59593-919-7. DOI: 10.1145/1411204.1411215. [Online]. Available: http://doi.acm.org/10.1145/1411204.1411215

L. Simon, A. Bansal, A. Mallya, *et al.*

Co-Logic Programming: Extending Logic Programming with Coinduction

Abstract In this paper we present the theory and practice of co-logic programming (co-LP for brevity), a paradigm that combines both inductive and coinductive logic programming. Co-LP is a natural generalization of logic programming and coinductive logic programming, which in turn generalizes other extensions of logic programming, such as infinite trees, lazy predicates, and concurrent communicating predicates. Co-LP has applications to rational trees, verifying infinitary properties, lazy evaluation, concurrent LP, model checking, bisimilarity proofs, etc.

References

L. Simon, A. Bansal, A. Mallya, *et al.*, "Co-logic programming: Extending logic programming with coinduction," in *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, L. Arge, C. Cachin, T. Jurdzinski, *et al.*, Eds., ser. Lecture Notes in Computer Science, vol. 4596, Springer, 2007, pp. 472–483, ISBN: 978-3-540-73419-2. DOI: 10.1007/978-3-540-73420-8_42. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73420-8_42

A. Simpson and G. Plotkin

Complete axioms for categorical fixed-point operators

Abstract We give an axiomatic treatment of fixed-point operators in categories. A notion of iteration operator is defined embodying the equational properties of iteration theories. We prove a general completeness theorem for iteration operators, relying on a new, purely syntactic characterisation of the free iteration theory. We then show how iteration operators arise in axiomatic domain theory. One result derives them from the existence of sufficiently many bifree algebras (exploiting the universal property Freyd introduced in his notion of algebraic compactness). Another result shows that, in the presence of a parameterized natural numbers object and an equational lifting monad, any uniform fixed-point operator is necessarily an iteration operator

References

A. Simpson and G. Plotkin, "Complete axioms for categorical fixed-point operators," *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pp. 30–41, 2000. [Online]. Available: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=855753`

S. Staton

An Algebraic Presentation of Predicate Logic - (Extended Abstract)

Abstract We present an algebraic theory for a fragment of predicate logic. The fragment has disjunction, existential quantification and equality. It is not an algebraic theory in the classical sense, but rather within a new framework that we call parameterized algebraic theories. We demonstrate the relevance of this algebraic presentation to computer science by identifying a programming language in which every type carries a model of the algebraic theory. The result is a simple functional logic programming language. We provide a syntax-free representation theorem which places terms in bijection with sieves, a concept from category theory. We study presentation-invariance for general parameterized algebraic theories by providing a theory of clones. We show that parameterized algebraic theories characterize a class of enriched monads.

References

S. Staton, "An algebraic presentation of predicate logic - (extended abstract)," in *Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, F. Pfenning, Ed., ser. Lecture Notes in Computer Science, vol. 7794, Springer, 2013, pp. 401–417, ISBN: 978-3-642-37074-8. DOI: 10.1007/978-3-642-37075-5_26. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37075-5_26

P. J. Stuckey and M. Sulzmann

A theory of overloading

Abstract We present a novel approach to allow for overloading of identifiers in the spirit of type classes. Our approach relies on a combination of the HM(X) type system framework with Constraint Handling Rules (CHRs). CHRs are a declarative language for writing incremental constraint solvers, that provide our scheme with a form of programmable type language. CHRs allow us to precisely describe the relationships among overloaded identifiers. Under some sufficient conditions on the CHRs we achieve decidable type inference and the semantic meaning of programs is unambiguous. Our approach provides a common formal basis for many type class extensions such as multiparameter type classes and functional dependencies.

References

P. J. Stuckey and M. Sulzmann, "A theory of overloading," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1216–1269, 2005. DOI: 10.1145/1108970.1108974. [Online]. Available: http://doi.acm.org/10.1145/1108970.1108974

M. Sulzmann, G. J. Duck, S. L. P. Jones, *et al.*

Understanding functional dependencies via constraint handling rules

Abstract Functional dependencies are a popular and useful extension to Haskell style type classes. We give a reformulation of functional dependencies in terms of Constraint Handling Rules (CHRs). In previous work, CHRs have been employed for describing user-programmable type extensions in the context of Haskell style type classes. Here, we make use of CHRs to provide for the first time a concise result that under some sufficient conditions, functional dependencies allow for sound, complete and decidable type inference. The sufficient conditions imposed on functional dependencies can be very limiting. We show how to safely relax these conditions and suggest several sound extensions of functional dependencies. Our results allow for a better understanding of functional dependencies and open up the opportunity for new applications.

References

M. Sulzmann, G. J. Duck, S. L. P. Jones, *et al.*, "Understanding functional dependencies via constraint handling rules," *J. Funct. Program.*, vol. 17, no. 1, pp. 83–129, 2007. DOI: 10.1017/S0956796806006137. [Online]. Available: http://dx.doi.org/10.1017/S0956796806006137

G. Sutcliffe

The TPTP problem library and associated infrastructure : the FOF and CNF Parts, v3.5.0

Abstract This paper describes the First-Order Form (FOF) and Clause Normal Form (CNF) parts of the TPTP problem library, and the associated infrastructure. TPTP v3.5.0 was the last release containing only FOF and CNF problems, and thus serves as the exemplar. This paper summarizes the history and development of the TPTP, describes the structure and contents of the TPTP, and gives an overview of TPTP related projects and tools.

References

G. Sutcliffe, "The tptp problem library and associated infrastructure : the fof and cnf parts, v3.5.0," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 337–362, 2009, ISSN: 01687433. DOI: 10.1007/s10817-009-9143-8

H. Thielemann

How to Refine Polynomial Functions

Abstract Research on refinable functions in wavelet theory is mostly focused to localized functions. However it is known, that polynomial functions are refinable, too. In our paper we investigate on conversions between refinement masks and polynomials and their uniqueness.

References

H. Thielemann, "How to refine polynomial functions," *IJWMIP*, vol. 10, no. 3, 2012. DOI: 10.1142/S0219691312500270. [Online]. Available: http://dx.doi.org/10.1142/S0219691312500270

D. Vytiniotis, S. L. P. Jones, T. Schrijvers, *et al.*

OutsideIn(X) Modular type inference with local assumptions

Abstract Advanced type system features, such as GADTs, type classes and type families, have proven to be invaluable language extensions for ensuring data invariants and program correctness. Unfortunately, they pose a tough problem for type inference when they are used as local type assumptions. Local type assumptions often result in the lack of principal types and cast the generalisation of local let-bindings prohibitively difficult to implement and specify. User-declared axioms only make this situation worse. In this paper, we explain the problems and perhaps controversially argue for abandoning local let-binding generalisation. We give empirical results that local let generalisation is only sporadically used by Haskell programmers. Moving on, we present a novel constraint-based type inference approach for local type assumptions. Our system, called OutsideIn(X), is parameterised over the particular underlying constraint domain X, in the same way as HM(X). This stratification allows us to use a common metatheory and inference algorithm. OutsideIn(X) extends the constraints of X by introducing implication constraints on top. We describe the strategy for solving these implication constraints, which, in turn, relies on a constraint solver for X. We characterise the properties of the constraint solver for X so that the resulting algorithm only accepts programs with principal types, even when the type system specification accepts programs that do not enjoy principal types. Going beyond the general framework, we give a particular constraint solver for X = type classes + GADTs + type families, a non-trivial challenge in its own right. This constraint solver has been implemented and distributed as part of GHC 7.

References

D. Vytiniotis, S. L. P. Jones, T. Schrijvers, *et al.*, "Outsidein(x) modular type inference with local assumptions," *J. Funct. Program.*, vol. 21, no. 4-5, pp. 333–412, 2011. DOI: 10.1017/S0956796811000098. [Online]. Available: http://dx.doi.org/10.1017/S0956796811000098

P. Wadler

Theorems for Free!

Abstract From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus.

References

P. Wadler, "Theorems for free!" In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, J. E. Stoy, Ed., ACM, 1989, pp. 347–359, ISBN: 0-201-51389-7. DOI: 10.1145/99370.99404. [Online]. Available: http://doi.acm.org/10.1145/99370.99404

P. Wadler and S. Blott

How to Make ad-hoc Polymorphism Less ad-hoc

Abstract This paper presents type classes, a new approach to ad-hoc polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the eqtype variables of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them formally by means of type inference rules.

References

P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad-hoc," in *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, ACM Press, 1989, pp. 60–76, ISBN: 0-89791-294-2. DOI: 10.1145/75277.75283. [Online]. Available: http://doi.acm.org/10.1145/75277.75283

N. Bjørner, A. Gurfinkel, K. McMillan, *et al.*

Horn Clause Solvers for Program Verification

Abstract **abstract**

References

N. Bjørner, A. Gurfinkel, K. McMillan, *et al.*, "Horn clause solvers for program verification," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9300, 2015, pp. 24–51, ISBN: 9783319235332. DOI: 10.1007/978-3-319-23534-9_2. [Online]. Available: http://link.springer.com/10.1007/978-3-319-23534-9{_}2

H. Geuvers and R. Nederpelt

N.G. de Bruijn's contribution to the formalization of mathematics

Abstract A 'process theory' is any theory of systems and processes which admits sequential and parallel composition. 'Terminality' unifies normalisation of pure states, trace-preservation of CP-maps, and adding up to identity of positive operators in quantum theory, and generalises this to arbitrary process theories. We show that terminality and non-signalling coincide in any process theory, provided one makes causal structure explicit. In fact, making causal structure explicit is necessary to even make sense of non-signalling in process theories. We conclude that because of its much simpler mathematical form, terminality should be taken to be a more fundamental notion than non-signalling.

References

H. Geuvers and R. Nederpelt, "N.g. de bruijn's contribution to the formalization of mathematics," *Indagationes Mathematicae*, vol. 24, no. 4, pp. 1034–1049, 2013, ISSN: 00193577. DOI: 10.1016/j.indag.2013.09.003. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0019357713000700

R. Jhala, R. Majumdar, and A. Rybalchenko

HMC : Verifying Functional Programs

Abstract We present Hindley-Milner-Cousots (HMC), an algorithm that allows any interprocedural analysis for first-order imperative programs to be used to ver-ify safety properties of typed higher-order functional programs. HMC works as follows. First, it uses the type structure of the functional program to generate a set of logical refinement constraints whose satisfaction implies the safety of the source program. Next, it transforms the logical refinement constraints into a simple first-order imperative program that is safe iff the constraints are satisfiable. Thus, in one swoop, HMC makes tools for invariant generation, e.g., based on abstract domains, predicate abstraction, counterexample-guided refinement, and Craig in-terpolation be directly applicable to verify safety properties of modern functional languages in a fully automatic manner. We have implemented HMC and describe preliminary experimental results using two imperative checkers ARMC and IN-TERPROC to verify OCAML programs. Thus, by composing type-based reasoning grounded in program syntax and state-based reasoning grounded in abstract interpretation, HMC opens the door to automatic verification of programs written in modern programming languages.

References

R. Jhala, R. Majumdar, and A. Rybalchenko, "Hmc : verifying functional programs," pp. 470–485, 2011

E. Komendantskaya and J. Power

Logic programming: laxness and saturation

Abstract A propositional logic program P may be identified with a P_fP_f-coalgebra on the set of atomic propositions in the program. The corresponding $C(P_fP_f)$-coalgebra, where $C(P_fP_f)$ is the cofree comonad on P_fP_f, describes derivations by resolution. That correspondence has been developed to model first-order programs in two ways, with lax semantics and saturated semantics, based on locally ordered categories and right Kan extensions respectively. We unify the two approaches, exhibiting them as complementary rather than competing, reflecting the theorem-proving and proof-search aspects of logic programming. While maintaining that unity, we further refine lax semantics to give finitary models of logic programs with existential variables, and to develop a precise semantic relationship between variables in logic programming and worlds in local state.

References

E. Komendantskaya and J. Power, "Logic programming: laxness and saturation," 2016. arXiv: 1608.07708. [Online]. Available: http://arxiv.org/abs/1608.07708

E. Komendantskaya and J. Power

Logic programming: laxness and saturation

Abstract A propositional logic program P may be identified with a P_fP_f-coalgebra on the set of atomic propositions in the program. The corresponding $C(P_fP_f)$-coalgebra, where $C(P_fP_f)$ is the cofree comonad on P_fP_f, describes derivations by resolution. That correspondence has been developed to model first-order programs in two ways, with lax semantics and saturated semantics, based on locally ordered categories and right Kan extensions respectively. We unify the two approaches, exhibiting them as complementary rather than competing, reflecting the theorem-proving and proof-search aspects of logic programming. While maintaining that unity, we further refine lax semantics to give finitary models of logic programs with existential variables, and to develop a precise semantic relationship between variables in logic programming and worlds in local state.

References

E. Komendantskaya and J. Power, "Logic programming: laxness and saturation," 2016. arXiv: 1608.07708. [Online]. Available: http://arxiv.org/abs/1608.07708

N. P. Mendler, P. Panangaden, P. J. Scott, *et al.*

A Logical View of Concurrent Constraint Programming

Abstract Concurrent Constraint Programming (CCP) has been the subject of growing interest as the focus of a new paradigm for concurrent computation. Like logic programming it claims close relations to logic. In fact CCP languages are logics in a certain sense that we make precise in this paper. In recent work it was shown that the denotational semantics of determinate concurrent constraint programming languages forms a fibred categorical structure called a hyperdoctrine, which is used as the basis of the categorical formulation of first-order logic. What this shows is that the combinators of determinate CCP can be viewed as logical connectives. In this paper we extend these ideas to the operational semantics of such languages and thus make available similar analogies for a much broader variety of languages including indeterminate CCP languages and concurrent block-structured imperative languages.

References

N. P. Mendler, P. Panangaden, P. J. Scott, *et al.*, "A logical view of concurrent constraint programming," *Nordic J. of Computing*, vol. 2, no. 2, pp. 181–220, 1995

M. Odersky, M. Sulzmann, and M. Wehr

Type inference with constrained types

Abstract Type inference in polymorphic, nominal type systems with \nsubtyping, additionally equipped with ad-hoc overloading is not easy. However most mainstream languages like C#, Java and C++ have all those features, which makes extending them with type inference cumbersome. We present a practical, sound, but not complete, type inference algorithm for such type systems. It is based on the online constraint solving combined with deferral of certain typing actions. The algorithm is successfully employed in functional and objectoriented language for the .NET platform called Nemerle.

References

M. Odersky, M. Sulzmann, and M. Wehr, "Type inference with constrained types," *Theory and Practice of Object Systems*, vol. 5, no. 1, pp. 35–55, 1999, ISSN: 1074-3227. DOI: 10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4. [Online]. Available: http://doi.wiley.com/10.1002/{\%}28SICI{\%}291096-9942{\%}28199901/03{\%}295{\%}3A1{\%}3C35{\%}3A{\%}3AAID-TAPO4{\%}3E3.0.CO{\%}3B2-4

P. W. O'Hearn and R. D. Tennent

Parametricity and local variables

Abstract **abstract**

References

P. W. O'Hearn and R. D. Tennent, "Parametricity and local variables," *Journal of the ACM*, vol. 42, no. 3, pp. 658–709, 1995, ISSN: 00045411. DOI: 10.1145/210346.210425. [Online]. Available: http://portal.acm.org/citation.cfm?doid=210346.210425

C.-H. L. Ong and S. J. Ramsay

Verifying higher-order functional programs with pattern-matching algebraic data types

Abstract Type-based model checking algorithms for higher-order recursion schemes have recently emerged as a promising approach to the verification of functional programs. We introduce pattern-matching recursion schemes (PMRS) as an accurate model of computation for functional programs that manipulate algebraic data-types. PMRS are a natural extension of higher-order recursion schemes that incorporate pattern-matching in the defining rules. This paper is concerned with the following (undecidable) verification problem: given a correctness property ϕ, a functional program (qua PMRS) and a regular input set , does every term that is reachable from under rewriting by satisfy ϕ? To solve the PMRS verification problem, we present a sound semi-algorithm which is based on model-checking and counterexample guided abstraction refinement. Given a no-instance of the verification problem, the method is guaranteed to terminate. From an order-n PMRS and an input set generated by a regular tree grammar, our method constructs an order-n weak PMRS which over-approximates only the first-order pattern-matching behaviour, whilst remaining completely faithful to the higher-order control flow. Using a variation of Kobayashi's type-based approach, we show that the (trivial automaton) model-checking problem for weak PMRS is decidable. When a violation of the property is detected in the abstraction which does not correspond to a violation in the model, the abstraction is automatically refined by 'unfolding' the pattern-matching rules in the program to give successively more and more accurate weak PMRS models.

References

C.-H. L. Ong and S. J. Ramsay, "Verifying higher-order functional programs with pattern-matching algebraic data types," *SIGPLAN Not.*, vol. 46, no. 1, pp. 587–598, 2011, ISSN: 0362-1340. DOI: 10.1145/1925844.1926453. [Online]. Available: http://doi.acm.org/10.1145/1925844.1926453{\%}5Cnhttp://dl.acm.org/ft{_}gateway.cfm?id=1926453{\&}type=pdf

F. Pfenning

Logic programming in the LF logical framework

Abstract **abstract**

References

F. Pfenning, "Logic programming in the lf logical framework," *First Workshop on Logical Frameworks*, pp. 1–25, 1991. [Online]. Available: `http://books.google.com/books?hl=en{\&}lr={\&}id=X9wfWwslFQIC{\&}oi=fnd{\&}pg=PA149{\&}dq=Logic+Programming+in+the+LF+Logical+Framework{\&}ots=LfrwT41GfT{\&}sig=AymIHgqAw{\_}M3EiIPGvlpnR2J34M`

F. Pfenning and C. Schürmann

System description: Twelf a meta-logical framework for deductive systems

Abstract Twelf is a meta-logical framework for the specification, implementation, and meta-theory of deductive systems from the theory of programming languages and logics. It relies on the LF type theory and the judgments-as-types methodology for specification [HHP93], a constraint logic programming interpreter for implementation [Pfe91], and the meta-logic M2 for reasoning about object languages encoded in LF [SP98]. It is a significant extension and complete reimplementation of the Elf system [Pfe94]. Twelf is written in Standard ML and runs under SML of New Jersey and MLWorks on Unix and Window platforms. The current version (1.2) is distributed with a complete manual, example suites, a tutorial in the form of on-line lecture notes [Pfe], and an Emacs interface. Source and binary distributions are accessible via the Twelf home page http://www.cs.cmu.edu/~twelf.

References

F. Pfenning and C. Schürmann, "System description: twelf a meta-logical framework for deductive systems," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1632, pp. 202–206, 1999, ISSN: 16113349. DOI: 10.1007/3-540-48660-7_14

V. Simonet and F. Pottier

A constraint-based approach to guarded algebraic data types

Abstract We study HMG(X), an extension of the constraint-based type system HM(X) with deep pattern matching, polymorphic recursion, and guarded algebraic data types. Guarded algebraic data types subsume the concepts known in the literature as indexed types, guarded recursive datatype constructors, (first-class) phantom types, and equality qualified types, and are closely related to inductive types. Their characteristic property is to allow every branch of a case construct to be typechecked under different assumptions about the type variables in scope. We prove that HMG(X) is sound and that, provided recursive definitions carry a type annotation, type inference can be reduced to constraint solving. Constraint solving is decidable, at least for some instances of X, but prohibitively expensive. Effective type inference for guarded algebraic data types is left as an issue for future research.

References

V. Simonet and F. Pottier, "A constraint-based approach to guarded algebraic data types," *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 1, 1–es, 2007, ISSN: 01640925. DOI: 10.1145/1180475.1180476. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1180475.1180476

Bibliography

[1] M. Abadi and M. P. Fiore, "Syntactic considerations on recursive types," in *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, IEEE Computer Society, 1996, pp. 242–252, ISBN: 0-8186-7463-6. DOI: 10.1109/LICS.1996.561324. [Online]. Available: http://dx.doi.org/10.1109/LICS.1996.561324.

[2] A. Abel and B. Pientka, "Wellfounded recursion with copatterns: A unified approach to termination and productivity," in *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, G. Morrisett and T. Uustalu, Eds., ACM, 2013, pp. 185–196, ISBN: 978-1-4503-2326-0. DOI: 10.1145/2500365.2500591. [Online]. Available: http://doi.acm.org/10.1145/2500365.2500591.

[3] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer, "Copatterns: Programming infinite structures by observations," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, R. Giacobazzi and R. Cousot, Eds., ACM, 2013, pp. 27–38, ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429075. [Online]. Available: http://doi.acm.org/10.1145/2429069.2429075.

[4] S. Abramsky and N. Tzevelekos, "Introduction to categories and categorical logic," *CoRR*, vol. abs/1102.1313, 2011. [Online]. Available: http://arxiv.org/abs/1102.1313.

[5] J. Adámek and V. Koubek, "Least fixed point of a functor," *J. Comput. Syst. Sci.*, vol. 19, no. 2, pp. 163–178, 1979. DOI: 10.1016/0022-0000(79)90026-6. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(79)90026-6.

[6] ——, "On the greatest fixed point of a set functor," *Theor. Comput. Sci.*, vol. 150, no. 1, pp. 57–75, 1995. DOI: 10.1016/0304-3975(95)00011-K. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(95)00011-K.

[7] D. Ancona and A. Dovier, "A theoretical perspective of coinductive logic programming," *Fundam. Inform.*, vol. 140, no. 3-4, pp. 221–246, 2015. DOI: 10.3233/FI-2015-1252. [Online]. Available: http://dx.doi.org/10.3233/FI-2015-1252.

[8] D. Ancona and G. Lagorio, "Idealized coinductive type systems for imperative object-oriented programs," *RAIRO - Theor. Inf. and Applic.*, vol. 45, no. 1, pp. 3–33, 2011. DOI: 10.1051/ita/2011009. [Online]. Available: http://dx.doi.org/10.1051/ita/2011009.

[9] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, "Proving correctness of imperative programs by linearizing constrained horn clauses," *TPLP*, vol. 15, no. 4-5, pp. 635–650, 2015. DOI: 10.1017/S1471068415000289. [Online]. Available: http://dx.doi.org/10.1017/S1471068415000289.

[10] R. Atkey, "What is a categorical model of arrows?" *Electr. Notes Theor. Comput. Sci.*, vol. 229, no. 5, pp. 19–37, 2011. DOI: 10.1016/j.entcs.2011.02.014. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2011.02.014.

[11] R. Backhouse, R. Crole, and J. Gibbons, Eds., *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2002, vol. 2297, ISBN: 3540436138. [Online]. Available: http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/acmmpc-toc.pdf.

[12] H. P. Barendregt, "Functional programming and lambda calculus," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 1990, pp. 321–363.

[13] H. Basold and H. H. Hansen, "Well-definedness and observational equivalence for inductive-coinductive programs," *Journal of Logic and Computation*, exw091, 2016, ISSN: 0955-792X. DOI: 10.1093/logcom/exv091. [Online]. Available: https://academic.oup.com/logcom/article-lookup/doi/10.1093/logcom/exv091.

[14] M. Bellia and G. Levi, "The relation between logic and functional languages: A survey," *J. Log. Program.*, vol. 3, no. 3, pp. 217–236, 1986. DOI: 10.1016/0743-1066(86)90014-2. [Online]. Available: http://dx.doi.org/10.1016/0743-1066(86)90014-2.

[15] B. van den Berg and F. D. Marchi, "Non-well-founded trees in categories," *Ann. Pure Appl. Logic*, vol. 146, no. 1, pp. 40–59, 2007. DOI: 10.1016/j.apal.2006.12.001. [Online]. Available: http://dx.doi.org/10.1016/j.apal.2006.12.001.

[16] J. Bernardy, P. Jansson, M. Zalewski, and S. Schupp, "Generic programming with C++ concepts and haskell type classes - a comparison," *J. Funct. Program.*, vol. 20, no. 3-4, pp. 271–302, 2010. DOI: 10.1017/S095679681000016X. [Online]. Available: http://dx.doi.org/10.1017/S095679681000016X.

[17] R. S. Bird, J. Gibbons, S. Mehner, J. Voigtländer, and T. Schrijvers, "Understanding idiomatic traversals backwards and forwards," in *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, C. Shan, Ed., ACM, 2013, pp. 25–36, ISBN: 978-1-4503-2383-3. DOI: 10.1145/2503778.2503781. [Online]. Available: http://doi.acm.org/10.1145/2503778.2503781.

[18] F. Bonchi and F. Zanasi, "Bialgebraic semantics for logic programming," *Logical Methods in Computer Science*, vol. 11, no. 1, 2015. DOI: 10.2168/LMCS-11(1:14)2015. [Online]. Available: http://dx.doi.org/10.2168/LMCS-11(1:14)2015.

[19] A. Colmerauer, "Equations and inequations on finite and infinite trees," in *FGCS*, 1984, pp. 85–99.

[20] P. Cousot and R. Cousot, "Inductive definitions, semantics and abstract interpretation," in *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, R. Sethi, Ed., ACM Press, 1992, pp. 83–94, ISBN: 0-89791-453-8. DOI: 10.1145/143165.143184. [Online]. Available: http://doi.acm.org/10.1145/143165.143184.

[21] K. Crary, R. Harper, and S. Puri, "What is a recursive module?" In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, B. G. Ryder and B. G. Zorn, Eds., ACM, 1999, pp. 50–63, ISBN: 1-58113-094-5. DOI: 10.1145/301618.301641. [Online]. Available: http://doi.acm.org/10.1145/301618.301641.

[22] D. Van Dalen, "Intuitionistic logic," in *Handbook of Philosophical Logic: Volume III: Alternatives in Classical Logic*, D. Gabbay and F. Guenthner, Eds. Dordrecht: Springer Netherlands, 1986, pp. 225–339, ISBN: 978-94-009-5203-4. DOI: 10.1007/978-94-009-5203-4_4. [Online]. Available: http://dx.doi.org/10.1007/978-94-009-5203-4_4.

[23] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, R. A. DeMillo, Ed., ACM Press, 1982, pp. 207–212, ISBN: 0-89791-065-6. DOI: 10.1145/582153.582176. [Online]. Available: http://doi.acm.org/10.1145/582153.582176.

[24] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons, "Fast and loose reasoning is morally correct," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. G. Morrisett and S. L. P. Jones, Eds., ACM, 2006, pp. 206–217, ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111056. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111056.

[25] E. DE ANGELIS, F. FIORAVANTI, A. PETTOROSSI, and M. PROIETTI, "Proving correctness of imperative programs by linearizing constrained horn clauses," *Theory and Practice of Logic Programming*, vol. 15, no. 4-5, pp. 635–650, 2015, ISSN: 1471-0684. DOI: 10.1017/S1471068415000289. [Online]. Available: http://dx.doi.org/10.1017/S1471068415000289http://www.journals.cambridge.org/abstract{_}S1471068415000289.

[26] D. De Schreye, V. Nys, and C. Nicholson, "Analysing and compiling coroutines with abstract conjunctive partial deduction," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8981, 2015, pp. 21–38, ISBN: 9783319178219. DOI: 10.1007/978-3-319-17822-6_2. [Online]. Available: http://link.springer.com/10.1007/978-3-319-17822-6{_}2.

[27] D. Devriese and F. Piessens, "On the bright side of type classes: Instance arguments in agda," in *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, M. M. T. Chakravarty, Z. Hu, and O. Danvy, Eds., ACM, 2011, pp. 143–155, ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034796. [Online]. Available: http://doi.acm.org/10.1145/2034773.2034796.

[28] A. Dijkstra, J. Fokker, and S. D. Swierstra, "The structure of the essential haskell compiler, or coping with compiler complexity," in *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, O. Chitil, Z. Horváth, and V. Zsók, Eds., ser. Lecture Notes in Computer Science, vol. 5083, Springer, 2007, pp. 57–74, ISBN: 978-3-540-85372-5. DOI: 10.1007/978-3-540-85373-2_4. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85373-2_4.

[29] M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli, "Declarative modeling of the operational behavior of logic languages," *Theor. Comput. Sci.*, vol. 69, no. 3, pp. 289–318, 1989. DOI: 10.1016/0304-3975(89)90070-4. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(89)90070-4.

[30] F. Farka, E. Komendantskaya, K. Hammond, and P. Fu, "Coinductive soundness of corecursive type class resolution," 2016. arXiv: 1608.05233. [Online]. Available: http://arxiv.org/abs/1608.05233.

[31] K. Faxén, "A static semantics for haskell," *J. Funct. Program.*, vol. 12, no. 4&5, pp. 295–357, 2002. DOI: 10.1017/S0956796802004380. [Online]. Available: http://dx.doi.org/10.1017/S0956796802004380.

[32] P. Fu and E. Komendantskaya, "A type-theoretic approach to resolution," in *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*, M. Falaschi, Ed., ser. Lecture Notes in Computer Science, vol. 9527, Springer, 2015, pp. 91–106, ISBN: 978-3-319-27435-5. DOI: 10.1007/978-3-319-27436-2_6. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-27436-2_6.

[33] ——, "Operational semantics of resolution and productivity in horn clause logic," *Formal Aspects of Computing*, pp. 1–22, 2016, ISSN: 0934-5043. DOI: 10.1007/s00165-016-0403-1. arXiv: 1604.04114. [Online]. Available: http://arxiv.org/abs/1604.04114http://link.springer.com/10.1007/s00165-016-0403-1.

[34] P. Fu, E. Komendantskaya, T. Schrijvers, and A. Pond, "Proof relevant corecursive resolution," in *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, O. Kiselyov and A. King, Eds., ser. Lecture Notes in Computer Science, vol. 9613, Springer, 2016, pp. 126–143, ISBN: 978-3-319-29603-6. DOI: 10.1007/978-3-319-29604-3_9. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-29604-3_9.

[35] N. Ghani and P. Hancock, "Containers, monads and induction recursion," *Mathematical Structures in Computer Science*, vol. 26, no. 1, pp. 89–113, 2016. DOI: 10.1017/S0960129514000127. [Online]. Available: http://dx.doi.org/10.1017/S0960129514000127.

[36] J. Gibbons and G. Hutton, "Proof methods for corecursive programs," *Fundam. Inform.*, vol. 66, no. 4, pp. 353–366, 2005. [Online]. Available: http://content.iospress.com/articles/fundamenta-informaticae/fi66-4-03.

[37] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer, "How to make ad hoc proof automation less ad hoc," *J. Funct. Program.*, vol. 23, no. 4, pp. 357–401, 2013. DOI: 10.1017/S0956796813000051. [Online]. Available: http://dx.doi.org/10.1017/S0956796813000051.

[38] G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya, "Coinductive logic programming and its applications," in *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, V. Dahl and I. Niemelä, Eds., ser. Lecture Notes in Computer Science, vol. 4670, Springer, 2007, pp. 27–44, ISBN: 978-3-540-74608-9. DOI: 10.1007/978-3-540-74610-2_4. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74610-2_4.

[39] C. V. Hall, K. Hammond, S. L. P. Jones, and P. Wadler, "Type classes in haskell," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 2, pp. 109–138, 1996. DOI: 10.1145/227699.227700. [Online]. Available: http://doi.acm.org/10.1145/227699.227700.

[40] R. Hinze and S. L. P. Jones, "Derivable type classes," *Electr. Notes Theor. Comput. Sci.*, vol. 41, no. 1, pp. 5–35, 2000. DOI: 10.1016/S1571-0661(05)80542-0. [Online]. Available: http://dx.doi.org/10.1016/S1571-0661(05)80542-0.

[41] R. Hinze and S. Peyton Jones, "Derivable type classes," *Electronic Notes in Theoretical Computer Science*, vol. 41, no. 1, pp. 5–35, 2001, ISSN: 15710661. DOI: 10.1016/S1571-0661(05)80542-0. [Online]. Available: http://dx.doi.org/10.1016/S1571-0661(05)80542-0http://linkinghub.elsevier.com/retrieve/pii/S1571066105805420.

[42] R. Hinze, N. Wu, and J. Gibbons, "Unifying structured recursion schemes," *ACM SIGPLAN Notices*, vol. 48, no. 9, pp. 209–220, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2544174.2500578.

[43] J. S. Hodas and D. Miller, "Logic programming in a fragment of intuitionistic linear logic," *Inf. Comput.*, vol. 110, no. 2, pp. 327–365, 1994. DOI: 10.1006/inco.1994.1036. [Online]. Available: http://dx.doi.org/10.1006/inco.1994.1036.

[44] P. Hudak, J. Hughes, S. L. P. Jones, and P. Wadler, "A history of haskell: Being lazy with class," in *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, B. G. Ryder and B. Hailpern, Eds., ACM, 2007, pp. 1–55. DOI: 10.1145/1238844.1238856. [Online]. Available: http://doi.acm.org/10.1145/1238844.1238856.

[45] G. Huet and A. Saibi, "Constructive category theory," in *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg*, Goteberg, 1995, p. 27.

[46] R. Iemhoff, "On the admissible rules of intuitionistic propositional logic," *J. Symb. Log.*, vol. 66, no. 1, pp. 281–294, 2001. DOI: 10.2307/2694922. [Online]. Available: http://dx.doi.org/10.2307/2694922.

[47] J. Jaffar and P. J. Stuckey, "Semantics of infinite tree logic programming," *Theor. Comput. Sci.*, vol. 46, no. 3, pp. 141–158, 1986. DOI: 10.1016/0304-3975(86)90027-7. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(86)90027-7.

[48] M. Jaskelioff and O. Rypacek, "An investigation of the laws of traversals," in *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012.*, J. Chapman and P. B. Levy, Eds., ser. EPTCS, vol. 76, 2012, pp. 40–49. DOI: 10.4204/EPTCS.76.5. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.76.5.

[49] P. Johann and N. Ghani, "A principled approach to programming with nested types in haskell," *Higher-Order and Symbolic Computation*, vol. 22, no. 2, pp. 155–189, 2009. DOI: 10.1007/s10990-009-9047-7. [Online]. Available: http://dx.doi.org/10.1007/s10990-009-9047-7.

[50] R. van Kesteren, M. C.J. D. van Eekelen, and M. de Mol, "Proof support for generic type classes," in *Revised Selected Papers from the Fifth Symposium on Trends in Functional Programming, TFP 2004, München, Germany, 25-26 November 2004.*, H. Loidl, Ed., ser. Trends in Functional Programming, vol. 5, Intellect, 2004, pp. 1–16, ISBN: 1-84150-144-1.

[51] O. Kiselyov and H. Ishii, "Freer monads, more extensible effects," in *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, B. Lippmeier, Ed., ACM, 2015, pp. 94–105, ISBN: 978-1-4503-3808-0. DOI: 10.1145/2804302.2804319. [Online]. Available: http://doi.acm.org/10.1145/2804302.2804319.

[52] P. Kokke and W. Swierstra, "Auto in agda - programming proof search using reflection," in *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, R. Hinze and J. Voigtländer, Eds., ser. Lecture Notes in Computer Science, vol. 9129, Springer, 2015, pp. 276–301, ISBN: 978-3-319-19796-8. DOI: 10.1007/978-3-319-19797-5_14. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-19797-5_14.

[53] E. Komendantskaya and J. Power, "Coalgebraic semantics for derivations in logic programming," in *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings*, A. Corradini, B. Klin, and C. Cîrstea, Eds., ser. Lecture Notes in Computer Science, vol. 6859, Springer, 2011, pp. 268–282, ISBN: 978-3-642-22943-5. DOI: 10.1007/978-3-642-22944-2_19. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22944-2_19.

[54] D. Kozen and A. Silva, "Practical coinduction," *Mathematical Structures in Computer Science*, 121, Feb. 2016. DOI: 10.1017/S0960129515000493.

[55] R. Lämmel and S. L. P. Jones, "Scrap your boilerplate with class: Extensible generic functions," in *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, O. Danvy and B. C. Pierce, Eds., ACM, 2005, pp. 204–215, ISBN: 1-59593-064-7. DOI: 10.1145/1086365.1086391. [Online]. Available: http://doi.acm.org/10.1145/1086365.1086391.

[56] C. S. Lee, N. D. Jones, and A. M. Ben-Amram, "The size-change principle for program termination," in *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, C. Hankin and D. Schmidt, Eds., ACM, 2001, pp. 81–92, ISBN: 1-58113-336-7. DOI: 10.1145/360204.360210. [Online]. Available: http://doi.acm.org/10.1145/360204.360210.

[57] M. Lenisa, J. Power, and H. Watanabe, "Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads," *Electr. Notes Theor. Comput. Sci.*, vol. 33, pp. 230–260, 2000. DOI: 10.1016/S1571-0661(05)80350-0. [Online]. Available: http://dx.doi.org/10.1016/S1571-0661(05)80350-0.

[58] F. Lindblad and M. Benke, "A tool for automated theorem proving in agda," in *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, J. Filliâtre, C. Paulin-Mohring, and B. Werner, Eds., ser. Lecture Notes in Computer Science, vol. 3839, Springer, 2004, pp. 154–169, ISBN: 3-540-31428-8. DOI: 10.1007/11617990_10. [Online]. Available: http://dx.doi.org/10.1007/11617990_10.

[59] J. W. Lloyd, *Foundations of Logic Programming, 2nd Edition*. Springer, 1987, ISBN: 3-540-18199-7.

[60] A. Löh and R. Hinze, "Open data types and open functions," in *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, A. Bossi and M. J. Maher, Eds., ACM, 2006, pp. 133–144, ISBN: 1-59593-388-3. DOI: 10.1145/1140335.1140352. [Online]. Available: http://doi.acm.org/10.1145/1140335.1140352.

[61] D. B. MacQueen, G. D. Plotkin, and R. Sethi, "An ideal model for recursive polymorphic types," *Information and Control*, vol. 71, no. 1/2, pp. 95–130, 1986. DOI: 10.1016/S0019-9958(86)80019-5. [Online]. Available: http://dx.doi.org/10.1016/S0019-9958(86)80019-5.

[62] C. McBride, "Faking it: Simulating dependent types in haskell," *J. Funct. Program.*, vol. 12, no. 4&5, pp. 375–392, 2002. DOI: 10.1017/S0956796802004355. [Online]. Available: http://dx.doi.org/10.1017/S0956796802004355.

[63] C. McBride and R. Paterson, "Applicative programming with effects," *J. Funct. Program.*, vol. 18, no. 1, pp. 1–13, 2008. DOI: 10.1017/S0956796807006326. [Online]. Available: http://dx.doi.org/10.1017/S0956796807006326.

[64] D. Miller and G. Nadathur, *Programming with Higher-Order Logic*. Cambridge University Press, 2012, ISBN: 978-0-521-87940-8. [Online]. Available: http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/programming-higher-order-logic?format=HB.

[65] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, "Uniform proofs as a foundation for logic programming," *Ann. Pure Appl. Logic*, vol. 51, no. 1-2, pp. 125–157, 1991. DOI: 10.1016/0168-0072(91)90068-W. [Online]. Available: http://dx.doi.org/10.1016/0168-0072(91)90068-W.

[66] R. Milner, "A theory of type polymorphism in programming," *J. Comput. Syst. Sci.*, vol. 17, no. 3, pp. 348–375, 1978. DOI: 10.1016/0022-0000(78)90014-4. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(78)90014-4.

[67] ——, *Communication and concurrency*, ser. PHI Series in computer science. Prentice Hall, 1989, ISBN: 978-0-13-115007-2.

[68] I. Moerdijk and E. Palmgren, "Wellfounded trees in categories," *Ann. Pure Appl. Logic*, vol. 104, no. 1-3, pp. 189–218, 2000. DOI: 10.1016/S0168-0072(00)00012-9. [Online]. Available: http://dx.doi.org/10.1016/S0168-0072(00)00012-9.

[69] J. H. Morris, "Lambda-calculus models of programming languages," PhD thesis, Massachusetts Institute of Technology, 1968, p. 134. DOI: 1721.1/64850.

[70] L. S. Moss and N. Danner, "On the foundations of corecursion," *Logic Journal of the IGPL*, vol. 5, no. 2, pp. 231–257, 1997. DOI: 10.1093/jigpal/5.2.231. [Online]. Available: http://dx.doi.org/10.1093/jigpal/5.2.231.

[71] U. Norell, "Dependently typed programming in agda," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5832 LNCS, 2009, pp. 230–266, ISBN: 3642046517. DOI: 10.1007/978-3-642-04652-0_5.

[72] B. C. d. S. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi, "The implicit calculus: A new foundation for generic programming," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds., ACM, 2012, pp. 35–44, ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254070. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254070.

[73] L. C. Paulson and A. W. Smith, "Logic programming, functional programming, and inductive definitions," *CoRR*, vol. cs.LO/9301109, 1993. [Online]. Available: http://arxiv.org/abs/cs.LO/9301109.

[74] B. C. Pierce, *Types and programming languages*. MIT Press, 2002, ISBN: 978-0-262-16209-8.

[75] A. M. Pitts, "Parametric polymorphism and operational equivalence," *Electr. Notes Theor. Comput. Sci.*, vol. 10, pp. 2–27, 1997. DOI: 10.1016/S1571-0661(05)80685-1. [Online]. Available: http://dx.doi.org/10.1016/S1571-0661(05)80685-1.

[76] G. D. Plotkin and J. Power, "Algebraic operations and generic effects," *Applied Categorical Structures*, vol. 11, no. 1, pp. 69–94, 2003. DOI: 10.1023/A:1023064908962. [Online]. Available: http://dx.doi.org/10.1023/A:1023064908962.

[77] J. Power and H. Watanabe, "Combining a monad and a comonad," *Theor. Comput. Sci.*, vol. 280, no. 1-2, pp. 137–162, 2002. DOI: 10.1016/S0304-3975(01)00024-X. [Online]. Available: http://dx.doi.org/10.1016/S0304-3975(01)00024-X.

[78] G. Rosu and D. Lucanu, "Circular coinduction: A proof theoretical foundation," in *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings*, A. Kurz, M. Lenisa, and A. Tarlecki, Eds., ser. Lecture Notes in Computer Science, vol. 5728, Springer, 2009, pp. 127–144, ISBN: 978-3-642-03740-5. DOI: 10.1007/978-3-642-03741-2_10. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03741-2_10.

[79] J. J.M. M. Rutten, "Behavioural differential equations: a coinductive calculus of streams, automata, and power series," *Theoretical Computer Science*, vol. 308, no. 1-3, pp. 1–53, 2003, ISSN: 03043975. DOI: 10.1016/S0304-3975(02)00895-2.

[80] D. Sangiorgi, "On the origins of bisimulation and coinduction," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 4, 2009. DOI: 10.1145/1516507.1516510. [Online]. Available: http://doi.acm.org/10.1145/1516507.1516510.

[81] D. D. Schreye, V. Nys, and C. J. Nicholson, "Analysing and compiling coroutines with abstract conjunctive partial deduction," in *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers*, M. Proietti and H. Seki, Eds., ser. Lecture Notes in Computer Science, vol. 8981, Springer, 2014, pp. 21–38, ISBN: 978-3-319-17821-9. DOI: 10.1007/978-3-319-17822-6_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-17822-6_2.

[82] T. Schrijvers, S. L. P. Jones, M. M. T. Chakravarty, and M. Sulzmann, "Type checking with open type functions," in *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, J. Hook and P. Thiemann, Eds., ACM, 2008, pp. 51–62, ISBN: 978-1-59593-919-7. DOI: 10.1145/1411204.1411215. [Online]. Available: http://doi.acm.org/10.1145/1411204.1411215.

[83] L. Simon, A. Bansal, A. Mallya, and G. Gupta, "Co-logic programming: Extending logic programming with coinduction," in *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, Eds., ser. Lecture Notes in Computer Science, vol. 4596, Springer, 2007, pp. 472–483, ISBN: 978-3-540-73419-2. DOI: 10.1007/978-3-540-73420-8_42. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73420-8_42.

[84] A. Simpson and G. Plotkin, "Complete axioms for categorical fixed-point operators," *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pp. 30–41, 2000. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=855753.

[85] S. Staton, "An algebraic presentation of predicate logic - (extended abstract)," in *Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, F. Pfenning, Ed., ser. Lecture Notes in Computer Science, vol. 7794, Springer, 2013, pp. 401–417, ISBN: 978-3-642-37074-8. DOI: 10.1007/978-3-642-37075-5_26. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37075-5_26.

[86] P. J. Stuckey and M. Sulzmann, "A theory of overloading," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1216–1269, 2005. DOI: 10.1145/1108970.1108974. [Online]. Available: http://doi.acm.org/10.1145/1108970.1108974.

[87] M. Sulzmann, G. J. Duck, S. L. P. Jones, and P. J. Stuckey, "Understanding functional dependencies via constraint handling rules," *J. Funct. Program.*, vol. 17, no. 1, pp. 83–129, 2007. DOI: 10.1017/S0956796806006137. [Online]. Available: http://dx.doi.org/10.1017/S0956796806006137.

[88] G. Sutcliffe, "The tptp problem library and associated infrastructure : the fof and cnf parts, v3.5.0," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 337–362, 2009, ISSN: 01687433. DOI: 10.1007/s10817-009-9143-8.

[89] H. Thielemann, "How to refine polynomial functions," *IJWMIP*, vol. 10, no. 3, 2012. DOI: 10.1142/S0219691312500270. [Online]. Available: http://dx.doi.org/10.1142/S0219691312500270.

[90] D. Vytiniotis, S. L. P. Jones, T. Schrijvers, and M. Sulzmann, "Outsidein(x) modular type inference with local assumptions," *J. Funct. Program.*, vol. 21, no. 4-5, pp. 333–412, 2011. DOI: 10.1017/S0956796811000098. [Online]. Available: http://dx.doi.org/10.1017/S0956796811000098.

[91] P. Wadler, "Theorems for free!" In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, J. E. Stoy, Ed., ACM, 1989, pp. 347–359, ISBN: 0-201-51389-7. DOI: 10.1145/99370.99404. [Online]. Available: http://doi.acm.org/10.1145/99370.99404.

[92] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad-hoc," in *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, ACM Press, 1989, pp. 60–76, ISBN: 0-89791-294-2. DOI: 10.1145/75277.75283. [Online]. Available: http://doi.acm.org/10.1145/75277.75283.

[93] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, "Horn clause solvers for program verification," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9300, 2015, pp. 24–51, ISBN: 9783319235332. DOI: 10.1007/978-3-319-23534-9_2. [Online]. Available: http://link.springer.com/10.1007/978-3-319-23534-9{_}2.

[94] H. Geuvers and R. Nederpelt, "N.g. de bruijn's contribution to the formalization of mathematics," *Indagationes Mathematicae*, vol. 24, no. 4, pp. 1034–1049, 2013, ISSN: 00193577. DOI: 10.1016/j.indag.2013.09.003. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0019357713000700.

[95] R. Jhala, R. Majumdar, and A. Rybalchenko, "Hmc : verifying functional programs," pp. 470–485, 2011.

[96] E. Komendantskaya and J. Power, "Logic programming: laxness and saturation," 2016. arXiv: 1608.07708. [Online]. Available: http://arxiv.org/abs/1608.07708.

[97] ——, "Logic programming: laxness and saturation," 2016. arXiv: 1608.07708. [Online]. Available: http://arxiv.org/abs/1608.07708.

[98] N. P. Mendler, P. Panangaden, P. J. Scott, and R. A. G. Seely, "A logical view of concurrent constraint programming," *Nordic J. of Computing*, vol. 2, no. 2, pp. 181–220, 1995.

[99] M. Odersky, M. Sulzmann, and M. Wehr, "Type inference with constrained types," *Theory and Practice of Object Systems*, vol. 5, no. 1, pp. 35–55, 1999, ISSN: 1074-3227. DOI: 10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4. [Online]. Available: http://doi.wiley.com/10.1002/{\%}28SICI{\%}291096-9942{\%}28199901/03{\%}295{\%}3A1{\%}3C35{\%}3A{\%}3AAID-TAPO4{\%}3E3.0.CO{\%}3B2-4.

[100] P. W. O'Hearn and R. D. Tennent, "Parametricity and local variables," *Journal of the ACM*, vol. 42, no. 3, pp. 658–709, 1995, ISSN: 00045411. DOI: 10.1145/210346.210425. [Online]. Available: http://portal.acm.org/citation.cfm?doid=210346.210425.

[101] C.-H. L. Ong and S. J. Ramsay, "Verifying higher-order functional programs with pattern-matching algebraic data types," *SIGPLAN Not.*, vol. 46, no. 1, pp. 587–598, 2011, ISSN: 0362-1340. DOI: 10.1145/1925844.1926453. [Online]. Available: http://doi.acm.org/10.1145/1925844.1926453{\%}5Cnhttp://dl.acm.org/ft{_}gateway.cfm?id=1926453{\&}type=pdf.

[102] F. Pfenning, "Logic programming in the lf logical framework," *First Workshop on Logical Frameworks*, pp. 1–25, 1991. [Online]. Available: http://books.google.com/books?hl=en{\&}lr={\&}id=X9wfWwslFQIC{\&}oi=fnd{\&}pg=PA149{\&}dq=Logic+Programming+in+the+LF+Logical+Framework{\&}ots=LfrwT41GfT{\&}sig=AymIHgqAw{_}M3EiIPGvlpnR2J34M.

[103] F. Pfenning and C. Schürmann, "System description: twelf a meta-logical framework for deductive systems," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1632, pp. 202–206, 1999, ISSN: 16113349. DOI: 10.1007/3-540-48660-7_14.

[104] V. Simonet and F. Pottier, "A constraint-based approach to guarded algebraic data types," *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 1, 1–es, 2007, ISSN: 01640925. DOI: 10.1145/1180475.1180476. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1180475.1180476.