Semantic Analysis of Normalisation for Directional Logic Programming

Vikraman Choudhury
vikraman.choudhury@unibo.it
Università di Bologna & Inria OLAS
Bologna, Italy

Neel Krishnaswami
nk480@cl.cam.ac.uk
University of Cambridge
Cambridge, UK

Ariadne Si Suo
ss3039@cam.ac.uk
University of Cambridge
Cambridge, UK

Abstract

This work is about using *directional logic programming* to give foundations to mode-correct bidirectional type systems [2, § 6.2]. Reddy[5] uses a term language for *classical linear logic*, adapted from Abramsky's *Linear Chemical Abstract Machine (LCHAM)* [1], to give a typed calculus for directional logic programs. We give a categorical semantics to Reddy's calculus, using polycategories. We give normalisation results for this calculus, which shows how to evaluate logic queries to normal forms, that gives output substitutions indicating whether queries fail or succeed.

Keywords: classical linear logic, categorical semantics, directional logic programming, bidirectional typechecking

Directional Logic Programming

Type systems are logic programs, which can be run forwards or backwards giving bidirectional type-checking – type checking and type synthesis [2]. However, there is a subtle issue about mode-correctness. Logic programs may not terminate, for example, consider the simple list reversal:

$\mathsf{reverse}([],[])$.
$\mathsf{reverse}(p::x,y) :\!-\ \mathsf{reverse}(x,z),\ \mathsf{append}(z,[p],y).$

If the implementation uses left-to-right literal selection, then $\mathsf{reverse}(x,[1,2,3])$ won't terminate. If the implementation uses right-to-left literal selection, then $\mathsf{reverse}([1,2,3],x)$ won't terminate.

We can ensure termination by adding mode annotations to arguments. In the example above, we may specify:

$$\textbf{mode}\ \mathsf{reverse}(+,-). \qquad \textbf{mode}\ \mathsf{append}(+,+,-).$$

where $+$ means the argument should always be in input position, and $-$ means the argument should be in output position. Then $\mathsf{reverse}(x,[1,2,3])$ won't be an acceptable goal, as it has a variable in the input position. Now, as long as the implementation uses left-to-right literal selection, all acceptable goals will terminate [5, Pg. 4].

Rather than treating entire arguments as inputs or outputs, we would like a more fine-grained notion of mode, that allows embedding inputs in outputs, and vice versa. Classical linear logic can be used to combine modes with types, where linear negation switches input and output modes. For example, reverse gets the type:

$$\mathsf{reverse} : \textbf{Proc}(\mathsf{List}(A) \otimes \mathsf{List}(A)^{\perp})$$

This is the idea behind *directional* logic programming – type-checking ensures that every well-typed program is mode-correct. Reddy[5]'s language for directional logic programming is presented as a sequent calculus for classical linear logic, with *terms*, *patterns*, *variables*, and *commands*, and an untyped operational semantics for commands, that enjoys subject reduction.

Polycategorical Semantics

To give an *adequate* denotational semantics to Reddy's language, we would like to produce a typed equational theory. First, we advocate for the use of polycategories for its categorical semantics, since this sequent calculus presentation is the internal language of polycategories.

We consider the free (symmetric) polycategory $\mathcal{P}$ on a polygraph of base types and constant polymaps. Following Reddy's presentation, this is written as a sequent calculus with two-sided sequents $\Gamma \vdash \{\theta\}\ \Delta$ as judgements, where commands θ correspond to derivations of polymaps, and equations (on typed derivations, or commands) correspond to (symmetric) polycategory axioms. The free polycategory has a presentation using polycategorical trees [3, Prop. 2.9] (the polycategorical version of directed paths giving free categories), and we observe that this gives *normal forms* for typed derivations (or commands) on base types.

Then, we consider the free linearly distributive category on $\mathcal{P}$, given by $\mathsf{LinDist}(\mathcal{P})$, which freely adds $\otimes$s and $\parr$s of formulas, and their *left* and *right* rules in the sequent calculus. This is Reddy's calculus, without duals or additives.

Finally, every linearly distributive category has an underlying polycategory, given by $\mathsf{Poly}(\mathsf{LinDist}(\mathcal{P}))$. By using the freeness of $\mathcal{P}$, we get a functor of polycategories $\eta_{\mathcal{P}} : \mathcal{P} \longrightarrow \mathsf{Poly}(\mathsf{LinDist}(\mathcal{P}))$, which is just the unit of the adjunction $\mathsf{LinDist} \dashv \mathsf{Poly} : \textbf{LinDistCat} \longrightarrow \textbf{PolyCat}$, at $\mathcal{P}$.

By the Hyland-Shulman envelope construction [4, 6], this functor is a polycategorical Yoneda-type embedding, making it fully-faithful. Syntactically, this corresponds to a logical relations proof of principal cut-elimination. Given a polymap in $\mathsf{Poly}(\mathsf{LinDist}(\mathcal{P}))$, we produce a *canonical* polymap in $\mathcal{P}$, which is a principal-cut-free derivation, giving a normal form for derivations (or commands) in Reddy's language. This corresponds to cut-elimination for LCHAM in [1]. In terms of logic programming, the normal form is the list of output substitutions obtained by running a Prolog query.

Further work is necessary to add (non-empty) additives, using products and coproducts, and duals $(-)^{\perp}$, using star-polycategories.

Acknowledgments

This work is partly supported by the EU Marie-Skłodowska-Curie action "ReGraDe-CS", grant № 101106046 (https://doi.org/10.3030/101106046), and partly by the European Research Council Consolidator Grant for the project "TypeFoundry", funded under the European Union's Horizon 2020 Framework Programme grant № 101002277 (https://doi.org/10.3030/101002277).

References

[1] Samson Abramsky. 1993. Computational interpretations of linear logic. *Theoretical Computer Science*, 111, 1, (Apr. 12, 1993), 3–57. DOI: 10.1016/0304-3975(93)90181-R (cit. on p. 1).

[2] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.*, 54, 5, (May 25, 2021), 98:1–98:38. DOI: 10.1145/3450952 (cit. on p. 1).

[3] Richard Garner and Tom Hirschowitz. 2018. Shapely monads and analytic functors. *Journal of Logic and Computation*, 28, 1, (Feb. 1, 2018), 33–83. DOI: 10.1093/logcom/exx029 (cit. on p. 1).

[4] J.M.E. Hyland. 2002. Proof theory in the abstract. *Annals of Pure and Applied Logic*, 114, 1-3, (Apr. 2002), 43–78. DOI: 10.1016/S0168-0072(01)00075-6 (cit. on p. 1).

[5] Uday S. Reddy. 1993. A typed foundation for directional logic programming. In *Extensions of Logic Programming*. Vol. 660. E. Lamma and P. Mello, (Eds.) Red. by J. Siekmann, G. Goos, and J. Hartmanis. Springer Berlin Heidelberg, Berlin, Heidelberg, 282–318. ISBN: 978-3-540-56454-6 978-3-540-47562-0. DOI: 10.1007/3-540-56454-3_15 (cit. on p. 1).

[6] Michael Shulman. 2021. *-Autonomous Envelopes and Conservativity. *Electronic Proceedings in Theoretical Computer Science*, 353, (Dec. 30, 2021), 175–194. DOI: 10.4204/EPTCS.353.9 (cit. on p. 1).