Denotational semantics in impredicative guarded dependent type theory

April 17, 2023

Jonathan Sterling (joint work with Gratzer & Birkedal)

Aarhus University

Table of Contents

What is denotational semantics?

Denotational semantics is an approach to studying programs comprised by the following scientific hypotheses:

1. a **type** $\boxed{\tau}$ denotes mathematical structure $[\![\tau]\!]$;
2. a **program** $\boxed{x : \sigma \vdash M : \tau}$ denotes a *homomorphism* of structures from $[\![M]\!] : [\![\sigma]\!] \longrightarrow [\![\tau]\!]$;
3. **compositionality:** the denotation of a program is built from the denotations of its subparts, *e.g.* $[\![M + N]\!] = [\![M]\!] + [\![N]\!]$.

Strengths and weaknesses of denotational semantics

Strengths of denotational semantics:

1. it is **modular** and **reusable**;
2. **mathematical abstractions** are available to solve problems;
3. **language extensibility** is built in from the start.

Weaknesses of denotational semantics:

1. some language features **challenge compositionality**;
2. denotational semantics for realistic languages with state and concurrency is **exceedingly complex**.

Weaknesses above led to the current (regrettable) hegemony of **operational semantics**: strong results but no explanations.

Operational *vs.* denotational semantics

Operational semantics is an approach to studying programs that emphasises the composition of **program execution steps** rather than the composition of **program fragments**.

Strengths of operational semantics:

1. it **scales** to realistic languages of extreme complexity;
2. it is **simple** enough for people who dislike mathematics;
3. it is easy to **mechanize** in proof assistants like Coq.

Weaknesses of operational semantics:

1. it is **monolithic**, impedes **composition** and **reuse**;
2. mathematical abstractions are **difficult to adapt**;
3. it struggles to accommodate language extensibility.

A return to denotations in birth…

Purely operational methods are giving way to a **hybrid regime** in which denotational ideas provide critical input:

- **abstraction**: step-indexed Kripke logical relations with recursive worlds *(oh my!)*, tamed by ***guarded domain theory***;
- **compositionality**: see the recent use of ***interaction trees*** in the DeepSpec project for compositional reasoning about impure first-order programs;
- **applications**: there is an increasing need to write programs on ***actual spaces***, as in differentiable programming for AI or probabilistic programming for the sciences.

Denotational semantics for realistic PLs

Domain theory is <u>**the**</u> account of general recursion, but has struggled to combine more complex features, including two that are now table-stakes in operational semantics:

- ***higher-order store***: where you can store effectful functions and pointers in the heap;
- ***concurrency***: many advances in the denotational semantics world (*e.g.* powerdomains & event structures), unfinished. **meanwhile:** great strides in practical operational accounts of concurrency.

Today's talk: I will show how to combine ***guarded recursion*** with ***polymorphic types*** to easily define denotational models of higher-order store with polymorphism.

Table of Contents

Monadic System $\mathbf{F}^\omega$ with reference types

Our language is a version of System $\mathbf{F}^\omega$ extended by an "IO monad" with reference types:

$$
\begin{aligned}
&\mathbf{IO} : \star \to \star \\
&\mathbf{IORef} : \star \to \star \\
&\mathbf{get} : \mathbf{IORef}\ \alpha \to \mathbf{IO}\ \alpha \\
&\mathbf{set} : \mathbf{IORef}\ \alpha \to \alpha \to \mathbf{IO}\ () \\
&\mathbf{new} : \alpha \to \mathbf{IO}\ (\mathbf{IORef}\ \alpha)
\end{aligned}
$$

Note: standard CBV translation lets you program in ML style without the monad.

Kripke semantics of reference types

The classic *state monad* handles a single cell of fixed type:

$$\mathbf{State}\ \sigma\ \alpha = \sigma \to (\sigma \times \alpha)$$

Our situation is harder: we can allocate new cells, and store anything we want in there.

Thus the denotation $[\![\mathbf{IORef}\ \alpha]\!]$ must depend on the "current" heap layout, which is always growing.

The solution is to *parameterize* $[\![-]\!]$ in heap layouts and require all denotations to be *monotone* in the growth of the heap (Reynolds, Oles, O'Hearn, *etc.*). Called **Kripke semantics**.

Defining the poset $\mathcal{W}$ of heap layouts

A ***heap layout*** w should map a finite set of global addresses to *semantic types*. A ***semantic type*** A should be a *family of sets* A_w indexed in heap layouts w with transition functions for each heap inclusion, *i.e.* a **functor** from heap layouts to sets.

Defining the poset $\mathcal{W}$ of heap layouts

A ***heap layout*** w should map a finite set of global addresses to *semantic types*. A ***semantic type*** A should be a *family of sets* A_w indexed in heap layouts w with transition functions for each heap inclusion, *i.e.* a **functor** from heap layouts to sets.

This is circular, in a bad way! When $\mathcal{U}$ is some non-trivial set of sets, we cannot solve the following system of equations:

$$\mathcal{W} \cong \mathsf{Addr} \xrightarrow{\mathit{fin.}} \mathcal{T}$$
$$\mathcal{T} \cong \mathbf{Functor}(\mathcal{W}, \mathcal{U})$$

Solved in the operational setting by Appel & McAllester's *step-indexing*, further developed by Amal Ahmed in her *tour-de-force* PhD thesis.

A step-indexed poset of heap layouts

The idea of Appel and McAllester was, roughly, to *stratify* the definition of $\mathcal{W}$ in its unrollings of finite depth. We can give a straightforward denotational version.

Idea: every set is replaced by a *contravariantly* ω-indexed family of sets, *i.e.* a functor $\omega^{op} \longrightarrow \mathbf{Set}$.

$$\mathcal{W}_n = \mathsf{Addr} \xrightarrow{\mathit{fin.}} \varprojlim_{k<n} \mathcal{T}_k$$
$$\mathcal{T}_n = \mathbf{Functor}(\mathcal{W}_n \times \omega^{op}, \mathbf{Set})$$

The above is well-defined! But it is also gnarly... **We can tame it with *guarded dependent type theory*.**

Denotational semantics in guarded type theory

Guarded dependent type theory / **GDTT** is a version of dependent type theory whose purpose is to speak of functors $\omega^{op} \longrightarrow \mathbf{Set}$.

GDTT has so far been used to give elegant denotational semantics to *non-polymorphic* languages with general recursion, recursive types, and non-determinism.

See the work of Birkedal, Møgelberg, Paviotti, Veltri, Vezzosi, *etc*.

The later modality in guarded type theory

Guarded type theory extends ordinary type theory / set theory with a "later modality" ▶ which allows you to interpret general recursion:

$$[\![\blacktriangleright A]\!]_n = \varprojlim_{k<n} [\![A]\!]_k$$

$$\frac{u : X}{\mathsf{next}\, u : \blacktriangleright X} \qquad \frac{F : \blacktriangleright X \to X}{\mathbf{fix}_X\, F : X \qquad \mathbf{fix}_X\, F = F(\mathsf{next}\,(\mathbf{fix}_X\, F))}$$

From guarded recursion to general recursion

The kinds of recursive functions supported by $\mathbf{fix}_X$ are *guarded*, in the sense that the recursive call is trapped under $\blacktriangleright$.

To actually program, we need an operation to get rid of $\blacktriangleright$; a type X equipped with an operation $\sigma_X : \blacktriangleright X \to X$ is called a ***guarded domain***.

Example

The universe **Set** is a guarded domain for which $\sigma_{\mathbf{Set}}(\mathsf{next}\, A) = \blacktriangleright A$.

Example

For any type A : **Set**, we have the *free guarded domain monad* $\mathbf{L}A = A + \blacktriangleright \mathbf{L}A$ on A, defined using $\mathbf{fix}_{\mathbf{Set}}$. Then $\sigma_{\blacktriangleright \mathbf{L}A} = \mathbf{inr}$.

Recursive types and functions in guarded type theory

Let $F : X \to X$ be an arbitrary operator on a guarded domain X.

$$\mu F \triangleq \mathsf{fix}_X(\lambda x.\boldsymbol{\sigma}_X(\blacktriangleright F x))$$

We have $\mu F = \boldsymbol{\sigma}_X(F(\mu F))$. The algebra $\boldsymbol{\sigma}_X$ tracks each "step" of unfolding an infinite object.

1. This gives recursive types when $X = \mathbf{Set}$.
2. It gives recursive functions à la CBV when $X = (\mathbb{N} \to \mathbf{L}\mathbb{N})$.

Returning to worlds and heaps...

We return to the painfully explicit definition of heap layouts (worlds) to clean it up using guarded type theory as a DSL.

$$\mathcal{W}_n = \mathsf{Addr} \xrightarrow{\mathit{fin.}} \varprojlim_{k<n} \mathcal{T}_k$$
$$\mathcal{T}_n = \mathbf{Functor}(\mathcal{W}_n \times \omega^{op}, \mathbf{Set})$$

Returning to worlds and heaps...

We return to the painfully explicit definition of heap layouts (worlds) to clean it up using guarded type theory as a DSL.

$$\mathcal{W} = \mathsf{Addr} \xrightarrow{\mathit{fin.}} \blacktriangleright\mathcal{T}$$
$$\mathcal{T} = \mathbf{Functor}(\mathcal{W}, \mathbf{Set})$$

Returning to worlds and heaps...

We return to the painfully explicit definition of heap layouts (worlds) to clean it up using guarded type theory as a DSL.

$$\mathcal{W} = \mathsf{Addr} \xrightarrow{\mathit{fin.}} \blacktriangleright\mathcal{T}$$
$$\mathcal{T} = \mathbf{Functor}(\mathcal{W}, \mathbf{Set})$$

The above means the exact same thing, but it is much easier to work with! It is our responsibility to say "No!" more often to ***index hell***.

The **IORef** type in guarded type theory

We can now give the denotation of the **IORef** type in **GDTT**.

$$
\begin{aligned}
&[\![\mathbf{IORef}]\!] : \mathcal{T} \to \mathcal{T} \\
&[\![\mathbf{IORef}]\!]\, A = \\
&\quad \lambda w : \mathcal{W}. \\
&\qquad \left\{ l \in |w| \;\middle|\; wl = \mathsf{next}\, A \right\}
\end{aligned}
$$

The **IORef** type in guarded type theory

We can now give the denotation of the **IORef** type in **GDTT**.

$$
\begin{aligned}
&[\![\mathbf{IORef}]\!] : \mathcal{T} \to \mathcal{T} \\
&[\![\mathbf{IORef}]\!]\, A = \\
&\quad \lambda w : \mathcal{W}. \\
&\qquad \left\{ l \in |w| \;\middle|\; wl = \mathsf{next}\, A \right\}
\end{aligned}
$$

Are we done? **No.**

Defining the IO monad?

Suppose we want to define **IO** as a kind of state monad. First we must define what the states (heaps) are:

$$\mathbf{H}_w : \mathbf{Set} \text{ for each } w : \mathcal{W}$$
$$\mathbf{H}_w = \prod_{l \in |w|} \boldsymbol{\sigma}_{\mathbf{Set}}(\blacktriangleright(-w)\ (wl))$$

A naïve attempt to define $[\![\mathbf{IO}]\!]$, using the *free guarded domain monad* $\mathbf{L}X = X + \blacktriangleright X$ to support general recursion.

$$
\begin{aligned}
&[\![\mathbf{IO}]\!] : \mathcal{T} \to \mathcal{T}\\
&[\![\mathbf{IO}]\!]\,A = \\
&\quad \lambda w : \mathcal{W}.\\
&\qquad \mathbf{H}_w \to \mathbf{L}\,(\mathbf{H}_w \times Aw)
\end{aligned}
$$

A naïve attempt to define $[\![\mathbf{IO}]\!]$, using the *free guarded domain monad* $\mathbf{L}X = X + \blacktriangleright X$ to support general recursion.

$$
\begin{aligned}
&[\![\mathbf{IO}]\!] : \mathcal{T} \to \mathcal{T}\\
&[\![\mathbf{IO}]\!]\, A = \\
&\quad \lambda w : \mathcal{W}.\\
&\qquad \mathbf{H}_w \to \mathbf{L}\,(\mathbf{H}_w \times Aw)
\end{aligned}
$$

Wrong in so many ways!

1. it is ill-defined / not monotone in $w : \mathcal{W}$;
2. it does not support allocating any new cells!

$$
\begin{aligned}
&[\![\mathbf{IO}]\!] : \mathcal{T} \to \mathcal{T} \\
&[\![\mathbf{IO}]\!]\, A = \\
&\quad \lambda w : \mathcal{W}. \\
&\qquad \mathbf{H}_w \to \mathbf{L}\,(\mathbf{H}_w \times A w)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathbf{IO}]\!] : \mathcal{T} \to \mathcal{T} \\
&[\![\mathbf{IO}]\!]\, A = \\
&\quad \lambda w : \mathcal{W}. \\
&\qquad \mathbf{H}_w \to \mathbf{L}\,(\mathbf{H}_w \times A w)
\end{aligned}
$$

- First we have to make it monotone in heap expansion.

$$
\begin{aligned}
&[\![\mathbf{IO}]\!] : \mathcal{T} \to \mathcal{T} \\
&[\![\mathbf{IO}]\!]\, A = \\
&\quad \lambda w : \mathcal{W}. \\
&\qquad \textstyle\prod_{w' \geqslant w} \mathbf{H}_{w'} \to \mathbf{L}\left(\mathbf{H}_{w'} \times A w'\right)
\end{aligned}
$$

- First we have to make it monotone in heap expansion.

$$
\begin{aligned}
&[\![\mathbf{IO}]\!] : \mathcal{T} \to \mathcal{T} \\
&[\![\mathbf{IO}]\!]\, A = \\
&\quad \lambda w : \mathcal{W}. \\
&\qquad \textstyle\prod_{w' \geqslant w} \mathbf{H}_{w'} \to \mathbf{L}\left(\mathbf{H}_{w'} \times A w'\right)
\end{aligned}
$$

- First we have to make it monotone in heap expansion.
- But we still can't allocate new cells during computation.

$$
\begin{aligned}
&[\![\mathbf{IO}]\!] : \mathcal{T} \to \mathcal{T}\\
&[\![\mathbf{IO}]\!]\, A = \\
&\quad \lambda w : \mathcal{W}.\\
&\qquad \textstyle\prod_{w' \geqslant w} \mathbf{H}_{w'} \to \mathbf{L} \sum_{w'' \geqslant w'} \mathbf{H}_{w''} \times A w''
\end{aligned}
$$

- First we have to make it monotone in heap expansion.
- But we still can't allocate new cells during computation.

$$
\begin{aligned}
&[\![\mathbf{IO}]\!] : \mathcal{T} \to \mathcal{T}\\
&[\![\mathbf{IO}]\!]\, A = \\
&\quad \lambda w : \mathcal{W}.\\
&\qquad \prod_{w' \geqslant w} \mathbf{H}_{w'} \to \mathbf{L} \sum_{w'' \geqslant w'} \mathbf{H}_{w''} \times A w''
\end{aligned}
$$

- First we have to make it monotone in heap expansion.
- But we still can't allocate new cells during computation.
- OK, but now the whole thing is ill-defined:
  - we are trying to construct a type $[\![\mathbf{IO}]\!]\, A\, w : \mathbf{Set}$
  - but **Set** is not closed under $\mathcal{W}$-indexed products and sums, since $\mathcal{W}$ is as big as **Set**!

$$
\begin{aligned}
&[\![\mathbf{IO}]\!] : \mathcal{T} \to \mathcal{T} \\
&[\![\mathbf{IO}]\!]\, A = \\
&\quad \lambda w : \mathcal{W}. \\
&\qquad \prod_{w' \geqslant w} \mathbf{H}_{w'} \to \mathbf{L} \sum_{w'' \geqslant w'} \mathbf{H}_{w''} \times A w''
\end{aligned}
$$

- First we have to make it monotone in heap expansion.
- But we still can't allocate new cells during computation.
- OK, but now the whole thing is ill-defined:
  - we are trying to construct a type $[\![\mathbf{IO}]\!]\, A\, w : \mathbf{Set}$
  - but **Set** is not closed under $\mathcal{W}$-indexed products and sums, since $\mathcal{W}$ is as big as **Set**!

Thus **GDTT** is *inadequate* for defining a typed denotational semantics of higher-order store.

A fork in the road

Two potential ways forward.

A fork in the road

Two potential ways forward.

1. **drop the dream and revert to *untyped* semantics**, replacing **Set**, $\prod$, $\sum$ with $\mathcal{P}(\mathbf{V})$, $\bigcap$, $\bigcup$ where $\mathbf{V}$ is some universal domain; this works because powersets are complete lattices.

A fork in the road

Two potential ways forward.

1. **drop the dream and revert to *untyped* semantics**, replacing **Set**, $\prod$, $\sum$ with $\mathcal{P}(\mathbf{V})$, $\bigcap$, $\bigcup$ where **V** is some universal domain; this works because powersets are complete lattices.
2. **or just add impredicative polymorphism to GDTT**:

$$[\![\mathbf{IO}]\!]\, A\, w = \forall_{w' \geqslant w} \mathbf{H}_{w'} \to \mathbf{L} \exists_{w'' \geqslant w'} \mathbf{H}_{w''} \times A w''$$

A fork in the road

Two potential ways forward.

1. **drop the dream and revert to *untyped* semantics**, replacing **Set**, $\prod$, $\sum$ with $\mathcal{P}(\mathbf{V})$, $\bigcap$, $\bigcup$ where $\mathbf{V}$ is some universal domain; this works because powersets are complete lattices.
2. **or just add impredicative polymorphism to GDTT**:

$$[\![\mathbf{IO}]\!]\, A\, w = \forall_{w' \geqslant w} \mathbf{H}_{w'} \to \mathbf{L} \exists_{w'' \geqslant w'} \mathbf{H}_{w''} \times A w''$$

(By the way, we must solve this problem already if we want to model System F's universal types anyway.)

Table of Contents

Impredicative guarded dependent type theory

iGDTT extends the the Birkedal–Møgelberg–Paviotti program of guarded denotational semantics to languages that combine **polymorphism** with **realistic computational effects**.

iGDTT augments **GDTT** with the “impredicative `Set`” universe from the old calculus of constructions / Coq, which we will call **iSet**.

The definition of iGDTT, formally

The structure of **iGDTT** is as follows:

1. a hierarchy of predicative universes $\mathbf{Type}_i$;
2. an **impredicative** universe $\mathbf{Prop} \in \mathbf{Type}_i$ of proof-irrelevant types satisfying propositional extensionality;
3. an **impredicative** universe $\mathbf{iSet} \in \mathbf{Type}_i$ with $\mathbf{Prop} \subseteq \mathbf{iSet}$;
4. all universes have $\prod$, $\sum$, $(=)$, inductive types, and $\blacktriangleright$.

Note that $\mathbf{Prop} \notin \mathbf{iSet}$ and $\mathbf{Prop}$ is *not* a subobject classifier!

Universal and existential types in iGDTT

An impredicative universe $\mathbb{X} \in \mathbf{Type}_i$ is one that is closed under *large* universal quantification:

$$\frac{A : \mathbf{Type}_i \qquad x : A \vdash Bx : \mathbb{X}}{\forall_{x:A} Bx : \mathbb{X}} \qquad \uparrow_{\mathbb{X}}^{\mathbf{Type}_i}\left(\forall_{x:A} Bx\right) \cong \prod_{x:A} \uparrow_{\mathbb{X}}^{\mathbf{Type}_i}(Bx)$$

Universal and existential types in iGDTT

An impredicative universe $\mathbb{X} \in \mathbf{Type}_i$ is one that is closed under *large* universal quantification:

$$\frac{A : \mathbf{Type}_i \qquad x : A \vdash Bx : \mathbb{X}}{\forall_{x:A} Bx : \mathbb{X}} \qquad \uparrow_{\mathbb{X}}^{\mathbf{Type}_i} (\forall_{x:A} Bx) \cong \prod_{x:A} \uparrow_{\mathbb{X}}^{\mathbf{Type}_i} (Bx)$$

If $\mathbb{X}$ is closed under $(=)$, then it is automatically closed under *existential quantification*, via the coherent impredicative encoding of Awodey, Frey, and Speight (2018).

$$(\exists_{x:A} Bx) \subseteq \forall_{C:\mathbb{X}} \forall_{k:\prod_{x:A} \prod_{b:Bx} C} C$$

Although $\forall_{x:A} Bx$ is the dependent product, it is *not* the case that $\exists_{x:A} Bx$ is the dependent sum. (It is a so-called “weak sum”.)

Simple denotational semantics of state in iGDTT

Finally our denotational semantics can be defined!

$$
\begin{aligned}
&\mathcal{W} = \mathsf{Addr} \xrightarrow{\mathit{fin.}} \blacktriangleright\mathcal{T} \\
&\mathcal{T} = \mathbf{Functor}(\mathcal{W}, \mathbf{iSet}) \\
&\mathbf{H}_w = \textstyle\prod_{l \in |w|} \boldsymbol{\sigma}_{\mathbf{iSet}}(\blacktriangleright(-w)\ (wl))
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathbf{IORef}]\!] : \mathcal{T} \to \mathcal{T} \\
&[\![\mathbf{IORef}]\!]\, A\, w = \left\{ l \in |w| \;\middle|\; wl = \mathsf{next}\, A \right\}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathbf{IO}]\!] : \mathcal{T} \to \mathcal{T} \\
&[\![\mathbf{IO}]\!]\, A\, w = \forall_{w' \geqslant w} \mathbf{H}_{w'} \to \mathbf{L} \exists_{w'' \geqslant w'} \mathbf{H}_{w''} \times Aw''
\end{aligned}
$$

Wait, how do we know this is OK?

The original **GDTT** was justified in the topos of trees $\mathbf{Functor}(\omega^{op}, \mathbf{Set})$. What about **iGDTT**?

1. Take *any* non-trivial realizability topos $\mathbf{E}$;
2. Take *any* non-trivial internal well-founded poset $\mathbb{O}$ in $\mathbf{E}$;
3. Then the category of *internal* diagrams $\mathbf{Functor}_{\mathbf{E}}(\mathbb{O}^{op}, \mathbf{E})$ is a non-trivial model of **iGDTT**.

Just black-box this (as you already do with results of *e.g.* set theory).

What I did not have time to tell you about…

1. ***Dependent types, now***: model construction *also* justifies a version of **iGDTT** with an **IO** monad! (Important for languages like Idris 2 and Lean 4, which currently have no semantics.)
2. ***Easy to extend*** with any additional algebraic effect.
3. ***Relational reasoning*** with imperative ADTs supported via LRAT.
4. ***Higher-order separation logic*** over denotational semantics: see manuscript of S., **Aagaard**, and B.

What you should take away from this...

1. **The past two decades of operationally-based exploration have been extremely fruitful;** applications of step-indexing have reached a high degree of sophistication while denotations slept.
2. **Denotational semantics is *not* as hopeless as you have been told;** state of the art languages *can* have straightforward denotational models that *simplify* and *clarify* the standard operational models.
3. **Denotational semantics scales effortlessly to full dependent type theory with higher-order state**, whereas operational semantics of dependent type theory is a tarpit.
4. **Full abstraction is a *non-goal* of working semanticists;** different models explain ***different facets*** of a program's behavior in terms of its components, period. Logics like **Iris** are essential to *reason* about **both** the operational and denotational models.
5. ***Go forth, and denote!***

Future work

1. Experiment with a *resumption-style* version of our monad, to prepare for concurrency.
2. Refine the model to support more fine-grained invariants (*e.g*. heap islands, transition systems, *etc*.).
3. Refine the model to validate the equational theory of *garbage collection* (joint work with O. Kammar).
4. Adapt the model of higher-order separation logic to support higher-order ghost state (joint work with F. L. Aagaard).

Thanks!

Funded by
the European Union

References I

- Ahmed, Amal Jamil (2004). "Semantics of Types for Mutable State". PhD thesis. Princeton University. URL: http://www.ccs.neu.edu/home/amal/ahmedthesis.pdf.
- Appel, Andrew W. and David McAllester (Sept. 2001). "An Indexed Model of Recursive Types for Foundational Proof-carrying Code". In: *ACM Transactions on Programming Languages and Systems* 23.5, pp. 657–683. ISSN: 0164-0925. DOI: 10.1145/504709.504712.
- Appel, Andrew W., Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon (2007). "A Very Modal Model of a Modern, Major, General Type System". In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Nice, France: Association for Computing Machinery, pp. 109–122. ISBN: 1-59593-575-4.
- Awodey, Steve, Jonas Frey, and Sam Speight (2018). "Impredicative Encodings of (Higher) Inductive Types". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. Oxford, United Kingdom: Association for Computing Machinery, pp. 76–85. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209130.
- Birkedal, Lars, Aleš Bizjak, et al. (2019). "Guarded Cubical Type Theory". In: *Journal of Automated Reasoning* 63.2, pp. 211–253. DOI: 10.1007/s10817-018-9471-7.
- Birkedal, Lars, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring (2011). "First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees". In: *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society, pp. 55–64. ISBN: 978-0-7695-4412-0. DOI: 10.1109/LICS.2011.16. arXiv: 1208.3596 [cs.LO].

References II

Birkedal, Lars, Kristian Støvring, and Jacob Thamsborg (2010). "Realisability semantics of parametric polymorphism, general references and recursive types". In: *Mathematical Structures in Computer Science* 20.4, pp. 655–703. DOI: 10.1017/S0960129510000162.

Bizjak, Aleš et al. (2016). "Guarded Dependent Type Theory with Coinductive Types". In: *Foundations of Software Science and Computation Structures: 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*. Ed. by Bart Jacobs and Christof Löding. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 20–35. ISBN: 978-3-662-49630-5. DOI: 10.1007/978-3-662-49630-5_2. arXiv: 1601.01586 [cs.LO].

Levy, Paul Blain (2003a). "Adjunction Models For Call-By-Push-Value With Stacks". In: *Electronic Notes in Theoretical Computer Science* 69. CTCS'02, Category Theory and Computer Science, pp. 248–271. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)80568-1.

— (Jan. 1, 2003b). *Call-by-Push-Value: A Functional/Imperative Synthesis*. Kluwer, Semantic Structures in Computation, 2. ISBN: 1-4020-1730-8.

Møgelberg, Rasmus Ejlers and Marco Paviotti (2016). "Denotational Semantics of Recursive Types in Synthetic Guarded Domain Theory". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. New York, NY, USA: Association for Computing Machinery, pp. 317–326. ISBN: 978-1-4503-4391-6. DOI: 10.1145/2933575.2934516.

References III

- Møgelberg, Rasmus Ejlers and Niccolò Veltri (Jan. 2019). "Bisimulation as Path Type for Guarded Recursive Types". In: *Proceedings of the ACM on Programming Languages* 3.POPL. DOI: 10.1145/3290317.
- Møgelberg, Rasmus Ejlers and Andrea Vezzosi (Dec. 2021). "Two Guarded Recursive Powerdomains for Applicative Simulation". In: *Proceedings 37th Conference on Mathematical Foundations of Programming Semantics*. Vol. 351. Electronic Proceedings in Theoretical Computer Science, pp. 200–217. DOI: 10.4204/EPTCS.351.13.
- Palombi, Daniele and Jonathan Sterling (Feb. 2023). "Classifying topoi in synthetic guarded domain theory". In: *Electronic Notes in Theoretical Informatics and Computer Science* Volume 1 - Proceedings of MFPS XXXVIII. DOI: 10.46298/entics.10323. URL: https://entics.episciences.org/10323.
- Paviotti, Marco (2016). "Denotational semantics in Synthetic Guarded Domain Theory". PhD thesis. Denmark: IT-Universitetet i København. ISBN: 978-87-7949-345-2.
- Paviotti, Marco, Rasmus Ejlers Møgelberg, and Lars Birkedal (2015). "A Model of PCF in Guarded Type Theory". In: *Electronic Notes in Theoretical Computer Science* 319.Supplement C. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI), pp. 333–349. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2015.12.020.
- Sterling, Jonathan, Daniel Gratzer, and Lars Birkedal (July 2022). "Denotational semantics of general store and polymorphism". Unpublished manuscript. DOI: 10.48550/arXiv.2210.02169.

References IV

Sterling, Jonathan and Robert Harper (Oct. 2021). “Logical Relations as Types: Proof-Relevant Parametricity for Program Modules”. In: *Journal of the ACM* 68.6. ISSN: 0004-5411. DOI: 10.1145/3474834. arXiv: 2010.08599 [cs.PL].

Veltri, Niccolò and Andrea Vezzosi (2020). “Formalizing π-Calculus in Guarded Cubical Agda”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. New Orleans, LA, USA: Association for Computing Machinery, pp. 270–283. ISBN: 978-1-4503-7097-4. DOI: 10.1145/3372885.3373814.