An invitation to univalent foundations

of mathematics and computer science

Dr Jon Sterling

Associate Professor in Logical
Foundations and Formal Methods

Computer Laboratory
University of Cambridge

Wednesday Seminar
25 October 2023

A refactoring problem...

Suppose we have written a program, *e.g*. a web server, that uses a *queue data structure* as part of its implementation.

A refactoring problem...

Suppose we have written a program, *e.g*. a web server, that uses a *queue data structure* as part of its implementation.

Because we were just prototyping, we started off with the naïve implementation based on linked lists — for which **dequeueing is linear** in the size of the queue. Our server is *dying* under trivial loads!

A refactoring problem...

Suppose we have written a program, *e.g*. a web server, that uses a *queue data structure* as part of its implementation.

Because we were just prototyping, we started off with the naïve implementation based on linked lists — for which **dequeueing is linear** in the size of the queue. Our server is *dying* under trivial loads!

We then replace the naïve queue implementation by a data structure with better amortised complexity, *e.g*. a batched queue.

A refactoring problem...

Suppose we have written a program, *e.g*. a web server, that uses a *queue data structure* as part of its implementation.

Because we were just prototyping, we started off with the naïve implementation based on linked lists — for which **dequeueing is linear** in the size of the queue. Our server is *dying* under trivial loads!

We then replace the naïve queue implementation by a data structure with better amortised complexity, *e.g*. a batched queue.

Question: *How can we be sure that the behavior of the web server is preserved by this refactoring?*

A non-rigorous (but correct!) argument...

1. The original server only interacted with the queue through the queue operations.
2. As the new queue implementation is also functionally correct, surely the web server remains as correct as it ever was.
3. QED ???

A non-rigorous (but correct!) argument...

1. The original server only interacted with the queue through the queue operations.
2. As the new queue implementation is also functionally correct, surely the web server remains as correct as it ever was.
3. QED ???

But how do we know that we did not secretly depend on "undocumented" behavior of the original queue structure? We need to argue that the two queues are equivalent (somehow) and that these equivalences are preserved by the construction of web servers.

A non-rigorous (but correct!) argument...

1. The original server only interacted with the queue through the queue operations.
2. As the new queue implementation is also functionally correct, surely the web server remains as correct as it ever was.
3. QED ???

But how do we know that we did not secretly depend on "undocumented" behavior of the original queue structure? We need to argue that the two queues are equivalent (somehow) and that these equivalences are preserved by the construction of web servers.

To make these arguments precise and rigorous, we need a ***foundation for computer programming*** *that lets us reason about and exploit equivalence of distinct data structures.*

What is a “foundation”?

A **foundation** concentrates prior scientific experience into a *language*. Like any human language, a foundation has both a *grammar* and a *vocabulary*, which can vary greatly in their fidelity to a given sphere of experience.

What is a “foundation”?

A **foundation** concentrates prior scientific experience into a *language*. Like any human language, a foundation has both a *grammar* and a *vocabulary*, which can vary greatly in their fidelity to a given sphere of experience.

- Euclid’s axioms for geometry
- Zermelo’s axioms for sets ($\in$-trees)
- Eilenberg–Steenrod axioms for homology theories
- Grothendieck’s axioms for Abelian categories
- Hyland and Phoa’s axioms for synthetic computability(*) theory
- Voevodsky’s univalent foundations

What is a “foundation”?

A **foundation** concentrates prior scientific experience into a *language*. Like any human language, a foundation has both a *grammar* and a *vocabulary*, which can vary greatly in their fidelity to a given sphere of experience.

- Euclid’s axioms for geometry
- Zermelo’s axioms for sets ($\in$-trees)
- Eilenberg–Steenrod axioms for homology theories
- Grothendieck’s axioms for Abelian categories
- Hyland and Phoa’s axioms for synthetic computability(*) theory
- Voevodsky’s univalent foundations

I am interested in the foundations of ***compositional reasoning about computer programs***.

Different approaches to foundations

The *grammar* of mathematics can be organised in many ways.

Material set theory: first-order logic over a theory with a *single* type of entity (sets), formulas built up from the $\in$-relation. Example: ZFC.

Categorical set theories: first-order logic over a theory with two types: sets and functions. Example: ETCS.

Simple type theory / HOL: first-order logic over a theory with many types, including a type of propositions. Example: Isabelle/HOL.

Dependent type theories: algebraic(*) theories allowing many types, each corresponding to a different kind of mathematical entity (*e.g.* integers, reals, automata, vector spaces, propositions, *etc.*). Examples: Martin-Löf Type Theory, Calculus of Constructions, **HoTT**.

Analytic *vs*. synthetic theories of sets

Mathematical practice is far-removed from the vocabulary of set theoretic foundations — for instance, when was the last time you mentioned the "axiom of regularity" in a proof? Even the axioms of "pairing" and "union" are deeply impertinent...

Everyday mathematical practice works at a higher level of abstraction. The basic move are things like:

1. The product of a family of sets is a set.
2. The disjoint sum of a family of sets is a set.
3. The quotient of a set by an equivalence relation is a set.

Type theory takes the above (and beyond) as its *primitive* rules.

Whereas collections in set theory are built analytically from $\in$-trees, type theory delivers a **synthetic theory of collections**.

Primer on (dependent) type theory à la Martin-Löf

There are two kinds of entity in type theory: *types* and *elements*. (*Sets* will be described later as special types.)

$$\boxed{A : \textbf{Type}} \qquad\qquad \boxed{u : A}$$

The colon is *not* a predicate: it is an annotation. Just as in Java or Haskell, the type of an element is part of that element.

The vocabulary of type theory unfolds by specifying rules to form (new) types and elements.

The vocabulary of type theory

Let $A : \mathbf{Type}$ be a type, and let $B(x) : \mathbf{Type}$ be a family of types indexed in $x : A$. Then:

1. The product $\prod_{(x:A)} B(x)$ is a type; an element of $\prod_{(x:A)} B(x)$ is given by a function $f(x) : B(x)$ varying in $x : A$.
2. The disjoint sum $\sum_{(x:A)} B(x)$ is a type; an element of $\sum_{(x:A)} B(x)$ is given by a pair (u, v) where $u : A$ and $v : B(u)$.

We also have a type $\mathbb{N} : \mathbf{Type}$ of natural numbers, defined inductively by two constructors: zero and successor.

Equality and identification in type theory

In traditional mathematics, there is an *equality* predicate $x = y$ defined for any entities x, y of the same type.

The principle of equivalence. The important thing about equality is that respected by *everything*! If $x = y$ then $P(x) = P(y)$.

Type theory takes (roughly) the above as a *definition* of what it means for two entities to be the same. **In type theory, this principle is compatible with more than just traditional equality** — a famous result of Hofmann and Streicher [7].

The principle of equivalence and univalent foundations

In type theory, traditional equality is *not* the only thing that is respected by all expressible notions! Hofmann and Streicher [7] and Voevodsky [18] proposed to take advantage of this fact.

"Univalence": If $f : A \to B$ is a bijective function, then any expressible property of A also holds of B.

Univalent type theory / *homotopy type theory* takes the above principle of univalence as a basic axiom, alongside the axioms of products and disjoint sums, *etc*.

Achieved by replacing equality with a more general and more flexible notion of "identification" that includes equality, isomorphism, and more.

Identification types

Fix a type $A : \mathbf{Type}$.

1. **Formation.** Given elements $x, y : A$ we may form the *identification type* $x =_A y : \mathbf{Type}$.
2. **Construction.** Given $x : A$ we may form the *reflexivity identification* $\mathrm{refl}_x : x =_A x$.
3. **Induction.** Given $x : A$, to define a family of functions $f(y, p) : B(y, p)$ varying in $y : A$ and $p : x =_A y$, it suffices to define $p(x, \mathrm{refl}_x) : B(x, \mathrm{refl}_x)$.

Identification induction

Induction. *Given $x : A$, to define a family of functions $f(y, p) : B(y, p)$ varying in $y : A$ and $p : x =_A y$, it suffices to define $p(x, \mathrm{refl}_x) : B(x, \mathrm{refl}_x)$.*

$\mathrm{symmetry} : \prod_{(x,y:A)} \prod_{(p:x=_A y)} y =_A x$
$\mathrm{symmetry}\, x\, x\, \mathrm{refl}_x :\equiv \mathrm{refl}_x$

$\mathrm{transitivity} : \prod_{(x,y:A)} \prod_{(p:x=_A y)} \prod_{(z:A)} \prod_{(q:y=_A z)} x =_A z$
$\mathrm{transitivity}\, x\, x\, \mathrm{refl}_x\, z\, q :\equiv q$

$\mathrm{congruence} : \prod_{(x,y:A)} \prod_{(p:x=_A y)} \prod_{(f:A\to B)} fx =_B fy$
$\mathrm{congruence}\, x\, x\, \mathrm{refl}_x\, f :\equiv \mathrm{refl}_{fx}$

But you cannot prove: $\prod_{(x,y:A)} \prod_{(p,q:x=_A y)} p =_{x=_A y} q$, except in the case of certain types A (which will be called “sets”).

Propositions and sets in univalent foundations

Voevodsky used identification types to isolate types that behave like *propositions (truth values)* and like *sets* from traditional foundations.

Propositions and sets in univalent foundations

Voevodsky used identification types to isolate types that behave like *propositions (truth values)* and like *sets* from traditional foundations.

Definition

A type $A : \mathbf{Type}$ is called a ***proposition*** when any two of its elements may be identified, *i.e*. we have:

$$\mathrm{isProp}(A) :\equiv \prod_{(x,y:A)} x =_A y$$

Propositions and sets in univalent foundations

Voevodsky used identification types to isolate types that behave like *propositions (truth values)* and like *sets* from traditional foundations.

Definition

A type $A : \mathbf{Type}$ is called a ***proposition*** when any two of its elements may be identified, *i.e.* we have:

$$\mathrm{isProp}(A) :\equiv \prod_{(x,y:A)} x =_A y$$

Definition

A type $A : \mathbf{Type}$ is called a ***set*** when for any $x, y : A$ the identification type $x =_A y$ is a proposition, *i.e.* we have:

$$\begin{aligned}\mathrm{isSet}(A) &:\equiv \prod_{(x,y:A)} \mathrm{isProp}(x =_A y) \\ &\equiv \prod_{(x,y:A)} \prod_{(p,q:x=_A y)} p =_{x=_A y} q\end{aligned}$$

Singletons and unique existence

Many definitions in mathematics take the form, “The unique function that . . . ”. Unique existence in univalent foundations is expressed here in terms of *singleton types*, a.k.a. *contractible types*.

Definition

A type $A : \mathbf{Type}$ is called a ***singleton*** when we have an element $a_0 : A$ (called the “center of contraction”) with which every other element may be identified, *i.e.* we have:

$$\mathrm{isSingl}(A) :\equiv \sum_{(a_0:A)} \prod_{(y:A)} y =_A a_0$$

We have an infinite hierarchy of “truncation levels”:

$$\mathrm{isSingl}(A) \to \mathrm{isProp}(A) \to \mathrm{isSet}(A) \to \cdots$$

Note that there exist types that do not live in *any* of these levels.

Non-propositions and non-sets in univalent foundations

The simplest set that is not a proposition is $\mathbf{2} \equiv \{0, 1\}$. For a type that is not a set, we may consider the type **Set** of all (small) sets!

In univalent foundations, identifications between sets are given by *isomorphisms*. **Set** fails to be a set *not* because of size paradoxes, but because there can be more than one isomorphism between two sets.

Characterizing identification types in univalent foundations

The built-in notion of identification is global, just like equality in set theory. For any specific kind of object (like sets, groups, *etc*.) we may prove a *characterization theorem* that makes the structure of identifications explicit. For example:

1. Identifications between functions $f, g : A \to B$ are given by *homotopies* $h : \prod_{(x:A)} fx =_A gx$.
2. Identifications between sets are given by isomorphisms.
3. Identifications between groups are given by *invertible group homomorphisms*.
4. Identifications between categories are given by *categorical equivalences* (full, faithful, and essentially surjective functors).

All these characterizations follow from the axioms.

Univalent foundations of computer programming

The system I have so far described is secretly a *programming language*. We can *run* the code and even analyze its complexity.

Dependent types handle two things:

1. Parameterization of program *modules* in their dependencies.
2. Specification and verification of functional correctness using induction and equational reasoning.

The impact of *univalence* is to expand the power of equational reasoning to include representation independence.

Fearless refactoring with univalence

We return to our original refactoring problem. We have written a web server that uses a *queue data structure* as part of its implementation.

Because we were just prototyping, we started off with the naïve implementation based on linked lists — for which **dequeueing is linear** in the size of the queue. Bad performance!

We then replace the naïve queue implementation by a data structure with better amortised complexity.

But how can we be sure that the resulting web server will still function correctly? **Univalence answers precisely this question**, as shown by Angiuli *et al*. [3].

Functional queues in univalent foundations

In univalent foundations, we may describe the type of *functional queues* as an iterated disjoint sum — which we write as a "record".

record QueueStructure : **Type** :≡

queue : **Set**
init : *queue*
enqeue : message × *queue* → *queue*
dequeue : *queue* → option (message × *queue*)

Functional queues in univalent foundations

In univalent foundations, we may describe the type of *functional queues* as an iterated disjoint sum — which we write as a "record".

record QueueStructure : **Type** :≡

$queue$: **Set**

$init$: $queue$

$enqeue$: message $\times$ $queue$ $\to$ $queue$

$dequeue$: $queue$ $\to$ option (message $\times$ $queue$)

...

$dequeueEnqueueLaw$:

$$\prod_{(m:\text{message})} \prod_{(q:queue)} dequeue\,(enqueue\,(m, q)) =$$

case $dequeue\ q$ **of**

| None $\hookrightarrow$ Some $(a, init)$

| Some $(m', q') \hookrightarrow$ Some $(m', enqueue\,(a, q'))$

...

A naïve queue implementation

We can define an implementation ListQueue : QueueStructure by taking to ListQueue.*queue* be the type of functional linked lists.

data List $(A : \mathbf{Set}) : \mathbf{Set} :\equiv$
$\quad \mid nil : \mathrm{List}\, A$
$\quad \mid cons : A \times \mathrm{List}\, A \to \mathrm{List}\, A$

ListQueue : QueueStructure
ListQueue.*queue* $:\equiv$ List message
ListQueue.*init* $:\equiv$ *nil*
ListQueue.*enqueue* $(m, q) :\equiv cons\,(m, q)$
$\ldots$

To implement ListQueue.*dequeue* we must reach the *end* of the list!

Back to our refactoring problem

Recall that we have written web server that uses ListQueue as part of its implementation; suppose we also proved that it is correct.

$$server : \text{WebServer}$$
$$serverCorrect : \text{WebServerCorrectness}\ server$$

As we only interacted with ListQueue through the operations specified by QueueStructure, it is not difficult to *generalise* our server into a function:

$$generalisedServer : \text{QueueStructure} \to \text{WebServer}$$

If we build a faster queue structure, we can plug it into our server.

A less naïve queue implementation

To achieve better amortised complexity, we can define a new queue implementation that uses *two* linked lists: one for the front and one for the back.

$$\begin{aligned}&\mathsf{BatchedQueue} : \mathsf{QueueStructure}\\&\mathsf{BatchedQueue}.\mathit{queue} :\equiv \mathsf{List}\ \mathsf{message} \times \mathsf{List}\ \mathsf{message}\\&\ldots\end{aligned}$$

All operations are amortised constant time.

But the *dequeueEnqueueLaw* does not hold, so this does not define a queue structure! Angiuli *et al*. [3] teach us how to proceed.

Quotient inductive types

We can define a *refinement* of the batched queue data structure that will actually satisfy the equational theory of queues using a **quotient inductive type**.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{DoubleList}\ (A : \textbf{Set}) : \textbf{Set}\ \textbf{where} \\
&\quad \mid \mathit{make} : \mathsf{List}\ A \times \mathsf{List}\ A \to \mathsf{DoubleList}\ A \\
&\quad \mid \mathit{tilt} : \prod_{(u,v:\mathsf{List}\ A)} \\
&\qquad \mathit{make}\ (u + \mathit{cons}\ (a, \mathit{nil}), v) =_{\mathsf{DoubleList}\ A} \\
&\qquad\quad \mathit{make}\ (u, v + \mathit{cons}\ (a, \mathit{nil}))
\end{aligned}
$$

To define a function *out* of a "double list", you must prove that you preserve the "tilting invariant".

We have an isomorphism $\mathsf{DoubleList}\ A \cong \mathsf{List}\ A$, and this isomorphism is linear in the size of the input!

A corrected batched queue structure

Using our quotiented double lists, we may try once again to define batched queues.

BatchedQueue : QueueStructure
BatchedQueue.*queue* :≡ DoubleList message
. . .

The new definition goes through! Possible because all the operations for batched queues obviously preserve the tilting invariant.

Although the representation is isomorphic to lists, we have constant amortised complexity.

Can we migrate the correctness proof?

We now have:

$$server, fastServer : \mathrm{WebServer}$$
$$serverCorrect : \mathrm{WebServerCorrectness}\ server$$

$$server \equiv generalisedServer\ \mathrm{ListQueue}$$
$$fastServer :\equiv generalisedServer\ \mathrm{BatchedQueue}$$

But we have lost our correctness proof: we want to have $fastServerCorrect : \mathrm{WebServerCorrectness}\ fastServer$.

Univalence provides a way to do this, using *queue structure isomorphisms*.

Queue structure isomorphisms

Let Q_1, Q_2 : QueueStructure be two queue implementations.

Definition

An *homomorphism* from Q_1 to Q_2 is given by a mapping

$$\phi : Q_1.queue \to Q_2.queue$$

that preserves the queue operations:

$$\phi\, Q_1.init = Q_2.init$$
$$\phi\,(Q_1.enqueue\,(m, q)) = Q_2.enqueue\,(m, \phi\, q)$$
$$\mathsf{mapOption}\,(\mathsf{id} \times \phi)\,(Q_1.dequeue\, q) = Q_2.dequeue\,(\phi\, q)$$

Definition

A *isomorphism* ϕ: $Q_1 \cong Q_2$ is given by mutually inverse queue homomorphisms ϕ: $Q_1 \to Q_2$ and ϕ^{-1}: $Q_2 \to Q_1$.

The univalence principle for queue structures

The following is a consequence of the univalence principles for sets, functions, and disjoint sums.

Theorem

Let $Q_1 : \mathsf{QueueStructure}$ *be a queue structure. Then the map*

$$\prod_{(Q_2:\mathsf{QueueStructure})} \prod_{(p:Q_1 =_{\mathsf{QueueStructure}} Q_2)} Q_1 \cong Q_2$$

sending Q_1, refl_{Q_1} *to the identity isomorphism* $Q_1 \cong Q_1$ *is an isomorphism.*

In other words, queue structure isomorphisms induce queue structure identifications!

An identification between the slow and fast servers

Recall that we have an isomorphism $\mathsf{DoubleList}\ A \cong \mathsf{List}\ A$; we can extend this isomorphism of sets to a *queue structure isomorphism* $\mathsf{BatchedQueue} \cong \mathsf{ListQueue}$.

Thus by the univalence principle, we have a queue structure identification $\mathsf{BatchedQueue} =_{\mathsf{QueueStructure}} \mathsf{ListQueue}$.

We may thus define by congruence an identification

$$\mathit{server} =_{\mathsf{WebServer}} \mathit{fastServer}$$

as we have both $\mathit{server} \equiv \mathit{generalisedServer}\ \mathsf{ListQueue}$ and $\mathit{fastServer} \equiv \mathit{generalisedServer}\ \mathsf{BatchedQueue}$.

Transporting correctness proofs along identifications

The laws of identification types allow us to transfer correctness proofs along identifications. Given $S_1 : \mathsf{WebServer}$, we define:

$$\prod_{(S_2 : \mathsf{WebServer})} \prod_{(p : S_1 =_{\mathsf{WebServer}} S_2)} \mathsf{WebServerCorrectness}\ S_1 \to \mathsf{WebServerCorrectness}\ S_2$$

to be the map sending S_1, refl_{S_1} to the identity function on $\mathsf{WebServerCorrectness}\ S_1$.

Applying the above to our identification $\mathit{server} =_{\mathsf{WebServer}} \mathit{fastServer}$ and our original correctness proof

$$\mathit{serverCorrect} : \mathsf{WebServerCorrectness}\ \mathit{server}$$

we obtain the desired correctness proof

$$\mathit{fastServerCorrect} : \mathsf{WebServerCorrectness}\ \mathit{fastServer}.$$

How do programs in univalent type theory run?

When we add a feature to a programming language, we always have to say *how it is to be executed*. So far we have spoken of the univalence principle as an *axiom*: how do we run it?

Discovering how to run programs written in univalent type theory was one of the main struggles for type theorists in the 2010s.

Thierry Coquand suggested the use of **cubical type theory** as a more refined language that might brook such an execution semantics.

Coquand's conjectures on cubes

Around 2016, Simon Huber and Carlo Angiuli independently discovered how to run cubical programs *without free variables* [8, 1], confirming the weaker version of **Coquand's conjecture**.

In 2020–2021, I verified the strong version of Coquand's conjecture with the assistance of Carlo Angiuli — providing a decision procedure for *open terms* that fully characterizes the equational theory of cubical type theory [14, 13] and implements a form of *symbolic/residualized computation* in the presence of variables.

Together, these results mean that we can (in principle) build programming languages based on univalent foundations. **Today you can play with Cubical Agda!** [17]

Some future directions

1. **Can univalent type theory incorporate realistic language features (like state, concurrency, *etc*.)?** S., Gratzer, and Birkedal [15] show that univalence has significant impact on equational reasoning for higher-order imperative/OO programs. More work is needed on the semantic side.

Some future directions

1. **Can univalent type theory incorporate realistic language features (like state, concurrency, *etc*.)?** S., Gratzer, and Birkedal [15] show that univalence has significant impact on equational reasoning for higher-order imperative/OO programs. More work is needed on the semantic side.
2. **Can univalent programming languages be compiled to efficient machine code?**

Some future directions

1. **Can univalent type theory incorporate realistic language features (like state, concurrency, *etc*.)?** S., Gratzer, and Birkedal [15] show that univalence has significant impact on equational reasoning for higher-order imperative/OO programs. More work is needed on the semantic side.
2. **Can univalent programming languages be compiled to efficient machine code?**
3. **Geometrical reasoning about higher-dimensional identifications?** *cf*. `homotopy.io`

Some future directions

1. **Can univalent type theory incorporate realistic language features (like state, concurrency, *etc*.)?** S., Gratzer, and Birkedal [15] show that univalence has significant impact on equational reasoning for higher-order imperative/OO programs. More work is needed on the semantic side.
2. **Can univalent programming languages be compiled to efficient machine code?**
3. **Geometrical reasoning about higher-dimensional identifications?** *cf*. `homotopy.io`
4. **Impact of univalence on semantics of true concurrency?** (Joyal–Nielsen–Winskel open map bisimulation [9] exploits *descent* for coproducts: univalence = descent for *all* colimits)

Thanks! Please email me or drop by my office (FE24) if you'd like to collaborate.

Selected references I

[1] C. Angiuli. *Computational Semantics of Cartesian Cubical Type Theory*. PhD thesis, Carnegie Mellon University, 2019.

[2] C. Angiuli, G. Brunerie, T. Coquand, K.-B. Hou (Favonia), R. Harper, and D. R. Licata. Syntax and models of Cartesian cubical type theory. *Mathematical Structures in Computer Science*, 31(4):424–468, 2021. doi: $10.1017/\mathrm{S}0960129521000347$.

[3] C. Angiuli, E. Cavallo, A. Mörtberg, and M. Zeuner. Internalizing representation independence with univalence. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–30, Jan. 2021. doi: $10.1145/3434293$.

[4] S. Awodey. Structuralism, invariance, and univalence. *Philosophia Mathematica*, 22(1):1–11, 2014.

Selected references II

[5] S. Awodey and M. A. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146(1):45–55, Jan. 2009. ISSN 0305-0041. doi: 10.1017/S0305004108001783.

[6] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. *IfCoLog Journal of Logics and their Applications*, 4(10): 3127–3169, Nov. 2017.

[7] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998. doi: 10.1093/oso/9780198501275.001.0001.

[8] S. Huber. *Cubical Interpretations of Type Theory*. PhD thesis, University of Gothenberg, 2016.

Selected references III

[9] A. Joyal, M. Nielsen, and G. Winskel. Bisimulation from open maps. *Information and Computation*, 127(2):164–185, 1996. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.1996.0057.

[10] RedPRL Development Team. `redtt`, 2018. URL `https://www.github.com/RedPRL/redtt`.

[11] RedPRL Development Team. `cooltt`, 2020. URL `https://www.github.com/RedPRL/cooltt`.

[12] E. Rijke. Introduction to homotopy type theory. To appear, Cambridge University Press, 2022.

[13] J. Sterling. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD thesis, Carnegie Mellon University, 2021. Version 1.1, revised May 2022.

Selected references IV

[14] J. Sterling and C. Angiuli. Normalization for cubical type theory. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–15, Los Alamitos, CA, USA, July 2021. IEEE Computer Society. doi: 10.1109/LICS52264.2021.9470719.

[15] J. Sterling, D. Gratzer, and L. Birkedal. Free theorems from univalent reference types, 2023.

[16] T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

Selected references V

[17] A. Vezzosi, A. Mörtberg, and A. Abel. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming*, Boston, Massachusetts, USA, 2019. Association for Computing Machinery. doi: $10.1145/3341691$.

[18] V. Voevodsky. A very short note on homotopy λ-calculus. Unpublished note, Sept. 2006. URL `https://www.math.ias.edu/Voevodsky/files/files-annotated/Dropbox/Unfinished_papers/Dynamic_logic/Stage_9_2012_09_01/2006_09_Hlambda.pdf`.