Contracts, Simulations, and Sessions

Formal Verification of Compilers, Smart Contracts, and Protocols

Kegan McIlwaine

May 1, 2026

Department of Electrical Engineering and Computer Science
College of Engineering and Physical Sciences
University of Wyoming

Committee:
James Caldwell (Chair)
Ruben Gamboa
John Hitchcock
Tyrrell McAllister (External)
Mike Borowczak (University of Central Florida)

This work was partially supported by a grant from IO Global (IOG)
and matching funds from the State of Wyoming.

Motivation

- Smart contracts move real money. Bugs lead to large losses.
- The programming languages and tools need to work as expected.
- Formal methods can be used to give confidence across the whole development pipeline.
    - Target Semantics
    - Compiler
    - Transformations
    - Protocols

Contributions

1. Marlowe: A small-step operational semantics and proof of correctness with respect to the preexisting evaluator.
2. Faustus: A smart contract programming language with parameterized abstractions, expressions for guarded commands based on CCS (the Calculus of Communicating Systems), a type checker, and verified compilation to Marlowe.
3. Transformation verification: Formal proof of observational equivalence between original and transformed contracts via weak bisimulation.
4. STILL: A standalone interactive tactic-based theorem prover for dependent session types with process extraction.
5. Verified protocols: The axiom of choice, natural numbers using polymorphic session types, a bit counter with changing network topology, and an English auction protocol.

Roadmap

Marlowe / Faustus / Transformation Verification Overview

STILL Design

Protocol Verifications

Conclusion

Marlowe / Faustus / Transformation Verification Overview

Marlowe

- A domain-specific language (DSL) for financial contracts[1].
- Runs on the Cardano blockchain.
- Contracts always terminate and release funds within a bounded amount of time (deadlock safety and liquidity).
- The evaluator has reference implementations in Isabelle/HOL and Haskell.

Goal

Define a proof-friendly small-step semantics and prove it agrees with the existing evaluator.

[1] Lamela Seijas and Thompson 2018.

Key Evaluator Functions

- Internal reductions: `reduceContractStep`
- Input matching at a `when` contract: `applyCases`
- Time interval update and trimming: `fixInterval`

Proof Strategy

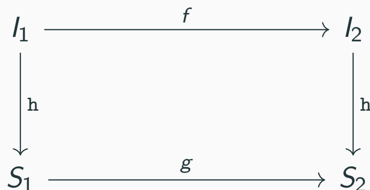
Proved three simulation diagrams:

1. Internal τ steps
2. Input processing
3. Timeout behavior

Formalized in a 1,738 line Isabelle/HOL theory file.

Verification Using Simulation

f, g, h are functions (more generally, relations). I_1 and I_2 are states in the implementation. S_1 and S_2 are states in the specification.



$$\text{Correctness:} \quad \forall I_1, I_2, S_1.\ f\ I_1 = I_2 \wedge h\ I_1 = S_1 \Rightarrow \exists S_2.\ h\ I_2 = S_2 \wedge g\ S_1 = S_2$$

$$\text{Adequacy:} \quad \forall I_1, S_1, S_2.\ h\ I_1 = S_1 \wedge g\ S_1 = S_2 \Rightarrow \exists I_2.\ f\ I_1 = I_2 \wedge h^{-1}\ S_2 = I_2$$

More simply

Correctness: $h \circ f = g \circ h$

Adequacy: $f \circ h^{-1} = h^{-1} \circ g$

Marlowe Timeout Behavior Diagram

The diagram showing the correctness of the timeout behavior is given below. To prove correctness we show that the diagram commutes.

$$\begin{array}{ccc} I_1 & \xrightarrow{\uparrow\texttt{fixInterval}\,\Delta\ \texttt{>>=}\ \uparrow\texttt{reduceContractStep}} & I_2 \\ \downarrow{\texttt{Id}} & & \downarrow{\texttt{Id}} \\ S_1 & \xrightarrow{\xrightarrow{(\tau,\Delta)}_m} & S_2 \end{array}$$

```
lemma marloweSmallStepTimeoutImpliesReduceStep:
  assumes "((When cases t c, state1, env, warnings, payments
      ) →m λ(None, lower, upper) (newCont, newState,
      newTimeEnv1, warnings, payments)) ∧
    fixInterval (lower, upper) state1 = IntervalTrimmed
        newTimeEnv1 newTimeState1"
  shows "reduceContractStep newTimeEnv1 newTimeState1 (When
      cases t c) = Reduced ReduceNoWarning ReduceNoPayment
      newState newCont"
```

Faustus

- Marlowe has no abstractions or looping control operators, i.e., there are no functions; all Marlowe code is inline. Contracts are huge and massively repetitive.
- Faustus extends Marlowe with parameterized abstractions and guarded commands together with CCS[2] inspired operators for: -> Sequencing, <+> Choice, and <*> Interleaving
- Faustus is statically type-checked and ensures all identifiers and their types are declared before use, and all uses are well-typed.
- The verified compilation to Marlowe maintains liquidity and deadlock safety guarantees.
- Formalized in 10,957 lines across six Isabelle/HOL files.

[2] Milner 1980.

Faustus Multi-Signature Example

This Faustus contract compiles to 138,253 lines of Marlowe.

```
when {
  "alice" deposits 1000 "" into @"carol" -> {
    var votes = 0;
    when {
        (("voter1" chooses "agree" -> votes := votes + 1 <+> "voter1" chooses not "agree") <*>
        ("voter2" chooses "agree" -> votes := votes + 1 <+> "voter2" chooses not "agree") <*>
        ("voter3" chooses "agree" -> votes := votes + 1 <+> "voter3" chooses not "agree") <*>
        ("voter4" chooses "agree" -> votes := votes + 1 <+> "voter4" chooses not "agree") <*>
        ("voter5" chooses "agree" -> votes := votes + 1 <+> "voter5" chooses not "agree"))
        -> { if votes >= 3
             then @"carol" pays !"bob" (1000, ""); close
             else @"carol" pays !"alice" (1000, ""); close }
    } after 100 -> { if votes >= 3
                     then @"carol" pays !"bob" (1000, ""); close
                     else @"carol" pays !"alice" (1000, ""); close }
  }
} after 90 -> { close }
```

Using r as the number of votes received timing out, the number of expanded traces is:

$$\Sigma_{r=0}^{5} \frac{5!}{(5-r)!} 2^r = 6331$$

Faustus Compilation

$\mathcal{M}_1, \mathcal{M}_2 \in$ Marlowe states

$\mathcal{I}_1, \mathcal{I}_2 \in$ Intermediate Faustus states

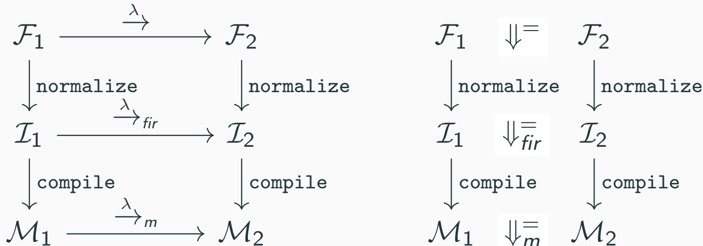
$\mathcal{F}_1, \mathcal{F}_2 \in$ Faustus states

$\xrightarrow{\lambda}_m$, $\xrightarrow{\lambda}_{fir}$, $\xrightarrow{\lambda}$ are input processing steps.

$\Downarrow^=$, $\Downarrow^=_{fir}$, $\Downarrow^=_m$ zero or all internal reductions.

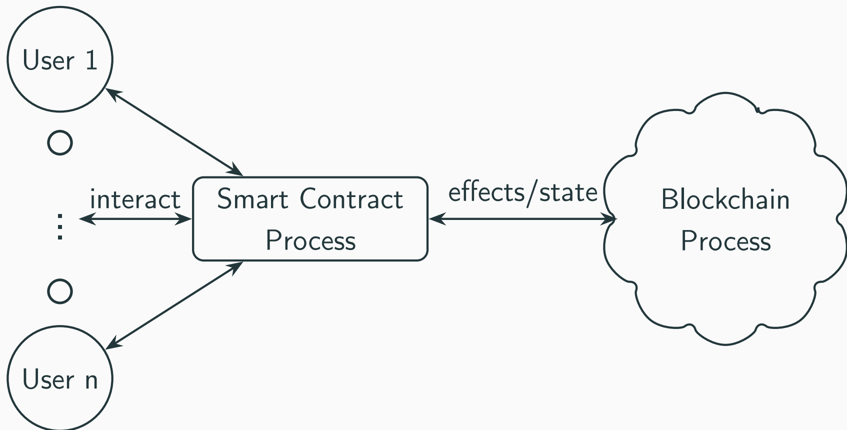
`normalize` – expands guard expressions

`compile` – expands abstractions



High-level idea:

Users and *contracts* modeled as processes that interact with the *blockchain*. We have not modeled the blockchain as π-calculus process, but interact with it through a limited interface.



Faustus Processes and LTS

- Define Faustus programs as concurrent processes with *observable* (λ) and *unobservable* (τ) transitions.
- Use a restriction set to hide observable transitions. We write $F/\mathcal{A}$ to denote the process F with $\mathcal{A}$ actions hidden.
- ***Proc*** is the set of all Faustus configurations.
- ***Label*** is the set of all actions.
- Define the Faustus *labeled transition system* (LTS) as $\langle \textbf{\textit{Proc}}, \textbf{\textit{Label}}, \rightarrow \rangle$.
- Define an *experiment* as a sequence of observable transitions, with zero or more unobservable transitions between each observable transition. $s = \lambda_1 \cdots \lambda_n \in \textbf{\textit{Label}}^*$.
- Faustus processes are *observationally equivalent*, written $\approx$, when there exists a *weak bisimulation* relation $\mathbb{R}$ over ***Proc*** that holds when applying the same experiment s.

Verified Transformation

- Motivation: avoid out-of-gas errors that can occur when evaluating long lists of guarded commands.
- Transform a long list of guarded commands into cascading `whens` using a *fresh* choice name.
- Requires disjoint guards to avoid uncovering previously unreachable code.
- Defined using the $\mathbb{SG}_c$ relation between Faustus processes where c is the fresh choice name.
- Verified[3] $F/\mathcal{A} \approx \texttt{split}(F, c)/\mathcal{A} \cup \{\alpha_c\}$.
- 3,356 lines across two Isabelle/HOL files.

[3] Verifying Smart Contract Transformations Using Bisimulations. McIlwaine and Caldwell. FMBC 2025.

Summary and Next Step

- Simulations described so far verify the stages of development in blue:
  - Target Semantics
  - Compiler
  - Transformations
  - Protocols
- Verifications preserve whatever bugs are present.
- We want correctness at the specification level.

STILL Design

Session Types

- Protocols specified as session types.
- Session types inhabited by π-calculus processes.
- Curry–Howard isomorphism with linear logic[4].
  - Linear logic propositions are session types.
  - Proofs are π-calculus processes.
- A proof of a protocol (session type) verifies that it is inhabited by a process (π-calculus term) of that type.
- The process term is extracted from the proof object.

[4] Caires and Pfenning 2010; Wadler 2012.

Operators in Session Types / Linear Logic

Multiplicatives (tensor and linear implication):

$A \otimes B$ send a channel/value of type A, then continue as B.
$A \multimap B$ receive a channel/value of type A, then continue as B.

Additives (with and plus):

$A \,\&\, B$ offer a choice to continue as A or B.
$A \oplus B$ select a choice to continue as A or B.

Exponential (bang):

$!A$ offer a server using protocol A with no resource constraints. Use A zero or more times.

First-order quantifiers (for all and exists):

$\forall x : t.A$ receive data $x : t$, then continue as $A[x]$.
$\exists x : t.A$ send a witness $N : t$, then continue as $A[N/x]$.

Session Type Sequents

Sequents

Session type sequents have the following form:

$$\Omega; \Psi; \Gamma; \Delta \vdash P :: z : A$$

Process P contains channel z that follows protocol A under the assumptions in the contexts Ω, Ψ, Γ, and Δ.

Session Type Contexts

A *context* is a map from channel names (variables) to session types.

The order of entries in the contexts (except for Ψ) does not matter. The names in Ω, Ψ, Γ, and Δ are pairwise disjoint.

We write C_1, C_2 to denote the context formed by the union of C_1 and C_2 under the constraint that the channel names are disjoint.

- Ω is a second-order session type context with variables that represent protocols.
- Ψ is a functional context with variables mapped to types in the Extended Calculus of Constructions.
- Γ is an unrestricted context with channel protocols that can be used zero or more times. Linear assumptions of the form $!A$ can be placed here.
- Δ is a linear context with channel protocols that must be consumed and that cannot be replicated.

Session Type Proof Rules

Id: The identity rule is an axiom, it has no premises.

$$\frac{}{\Omega; \Psi; \Gamma; a : A \vdash [a \leftrightarrow z] :: z : A}\ \text{Id}$$

Note that the linear context $a : A$ for the Id rule has exactly one entry.

$\otimes$R: The tensor right rule has two premises and partitions the linear context.

$$\frac{\Omega; \Psi; \Gamma; \Delta_1 \vdash P :: x : A \qquad \Omega; \Psi; \Gamma; \Delta_2 \vdash Q :: z : B}{\Omega; \Psi; \Gamma; \Delta_1, \Delta_2 \vdash (\nu x)z[x].(P|Q) :: z : A \otimes B}\ \otimes\text{R}$$

Note that in the linear context (Δ_1, Δ_2), Δ_1 and Δ_2 are disjoint.

STILL

STILL stands for "Session Types and Intuitionistic Linear Logic".

- Standalone interactive tactic-based theorem prover for dependent session types
- Two-layer system:
  - Functional layer: Extended Calculus of Constructions[5] (ECC).
  - Session layer: intuitionistic linear logic with dependency[6], polymorphism[7], and coinduction[8].
- Delays linear context partitioning without modifying the underlying logical system.
- Extracts π-calculus terms from completed proofs.

---

[5] Luo 1990.
[6] Toninho, Caires, and Pfenning 2011.
[7] Caires, Pérez, et al. 2013.
[8] Toninho, Caires, and Pfenning 2014.

Dependency

- Session types in STILL quantify over terms in the functional layer.
- Enables protocols where behavior depends on properties of the data being communicated.
- E.g. send a bid amount and then require a proof that the new bid is valid.

Recall:

$\forall x : t.A$ receive data $x : t$, then continue as $A[x]$.

$\exists x : t.A$ send a witness $N : t$, then continue as $A[N/x]$.

Polymorphism and Coinduction

- **Polymorphic session types**[9]: quantify over session types.

  $\forall X : \texttt{stype}.A$ receive protocol X, then continue as $A[X]$.

  $\exists X : \texttt{stype}.A$ send a protocol P, then continue as $A[P/X]$.

- **(Co)inductive session types**[10]: allow for nonterminating processes.

  $\nu X.A$ begin a corecursive process that behaves as $A[X]$.

  $\mu X.A$ Inductive processes are defined by consuming coinductive channels. μ notation is not part of the session type syntax.

[9] Caires, Pérez, et al. 2013.

[10] Toninho, Caires, and Pfenning 2014.

How To Create A Tactic

- Tactics are created by following rules bottom-up.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \land B} \land\mathrm{R}$$

- Tactics are functions that have the type[11]:

```
tactic = goal -> ([goal], justification)
justification = [theorem] -> theorem
```

- Tacticals allow the composition of tactics:

```
tactical = tactic -> tactic -> tactic
```

- Tactics provide goal-directed theorem proving.

[11] Gordon, Milner, and Wadsworth 1979.

Tactics In Linear Logic

$$\frac{\Omega; \Psi; \Gamma; \Delta_1 \vdash A \qquad \Omega; \Psi; \Gamma; \Delta_2 \vdash B}{\Omega; \Psi; \Gamma; \Delta_1, \Delta_2 \vdash A \otimes B} \otimes\mathrm{R}$$

- Linear logic complicates tactical theorem proving because the partition $\Delta_1, \Delta_2 = \Delta$ cannot be determined at the point of tactic (rule) application.
- Previous approaches:
  - User must explicitly provide the partition $\Delta = \Delta_1, \Delta_2$.
  - Modify the inference rules with constraints or leftovers. This requires revalidating the metatheory of the logical system.
- In the STILL prover we have introduced a novel representation of nondeterministic linear contexts and have redefined tactics.
- Tactics that consume entries in the linear context incrementally eliminate nondeterministic entries in the linear context, eventually making the partition concrete.
- Avoids modifications that require revalidating the metatheory.

Delaying Linear Context Partitioning

$$\frac{\Omega; \Psi; \Gamma; \Delta_1 \vdash A \qquad \Omega; \Psi; \Gamma; \Delta_2 \vdash B}{\Omega; \Psi; \Gamma; \Delta \vdash A \otimes B} \otimes\mathsf{R}\ (\Delta_1, \Delta_2 = \Delta)$$

- STILL represents the partition Δ_1, Δ_2 nondeterministically by using a shared global state:
  - A STILL proof is a tree of rules where each leaf of the tree is an axiom rule **and** nondeterministic partitions are completely determined, *i.e.* at the proof node where a split occurs $\Delta_1 \cap \Delta_2 = \emptyset$.
  - Initially, Δ_1 and Δ_2 appear to contain all resources in Δ. As proofs are constructed for the subgoal using Δ_1 (Δ_2), when a channel, say x, is used, the tactic engine eliminates x from the nondeterministic context Δ_2 (Δ_1). As a proof is constructed and resources are consumed, the paired nondeterministic contexts Δ_1, Δ_2 get "closer" to Δ.

STILL Demo

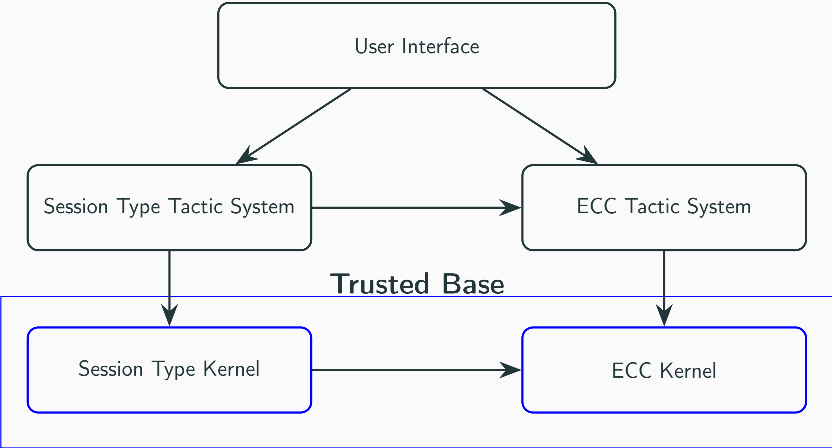
TestingFiles > TensorCommutative.still

```
1   module TensorCommutative begin
2
3   theorem tensorCommutative: "(A * B) -o (B * A)"
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE PORTS STILL

```
State:
New theorem started
*?a>>
  Ω: ;
  Ψ:  ;
  Γ:  ;
  Δ:
  |- z:A ⊗ B ⊸ B ⊗ A
```

STILL Architecture



Process Extraction

- Proof object corresponds to a well-typed π-calculus process.
- Extraction follows the type-checking rules.

$$\frac{}{\Omega; \Psi; \Gamma; a : A \vdash [a \leftrightarrow z] :: z : A}\ \mathsf{Id}$$

$$\frac{\Omega; \Psi; \Gamma; \Delta_1 \vdash P :: x : A \qquad \Omega; \Psi; \Gamma; \Delta_2 \vdash Q :: z : B}{\Omega; \Psi; \Gamma; \Delta_1, \Delta_2 \vdash (\nu x) z[x].(P | Q) :: z : A \otimes B}\ \otimes\mathsf{R}$$

- The π-calculus term (process) is correct-by-construction with respect to the protocol type.

STILL Contributions Summary

- STILL is a standalone interactive tactic-based theorem prover for dependent, polymorphic, and coinductive session types.
- STILL delays context partitioning without changing the underlying proof rules.
- STILL provides a layer for session types that communicates with an ECC layer.
- STILL extracts π-calculus terms from proof objects.
- The implementation consists of 5,926 lines of Haskell across 9 files.

Protocol Verifications

Protocol Verifications

Axiom of choice. Demonstrates power of Σ-types in ECC.

Natural numbers. Encoding inductive types using polymorphism.

Bit counter. Long-running protocol with changing topology.

English auction. A familiar protocol that uses all of the features.

Axiom of Choice

Intuition:

- Given $\Pi x : A.\Sigma y : B.P(x, y)$ in ECC.
- Produce $f : (A \to B)$ such that $\forall x : A.P(x, f(x))$.
- Protocol: send f first, later send proof $P(x, f(x))$.

```
theorem AC: "forall A:Type 1.
    forall B:Type 1.
    forall P:(Pi x:A. Pi y:B. Type 1).
    $(Pi x:A.Sigma y:B.P x y) -o
    exists f:(Pi v:A.B).
    forall x:A.
    $(P x (f x))"
```

Natural Numbers

- System F[12] encoding of inductive types[13]:

$$\mu X.F(X) = \forall X.(F(X) \rightarrow X) \rightarrow X$$

- Polymorphic session encoding of inductive types[14]:

$$\mu X.F(X) = \forall X.!(F(X) \multimap X) \multimap X$$

- Extracts of `zero`, `succ`, `one`, and `two` are given in the STILL source code.

```
stype nat = "forall X : stype. !((1 + X) -o X) -o X"

theorem zero: "nat"

theorem succ: "nat -o nat"

theorem one: "nat"

theorem two: "nat"
```

[12] Girard 1972.
[13] Wraith 1989.
[14] Toninho and Yoshida 2019.

Bit Counter

- Each Node process stores a single bit.
- The ripple carry counter is a network of Node processes that spawns a new Node when overflow occurs.
- The counter supports increment, query, and halt operations.
- Verification demonstrates coinductive session types and dynamic topology.

```
-- CImpl = nu X. &{val: int => int /\ X, inc: + {carry:X,
     done:X}, halt:1}
theorem epsilon: "epsilonT"

-- CImpl -o CImpl
theorem Node: "NodeT"

-- Counter nu X. {val:int /\ X, inc:X, halt:1}
theorem Counter: "CounterT"
```

English Auction

- Seller provides the item.
- Bidders provide bids and a channel that can receive refunds or the item.
- A higher bid refunds the previous bidder.
- Sends item to the winner and funds to the seller at the end.
- This is all guaranteed by the protocol.

```
stype englishSession = "forall ItemType:stype.
    forall bid:Type 1.
    forall zeroB:bid.
    forall validNewB:(Pi b1:bid. Pi b2:bid. Type 1).
    forall currency:(Pi b:bid.Type 1).
    ItemType -o
    (forall b:bid. $currency b -o 1) -o
    forall firstB:bid.
    forall firstValid:validNewB zeroB firstB.
    $currency firstB -o
    (ItemType + $currency firstB -o 1) -o
    nu Y. (forall newB:bid.
        forall nextValid:validNewB firstB newB.
        $currency newB -o
        (ItemType + $currency newB -o 1) -o Y) & 1"
```

Conclusion

Summary

- A new small-step semantics for Marlowe that is correct with respect to the existing evaluator.
- Faustus: a statically typed smart contract programming language with guarded commands and parameterized abstractions.
- Verified compilation from Faustus to Marlowe.
- Verified transformations using weak bisimulation.
- STILL: a standalone tactic-based theorem prover for dependent session types that extracts π-calculus processes from completed proofs.
- Verified protocols for the axiom of choice, natural numbers, a bit counter, and an English auction.

Limitations and Possible Future Work

- Formalize the STILL kernels, resource-hiding in the tactic system, and make extracted processes executable.
- Close the Isabelle and Haskell evaluator gap using extraction from Isabelle or porting the formalization to a system that supports extraction as a first-class feature.
- Verify a transformation that makes guarded command lists disjoint. This likely requires fixing the Faustus and Marlowe deposit processing semantics to disallow negative deposits.
- Extend Faustus with more abstractions.

References

Caires, Luís, Jorge A. Pérez, et al. (2013). **"Behavioral Polymorphism and Parametricity in Session-Based Communication".** In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 330–349. ISBN: 978-3-642-37036-6.

Caires, Luís and Frank Pfenning (2010). **"Session Types as Intuitionistic Linear Propositions".** en. In: *CONCUR 2010 - Concurrency Theory*. Ed. by Paul Gastin and François Laroussinie. Berlin, Heidelberg: Springer, pp. 222–236. ISBN: 978-3-642-15375-4. DOI: 10.1007/978-3-642-15375-4_16.

Girard, Jean-Yves (June 1972). **"Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur".** PhD Thesis. University of Paris.

Gordon, Michael J., Robin Milner, and Christopher P. Wadsworth (1979). ***Edinburgh LCF*.** Vol. 78. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. ISBN: 978-3-540-09724-2. DOI: 10.1007/3-540-09724-4. URL: http://link.springer.com/10.1007/3-540-09724-4.

Lamela Seijas, Pablo and Simon Thompson (2018). **"Marlowe: Financial Contracts on Blockchain".** In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, pp. 356–375. ISBN: 978-3-030-03427-6.

Luo, Zhaohui (1990). **"Extended calculus of constructions".** en. Accepted: 2016-01-19T16:28:16Z publisher: The University of Edinburgh. PhD thesis. The University of Edinburgh. URL: https://era.ed.ac.uk/handle/1842/12487.

Milner, Robin (1980). ***A Calculus of Communicating Systems.*** Vol. 92. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. ISBN: 978-3-540-10235-9. DOI: 10.1007/3-540-10235-3. URL: http://link.springer.com/10.1007/3-540-10235-3.

Toninho, Bernardo, Luís Caires, and Frank Pfenning (July 2011). **"Dependent session types via intuitionistic linear type theory".** en. In: *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*. Odense Denmark: ACM, pp. 161–172. ISBN: 978-1-4503-0776-5. DOI: 10.1145/2003476.2003499. URL: https://dl.acm.org/doi/10.1145/2003476.2003499.

References iii

Toninho, Bernardo, Luís Caires, and Frank Pfenning (2014). **"Corecursion and Non-divergence in Session-Typed Processes".** en. In: *Trustworthy Global Computing*. Ed. by Matteo Maffei and Emilio Tuosto. Berlin, Heidelberg: Springer, pp. 159–175. ISBN: 978-3-662-45917-1. DOI: 10.1007/978-3-662-45917-1_11.

Toninho, Bernardo and Nobuko Yoshida (2019). **"Polymorphic Session Processes as Morphisms".** In: *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy: Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday*. Cham: Springer International Publishing, pp. 101–117. ISBN: 978-3-030-31175-9. DOI: 10.1007/978-3-030-31175-9_7. URL: https://doi.org/10.1007/978-3-030-31175-9_7.

Wadler, Philip (2012). **"Propositions as sessions".** In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. Copenhagen, Denmark: Association for Computing Machinery, pp. 273–286. ISBN: 9781450310543. DOI: 10.1145/2364527.2364568. URL: https://doi.org/10.1145/2364527.2364568.

Wraith, GC (1989). **"A note on categorical data types".** In: *Category Theory and Computer Science, LNCS 389*. Ed. by DH Pitt et al.

Thank You

Questions?

Email: contact@kegan.dev

Artifacts:

Faustus `https://gitlab.com/UWyo-SSC/public/wabl/faustus-v2`

STILL `https://github.com/STILL-prover`

More Information

Full Marlowe Semantics Simulation Diagrams

$$\begin{array}{ccc} M_1 & \xrightarrow{\uparrow\texttt{reduceContractStep}} & M_2 \\ \mathcal{ID} & & \mathcal{ID} \\ S_1 & \xrightarrow{\xrightarrow{\tau}_m} & S_2 \end{array}$$

$$\begin{array}{ccc} M_1 & \xrightarrow{\uparrow\texttt{fixInterval}\,\Delta\ \texttt{>>=}\ \uparrow\texttt{applyCases}\,\lambda} & M_2 \\ \mathcal{ID} & & \mathcal{ID} \\ S_1 & \xrightarrow{\xrightarrow{(\lambda,\Delta)}_m} & S_2 \end{array}$$

$$\begin{array}{ccc} M_1 & \xrightarrow{\uparrow\texttt{fixInterval}\,\Delta\ \texttt{>>=}\ \uparrow\texttt{reduceContractStep}} & M_2 \\ \mathcal{ID} & & \mathcal{ID} \\ S_1 & \xrightarrow{\xrightarrow{(\tau,\Delta)}_m} & S_2 \end{array}$$

Full Faustus Compiler Simulation Diagrams