Verified compilation from BitML to Bitcoin: An Agda Odyssey

Orestis Melkonian

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2024

Abstract

Blockchain technology has taken the financial world by storm in recent years, allowing for programmable contracts to be enacted amongst participants in a decentralised fashion. Bugs in those programs, however, can lead to huge monetary losses and cannot in principle be amended after detection, due to the blockchain being an immutable data structure.

This incentivizes a high-assurance approach to developing smart contracts, which so far has mainly consisted of approximate methods of static analysis. Here, we strive for something more radical, namely the use of interactive proof assistants grounded in Type Theory to develop such contracts and formally verify their correctness by proving logical propositions within the same system.

Specifically, we take existing work on the *Bitcoin Modelling Language* (BitML) — a high-level process calculus for expressing contracts that compile down to Bitcoin transactions — and encode its definitions, semantics, and translation procedure in the Agda proof assistant.

BitML is one of the most mature works at the confluence of Blockchain and Programming Languages, which justifies the tremendous amount of effort required to mechanise the intricate results of the original paper, compared to various more lightweight alternatives such as model checking.

We can then prove properties about BitML contracts as Agda programs, in particular the main meta-theoretical result of the BitML paper, *compilation correctness*, which states that it suffices to prove properties at the more abstract level of BitML contracts, and then provably transfer them to the low-level of Bitcoin transactions.

By virtue of working in a type-theoretic proof assistant whose underlying logic is *constructive*, we can say that the central research goal of this thesis amounts to producing a **verified compiler** from BitML contracts to Bitcoin transactions.

This whole dissertation is a type-checked Agda script, and the corresponding formalisations are publicly available in HTML format:

- `https://omelkonian.github.io/formal-bitcoin/`
- `https://omelkonian.github.io/formal-bitml/`
- `https://omelkonian.github.io/formal-bitml-to-bitcoin/`

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Orestis Melkonian*)

To E.

for all her love, support, and devotion.

Table of Contents

1 Introduction **1**

1.1 Format of the thesis . . . 3

1.2 Agda setup . . . 3

1.3 Thesis overview . . . 4

2 Literature Review **6**

2.1 Blockchain technology . . . 6

2.2 Smart contracts . . . 9

2.3 State machines . . . 12

2.4 Process calculi . . . 13

2.5 Programming language theory . . . 16

3 Target Language: Bitcoin **23**

3.1 Basic entities . . . 25

3.2 Cryptographic operations . . . 26

3.3 Bitcoin scripts *(§3.1 of* [Atz+18b]*)* . . . 28

3.3.1 Denotational semantics: scripts *(§3.4 of* [Atz+18b]*)* . . . 29

3.4 Transactions *(§3.2 of* [Atz+18b]*)* . . . 31

3.4.1 Cryptographic operations on transactions *(§3.3 of* [Atz+18b]*)* . 33

3.4.2 Denotational semantics: transactions *(§3.5 of* [Atz+18b]*)* . . . 37

3.5 Blockchain consistency *(§3.6 of* [Atz+18b]*)* . . . 42

4 Source Language: BitML **48**

4.1 Basic entities . . . 49

4.2 Expressions *(§4,Fig.1 of* [BZ18]*)* . . . 49

4.3 Contract preconditions *(§4,Def.1 of* [BZ18]*)* . . . 50

4.4 Contracts *(§4,Def.2 & Fig.1 of* [BZ18]*)* . . . 51

4.5 Contract advertisements *(§4,Def.3 of* [BZ18]*)* . . . 53

4.6 Example BitML contracts *(§2 & §A.1 of* [BZ18]*)* . . . 55

4.7 Operational semantics *(§A.2,Def.6 of* [BZ18]*)* 62
4.7.1 Actions *(§A.2,Fig.2 of* [BZ18]*)* 62
4.7.2 Configurations *(§4,Def.4 & §A.2,Def.5 of* [BZ18]*)* 63
4.7.3 Labels *(§A.2,Fig.3 of* [BZ18]*)* 67
4.7.4 Semantics of predicates *(§A.2,Fig.5 of* [BZ18]*)* 68
4.7.5 Inference rules *(§A.2,Fig3-5 of* [BZ18]*)* 68
4.7.6 Decidability of inference rules 78
4.8 Example: the *timed commitment* protocol 79

5 From Source To Target: The BitML Compiler **83**
5.1 Formulating the type of the compiler 84
5.1.1 A primer on mappings with a finite domain 85
5.1.2 Compilation parameters 87
5.1.3 The type of compilation results 88
5.2 Defining the compilation procedure 90
5.2.1 A primer on well-founded recursion 90
5.2.2 A well-founded recursion for BitML contracts 92
5.2.3 The translation as a recursive function 93
5.3 Compilation examples 101
5.3.1 Compiling primitives *(§7 of* [BZ18]*)* 102
5.3.2 Timed-commitment protocol *(§A.5,Fig.8 of* [BZ18]*)* 107

6 Relating Source To Target: Coherence **110**
6.1 Computational model *(§6 & A.4 of* [BZ18]*)* 111
6.1.1 Serialization 111
6.1.2 Encoding contract advertisements *(§A.7 of* [BZ18]*)* 114
6.1.3 Computational runs *(§6 & A.4 of* [BZ18]*)* 116
6.2 Symbolic model *(§5 of* [BZ18]*)* 118
6.2.1 Symbolic runs 118
6.2.2 Properties of symbolic mappings 122
6.2.3 BitML's trace properties 125
6.2.4 Invoking the BitML compiler 138
6.3 The coherence relation *(§8 & A.6 of* [BZ18]*)* 142
6.3.1 The relevant inductive cases *(§A.6,Def.20 of* [BZ18]*)* 143
6.3.2 The Stratified relation *(§8 & §A.6,Def.20 of* [BZ18]*)* 166
6.3.3 An example property of coherence 170
6.3.4 Example coherence proof: timed commitment protocol 172

7 Towards Correct Compilation **184**
7.1 Symbolic strategies *(§5 & A.3 of* [BZ18]*)* 186
7.2 Computational strategies *(§6 & A.4 of* [BZ18]*)* 190
7.3 Translating strategies *(§A.7, Def.22 of* [BZ18]*)* 193
7.3.1 Parsing computational runs 195
7.3.2 Interpreting symbolic moves 202
7.4 Computational soundness *(§9 & A.8 of* [BZ18]*)* 205

8 Conclusion **208**
8.1 Reflection 208
8.2 Future Work 209
8.3 Epigraph 211

A `formal-prelude`: Infrastructure **215**
A.1 Syntax for unit tests 216
A.2 Syntax for inference rules 217
A.3 Deriving strategies 219
A.4 Decidability 220
A.5 Accessors 221
A.6 Collections 223
A.7 Coercions 225
A.8 No-nil list notation 226
A.9 Relation closures 227
A.10 Views 231
A.11 Measure-based termination 234
A.12 Substitution syntax 235
A.13 Meta-properties 237

Lay Summary

Bugs in blockchain software can lead to tremendous financial losses, hence there has been a growing demand for the application of *formal methods* to assure us that "nothing goes wrong".

In the case of Bitcoin — the first and still most prominent blockchain in existence — clever people have thought about this long and hard and came up with an abstract language, called **BitML**, for expressing programs running on the blockchain in a concise way, as well as allowing formal reasoning about their behaviour and translating them down to plain old Bitcoin transactions that eventually get submitted to the blockchain.

Alas, these results only exist on paper omitting too many details for us to be absolutely certain that "nothing goes wrong". This thesis tries to remedy that by encoding these existing results *formally* and *mechanically* in the Agda proof assistant, where all details are fleshed out and there is absolutely no room for error.

Apart from encoding the syntax and semantics of BitML programs in the logical language of Type Theory — upon which Agda's foundations rest — we also specify the compilation process and prove the central property of *computational soundness*: any adversarial attack possible on Bitcoin also manifests itself on the BitML source, thus it is *safe* to conduct all reasoning at this higher level of abstraction.

While these contributions had theoretically been described in the original BitML work, the resulting Agda artefact brings us a much higher level of confidence but also provides an *executable model* that effectively constitutes a **verified compiler** for BitML.

Chapter 1

Introduction

Blockchain technology has opened a whole array of interesting new applications, such as secure multi-party computation [And+14a], fair protocol design [BK14], and zero-knowledge proof systems [GMW91]), to name a few. Reasoning about the behaviour of such systems, however, is an exceptionally hard task, mainly due to their distributed nature. Moreover, the fiscal nature of the majority of these applications mandates a much higher degree of rigour compared to conventional IT applications, hence the need for a more formal account of their behaviour.

Furthermore, the advent of smart contracts (programs that run on the blockchain itself) gave rise to another source of vulnerabilities. One primary example of such a vulnerability caused by the use of smart contracts is the DAO attack,[1] where a security flaw in an Ethereum smart contract enabled the exploitation of a venture capital fund worth 150 million dollars at the time. The solution was to create a hard fork of the Ethereum blockchain, *i.e.,* reverse all transactions from the time of the attack, clearly going against the decentralised spirit of cryptocurrencies. Since these programs often deal with transactions of significant funds, it is of utmost importance that one can reason and ideally provide formal proofs about their behaviour in a concurrent/distributed setting.

My thesis lies at the intersection of Blockchain Theory and Programming Languages (PL), namely to lay the foundations for a more robust development process for smart contracts, where blockchain developers specify the functionality they wish to implement in a declarative style closer to mathematics and then prove logical propositions about the behavioural properties they deem interesting.

More specifically, I believe interactive proof assistants to be the ideal environment for these goals, especially when they are based on firm foundations such as Type Theory. Since we are concerned with highly complex systems that require rigorous investigation,

[1] https://en.wikipedia.org/wiki/The_DAO_(organization)

conducting our formal study in a mechanised fashion allows us to discover edge cases and increase the confidence of the model under investigation.

Goal. Which brings me to the actual object of study of this dissertation, the mechanisation of the *Bitcoin Modelling Language* (BitML), an idealised process calculus for Bitcoin smart contracts that is already accompanied by a rigorous meta-theoretical study:

> Massimo Bartoletti and Roberto Zunino. "BitML: a calculus for Bitcoin smart contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 83–100

I chose BitML as I found it to be one of the most promising existing results within the narrow intersection between Blockchain and PL, that also gives us plenty of existing material to work with: compilation to Bitcoin has already been specified in extreme detail and a intricate proof of correctness has already been fleshed out by the authors of the original BitML paper. From a formalist point of view, the goal of my thesis can thus be phrased as:

> *Mechanise the meta-theoretical results of the BitML paper in a (constructive) proof assistant.*

That is, encode the definitions and proofs of the paper type-theoretically, resulting in an *executable* specification that is necessarily formulated in much more detail, but also comes with a much higher degree of confidence.

Alternatively, I would like to think of this task in a similar vein to what CompCert [Ler06; Ler09] has achieved for verified compilation of the C programming language – one of the success stories in formal verification of this kind – but now applied to a completely fresh application domain; a CompCert for blockchain, if you will. Therefore, I take the ultimate goal of my thesis to be:

> *Develop a verified compiler from BitML contracts to Bitcoin transactions.*

Scope. Although my personal research these past four years touches upon the broader topic of formalised semantics of UTxO-based blockchains [Cha+20a; Cha+20c; Cha+20b; MSC23], I chose to focus on just the BitML mechanisation to keep a more *coherent* (pun intended, as you will soon realise) theme, but also because it is the only part of my research that I carried out all by my lonesome, in contrast to the meta-theoretical results of the EUTxO model that resulted from collaboration within the Plutus team within IOG and the separation logic for UTxO ledgers that we cooked up with Wouter Swierstra from Utrecht University and James Chapman from IOG. Anyway, it turns out the narrow

scope of BitML has incredible technical depth to keep us occupied for a whole dissertation on its own, so I do not hold any regrets about this decision.

1.1 Format of the thesis

This thesis is written as a *literate* Agda script, meaning that all rendered code blocks have been type checked with the Agda proof assistant [Nor08].

However, given the sheer size of the formalisations involved (just the Prelude — containing important infrastructure such as data structures we need — amounts to ∼300 pages of PDF), it is impossible to include all the gory details here and make the formalisation aspect of the thesis completely self-contained.

Moreover, the parts that we *do* include are themselves not complete, in the sense that we had to selectively show the code of each Agda file involved (*i.e.,* there are many hidden code blocks in between the visible ones). Consequently, one needs to inspect the actual non-literate development of each part to get a full understanding of the formalisation; we will make sure references to these are always available through hyper-links.

Disclaimer

Sadly, for lack of an existing technology in the Agda ecosystem to connect LaTeX and HTML artefacts of a formalisation through such URL hyper-links, I had to hack together an *ad hoc* amalgam of SQL databases and Bash scripts to do that. It is highly unstable though, so be aware that some identifiers will not have a URL link attached to them or, even worse, there will be a link pointing to a wrong definition of the same name.

1.2 Agda setup

Our formal development depends on the following software versions:

- **Agda:** version 2.6.3
- **Standard library:** version 1.7.2
- **Prelude:** commit #c10fe94

We developed the Prelude alongside the BitML formalisation (amongst other parallel projects, all sharing the same Prelude), so as to provide necessary infrastructure that was missing from the standard library. It depends on the standard library and re-exports the parts that we are interested in, albeit following a different module structure and adopting

different naming conventions (*e.g.,* importing the Prelude implicitly renames the sort of types from `Set` to `Type`). The whole development is available online here:

https://omelkonian.github.io/formal-prelude/

We will not discuss the Prelude in its entirety, as we believe such technical details not to be worthy of an entire thesis chapter. Instead, in the few places where the content of the Prelude is itself interesting or important, we will make a passing note with a brief exposition and refer the reader to more elaborate descriptions in Appendix A.

Last but not least, it should be noted that the gap filled by the Prelude cannot be trivially merged upstream into the standard library, due to concerns about type-checking speed and fundamentally different designs. Having said that, I have managed to factor out certain parts of it into reusable libraries that other users might now benefit from:

- https://github.com/omelkonian/agda-stdlib-classes:
  a typeclass-friendly wrapper to Agda's standard library;
- https://github.com/omelkonian/agda-stdlib-meta:
  utilities for meta-programming via *elaborator reflection*, such as deriving typeclass instances akin to Haskell's `deriving`;
- https://github.com/omelkonian/agda-lenses:
  programming with *lenses*, an essential toolkit when working with large record types.

1.3 Thesis overview

We start with a literature survey on related work in Chapter 2 to gain some background (and a bit of fresh air) before we embark on the rest of the deeply technical chapters that revolve around the mechanisation results.

First, we develop a mechanised model of Bitcoin transactions in Chapter 3, following prior work from the BitML authors [Atz+18b]. This will provide the source language of our verified compilation, by formulating the exact definitions of Bitcoin ledgers, transactions, and scripts, as well as an executable definition of their *denotational semantics* and the notion of *consistency* of a blockchain.

We then do the same for the BitML calculus in Chapter 4, *i.e.,* precisely define its syntax and operational semantics, as well as formally verified examples of BitML contracts and their execution under the transitions prescribed by their operational semantics. This will provide the target language of our verified compilation.

Chapter 5 concerns the definition of the BitML compiler: a BitML contract goes in, a set of Bitcoin transactions comes out. The intuition is that steps of execution in the BitML contract will correspond to submitting a corresponding transaction to the Bitcoin blockchain, amongst other messages that need to be communicated within the Bitcoin network for participants to coordinate (*a.k.a.* off-chain communication). This will provide the translation that our mechanisation is tasked with verifying.

Proving that the compilation is correct relies on a *refinement* argument [DE98] between BitML and Bitcoin transitions; one could interpret this as a form of *weak simulation* [Mil80], since the low-level of Bitcoin can always perform *internal* steps that have no corresponding match on BitML. Chapter 6 formalises a *logical relation* which establishes this correspondence between the *high* level of BitML and the *low* level of Bitcoin. This will be the technical "meat" of the thesis, as it drives how all subsequent definitions and proofs take shape.

Finally, we attempt to finalise the proof of compilation correctness in Chapter 7, stating that the refinement relation of the previous chapter can indeed be established for the transactions that the BitML compiler generates. Unfortunately, the mechanisation of the proof and its underlying definitions does not reach completion; there are still a few lemmas to prove, particularly concerning the game-theoretic aspects of the proof.

The anticlimactic nature of the final result might leave readers with a sour taste, which Chapter 8 tries to alleviate by ending on a more positive note, reflecting on the process and discussing some exciting future directions for research.

Chapter 2

Literature Review

This chapter conducts an extensive review of literature relevant to our research topic: formally verified compilation. Previous work is drawn from a wide range of scientific fields, from *Blockchain* to *Programming Languages*. The purpose of such an attempt was twofold; first, to thoroughly understand what has already been achieved and, second, to draw inspiration for our own work. While most of the sources cited here do not directly pertain to the study of the BitML calculus, we still deem it worthy to include here so as to situate the reader in the broader scope of things before embarking on a mechanisation voyage with a more specific goal in mind.

2.1 Blockchain technology

Although a very recent concept, blockchains have gained major popularity and revealed new research direction in a wide range of fields. While initially introduced only informally, there has been copious amounts of research to formally understand its mechanisms and reason about its behaviour since its conception in 2009 [Bon+15], a story reminiscent of *BitTorrent* in the context of *peer-to-peer file sharing* [Coh03].

The Dolev-Yao Model. Before we investigate blockchain-specific work, it is worthwhile to notice a prominent technique used for proving that certain security protocols are secure, namely the *Dolev-Yao* model [DY83].

One of its main characteristics is that primitive cryptographic operations are postulated to exist with ideal properties, hence allowing for emphasis on the interesting properties currently under scrutiny. Enforcing such separation of concerns — by abstracting away from these details — is a technique that appears throughout our formal development. Keeping a certain amount of abstraction is paramount to efficient prototyping, so as to get sufficient results in a reasonable amount of time.

Another highly influential aspect of the Dolev-Yao model is the consideration of active malicious participants, who are able to impersonate other users, alter transmitted messages and perform other similar actions.

Bitcoin. In 2009, a person/group by the alias of *Satoshi Nakamoto* proposed a decentralised system, *Bitcoin*, able to provide a completely decentralised monetary system [Nak08]. The system allows users to exchange currency, without the need for any trusted central entity, by providing a novel consensus algorithm that decides the order of transactions and prevents *double-spending*, *i.e.,* makes sure that there is a certain amount of currency at each point in time and it cannot be spent twice.

Instead of keeping track of explicit accounts, Bitcoin's accounting model is based on *unspent transaction outputs* (UTxO). These are locked transaction outputs carrying some monetary value, which can be spent by anyone who is able to unlock it. The simplest form of locking an output would be using a public key, requiring that the spending transaction is signed with the corresponding private key. In reality, the lock is a *validation script* which expects some arguments; the output can be spent by anyone that provides the arguments that leads script execution to succeed. Therefore, one can design and implement more complicated transactional schemes than the simple, key-based one. Bitcoin offers a low-level, Forth-like, stack-based language (Script), which has rather limited expressiveness.

For a more thorough description of all integral components of the Bitcoin system, we refer the reader to Princeton's Bitcoin book [Nar+15].

Ethereum. The next most popular blockchain currently is *Ethereum*, which proposes a wholly different approach to conceptualising the blockchain and advocates for adopting blockchain technology for general distributed applications (*dApps*), rather than restricting usage to tasks of a fiscal nature [But14].

The main difference lies in the underlying accounting model which, in contrast to Bitcoin's UTxO-based approach, consists of a global mutable state of accounts along with the total amount of currency they possess. Transactions that are then submitted to the blockchain change this global state accordingly. This is coupled with a feature-heavy, Javascript-like, Turing-complete scripting language, called *Solidity*, which compiles down to bytecode targeting the *Ethereum Virtual Machine* (EVM). EVM programs are then executed as imperative programs that mutate the blockchain's global state.

Consensus & Relation to Concurrency. There is great similarity between the notion of consensus inherent to how a blockchain operates and classical consensus problems

in *Concurrency Theory*, where the question is how to reach agreement on a value in a distributed setting.

First, the consensus algorithm used by both Bitcoin and Ethereum is based on *Proof-of-Work*, requiring blockchain participants to solve hard cryptographic puzzles in order to submit blocks/transactions, thus defending against *denial-of-service* (DoS) attacks. There exist alternative consensus mechanisms, used by other blockchain systems, such as *Proof-of-Stake*, but we refrain from investigating this blockchain component further, as we will work in a higher level of abstraction throughout this thesis. This is due to the fact that we are concerned with a component orthogonal to the consensus layer, namely the underlying accounting model.

One notable difference between classical consensus on distributed systems and blockchain-specific consensus is the fiscal nature of blockchain technology, leading to game-theoretic investigations of financial incentives, in addition to reasoning about computational behaviour/resources. Nonetheless, blockchains shed a new light on the consensus problem and have spurred enough interest to reshape the landscape of consensus research [GK18].

Apart from the common background in consensus protocols, the problems one faces when reasoning about the security and behaviour of blockchains bear a striking resemblance with those encountered when reasoning about computer programs that run concurrently or even across different machines, *i.e.,* the main research focus of *Concurrency Theory* and *Distributed Computing*. This particularly pertains to the way programs, running simultaneously on a blockchain, interact with one another [Her19].

Formal Blockchain Models. While research is still in its infant steps, there have been considerable efforts to provide formal models of its operations, especially from the cryptographic community [GKL15; Bad+17; HP17; Cac+17]. Alas, the landscape of mechanically verified formal models for blockchain systems is still scarce.

An exception is Anton Setzer's formal model of Bitcoin in Agda [Set18], which provides a mechanically-verified, executable specification of how Bitcoin operates. In order to make such a mechanisation effort tractable, cryptographic primitives and other irrelevant components are dealt with abstractly, by postulating such functionality and the corresponding desired properties; a technique we also employ throughout our formal development. While this work is a wonderful exercise in dependently-typed programming and uses advanced modelling techniques provided by Agda, such as *induction-recursion*, no further meta-theory is formalised and the feasibility of such a static model for proving any kind of non-trivial property of the blockchain remains questionable.

Another noteworthy attempt is [Mar19], where a tool for specifying the behaviour of *dynamic architectures*, FACTUM, is used to model blockchains and automatically generate code for the Isabelle/HOL proof assistant [NPW02]. Then, it is possible to interactively prove desired properties of the system, such as persistence of blockchain entries, using temporal logic to reason about traces.

Lastly, [Hir18] is an interesting work in this direction, which formally models blockchains as state-transition systems and reasons about possible properties using logical formulas in temporal-epistemic terms, drawing inspiration from the fields of *modal logic* and its interpretation in terms of *Kripke structures*. More specifically, the modality of these logical formulas allow for expressing and analysing the *atomicity* of a cross-chain swap.

2.2 Smart contracts

Smart contracts are computer programs that reside on the blockchain itself and, consequently, are executed on-chain. Their name is derived from their fiscal nature (*i.e.,* they resemble legal/financial contracts) and the fact that they are self-enforced without requiring a trusted intermediary. As was briefly mentioned in Subsection 2.1, Bitcoin only provides simple script templates, while Ethereum users have access to a full-blown imperative language.

Attacks & Analysis Tools. Since smart contracts handle monetary transactions, they provide a tempting battleground for malicious attackers to exploit undiscovered vulnerabilities, sometimes leading to a tremendous loss of capital (*e.g.,* the infamous DAO attack[1]). This is exactly the reason why a lot of research effort has been put on validating contracts, either by analysing their source code or developing appropriate formal methods for reasoning about the behaviour and security properties.

[SH17] examine different attacks on Ethereum smart contracts and recast them as classical problems found in the settings of concurrent program execution, in order to advocate for a *contracts-as-concurrent-objects* analogy. While it seems surprising that deterministic programs written in Solidity exhibit similar behaviour to concurrent programs with shared memory, this becomes clear when one considers the races in terms of transaction submission (*i.e.,* one cannot decide on the exact order transactions will get incorporated).

[1] https://en.wikipedia.org/wiki/The_DAO_(organization)

A taxonomy of security vulnerabilities of Ethereum smart contracts is systematised in [ABC17], encompassing issues introduced at the levels of Solidity, EVM and the blockchain mechanism itself. These vulnerabilities concern Solidity's function-calling, error-handling and resource-monitoring mechanisms, as well the inherent immutability of EVM bytecode and the limitations of blockchain's inner workings. Finally, a collection of smart contracts are presented, which exhibit issues that fall into one of the investigated categories.

Another categorisation of attacks possible on Ethereum is provided in [Luu+16], although the kind of attacks are more focused on timestamp dependencies, improper handling of exception and a particular form of function callbacks, called *re-entrancy*, which led to the aforementioned DAO attack. Having identified how these attacks are possible through examples, an operational semantics for Ethereum is introduced and suggestions for improvement and tighter security are recommended. Alas, these extensions require all network clients to upgrade; something not realistically feasible. To remedy this limitation, the authors introduce an analysis tool based on symbolic execution, OYENTE, that can automatically detect flaws in individual contracts, helping *developers* to program safer contracts and *users* to avoid interacting with problematic ones. The tool is accompanied by experimental evaluation, which demonstrates that OYENTE is a pragmatic tool for analysing smart contracts in the wild.

As expected from the fact that certain vulnerabilities have been identified and led to significant financial losses, there is currently a plethora of analysis tools, which check smart contracts against well-known vulnerabilities and "code smells". These can be divided between tools that statically analyse the program under question, *e.g.,* using the Datalog logic programming language [Gre+18; Tsa18], and the ones that dynamically monitor contracts for violations and possibly provide recovery mechanisms [CEP18].

Formal Methods & Verification. While analysis tools are a pragmatic approach to detecting known vulnerabilities, they prove inadequate for security issues that are yet to appear. Most importantly, once smart contracts are deployed on the blockchain, they are immutable. Therefore, developing appropriate formal methods to reason about contract behaviour and safeguard against future attacks is of paramount importance, as evidenced by the amount of surveys available in this domain [MCJ18; HK18; BZ19a]. One approach that proves quite attractive for these purposes is *language-based verification*, where issues are avoided by carefully designing the semantics of the language itself [SSW10].

A notable example of a language specifically designed to accommodate formal verification is SCILLA, an intermediate language for Ethereum-like smart

contracts [SKH18]. SCILLA's semantic model is based on *communicating automata*, which provide a clean separation between the language's purely functional fragment and its asynchronous messaging mechanism. Its development comprises of a *shallow embedding* in Coq [Bar+97], where one can formally verify properties of contracts. The properties under question are of a temporal nature and are subsequently divided in two categories: *safety* properties that hold throughout a contract's execution and *consistency* predicates that hold under certain assumptions, which mandate examining a contract's *execution traces* and interactions with other contracts. SCILLA is used in the Zilliqa blockchain, is accompanied by well-documented operational semantics and integrated static analyses to aid contract developers and has been used to verify crucial properties of several example contracts [Ser+19].

Another framework for certifying smart contracts in Coq is *ConCert* [ANS20; AS19; NS19]. Although it specifically targets the Oak programming language for smart contracts, the techniques used are applicable to any *functional* smart-contract language. The problem addressed here is the semantic gap between a *deep* and a *shallow* embedding of a language, which are independently useful in different context; one would use a deep embedding to reason about a language's *meta-theory*, while the shallow embedding is convenient for validating concrete, individual contracts. This is solved by a clever use of recently introduced meta-programming facilities of the Coq ecosystem, which allows for a more principled way to connect the semantics of the two alternative embeddings.

A Coq framework is also being developed in the context of Michelson, a low-level, stack-based scripting language for the Tezos blockchain [Goo14]. Again, the language is designed with formal verification in mind. *Mi-Cho-Coq* is the Coq embedding of Michelson, which provides facilities for formally verifying the functional correctness of Michelson programs [Ber+19]. To make this possible, the original Michelson interpreter written in OCaml was ported/embedded in Coq and then contract properties are proven using Djikstra's famous *weakest precondition calculus* [Dij75]. In contrast to ConCert though, there is no semantic connection between the Coq embedding and the actual Michelson interpreter. In contrast to SCILLA, the contract properties one can verify are still quite limited, e.g., do not include an adversarial model.

A different point in the design space of formal frameworks for smart contracts is investigated in [CPR18]. It is based on the $\mathbb{K}$ framework, where semantics are defined in a *language-independent* fashion and several language components (e.g. parser, interpreter) are automatically derived in a *correct-by-construction* manner. To that end, a formal semantics of EVM are defined in the $\mathbb{K}$ framework, dubbed *KEVM*, but no reasoning mechanisms/logics have been formulated yet.

Lastly, an alternative approach that circumvents the use of a proof assistant are *SMT solvers*, such as Microsoft's Z3 [DB08]. These allow for automatically deciding first-order formulas (in a suitable decidable theory), rather than providing a complete proof/derivation interactively. The VeriSol verifier is one such example, used in Microsoft's Azure blockchain [Wan+18]. There, Azure contracts are annotated with function pre-/post-conditions, which get checked automatically by an external SMT solver. A rather similar approach is employed in the context of Ethereum, where Solidity programs are statically analysed to derive an SMT-encoding of the same program, whose `assert` statements can then be automatically verified by an SMT solver [AR18]. However, there is an inherent limitation to the kind of properties that can be automatically verified by an SMT solver, a process commonly referred to as *push-button verification*. For instance, loop invariants in imperative programs are typically inserted manually by humans to aid the SMT solver, although there exist techniques for automatically inferring these in some cases, such as *monomial predicate abstraction* [LQ09].

2.3 State machines

As has become quite obvious by now, opinions are converging towards an automata-based interpretation of smart contracts.

Smart Contracts. First and foremost, the operational semantics of the aforementioned smart-contract language Scilla is based on a particular form of communicating automata, called *Communicating State Transition Systems* [SKH18]. This was initially employed for specifying arbitrary resource protocols to reason about concurrent resources, combining the compositionality of *Concurrent Separation Logic* (CSL) and the fine-grained resource control of *Rely-Guarantee* (RG) [Nan+14].

Another automata-based approach to specifying Ethereum smart contracts is employed in the *FSolidM* tool, where the user can design automata in a graphical interface, which can then be automatically translated to Solidity contracts [ML18]. The authors have also identified common security vulnerabilities and specified corresponding *design patterns*, allowing a user to effortlessly integrate it in their state machine in the form of an *extension plugin*. The easy-to-use graphical interface allows for easier adoption by the community, while the plugin mechanism aids in the development of more secure contracts. For instance, a 'locking' plugin is provided to "foolproof" contracts against re-entrancy attacks, by excluding mutually recursive calls for all functions inside the contract.

In the context of Bitcoin, [And+14b] utilises the notion of *Timed Automata* to model Bitcoin smart contracts, which are specified in the Uppaal model checker [LPY97]. This provides a pragmatic way to verify temporal properties of concrete smart contracts; Uppaal can verify properties written as *timed computation tree logic* (TCTL) formulas. Alas, the authors provide no formal claim that this class of automata actually corresponds to Bitcoin smart contracts, hence there is still a significant semantic gap that hinders further formalisation.

Mealy Machines. One particular form of state machines that might prove useful in the context of understanding smart contract behaviour, are *Mealy machines*, an extension of standard state machines to allow transitions to additionally produce output [Mea55].

This formulation proves attractive, as it is well-established, has been thoroughly studied in the past, and has even been proved isomorphic to a coalgebraic logic, *i.e.,* every finite Mealy machine corresponds to a finite formula of this logic [BRS08].

Although initially conceived to specify sequential digital circuits, the added ability of emitting outputs seems appropriate in the context of a blockchain ledger, where transaction submissions (*i.e.,* transitions) might have side-effects on the blockchain (*i.e.,* outputs). Note that this is a major inspiration for our own work on UTxO meta-theory where we establish a bisimulation between (a slightly more expressive) UTxO model with scripts and a variation of Mealy machines [Cha+20a; Cha+20b].

2.4 Process calculi

It is generally agreed that the λ-calculus provides a *canonical* model for purely functional computation, Unfortunately, there is no established equivalent for concurrent computation, but there has been a rich body of research towards establishing such a core calculus for concurrency [Pie97]. These formal models for concurrency are typically called *process calculi*, while the scientific area that emerged through their study is sometimes referred to as *process algebra* [Bae05]. Here, we give a brief overview of some important attempts in this direction, since BitML, the central topic of interest in this thesis, can itself be thought of as a process calculus and is obviously influenced by previous results in the area.

CCS. One of the most influential theoretical models towards a core calculus for concurrency is the *Calculus of Communicating Systems* (CCS), which introduces constructs for variable renaming/restriction, parallel composition and non-deterministic choice, among others [Mil80].

Its main innovation is the abandoning the idea of programs as pure functions from input to output, which is achieved by introducing an alternative *semantic domain* for processes, based on *labelled transition systems*. In fact, this led to the related development of *structural operational semantics* [Plo81], a highly influential technique that is employed by the state-of-the-art in the semantics of programming languages up to this day.

π-calculus. Following a series of publication by Milner [Mil79b; MM79; Mil79a], whose purpose was to refine the theory and capabilities of CCS, a further extension of CCS was later introduced to add support for *mobility*, *i.e.,* the ability for processes to give explicit names to communication channels and communicate the names themselves across channels, called the π-calculus [Mil89]. There, a formal model of π-calculus is introduced, along with the notion of *bisimulation* that identifies equivalent processes when they behave the same in any context (more on this in Subsection 2.5).

CSP. Around the same time as CCS, another process calculus for concurrency appears by the name of *Communicating Sequential Processes* (CSP) [Hoa78]. CSP is based on Djikstra's *guarded commands* [Dij75], which are commands prefixed by a boolean predicate; sequential composition can proceed only when the predicate expression evaluates to `true`. Furthermore, CSP employs synchronised communication, which in addition to guarded commands, proves to be sufficient to express commonly used programming constructs, such as co-routines, procedures and functions, as well as common concurrent primitives, such as monitors and conditional critical regions.

In contrast to previous approaches that use global variables to express communication, CSP introduces a paradigm shift to *message passing*. Surprisingly, the paradigm of message passing introduced in CSP was not apparent in the initial form of CCS, but inspired Milner to incorporate it in its revised versions and the π-calculus [Bae05].

Although not initially accompanied by a proper semantics and lacking rigorous techniques for proving correctness, further investigations led to what came to be known as *Theoretical CSP* [Hoa80; HBR81], which is based on *trace theory* and *failure pairs* preserving deadlock behaviour.

ACP. A more axiomatic approach is taken in the *Algebra of Communicating Processes* (ACP), where models for concurrent processes are specified as axiomatic systems, consisting of equational rules [BK87].

There, a more algebraic treatment of processes is investigated, given by a series of axiomatic systems with gradually increasing complexity. First, a simple axiom system, called *Basic Process Algebra*, provides the core of all subsequent refinements and contains

no communication constructs whatsoever (although it contains parallel composition, which freely interleaves processes). Then, communication is added to arrive at the definition of the axiomatic system, called ACP, which is then extended with *abstraction* to yield ACP_τ, allowing for more scalability and modular proof techniques.

In contrast to CCS that provides communication tangled with abstraction, and CSP that merges messaging with restriction, ACP employs a more general communication scheme by introducing all these features independently [Bae05].

BitML. A process calculus specifically designed for blockchain smart contracts is the *Bitcoin Modelling Language* (BitML) [BZ18], whose mechanisation is the central research topic of this thesis.

BitML contains constructs for fundamental smart contract operations, such as withdrawing funds, specifying deadlines and revealing secrets. By combining these core constructs, one gets a highly flexible, yet minimal, calculus, proven to be adequate for expressing a diverse set of contract examples [BCZ18; Atz+18a].

BitML's operational semantics is defined as a *labelled transition system* (LTS) between configurations, which indicate the funds of each participant and action authorisations among others. Then, a *symbolic model* is defined over the execution traces allowed by the operational semantics, upon which a game-theoretic notion of honest and adversarial strategies is defined. Briefly, an honest strategy outputs a set of possible next moves, given the current execution trace, while the adversary has the final call of which of all possible moves by all honest participant is actually realised.

The authors also provide a compilation scheme from BitML to 'standard' Bitcoin contracts, accompanied by a compilation correctness proof, stating that any attack possible on the Bitcoin level is also reflected in BitML's symbolic level. The aforementioned translation targets a formal model of Bitcoin transactions that precisely defines transactions and their consistency with respect to a given ledger [Atz+18b]. A similar game-theoretic model is defined on the low level of Bitcoin transactions, dubbed *computational model*. Compilation correctness is then formulated as a correspondence between symbolic and computational runs.

Although BitML does not contain constructors for *explicit message passing*, it can nevertheless be accurately described as a process calculus, due to the non-determinism inherent in the rules of its operational semantics. In fact, when one starts reasoning about actions different participants may employ, it quickly becomes clear that such strategies are defined via inter-participant communication. In other words, it turns out

that BitML shares messaging capabilities with more typical process calculi, although they arise implicitly through BitML's game-theoretic symbolic model.

The same authors investigated the property of *liquidity* in BitML smart contracts, *i.e.,* that the funds stored within a smart contract do not remain frozen indefinitely [BZ19b]. The result is several alternative definitions of liquidity, all proven to be *decidable*, thus allowing *model-checking* to automatically decide these properties for an arbitrary contract. Remarkably, one can specify arbitrary temporal properties in *linear temporal logic*, in addition to liquidity, which can then be automatically checked by the model-checker. This constitutes an addition to the list of static analysis tools presented in Subsection 2.2, arguably increasing confidence during the development of secure contracts [Atz+19].

2.5 Programming language theory

In this section, we give an overview of results in the theory of programming languages, which are closely related to our main research questions. Although the body of work in this area is vast, we provide a coarse stratification of relevant sub-fields and provide pointers for further investigation.

Semantics of Programming Languages. Since most of the work discussed in this section deals with the semantics of programming languages, it would be helpful to review the prominent styles of specifying such semantics.

Denotational semantics study programming constructs as mathematical objects in some *semantic domain* $\boldsymbol{D}$, where each type of the language is mapped to its denotation via a function $[\![_]\!] : \boldsymbol{Type} \to \boldsymbol{D}$ [Sco70]. This approach is *compositional*, since the denotation of a large program can be derived from the denotations of its constituents, *e.g.,* the previous definition naturally extends to *contexts* and *typing derivations*:

$$[\![\Gamma \vdash t : \sigma]\!] : [\![\Gamma]\!] \to [\![\sigma]\!]$$

In contrast to the abstract nature of denotational semantics, *operational semantics* try to capture a more concrete description of a programs execution. Here, we are particularly interested in the so-called *small-step* or *structural* operational semantics [Plo81], where a program's meaning corresponds to a series of individual computational steps, rather than describing the overall result of computation as in *big-step* or *natural* operational semantics. An important advantage of such an approach is that it is closer to the intuitive computational behaviour and allows a *syntax-directed*, *inductive* definition, which is amenable to rigorous verification. While big-step semantics are also structural,

it abstracts away evaluation details that we might want to consider, in order to have an accurate understanding of a program's execution. *Inference rules* are given as a set of transitions from initial to target configuration, which are typically modelled by a (labelled) transition system, a reoccurring phenomenon throughout our own work.

Last but not least, a more recent development was the formulation of a language's semantics in game-theoretic terms, known as *game semantics* [Abr97]. This is of interest to us, since it inspired BitML's semantics and, moreover, provided solutions to long-standing problems on *full abstraction* of several programming languages (more on this later on).

Bisimulation. In [Pie97], two foundational calculi are presented: the λ-calculus for purely functional computations and the π-calculus for concurrent systems. There are many possible definitions of equivalence between different terms/processes.

Denotational equivalence states that two terms are equivalent if their semantic counterpart is equivalent:

$$M = N \iff [\![M]\!] = [\![N]\!]$$

This definition is not always satisfactory, since the semantic domain may contain *too many* values, hence differentiating between terms/processes that behave the same. As an artificial example, consider the case where we denote terms as the sequence of the syntactic elements; then $1+2$ and $2+1$ would not be denotationally equivalent, although they behave the same.

To remedy this, the notion of *observational equivalence* was introduced, in which two terms are considered equivalent if they behave exactly the same based on the observations we can extract from them. The most common form of this type of equivalence is *contextual equivalence*, where we consider two terms equivalent, only when their impact is the same on any surrounding context C (where $\downarrow$ denotes normalisation):

$$M = N \iff \forall C.\ C[M]\downarrow \ \textit{iff}\ \ C[N]\downarrow$$

While this notion of equivalence captures the semantics more strictly and is closer to general intuition by virtue of being operational in character, it may prove difficult to prove due to the universal quantification of arbitrary contexts.

An alternative formulation, based on a language's operational semantics, is *bisimulation*, where two systems are considered equivalent if they execute corresponding steps that start and finish on *bisimilar* states. Thus, proving bisimulation entails coming up with some relation to connect system states, as well as a proof that steps allowed by the operational semantics move between states that are related in this way.

In its original form, *strong bisimulation*, the two systems must run in "lock-step", *i.e.,* there is an one-to-one correspondence between their steps. A more common form of bisimulation is the so-called *weak bisimulation*, where systems are allowed to perform an arbitrary number of internal steps, *i.e.,* a single step in the source might correspond to multiple steps in the target.

The main advantage of bisimulation over contextual equivalence is the absence of the aforementioned universal quantification, therefore lending itself to a pragmatic proof technique, *coinduction* [San11], where one can assume bisimilarity and, if that leads to no contradiction, we get the desired proof of bisimulation. In contrast, an inductive proof would make it harder to identify all the possible way such non-deterministic systems behave the same, although their internal structure might be completely different. For instance, this kind of proof technique is used to study process equivalence in the context of the π-calculus [San96].

Lastly, a language-agnostic technique for relating different state-based system, reminiscent of bisimulation, is presented in the seminal work of [AL88]. Specifically, the goal is to prove that a high-level specification is correctly implemented by a lower-level one. This can be achieved by providing a *refinement mapping* from high-level states/steps to low-level ones. The authors continue to show that there exist certain techniques to augment the low-level definition (*e.g.,* by introducing auxiliary variables), so as to guarantee the existence of such a mapping.

Full Abstraction for Denotational Semantics. It is commonplace to define both denotational and operational semantics for a language, given that each has complementary advantages. An issue that arises, however, is whether these two different semantics coincide; denotationally equivalent terms should also be observationally equivalent.

This question was introduced in the context of the *Programming Computable Functions* (PCF) programming language, an extension of the simply-typed λ-calculus, where a semantics is dubbed *fully-abstract* when both equivalences coincide; observational equivalence corresponds to denotational equivalence [Plo77]. [2] Around the same time, Milner discussed full abstraction for the more general setting of typed λ-calculi [Mil77]. Another interesting use of *full abstraction* concerns the comparison of different evaluation strategies, as exemplified in [Rie93], where call-by-value, call-by-name and lazy evaluations are considered.

[2] A very similar notion is *computational adequacy*, which states that observationally distinct terms have distinct denotations.

There have been alternative definitions that generalise the notion of equivalence, such as *logical full abstraction* [LP00]. Game semantics have proven quite flexible in tackling the problem of full abstraction, as seen by the diverse range of programming constructs approached this way, which include subtyping, probabilistic choice, references, non-determinism and concurrency [Cur07].

Other examples of full abstraction results include a fully-abstract *trace semantics* for a λ-calculus extended with *references* [Lai07], as well as a fully-abstract semantics for classical processes [KMP19].

Full Abstraction for Expressiveness. It turns out that finding such fully-abstract translations is a notoriously hard process, witnessed by the sheer amount of publications on full abstraction for PCF [AJM13], sometimes not even being satisfiable [Par16]. More surprisingly, it is debatable whether full abstraction actually is a good global criterion for the systematic investigation of the relative expressive power of programming languages, as shown by demonstrative counterexamples in [GN16].

Therefore, a central problem in the theory of programming languages is systematically comparing the expressiveness of different languages. Several theoretical frameworks have been proposed to tackle this question, although there is still no consensus towards a universally accepted solution.

A framework suited to formally study the expressiveness of different language extensions appears in [Fel91], where different extensions to the basic λ-calculus are shown and compared with the core language. Some are proven to strictly raise the expressive level of the language, hence rendering a local embedding impossible.

A more generic framework is provided in [Sha91], where the languages compared need not have a common semantic basis. In this case, several concurrent languages are compared, though the method applies to sequential ones as well.

In [Mit93], another technique is introduced based on *abstraction-preserving reductions*, where several examples and counterexamples are shown.

Certified Compilation. Proving compilers correct has had a long history in computer science, starting from the 1960s [MP67]. The primary purpose is to give formal guarantees that compiling from a high-level language to a lower-level ones does not alter the source semantics.

A renewed interest in correct compilation arose in recent years by CompCert, a compiler for a large subset of C (called Clight), targeting all common instruction

sets [Ler09].[3] Entirely programmed in the Coq proof assistant, the compiler itself is programmed in Coq's purely-functional subset Gallina and its proofs of correctness are expressed with *dependent types* and proven using *tactics* [Bar+97]. CompCert demonstrated that full compiler verification is *feasible* and gives a lot of benefits, since the compiler backend is highly optimised, consisting of multiple phases, each proven to preserve the desired safety properties via bisimulation and the resulting performance is on par with GCC [Ler06].

An alternative approach for certified compilation, which does not require that the compiler is written in a dependently-typed language, is *type-preserving* compilation. The seminal paper of [Mor+99] describes the design of an optimising compiler from System F to a typed assembly language TAL, where high-level abstractions (*e.g.,* closures) are enforced on the type level. Therefore, preserving the types through compilation guarantees that no such violations arise in the produced low-level code. Of course, as we require more intricate properties to be preserved through compilation, we will need ever more expressive type systems, which is by itself a fruitful direction for future research in secure compilation.

Certifying compilers, on the other hand, opt out of verifying the whole compilation process and, instead, verify the validity of each individual output through a program outside the compiler, called the *certifier*. Note that this technique is sometimes also referred to as *translation validation*. A primary example of this is the Touchstone compiler from a type-safe subset of C to the DEC Alpha assembly language [NL98]. Although the compiler cannot be said to be *bug-free*, it will nonetheless report any incorrect compiler outputs and is, arguably, much easier to prove and amenable to future compiler extensions (*e.g.,* new optimisation phases), since the certifier can be developed independently.

Secure Compilation. If we would also like to preserve security properties through compilation, we arrive back to the notion of full abstraction, since it is the technique most often employed in the context of *secure compilation* [PAC19].

The seminal work of [Aba99] uses full abstraction in two different compilation issues; first, preserving security properties of Java classes arising from access modifiers (*e.g.,* `public`/`private`) down to the generated intermediate bytecode and, secondly, implementing private channel communication on top of the π-calculus via the use of cryptographic primitives.

Other notable examples include a fully-abstract compiler from an ML-like subset of F* [Swa+11] to Javascript, enabling developers to reason about their programs

[3] CompCert initially only compiled down to PowerPC, although it supports x86, ARM and RISC-V currently.

on the high level of ML, without having to thoroughly understand the intricacies of Javascript [Fou+13]. The full abstraction proof of the compiler is mechanised in F* and utilises the notion of bisimulation discussed previously.

Alas, full abstraction does not always capture security properties we would like to enforce or could even be impossible to find such translations [Par16; PAC19]. Hence the need to explore a more open space of secure compilation criteria that can be preserved against any adversarial context. In [Aba+19], different properties are investigated, ranging from simple *trace properties* (*e.g.,* safety) to *hyperproperties* (*e.g.,* non-interference) and *relational hyperproperties* (*e.g.,* trace equivalence). Results indicate that most are easier to prove than full abstraction and provide strictly stronger security guarantees.

Another limitation of full abstraction manifests when the source language cannot express some constructs of the target language, thus creating issues in *back-translation*; a crucial step in the full abstraction proof, where target-level contexts are translated to equivalent source-level ones. A solution is examined in [DPP16a], where back-translation is only considered *approximately*, up to a number of reduction steps. Furthermore, this work employs commonly-used techniques to overcome the limitation of contextual equivalence, such as *cross-language logical relations* and *step-indexing*.

Kripke logical relations (KLR) are a particular form of logical relations, taking a form similar to Kripke semantics in intuitionistic logic. They have been extensively used to solve long-standing full abstraction problems, such as the aforementioned obstacle in the context of PCF [OR95]. When dealing with languages that support recursion, *step-indexing* restricts recursive operations to approximations up to a certain number of iterations. Step-indexing, along with other properties such as bi-orthogonality and realizability [BHP10], has been successfully used to prove compilation correctness from a simply-typed functional language to a variant SECD machine [BH09], as well as from an impure ML-like λ-calculus to idealised assembly [HD11].

However, bisimulation and Kripke logical relations have complementary advantages, leading to an increased interest in combining them to get a more flexible and modular proof technique. [Hur+12] identifies the limitations of each technique and arrives at a novel formulation of a *relation transition system* (RTS). Specifically, RTSs marry the ability of bisimulation to reason about recursive structure coinductively, while at the same time supporting local-state reasoning via the transition semantics of KLRs.

Similarly, [Nei+15] introduces the notion of *parametric inter-language simulations* (PILS), in order to overcome limitations that manifest when considering *compositional* compilation correctness, rather than whole-program compilation. All this is done in the

context of a compiler from a higher-order imperative language to assembly with multiple optimisation passes, extending the previous work of [HD11] and [Hur+12].

Chapter 3

Target Language: Bitcoin

Before we attempt to formulate the correctness of the BitML compiler, we first need to define the semantics of the target language of the compilation, namely the low-level model of Bitcoin transactions and scripts.

Our mechanisation closely follows the formal model of Bitcoin transactions introduced by a previous paper by the BitML team [Atz+18b]:

> Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. "A Formal Model of Bitcoin Transactions". In: *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*. Ed. by Sarah Meiklejohn and Kazue Sako. Vol. 10957. Lecture Notes in Computer Science. Springer, 2018, pp. 541–560. DOI: `10.1007/978-3-662-58387-6_29`. URL: `https://doi.org/10.1007/978-3-662-58387-6_29`

This chapter can thus be read as a mechanised presentation of the same material in literate Agda form. In particular, the formal model mechanised here is described in §3 of the original paper; we will provide further such correspondences for subsections.

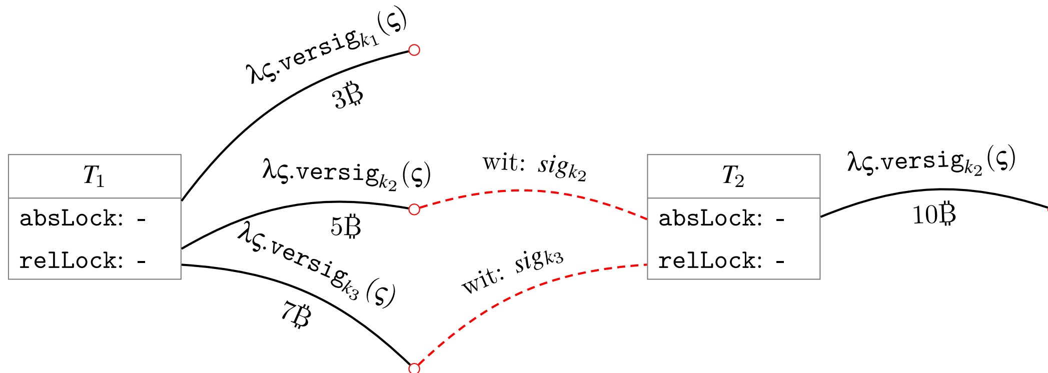
The entirety of the Agda development is publicly accessible in HTML format here:

`https://omelkonian.github.io/formal-bitcoin/`

Intuitive example. Let us start off with an informal description of how a UTxO-based blockchain such as Bitcoin operates. The full workings of a blockchain comprise a very complex system in its entirety: the *network* layer taking care of messages exchanged amongst peers/nodes in a distributed setting, the *consensus* layer deciding a common order of transactions submitted to the network, and the *ledger* layer recording transactions and keeping track of the current set of unspent transaction outputs (a.k.a. the UTxO set). While an exhaustive description of a blockchain is outside the scope of this work,

here we will focus on the *ledger* layer and how transactions submitted by the users of the blockchain network modify the *ledger state*.

A blockchain is a *directed acyclic graph* (DAG), where nodes represent transactions and edges indicate resource usage, as depicted below.



Outgoing edges (coloured in **black**) stand for fresh UTxOs that a transaction *produces* which carry a monetary value locked by a script that dictates who has access to these funds. *Incoming* edges (coloured in **red**) *consume* such UTxOs, providing some *witness* to the locking script in order to unlock it and thus gain access to the locked funds. Furthermore, a transaction also contains *timelocks* that constrain at which point in the evolution of a blockchain it should be considered valid.

Through this lens, the current state of a blockchain ledger can be thought of as the *fringe* of the DAG, which we commonly refer to as the UTxO set (*i.e.,* all dangling outputs that are yet to be consumed). Transactions modify the graph's fringe by connecting to a subset of its dangling outputs and extending with some newly introduced outputs.

Necessarily everything originates from a *genesis* transaction that has no inputs, and transactions are typically bundled in *blocks*, hence the term *blockchain*, but we do not consider these complications here.

In the example above, transaction T_1 produces three outputs holding different monetary values and locked by distinct *public* keys k_i; T_2 then consumes two of them (holding 5B and 7B resp.) by signing (parts of) the transaction with the corresponding *private* keys that unlocks said funds and hence has sufficient funds to output a new UTxO carrying a total of 10B.

The rest of this chapter presents a formal model to explicate and analyse such blockchains, culminating into a mechanised version of the example above where consistency can provably be established.

Chapter overview. We will start with some basic definitions in §3.1 and postulate the necessary cryptography in §3.2; these will provide the building blocks to let us define Bitcoin scripts in §3.3 and Bitcoin transactions in §3.4.

Finally, based on a denotational semantics of scripts (§3.3.1) and transactions (§3.4.2), we will formalise a notion of *consistency* for Bitcoin blockchains in §3.5.

3.1 Basic entities

We start out with some basic types: using natural numbers to represent monetary values (amount in Satoshis) as well as discrete time points, and integers to represent hashes and signatures.

```
Value  = ℕ
Time   = ℕ
HashId = ℤ
```

```
variable
  n n′ : ℕ
  t t′ : Time
```

On the right we associate specific variable names to these types. However, we will omit such definitions going forward for the sake of brevity; when such a *generalizable variable*[1] is encountered in some larger Agda type, it will be clear from the context what its type should be.

Disclaimer

While this is a crude model of cryptographic hashes/signatures, compared for instance to an abstract type of bitstrings that satisfy some key properties, we opt to strictly follow the formulation in the original paper here as integer arithmetic is actually employed later in the semantics of scripts and we will eventually need a concrete definition to compute with examples in subsequent sections.

As most examples will be phrased using Bitcoins (equal to 100 million Satoshis), we define a shorthand:

```
_B : Op₁ Value
_B = _* 100000000
```

While it is not important what precisely the discrete time points represent, we will follow the Bitcoin implementation and use *Unix time*, *i.e.,* the time number represents seconds since the *Unit epoch*: January 1st, 1970. (For the sake of simplicity, we will assume an idealised 30-day month, as we do not really care about exact dates in our formal model.) That way, we can easily define a convenient notation to specify other

[1] https://agda.readthedocs.io/en/v2.6.3/language/generalization-of-declared-variables.html

time units (hours, etc.) or refer to a particular calendar date, which we will use in subsequent examples.

```
_seconds _minutes _hours _days _months _years : Op₁ Time
_seconds = id
_minutes = _* 60
_hours   = _minutes ∘ (_* 60)
_days    = _hours   ∘ (_* 24)
_months  = _days    ∘ (_* 30)
_years   = _months  ∘ (_* 12)

record Date : Type where
 constructor _/_/_
 field day month year : ℕ

date:_ : Date → Time
date: d / m / y = if y <ᵇ 1970 then 0 else (y ∸ 1970) years + m months + d days
```

3.2 Cryptographic operations

Next, we need to postulate some cryptographic operations we will need throughout our development. For higher assurance, we could instead develop a machine-checked cryptography library from the ground up without any postulates, but this lies outside the scope of this thesis as we consider cryptographic concerns orthogonal to the correctness of the BitML compiler.

We will make use of key pairs consisting of a public and private key, as well as a postulated *universal* hashing function parametric over any type we might need it for.

```
record KeyPair : Type where
 field pub : HashId
       sec : HashId
open KeyPair public
unquoteDecl DecEqᵏᵖ = DERIVE DecEq [ quote KeyPair , DecEqᵏᵖ ]

postulate _♯ : ∀ {A : Type ℓ} → A → HashId
```

Note the use of the derivation strategy for decidable equality of key pairs, described later in Appendix A.3. The verbosity of this 'deriving' is unfortunate, and certainly does not fair well in comparison to Haskell's syntax, but stems from certain limitations in the Agda type-checker itself, namely the fact that *scope checking* is performed prior to *type checking* and no interleaving of the two is possible (e.g. to allow meta-programs to generate names).[2] In order to save space, we will omit similar boilerplate code for the

[2]See the following open issues in Agda's Github repository: #3699 , #5399 , #5864 .

following sections, but the reader can assume decidable equality is always derived for types that resemble *first-order data*.

We further postulate the two operations of *signing* with a key pair and *verifying* a signature, along with two corresponding laws: verifying only succeeds when applied to a signed element of matching keys, and signing is injective in both of its arguments (keys used and elements signed).

```
postulate
  SIG : KeyPair → A → HashId
  VER : KeyPair → HashId → A → Bool

  VERSIG : True (VER k σ x)
           ════════════════
           σ ≡ SIG k x

  SIG-injective : (∀ {x : A} → Injective≡ (λ k → SIG k x))
                × (∀ {k} → Injective≡ (λ (x : A) → SIG k x))
```

Notation

Horizontal syntax for equivalences. The double horizontal line in the above is the bidirectional version of the horizontal inference rules we have seen so far, standing for equivalence between statements and requires proof of both directions (*c.f.*, Appendix A.2).

From these, we then derive useful lemmas that indicate whether a signature verification is successful or not.

```
VERSIG≡ : True (VER k (SIG k x) x)
VERSIG≡ = VERSIG .proj₂ refl

VERSIG≢ : k ≢ k′ → ¬ True (VER k (SIG k′ x) x)
VERSIG≢ k≢ = ⊥-elim ∘ k≢ ∘ sym ∘ SIG-injective .proj₁ ∘ VERSIG .proj₁

VERSIG≢′ : x ≢ x′ → ¬ True (VER k (SIG k x) x′)
VERSIG≢′ x≢ p = ⊥-elim $ x≢ $ SIG-injective .proj₂ (VERSIG .proj₁ p)
```

This concludes the general cryptographic primitives that apply to all types, but we will later come across more involved operations for specific types that build upon these primitives.

As a last remark, it should be made clear that these cryptographic operations are *idealistic*, since it is not possible to implement them in practice. In particular, injectivity of hash functions and signing necessarily breaks at some point, albeit within a negligible probability. We believe these issues to be orthogonal to the investigation

we are conducting here, and so can safely mitigate the need for probabilistic reasoning and trust that our results hold within an overarching assumption that the cryptography behind is sound.

3.3 Bitcoin scripts *(§3.1 of [Atz+18b])*

We will employ a *well-scoped*, *well-typed* representation of script expressions.

In order to enforce script expressions are *well-scoped*, we will index them by their context, which in this case is simply the number of bound variables:

```
ScriptContext : Type
ScriptContext = ℕ
```

To further guarantee that scripts are also *well-typed*, our expression data type will also be indexed by a suitable embedding of the type universe associated to values returned by such scripts which are either booleans or integers:

```
data ScriptType : Type where
  `𝔹 `ℤ : ScriptType
```

Scripts consist of integer variables (*i.e.,* indices) pointing to an entry in the current scope/context (`var`), constant numbers (`` ` ``), the usual arithmetic and boolean expressions (`+`, `-`, `=`, `<`, `if_then_else`), as well as cryptographic operations related to hashing and signatures (given an arithmetic expression, `hash` calculates its hash and `|_|` calculates its length in bytes, while `versig` represents an *m*-out-of-*n* signature scheme — *c.f.,* Section 3.4.1 for more details), and a way to add temporal constraints to a sub-script (`absAfter`, `relAfter`). Therefore, variables are always numeric (referring to the arguments the script has been called with) and the result type can be either numeric or boolean.

```
module _ (ctx : ScriptContext) where
  data Script : ScriptType → Type where
    var              : Fin ctx → Script `ℤ
    `                : ℤ → Script `ℤ
    _`+_ _`-_        : Script `ℤ → Script `ℤ → Script `ℤ
    _`=_ _`<_        : Script `ℤ → Script `ℤ → Script `𝔹
    `if_then_else_   : Script `𝔹 → Script ty → Script ty → Script ty
    hash |_|         : Script `ℤ → Script `ℤ
    versig           : List KeyPair → List (Fin ctx) → Script `𝔹
    absAfter_⇒_ relAfter_⇒_ : Time → Script ty → Script ty
```

Notice how the context remains unchanged throughout all the constructors, hence we promote it to a static *parameter*, while the result type is dynamic ad therefore has to be a datatype *index*. Moreover, `versig` can only verify signatures that are arguments to

the script (`Fin ctx`), *i.e.*, there is no intrinsic way to construct and verify signatures within the scripting language.

Using these building blocks we can derive other useful operations, which we will make use of later to express the BitML compiler. In particular, equality comparisons and boolean conditionals via `if-then-else` are sufficient to derive all the usual boolean operations:

```
`false `true : Script ctx `𝔹
`false = ` (+ 1) `= ` (+ 0)
`true  = ` (+ 1) `= ` (+ 1)

_`∧_ _`∨_ : Script ctx `𝔹 → Script ctx `𝔹 → Script ctx `𝔹
e `∧ e′ = `if e then e′     else `false
e `∨ e′ = `if e then `true else e′

`not : Script ctx `𝔹 → Script ctx `𝔹
`not e = `if e then `false else `true
```

Bitcoin scripts are the ones that return a boolean value:

```
data BitcoinScript (ctx : ScriptContext) : Type where
  ƛ_ : Script ctx `𝔹 → BitcoinScript ctx
```

Here's an example Bitcoin script, which demonstrates the ability of the type system to keep us in check so as not to reference any *out-of-scope* variables: operating under the context of two variables and some hypothetical key `k`, this script ascertains that the first witness is signed with the private part of `k` and that the hash of the two witnesses match (*i.e.*, a contrived way of saying they are equal).

```
_ = BitcoinScript 2
  ∋ ƛ versig [ k ] [ 0F ] `∧ hash (var 1F) `= hash (var 0F)
```

(Patterns `0F` and `1F` construct indices of type `Fin 2`.)

Note that operator precedence (`ƛ` < `` `∧ `` < `` `= ``) relieves us of the burden of fully parenthesising the expression, which in this case would be:

```
_ = BitcoinScript 2
  ∋ ƛ (  (versig [] [ 0F ])
      `∧ (hash (var 1F) `= hash (var 0F)) )
```

3.3.1 Denotational semantics: scripts *(§3.4 of [Atz+18b])*

The denotational semantics of Bitcoin scripts will be formulated in an environment which gives a valuation of each variable in the context and also specifies the current witness redeeming the script (i.e. a transaction coupled with an index of its inputs).

```
record Environment (i o : ℕ) (ctx : ScriptContext) : Type where
  field tx  : Tx i o
        ix  : Fin i
        val : Witness ctx
```

The obvious Agda types will denote our script (embedded) types.

```
⟦_⟧ₜ : ScriptType → Type
⟦ `𝔹 ⟧ₜ = Bool
⟦ `ℤ ⟧ₜ = ℤ
```

The denotational semantics of a script is a function which, given the current environment, *may* return a value (of the denoted type).

```
⟦_⟧′_ : Script ctx ty → Environment i o ctx → Maybe ⟦ ty ⟧ₜ
⟦ e ⟧′ ρ = case e of λ where
  (var x)                → ⦇ (lookup (ρ .val) x) ⦈
  (` x)                  → ⦇ x ⦈
  (e `+ e′)              → ⦇ ⟦ e ⟧′ ρ + ⟦ e′ ⟧′ ρ ⦈
  (e `- e′)              → ⦇ ⟦ e ⟧′ ρ - ⟦ e′ ⟧′ ρ ⦈
  (e `= e′)              → ⦇ ⟦ e ⟧′ ρ == ⟦ e′ ⟧′ ρ ⦈
  (e `< e′)              → ⦇ ⟦ e ⟧′ ρ <ᵇ ⟦ e′ ⟧′ ρ ⦈
  (`if b then e else e′) → ⦇ if ⟦ b ⟧′ ρ then ⟦ e ⟧′ ρ else ⟦ e′ ⟧′ ρ ⦈
  (∣ e ∣)                → ⦇ size (⟦ e ⟧′ ρ) ⦈
  (hash e)               → ⦇ (⟦ e ⟧′ ρ) ♯ ⦈
  (versig k σ)           → just $ ver⋆ k (lookup (ρ .val) <$> σ) (ρ .tx) (ρ .ix)
  (absAfter t ⇒ e)       → if ρ .tx .absLock ≥ᵇ t then ⟦ e ⟧′ ρ else nothing
  (relAfter t ⇒ e)       → if ρ .tx ‼ʳ ρ .ix  ≥ᵇ t then ⟦ e ⟧′ ρ else nothing
 where size : ℤ → ℤ
       size x = + (suc (digits (fromℕ Integer.∣ x ∣)) / 7)
```

Note the use of *idiom brackets*,[3] which lets us work under an *applicative functor* — in this case the effect of erroneous execution via `Maybe` — as if we were writing a pure program [MP08]. (The `size` calculation is copied verbatim from the original paper; it represents the length in bytes with some particular encoding in mind for this particular setting; this is unimportant throughout our work as no proof or example actually relies on this calculation.)

Bitcoin scripts will naturally be interpreted as functions with a boolean co-domain.

```
⟦_⟧_ : BitcoinScript ctx → Environment i o ctx → Bool
⟦ ƛ e ⟧ ρ = M.fromMaybe false (⟦ e ⟧′ ρ)
```

In order to verify a script, we interpret it in the context of a specific transaction input and check that the return value is `true`.

[3] https://agda.readthedocs.io/en/v2.6.3/language/syntactic-sugar.html#idiom-brackets

```
_,_⊨_ : (tx : Tx i o) (j : Fin i) → BitcoinScript ctx
      → ⦃ ctx ≡ (tx ‼ʷ j) .proj₁ ⦄ → Type
(tx , j ⊨ e) ⦃ refl ⦄ = T $ ⟦ e ⟧ record {tx = tx; ix = j; val = (tx ‼ʷ j) .proj₂}
```

As promised in Section 3.4, this is the point where we utilise the indices of the type families of scripts expressions and lengths of witnesses, respectively. We do so by equating them, thus making sure witnesses carry as many arguments as required by the top-level script expression.

Example 1. *Script execution* *(Example 2 of* [Atz+18b]*)*

Here is an example from the original paper that showcases a successful script execution: the witnesses and hashes have been artificially setup for the simple expression of the script to evaluate to `true`.

```
module _
  {σ s : HashId} {t k txi}
  {i} {is : Vec TxInput i} {ws} {rs}
  {o} {os : Vec ∃TxOutput o}
  (let h = s ♯
       T = record { inputs = txi ∷ is
                  ; wit = (-, [ σ ⨾ s ]) ∷ ws
                  ; relLock = rs
                  ; outputs = os
                  ; absLock = t })
  (σ≡ : σ ≡ SIG k (μ T 0F))
  where
  _ : T , 0F ⊨ ƛ versig [ k ] [ 0F ] `∧ (hash (var 1F) `= ` h)
  _ rewrite begin ver⋆ [ k ] [ σ ] T 0F ≡⟨ if-eta _ ⟩
                  VER k σ _            ≡⟨ cong (λ ◆ → VER _ ◆ _) σ≡ ⟩
                  VER k (SIG k _) _    ≡⟨ T⇒true VERSIG≡ ⟩
                  true                 ∎
      |     ≟-refl h
      =     tt
```

3.4 Transactions *(§3.2 of* [Atz+18b]*)*

Bitcoin transactions spend as inputs previously unspent outputs, to which they refer by the hash of the enclosing transaction and the index into its outputs.

```
record TxInput : Type where
  constructor _at_
  field txId  : HashId
        index : ℕ
```

Outputs carry a monetary value, along with a validator script which will be used to unlock these funds.

```
record TxOutput (ctx : ScriptContext) : Type where
  constructor _locked-by_
  field value     : Value
        validator : BitcoinScript ctx
```

One can unlock an output by providing a *witness* of script arguments to the locking validator such that it evaluates to `true`.

```
Witness : ℕ → Type
Witness n = Vec HashId n
```

Keeping track of the witness length rather than simply working with lists might seem superfluous now, but its usefulness will become clear when we later want to ensure the script context matches how many witnesses we provide as input (Section 3.3.1).

Transactions bundle up inputs, outputs, and an *absolute time lock* which constrains the time period for which the transaction is considered valid. Furthermore, for each input there is a corresponding witness and a *relative time lock* which constrains when this particular input can be spent.

```
record Tx (i o : ℕ) : Type where
  field
    inputs  : Vec TxInput      i
    wit     : Vec (∃ Witness)  i
    relLock : Vec Time         i

    outputs : Vec (∃ TxOutput) o
    absLock : Time
```

Notice how we existentially pack the indices of the `Witness` and `TxOutput` type for now, only to unpack them when we actually need to utilise their values. (As multiple levels of such plumbing of the existential quantifier and accessing the resulting packed indices will quickly become tedious, we will henceforth employ the intuitive convention of prefixing types/fields with an ∃ and omit the exact computations to save space; these will always be clear from the context. For instance, `∃Tx` denotes `∃[ i ] ∃[ o ] Tx i o`, and `∃o` projects the second index `o` out of this "existential" transaction.)

The indirection arising from the hashes on the inputs is not as well behaved as we would want, since there is nothing preventing us from an *out-of-bounds* error — either referring to (a hash of) a non-existent transaction or an index larger than its outputs. An alternative way to represent inputs is to actually include the whole transaction and a well-scoped index into its outputs. While this is certainly not faithful to how Bitcoin operates, it provides a useful abstraction that we will make use of in later sections.

```
record TxInput′ : Type where
  constructor _at_
  field tx′    : ∃Tx
        index′ : Fin (∃o tx′)
```

It is finally time to harvest the fruits of our well-scoped labour, in way of safe lookup functions to retrieve a transaction's information.

```
_‼ⁱ_ : Tx i o → Fin i → TxInput
_‼ⁱ_ = V.lookup ∘ inputs

_‼ʷ_ : Tx i o → Fin i → ∃Witness
_‼ʷ_ = V.lookup ∘ wit

_‼ʳ_ : Tx i o → Fin i → Time
_‼ʳ_ = V.lookup ∘ relLock

_‼ᵒ_ : Tx i o → Fin o → ∃TxOutput
_‼ᵒ_ = V.lookup ∘ outputs
```

Actually, this is merely a taste of the advantages we gain from our intrinsically-typed scripts and transactions; when we later formulate our dependently-typed BitML compiler in Chapter 5 is when these benefits will really shine!

Finally, the building block of a Bitcoin blockchain is a transaction at a certain point in time.

```
record TimedTx : Type where
  constructor _at_
  field transaction : ∃Tx
        time        : Time
```

3.4.1 Cryptographic operations on transactions *(§3.3 of [Atz+18b])*

First and foremost, the alternative well-scoped definition of transaction inputs can be readily converted to the original form that uses hashes (`_♯`) and unbound indices.

```
_♯at_ : Tx i o → Fin o → TxInput
tx ♯at j = (tx ♯) at toℕ j

hashTxⁱ : TxInput′ → TxInput
hashTxⁱ ((_ , _ , tx) at j) = tx ♯at j
```

We also need a standard way to adhere to Bitcoin's support for *segregated witnesses* [Lau16] where the witnesses are separated from the base data of a transaction

to prevent malleability and, as a consequence, do not influence the computation of its hash. This is precisely why witnesses and relative time locks are separate fields from the inputs in a transaction (as defined in Section 3.4); we now enforce this before computing any hash by completely erasing witnesses.

```
wit⊥ : Vec ∃Witness n
wit⊥ = replicate (-, [])

wit→⊥ : Tx i o → Tx i o
wit→⊥ tx = record tx { wit = wit⊥ }
```

Notation

Implicit indices for existentials. At this point, it should be clear that we are dealing frequently with existentially-packed values of a dependent product. In our case, values for witnesses always carry the length of the corresponding vector as the first component of the tuple. When constructing such values, it turns out most of these can be automatically inferred by Agda's type checker, since the type dependencies rule out all but a single value, *e.g.*, all of the following expressions construct a singleton vector packed with its length: `(1 , [ x ])`; `(_ , [ x ])`; `(-, [ x ])`.

As regards to signing transactions and verifying signatures, we simplify the formulation in the original paper that provides multiple ways of signing via *signature modifiers* as we will only ever need to use the simplest case in our work (in [Atz+18b]'s terminology: $\mu = \alpha\alpha$).

```
μ : Tx i o → Fin i → Tx i o
μ {i = suc _} tx j = record tx { wit = wit⊥ [ 0F ]≔ (-, [ + (toℕ j) ]) }
```

The purpose of the `μ` signature modifier for an input j is to completely erase the witnesses of a given transaction, only retaining the index of said output at the first witness slot.

Signing a transaction's j-th input consists of filling in the j-th witness slot with a signature of the base transaction, *i.e.*, one post-processed using the signature modifier `μ`.

```
sig : List KeyPair → Tx i o → Fin i → Tx i o
sig ks tx j = record tx
  { wit = tx .wit [ j ]≔ (-, fromList (flip SIG (μ tx j) <$> ks)) }
```

We can then verify a transaction's j-th input signature by again computing the bare transaction using `μ`.

```
ver : KeyPair → HashId → Tx i o → Fin i → Bool
ver k σ tx j = VER k σ (μ tx j)
```

The whole point of the signature juggling above is to provide a safe API for the user to apply cryptographic operations on transactions that respect certain invariants; these could easily be broken if the user was allowed to freely run the primitive (postulated) `SIG` and `VER` procedures.

This construction can be generalised to a *m-of-n signature scheme*, where we instead work on sequences of keys and signatures: `sig⋆` now updates all input witnesses of a transaction using a vector of equal length as the inputs, and `ver⋆` now asserts that at least *m* keys have been used for the given *n* signatures.

```
sig⋆ : Vec (List KeyPair) i → Tx i o → Tx i o
sig⋆ kss tx = record tx
  { wit = (λ j → -, fromList (flip SIG (μ tx j) <$> lookup kss j)) <$> allFin _ }
```

```
ver⋆ : List KeyPair → List HashId → Tx i o → Fin i → Bool
ver⋆ _        []        _ _ = true
ver⋆ []       (_ ∷ _)   _ _ = false
ver⋆ (k ∷ ks) (σ ∷ σs) T j =
  if ver k σ T j then ver⋆ ks σs T j else ver⋆ ks (σ ∷ σs) T j
```

In subsequent examples, it will be useful to lift the signing law `VERSIG≡` that we postulated in Section 3.2 to one that works on singleton lists arising from the multi-signature scheme:

```
ver⋆sig≡ : ∀ {k} (t : Tx i o) (j : Fin i) → True $ ver⋆ L.[ k ] L.[ SIG k (μ t j) ] t j
ver⋆sig≡ {k = k} t j rewrite if-eta $ ver k (SIG k (μ t j)) t j = VERSIG≡
```

Note also how the multi-signature verification is order-dependent, *i.e.,* keys are consumed in sequence until they verify the first signature before moving on to subsequent signatures; this is clearly demonstrated by Example 2 below.

Example 2. 2-of-3 multi-signature (Example 1 of [Atz+18b])

Given 3 keys `kᵃ,kᵇ,kᶜ`, we will verify 2 signatures of a transaction `t`: operation `σp` signs the transaction using `kᵃ` and `σq` uses `kᵇ`.

```
module _
  (kᵃ kᵇ kᶜ : KeyPair) (kᶜ≢kᵇ : kᶜ ≢ kᵇ) (kᶜ≢kᵃ : kᶜ ≢ kᵃ) (kᵇ≢kᵃ : kᵇ ≢ kᵃ)
  (t : Tx (suc i) o)
  where
  σp  = SIG kᵃ (μ t 0F)
  σq  = SIG kᵇ (μ t 0F)
  ks  = List KeyPair ∋ [ kᶜ ⨾ kᵇ ⨾ kᵃ ]
  σs  = List HashId  ∋ [ σq ⨾ σp ]
  σs′ = List HashId  ∋ [ σp ⨾ σq ]
```

Notation

Literals for sequences. In order not to clutter presentation, we avoid constructing list-like data structures using immediately their 'nil' ([ ]) and 'cons' (∷) constructors, but rather define pattern synonyms for lists/vectors/etc. of a certain length:

```
pattern [_] x = x ∷ []
pattern [_⨾_] x y = x ∷ [ y ]
pattern [_⨾_⨾_] x y z = x ∷ [ y ⨾ z ]
```

... and so forth. The trick is that ∷ is an overloaded constructor for many types related to sequences, so we declare these pattern synonyms once but can reuse them for all of those types.

Verification will succeed only when the keys are given in the same order as the corresponding signatures, otherwise kᵇ will be consumed when trying to verify σp before getting the chance to verify σq.

We first demonstrate the two cases using *equational reasoning*:

```
_ : True (ver⋆ ks σs t 0F)
_ = true⇒T
  $ begin
      ver⋆ ks σs t 0F
    ≡⟨⟩
      (if VER kᶜ σq (μ t 0F) then
        ver⋆ [ kᵇ ⨾ kᵃ ] [ σp ] t 0F
      else
        ver⋆ [ kᵇ ⨾ kᵃ ] [ σq ⨾ σp ] t 0F)
    ≡⟨ if-false $ ¬T⇒false $ VERSIG≢ kᶜ≢kᵇ ⟩
      ver⋆ [ kᵇ ⨾ kᵃ ] σs t 0F
    ≡⟨⟩
      (if VER kᵇ σq (μ t 0F) then
        ver⋆ [ kᵃ ] [ σp ] t 0F
      else
        ver⋆ [ kᵃ ] [ σq ⨾ σp ] t 0F)
    ≡⟨ if-true $ T⇒true VERSIG≡ ⟩
      ver⋆ [ kᵃ ] [ σp ] t 0F
    ≡⟨⟩
      (if VER kᵃ σp (μ t 0F) then
        ver⋆ [] [] t 0F
      else
        ver⋆ [] [ σp ] t 0F)
    ≡⟨ if-true $ T⇒true VERSIG≡ ⟩
      ver⋆ [] [] t 0F
```

```
  ≡⟨⟩
    true
  ∎
```

We have explicitly included the intermediate calculation steps to inspect the internal evaluation of the type theory and showcase the merits of *referential transparency* inherent in (pure) functional programming languages, however we can move closer to the notation used in the paper by omitting these verbose unfoldings:

```
_ : ¬ True (ver⋆ ks σs′ t 0F)
_ = false⇒¬T
  $ begin
      ver⋆ ks σs′ t 0F
    ≡⟨ if-false $ ¬T⇒false $ VERSIG≢ kᶜ≢kᵃ ⟩
      ver⋆ [ kᵇ ; kᵃ ] [ σp ; σq ] t 0F
    ≡⟨ if-false $ ¬T⇒false $ VERSIG≢ kᵇ≢kᵃ ⟩
      ver⋆ [ kᵃ ] [ σp ; σq ] t 0F
    ≡⟨ if-true $ T⇒true VERSIG≡ ⟩
      ver⋆ [] [ σq ] t 0F
    ≡⟨⟩
      false
    ∎
```

A much more concise proof — that is also better behaved with respect to code evolution and refactoring [Chl13] — can be achieved using Agda's *rewrite mechanism*:[4]

```
_ : True (ver⋆ ks σs t 0F)
_ rewrite ¬T⇒false $ VERSIG≢ {x = μ t 0F} kᶜ≢kᵇ
  |       T⇒true   $ VERSIG≡ {k = kᵇ} {x = μ t 0F}
  |       T⇒true   $ VERSIG≡ {k = kᵃ} {x = μ t 0F}
  =       tt

_ : ¬ True (ver⋆ ks σs′ t 0F)
_ rewrite ¬T⇒false $ VERSIG≢ {x = μ t 0F} kᶜ≢kᵃ
  |       ¬T⇒false $ VERSIG≢ {x = μ t 0F} kᵇ≢kᵃ
  |       T⇒true   $ VERSIG≡ {k = kᵃ} {x = μ t 0F}
  =       id
```

In the rest of this document, we will use `rewrite` to save space whenever we do not really care about the exact evaluation steps taken by the Agda type checker.

3.4.2 Denotational semantics: transactions

(§3.5 of [Atz+18b])

[4] https://agda.readthedocs.io/en/v2.6.3/language/with-abstraction.html#with-rewrite

Notation

Accessors. In Agda, fields of a record value can be accessed using 'dot' notation, *e.g.,* `o .value` extracts the value of an output. Alas, it is often the case that the value lies deeper within some larger structure, and it quickly becomes tedious to chain all such projection functions to "zoom" in the precise level required.

From now on, we will use a stronger 'fat-dot' notation to automatically find the right zooming level for us, *e.g.,* `(0 , o) •value`, but omit their (uninteresting) definitions to save space.

Actually, our Prelude provides a general framework for generating such 'accessors', described in detail in Appendix A.5.

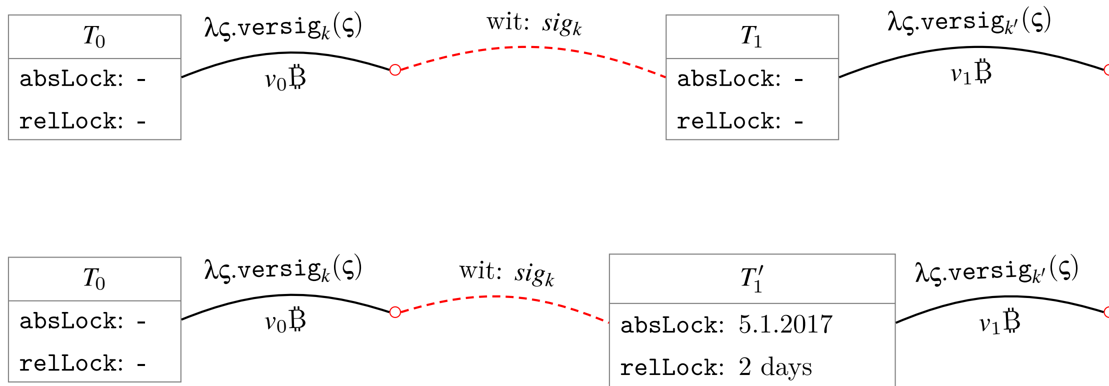
The semantics of transactions are encapsulated in the *output-redeeming* relation, which specifies the conditions under which a previously unspent output j can be redeemed from a new transaction input j' at a later time: $T, j, t \overset{v}{\rightsquigarrow} T', j', t'$.

```
record _,_,_~[_]_,_,_ (tx : Tx i o)    (j : Fin o)   (t : Time)
                       (v : Value)
                       (tx′ : Tx i′ o′) (j′ : Fin i′) (t′ : Time) : Type where
  field

   ⦃ witness~validator ⦄ :
     (tx ‼ᵒ j) .proj₁ ≡ (tx′ ‼ʷ j′) .proj₁

   input~output :
     tx′ ‼ⁱ j′ ≡ tx ♯at j

   scriptValidates :
     tx′ , j′ ⊨ (tx ‼ᵒ j) •validator

   value≡ :
     v ≡ (tx ‼ᵒ j) •value

   satisfiesAbsLock :
     t′ ≥ tx′ .absLock

   satisfiesRelLock :
       (t′ ≥ t)
     × (t′ ∸ t ≥ tx′ ‼ʳ j′)
```

The relation ensures that the validator scripts match, the validator script executes successfully, the monetary value redeemed is exactly equal to the one carried on the output being spend, and the temporal constraints are satisfied.

The above semantics, along with the signature scheme defined in Section 3.4.1 and the semantics of scripts defined in Section 3.3.1, are adequate to replay Examples 4 and 5 from the original paper in a mechanised fashion.

Example 3. *Output redeeming I* *(Example 3 of* [Atz+18b]*)*
Consider the three transactions T_0, T_1, T_1' depicted below:



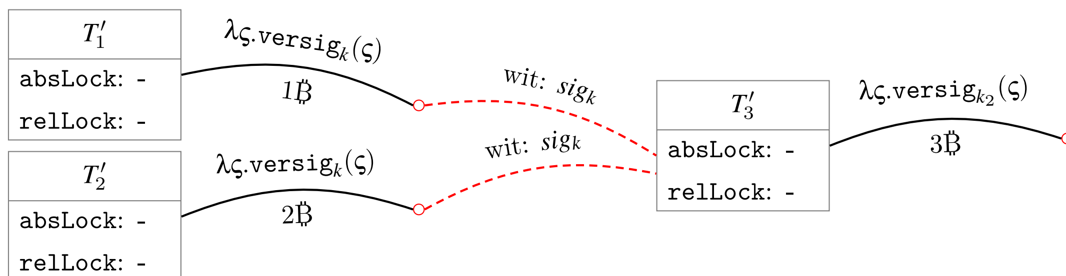
There are two possible scenarios for spending T_0's output, either by T_1 or T_1' respectively, which we can easily verify by proving that the redeeming relation holds, *i.e.*, construct a value of the corresponding `record` type.

```
module Example3 {k k′ : KeyPair} {v₀ v₁ : Value} where

  t₀ = date: 2 / 1 / 2017
  t₁ = date: 6 / 1 / 2017

  T₀ : Tx 0 1
  T₀ = record
    { inputs  = []
    ; wit     = []
    ; relLock = []
    ; outputs = [ 1 , v₀ locked-by ƛ versig [ k ] [ 0F ] ]
    ; absLock = t₀ }

  T₁ : Tx 1 1
  T₁ = sig⋆ [ [ k ] ] record
    { inputs  = [ (T₀ ♯) at 0 ]
    ; wit     = wit⊥
    ; relLock = [ 0 ]
    ; outputs = [ 1 , v₁ locked-by ƛ versig [ k′ ] [ 0F ] ]
    ; absLock = t₁ }

  T₁′ : Tx 1 1
  T₁′ = sig⋆ [ [ k ] ] record
    { inputs  = [ (T₀ ♯) at 0 ]
    ; wit     = wit⊥
    ; relLock = [ 2 days ]
    ; outputs = [ 1 , v₁ locked-by ƛ versig [ k′ ] [ 0F ] ]
```

```
  ; absLock = date: 5 / 1 / 2017 }

T₀↝T₁ : T₀ , 0F , t₀ ↝[ v₀ ] T₁ , 0F , t₁
T₀↝T₁ = record
  { input~output      = refl
  ; scriptValidates  = ver⋆sig≡ T₁ 0F
  ; value≡            = refl
  ; satisfiesAbsLock = ≤-refl
  ; satisfiesRelLock = m≤m+n _ (4 days) , z≤n
  }

T₀↝T₁′ : T₀ , 0F , t₀ ↝[ v₀ ] T₁′ , 0F , t₁
T₀↝T₁′ = record
  { input~output      = refl
  ; scriptValidates  = ver⋆sig≡ T₁′ 0F
  ; value≡            = refl
  ; satisfiesAbsLock = m≤m+n _ (1 days)
  ; satisfiesRelLock = m≤m+n _ (4 days) , m≤m+n _ (2 days)
  }
```

Example 4. *Output redeeming II* *(Example 4 of [Atz+18b])*

Here is an example transaction T_3' that spends two inputs from T_1' and T_2' respectively and outputs the sum of their values.



We can mechanically verify this as before:

```
module Example4 {k k₂ : KeyPair} {t t′ : Time} (t≤t′ : t ≤ t′) where

  T₁′ : Tx 0 1
  T₁′ = record
    { inputs  = []
    ; wit     = []
    ; relLock = []
    ; outputs = [ 1 , 1 locked-by ƛ versig [ k ] [ 0F ] ]
    ; absLock = t }

  T₂′ : Tx 0 1
```

```
T₂′ = record
  { inputs  = []
  ; wit     = []
  ; relLock = []
  ; outputs = [ 1 , 2 locked-by ƛ versig [ k ] [ 0F ] ]
  ; absLock = t }

T₃′ : Tx 2 1
T₃′ = sig⋆ [ [ k ] ; [ k ] ] record
  { inputs  = [ (T₁′ ♯) at 0 ; (T₂′ ♯) at 0 ]
  ; wit     = wit⊥
  ; relLock = [ 0 ; 0 ]
  ; outputs = [ 1 , 3 locked-by ƛ versig [ k₂ ] [ 0F ] ]
  ; absLock = t′ }

T₁′⇝T₃′ : T₁′ , 0F , t ⇝[ 1 ] T₃′ , 0F , t′
T₁′⇝T₃′ = record
  { input~output     = refl
  ; scriptValidates  = ver⋆sig≡ T₃′ 0F
  ; value≡           = refl
  ; satisfiesAbsLock = ≤-refl
  ; satisfiesRelLock = t≤t′ , z≤n
  }

T₂′⇝T₃′ : T₂′ , 0F , t ⇝[ 2 ] T₃′ , 1F , t′
T₂′⇝T₃′ = record
  { input~output     = refl
  ; scriptValidates  = ver⋆sig≡ T₃′ 1F
  ; value≡           = refl
  ; satisfiesAbsLock = ≤-refl
  ; satisfiesRelLock = t≤t′ , z≤n
  }
```

However, if we tamper with the witnesses of T_3', we can verify that validation fails, *i.e.*, prove the *negation* of the redeeming relation.

```
T₃″ : Tx 2 1
T₃″ = record T₃′ {wit = [ T₃′ ‼ʷ 0F ; -, [ SIG k (μ T₃′ 0F) ] ]}

T₂′⇏T₃″ : T₂′ , 0F , t ⇏ T₃″ , 1F , t′
T₂′⇏T₃″ (_ , record {scriptValidates = ver≡})
  = ⊨-elim T₃″ 1F _ ver≡ $ false⇒¬T $
  begin
    ver⋆ [ k ] [ SIG k (μ T₃′ 0F) ] T₃″ 1F
  ≡⟨ if-eta _ ⟩
    VER k (SIG k _) _
```

```
  ≡⟨ ¬T⇒false $ VERSIG≢′ (λ ()) ⟩
    false
  ∎
```

3.5 Blockchain consistency (§3.6 of [Atz+18b])

We can finally formulate the notion of a blockchain as a timed sequence of transactions.

```
Blockchain : Type
Blockchain = List TimedTx
```

A useful operation to is to filter all transactions in a blockchain that have a given hash; we will later ensure only up to a single match can be returned in a well-formed blockchain.

```
match : Blockchain → HashId → Set⟨ TimedTx ⟩
match []                                     _    = ∅
match (∃tx@((_ , _ , tx) at _) ∷ txs) tx♯ =
  if tx ♯ == tx♯ then
    singleton ∃tx ∪ match txs tx♯
  else
    match txs tx♯
```

(Set⟨ A ⟩ denotes finite sets consisting of elements drawn from some type A with decidable equality; the details of their implementation do not matter here.)

Another ubiquitous operation on blockchains is computing the *fringe* of the transaction graph, i.e. the set of all unspent transaction outputs (hence the name 'UTxO').

```
UTXOₜₓ : ∃Tx → Set⟨ TxInput ⟩
UTXOₜₓ (_ , o , tx) = fromList $ (tx ♯) at_ <$> upTo o

STXOₜₓ : ∃Tx → Set⟨ TxInput ⟩
STXOₜₓ (_ , _ , tx) = tx .inputs ∙toList ∙fromList

UTXO : Blockchain → Set⟨ TxInput ⟩
UTXO = λ where
  []                  → ∅
  (tx at _ ∷ txs) → UTXO txs ─ STXOₜₓ tx
                      ∪ UTXOₜₓ tx
```

It is worth noting that this set-theoretic definition greatly diverges from the original paper, where a more indirect, logical but not immediately decidable formulation of *unspent-ness* is given:

```
module _ (b : Blockchain) (i : Index b) (let (_ , o , Tⱼ) at tⱼ = b ‼ i) where
  Unspent : Pred₀ (Fin o)
```

```
Unspent j =
  ∀ (i′ : Index b) → i′ ≤ i →
    let (i′ , _ , Tⱼ′) at tⱼ′ = b ‼ i′ in
    ∀ (j′ : Fin i′) →
      Tⱼ , j , tⱼ ↛ Tⱼ′ , j′ , tⱼ′
```

However, we opt for the set-theoretic formulation above which has proven easier to work with and has become standard in subsequent meta-theoretical studies of the UTxO model [Zah18; Cha+20a; Zah20; Cha+20b; Cha+20c].

We can finally express the exact conditions under which a transaction can be considered consistent with an existing blockchain at a given time:

```
record _▷_,_ (txs : Blockchain) (tx : Tx i o) (t : Time) : Type where
  field
    inputsUnique :
      Unique $ tx .inputs ∙toList

    singleMatch : ∀ (j : Fin i) →
      let
        (tx♯ at _) = tx ‼ⁱ j
      in
        ∃[ tx′ ] (match txs tx♯ ≡ singleton tx′)

    noOutOfBounds : ∀ (j : Fin i) →
      let
        (_ at oⱼ) = tx ‼ⁱ j
        (((_ , o , _) at _) , _) = singleMatch j
      in
        oⱼ < o

  private
    getI : ∀ (j : Fin i) → let i′ , o′ , _ = singleMatch j .proj₁ .transaction in
      Tx i′ o′ × Fin o′ × Time × Value
    getI j =
      let
        (_ at oⱼ) = tx ‼ⁱ j
        (((_ , o , Tⱼ) at tⱼ) , _) = singleMatch j
        oⱼ = F.fromℕ< {m = oⱼ} {n = o} (noOutOfBounds j)
        vⱼ = (Tⱼ ‼ᵒ oⱼ) ∙value
      in
        Tⱼ , oⱼ , tⱼ , vⱼ

  field
    inputs∈UTXO : ∀ (j : Fin i) →
      tx ‼ⁱ j ∈ˢ UTXO txs
```

```
inputsRedeemable : ∀ (j : Fin i) → let Tⱼ , oⱼ , tⱼ , vⱼ = getI j in
  Tⱼ , oⱼ , tⱼ ⇝[ vⱼ ] tx , j , t

valuesPreserved :
 let
  ins  = V.tabulate λ j → let Tⱼ , oⱼ , _ = getI j in (Tⱼ ‼ᵒ oⱼ) •value
  outs = _•value <$> tx .outputs
 in
  V.sum ins ≥ V.sum outs

laterTime :
 t ≥ latestTime txs
```

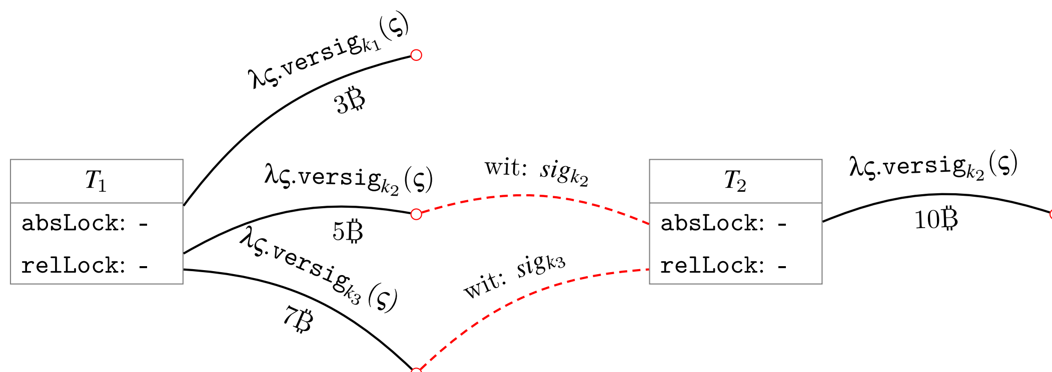
Setting aside the well-formedness conditions that rule out nonsensical scenarios and derive the helper `getI` total function that resolves references, consistency makes sure that no *double spending* occurs, outputs are redeemed properly w.r.t. to the semantics of transactions, the overall value flowing through a transaction is preserved ("you get what you put in"), and time is advanced as expected.

A *consistent blockchain* is one that has been inductively defined by appending a transaction one-by-one, but also providing a proof of consistency for each such inductive step.

```
data ConsistentBlockchain : Blockchain → Type where
 ∎      : ConsistentBlockchain []
 _⊕_∶-_ : ConsistentBlockchain txs
         → (tx : Tx i o)
         → txs ▷ tx , t
         → ConsistentBlockchain ((-, -, tx) at t ∷ txs)
```

Example 5. *UTxO calculation* *(Example 5 of* [Atz+18b]*)*

Consider the two transactions T_1 and T_2 depicted below:



Since our formulation of the UTxO set is *constructive*, we can *calculate* the UTxO set of the blockchain formed by conjoining these two transactions. Alas this is not entirely true: there is a small caveat stemming from the fact that we have postulated our hash functions, rendering the UTxO calculation impossible to compute in full. Fortunately, Agda provides a means to extend the evaluation of the core type theory with additional computational reductions [Coc19] in the form of *rewrite rules*,[5] a technique more broadly known as *Rewriting Type Theory (RTT)* [CTW21].

```
T₁ : Tx 0 3
T₁ = record
  { inputs  = []
  ; wit     = []
  ; relLock = []
  ; outputs = [ (1 , 3 ₿ locked-by ƛ versig [ k₁ ] [ 0F ])
              ⨾ (1 , 5 ₿ locked-by ƛ versig [ k₂ ] [ 0F ])
              ⨾ (1 , 7 ₿ locked-by ƛ versig [ k₃ ] [ 0F ]) ]
  ; absLock = 0 }

T₂ : Tx 2 1
T₂ = sig⋆ [ [ k₂ ] ⨾ [ k₃ ] ] record
  { inputs  = [ (T₁ ♯) at 1 ⨾ (T₁ ♯) at 2 ]
  ; wit     = wit⊥
  ; relLock = [ 0           ⨾ 0           ]
  ; outputs = [ 1 , 10 ₿ locked-by ƛ versig [ k₂ ] [ 0F ] ]
  ; absLock = t₂ }

T₃ : Tx 1 1
T₃ = sig⋆ [ [ k₂ ] ] record
  { inputs  = [ (T₁ ♯) at 1 ]
  ; wit     = wit⊥
  ; relLock = [ 0           ]
  ; outputs = [ 1 , 5 ₿ locked-by ƛ versig [ k₂ ] [ 0F ] ]
  ; absLock = t₃ }

B : Blockchain
B = [ (-, -, T₂) at t₂ ⨾ (-, -, T₁) at 0 ]

postulate
  eq1 : (T₁ ♯) ≡ + 1
  eq2 : (T₂ ♯) ≡ + 2
  eq3 : (T₃ ♯) ≡ + 3
{-# REWRITE eq1 #-}
{-# REWRITE eq2 #-}
{-# REWRITE eq3 #-}
```

[5] https://agda.readthedocs.io/en/v2.6.3/language/rewriting.html

```
b = List TxInput ∋ [ (T₁ ♯) at 0 ; (T₂ ♯) at 0 ]

_ : UTXO B ≈ fromList b
_ = toWitness {Q = UTXO B ≈?ˢ fromList b} tt

B₀ : Blockchain
B₀ = [ (-, -, T₁) at 0 ]

b₀ = List TxInput ∋ [ (T₁ ♯) at 0 ; (T₁ ♯) at 1 ; (T₁ ♯) at 2 ]

_ : UTXO B₀ ≈ fromList b₀
_ = toWitness {Q = UTXO B₀ ≈?ˢ fromList b₀} tt
```

Example 6. *Consistent update* *(Example 6 of [Atz+18b])*

Continuing on the same example transactions from Example 5, we can easily prove that appending T_2 to an blockchain consisting only of T_1 is in fact a consistent update:

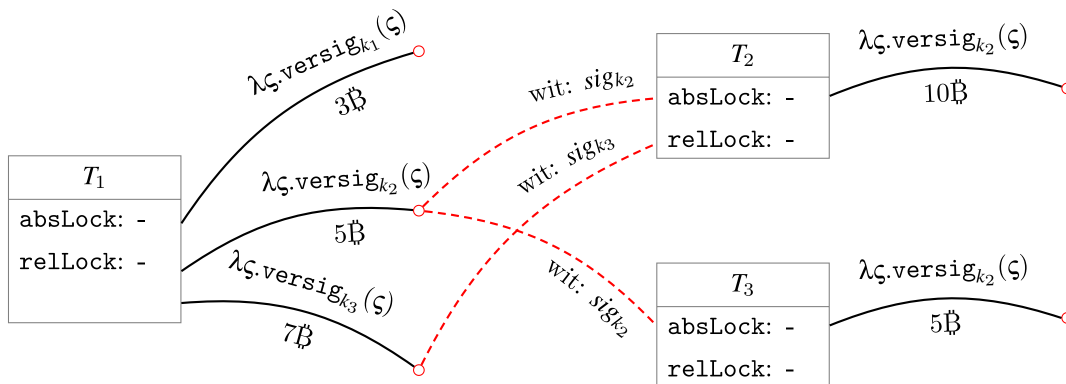
```
_ : B₀ ▷ T₂ , t₂
_ = record
    { inputsUnique = auto
    ; singleMatch = λ where
        0F → -, refl
        1F → -, refl
    ; noOutOfBounds = λ where
        0F → m≤m+n _ 1
        1F → ≤-refl
    ; inputs∈UTXO = λ where
        0F → auto
        1F → auto
    ; inputsRedeemable = λ where
        0F → record
          { input~output      = refl
          ; scriptValidates   = ver⋆sig≡ T₂ 0F
          ; value≡            = refl
          ; satisfiesAbsLock = ≤-refl
          ; satisfiesRelLock = z≤n , z≤n
          }
        1F → record
          { input~output      = refl
          ; scriptValidates   = ver⋆sig≡ T₂ 1F
          ; value≡            = refl
          ; satisfiesAbsLock = ≤-refl
          ; satisfiesRelLock = z≤n , z≤n
          }
    ; valuesPreserved = m≤m+n _ (2 ₿)
```

```
  ; laterTime = z≤n
  }
```

(Some proofs are automatically discharged by appealing to the mystical `auto`; this is a *typeclass*-based inference procedure for retrieving a decision procedure for any decidable type, which will be described in more detail in Appendix A.4.)

Example 7. *Inconsistent update* *(Example 7 of* [Atz+18b]*)*

If we now try to extend the blockchain of Example 5 with another transaction T_3 that again spends the second output of T_1 we arrive at the phenomenon of *double spending*, observed visually below in the transaction graph by the conflicting output having two red outgoing edges.



Thankfully, our mechanised model allows us to verify that this is not a consistent update:

```
_ : ¬ (B ▷ T₃ , t₃)
_ = λ where record {inputs∈UTXO = p} → contradict (p 0F)
```

Chapter 4

Source Language: BitML

Before we attempt to formulate the correctness of the BitML compiler, we first need to define the semantics of the source language of the compilation, namely the high-level model of BitML smart contracts.

Our mechanisation closely follows the operational semantics of BitML contracts introduced in the first half of the BitML paper [BZ18]:

> Massimo Bartoletti and Roberto Zunino. "BitML: a calculus for Bitcoin smart contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 83–100

Examples have also been drawn from the following papers by Bartoletti *et al.*:

- Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. "Fun with Bitcoin smart contracts". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 432–449
- Nicola Atzei, Massimo Bartoletti, Stefano Lande, Nobuko Yoshida, and Roberto Zunino. "Developing secure bitcoin contracts with BitML". in: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo. ACM, 2019, pp. 1124–1128. DOI: 10.1145/3338906.3341173. URL: https://doi.org/10.1145/3338906.3341173
- Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, and Roberto Zunino. "SoK: Unraveling Bitcoin Smart Contracts". In: *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Lujo Bauer and Ralf Küsters. Vol. 10804. Lecture Notes in Computer Science. Springer, 2018, pp. 217–242. DOI: 10.1007/978-3-319-89722-6_9. URL: https://doi.org/10.1007/978-3-319-89722-6_9

This chapter can thus be read as a mechanised presentation of the same material in literate Agda form. In particular, the formal model mechanised here is described in §4 and §A.2 of the original paper; we will provide further such correspondences for each subsection.

The entirety of the Agda development is publicly accessible in HTML format here:

https://omelkonian.github.io/formal-bitml/

Chapter overview. After setting up with a few basic definitions in §4.1, we will define the terms of the BitML calculus in §4.2, followed by some demonstrative examples of BitML contracts in §4.6.

The behaviour of such contracts will be studied under an operational semantics in §4.7, presented as a labelled transition system between *configurations*. Two distinct transitions, representing possible executions, will be illustrated for the *timed commitment* protocol in §4.8, which will act as our running example for the rest of the document.

4.1 Basic entities

First and foremost, smart contracts will hold and exchange some amount of monetary value, which we model by a natural number representing the amount in Satoshis, much like we did for the Bitcoin formalisation of Chapter 3.

```
Value  = ℕ
Values = List Value
```

Contracts will also handle two types of names: identifiers (`Id`) that refer to deposits holding some monetary `Value` and secrets (`Secret`) that are strings to be revealed during the execution of a contract.

```
Secret  = String
Secrets = List Secret

Id  = String
Ids = List String

Name  = Secret ⊎ Id
Names = List Name
```

We once again employ a discrete model of time using natural numbers:

```
Time = ℕ
```

Following Bitcoin's use of Unix time, these represent number of seconds since 1/1/1970.

4.2 Expressions (§4,Fig.1 of [BZ18])

Arithmetic expressions consist of constants, adding and multiplying, as well as calculating the length of a secret.

```
data Arith : Type where
  `        : ℤ → Arith
  _`+_ _`-_ : Arith → Arith → Arith
  ∥_∥       : Secret → Arith
```

Predicates, *i.e.*, logical expressions, consist of constants, negation, conjunction, and arithmetic comparisons.

```
data Predicate : Type where
  `true     : Predicate
  `¬_       : Predicate → Predicate
  _`∧_      : Predicate → Predicate → Predicate
  _`=_ _`<_ : Arith → Arith → Predicate
```

Other usual operations are derived from the primitive ones in the obvious way, so we will omit them to save space. Just to see one example, logical disjunction can be derived from primitive negation and conjunction:

```
p `∨ q = `¬ (`¬ p `∧ `¬ q)
```

Here is an example predicate that should hold, since we are checking equality between lengths of identical secrets and values of equivalent arithmetic expressions.

```
_ =  Predicate
  ∋  ∥ "change_me" ∥ `= ∥ "change_me" ∥
  `∧ ` 5ℤ            `= ` 3ℤ `+ ` 2ℤ
```

We still cannot verify that this evaluates to `` `true ``; we will define the denotational semantics of these expressions later in Section 4.7.4.

Notation

Unit tests. Whenever we want to write a *unit test* we use a definition with the "don't care" identifier `_`. Furthermore, if we do not want to repeat the identifier for both the type and definition, we write it more succinctly like above using the `∋` type annotation mechanism from the standard library.

4.3 Contract preconditions (§4,Def.1 of [BZ18])

The rest of the development is parameterised over an abstract set of *participants* that admits decidable equality and has a non-empty subset of *honest participants*.

```
module ...
  (Participant : Type)
  ⦃ _ : DecEq Participant ⦄
  (Honest : List⁺ Participant)
  where
```

Contract preconditions express certain condition which have to be met before the underlying contract can been activated (henceforth called 'stipulation').

```
data Precondition : Type where
  _:?_at_ _:!_at_ : Participant → Value → Id → Precondition
  _:secret_       : Participant → Secret → Precondition
  _|_             : Precondition → Precondition → Precondition
```

The first two constructs refer to participants to hold `Value` balances in a *deposit* with identifier `Id`:

- *volatile deposits*, denoted with `:?` , are simply pre-authorised by the participant and will be requested to actually get spent midway during the execution of the contract;
- *persistent deposits*, denoted with `:!` , should be available at the time of stipulation and are spent immediately.

We will later need to refer to these deposits individually, so we define a useful alias type for the necessary information contained within each *deposit reference*:

```
DepositRef = Participant × Value × Id
```

We might also need to specify whether the type of the referenced deposit is volatile or persistent, which we capture with an additional alias for *typed deposit references*:

```
data DepositType : Type where
  volatile persistent : DepositType

TDepositRef = DepositType × DepositRef
```

Using the `:secret` construct, participants can also commit to a *random nonce* before stipulation. Contract execution can then be conditional on whether the participant reveals the committed secret.

Finally, multiple preconditions can be composed together using `|`, requiring that all of them hold before stipulation.

4.4 Contracts (§4,Def.2 & Fig.1 of [BZ18])

Contracts will be mutually defined by the following types:

```
data Branch : Type
Contract    = List Branch
VContracts  = List (Value × Contract)
```

A BitML contract (`Contract`) is a choice amongst branches (`Branch`); a list of contracts associated to their value (`VContracts`) is needed for the *splitting* construct.

Contract branches typically have a continuation contract that resumes execution after the atomic operation prescribed by the branch finishes successfully.

```
data Branch where
  put_&reveal_if_⇒_ : Ids → Secrets → Predicate → Contract → Branch
  split             : VContracts → Branch
  withdraw          : Participant → Branch

  _:_               : Participant → Branch → Branch
  after_:_          : Time → Branch → Branch
```

The multi-purpose `put_&reveal_if_⇒_` construct serves the following functions:

- extending the contract balance by spending given **volatile deposits** (drawn from the contract's preconditions)
- checking that certain secrets (as specified in the contract's preconditions) have been **revealed**
- checking that the given **predicate** is true, otherwise aborting
- after all the above, continuing execution as the given sub-contract

The splitting construct `split` distributes the contract's balance amongst several other contracts and continues execution on each one of them concurrently.

The continuation-based recursion stops at the base case of `withdraw`, which transfers all remaining funds held in the contract to a given participant and terminates the contract execution.

Finally, contracts can be prefixed with two types of *decorations* to restrict access to the continuation of the contract:

- `_:_` requires authorisation by a specific participant
- `after_:_` allows access only after a given time

Notation

Synonyms for `put`. For notational convenience, we define several variants of the put-reveal construct that utilise only part of the functionality inherent in `put`.

```
pattern put_&reveal_⇒_ xs as c = put xs &reveal as if `true ⇒ c
pattern put_⇒_ xs c            = put xs &reveal []          ⇒ c
pattern put_if_⇒_ xs p c       = put xs &reveal [] if p     ⇒ c
pattern reveal_if_⇒_ as p c    = put [] &reveal as if p     ⇒ c
pattern reveal_⇒_ as c         = put [] &reveal as          ⇒ c
```

4.5 Contract advertisements (§4,Def.3 of [BZ18])

A contract `C` is *advertised* along with its preconditions `G`, which we denote as `⟨ G ⟩ C`.

```
record Ad : Type where
  constructor ⟨_⟩_
  field G : Precondition
        C : Contract
```

Not all contract advertisements are valid however; there are certain conditions that have to be satisfied.

First, we need to make sure the contract itself is *well-formed* with respect to the monetary balances involved, *i.e.*, :

1. when a contract is `split` into several continuations, the respective values distributed amongst these should sum up to the current balance of the running contract
2. when a deposit is dynamically `put` into the contract, there should be a corresponding *volatile* deposit specified in the contract's preconditions

```
splitsOK : Precondition → Contract → Bool
splitsOK G C₀ = goᶜ C₀ (persistentValue G)
  where
    goᵈ : Branch → Value → Bool
    goᶜ : Contract → Value → Bool
    goᵛᶜ : VContracts → Bool

    goᵛᶜ [] = true
    goᵛᶜ ((v , cs) ∷ vcs) = goᶜ cs v ∧ goᵛᶜ vcs
```

```
goᶜ [] _ = true
goᶜ (c ∷ cs) v = goᵈ c v ∧ goᶜ cs v

goᵈ (put xs &reveal as if p ⇒ c) v =
  case sequenceM $ map (λ x → checkDeposit volatile x G) xs of λ where
    nothing → false
    (just vs) → goᶜ c (v + ∑ℕ vs)
goᵈ (split vcs)  v = (∑₁ vcs == v) ∧ goᵛᶜ vcs
goᵈ (after _ : c) v = goᵈ c v
goᵈ (_ : c)       v = goᵈ c v
goᵈ (withdraw _) _ = true
```

($\sum\mathbb{N}$ calculates the sum of a list of numbers and $\sum_1$ only considered the first components of a list of tuples.)

A (well-formed) contract advertisement is then considered valid *iff* all of the following conditions are met:

(i) the names appearing in the preconditions are distinct

(ii) all names occurring in the contract have a corresponding clause in the preconditions

(iii) all volatile deposits references by a `put` command are distinct, and the predicate used by the `if` part can only use names introduced by the `reveal` part

(iv) every participant involved in the contract has a corresponding *persistent deposit* in the contract's precondition (via `:!`)

```
module _ (ad : Ad) (let ⟨ G ⟩ C = ad) where
  record ValidAd : Type where
    field
      splits-OK  :
        T $ splitsOK G C

      names-uniq : -- (i)
        Unique $ names G

      names-⊆    : -- (ii)
        names C ⊆ names G

      names-put  : -- (iii)
        All (λ (xs , as , p) → Unique xs × secrets p ⊆ as) (putComponents C)

      parts-⊆    : -- (iv)
        participants G ++ participants C ⊆ persistentParticipants G
```

We have deliberately omitted the helper definitions for collecting sub-components such as `putComponents`, `names` and `participants`, as these are largely uninteresting; simply traverse a given structure (be it a contract, a configuration or what have you) collecting any values of the requested type into a list. Alas, defining type classes for each such association of a larger type and the type of elements being collected, while manually implementing the traversals each time, requires a lot of boilerplate code and is time-consuming to maintain. To remedy this, the Prelude provides a more streamlined way to achieve this using a single generic class of collection associations, as described in Appendix A.6.

It is quite straightforward to show that the validity of a contract advertisement is *decidable*, *i.e.*, we can decide whether a given advertisement is valid, by relying on the decidability of the primitive constructs (*e.g.*, for `Unique`, `⊆`, *etc.*) used in its formulation.

```
instance
 Dec-VAd : ValidAd ⁇¹
 Dec-VAd {x = ⟨ G ⟩ C} .dec
  with T? $ splitsOK G C
   |    unique? $ names G
   |    names C ⊆? names G
   |    all? (λ (xs , as , p) → unique? xs ×-dec secrets p ⊆? as) (putComponents C)
   |    participants G ++ participants C ⊆? persistentParticipants G
 ... | no ¬splits-OK | _ | _ | _ | _                = no $ ¬splits-OK  ∘ splits-OK
 ... | _ | no ¬names-uniq | _ | _ | _               = no $ ¬names-uniq ∘ names-uniq
 ... | _ | _ | no ¬names-⊆ | _ | _                  = no $ ¬names-⊆    ∘ names-⊆
 ... | _ | _ | _ | no ¬names-put | _                = no $ ¬names-put  ∘ names-put
 ... | _ | _ | _ | _ | no ¬parts-⊆                  = no $ ¬parts-⊆    ∘ parts-⊆
 ... | yes p₁ | yes p₂ | yes p₃ | yes p₄ | yes p₅ = yes λ where
  .splits-OK → p₁; .names-uniq → p₂; .names-⊆ → p₃; .names-put → p₄; .parts-⊆ → p₅
```

Actually, a clever use of the `Decidable` typeclass described in Appendix A.4 lets us omit the tedious repetition of the conditions we have already stated.

```
Dec-VAd′ : ValidAd ⁇¹
Dec-VAd′ .dec with dec | dec | dec | dec | dec
... | no ¬splits-OK | _ | _ | _ | _                = no $ ¬splits-OK  ∘ splits-OK
... | _ | no ¬names-uniq | _ | _ | _               = no $ ¬names-uniq ∘ names-uniq
... | _ | _ | no ¬names-⊆ | _ | _                  = no $ ¬names-⊆    ∘ names-⊆
... | _ | _ | _ | no ¬names-put | _                = no $ ¬names-put  ∘ names-put
... | _ | _ | _ | _ | no ¬parts-⊆                  = no $ ¬parts-⊆    ∘ parts-⊆
... | yes p₁ | yes p₂ | yes p₃ | yes p₄ | yes p₅ = yes λ where
 .splits-OK → p₁; .names-uniq → p₂; .names-⊆ → p₃; .names-put → p₄; .parts-⊆ → p₅
```

4.6 Example BitML contracts *(§2 & §A.1 of [BZ18])*

To clarify the constructs allowed in the syntax of BitML contracts, let us go through some intuitive examples.

Since we will also want to prove that our contract advertisements are valid, and do so automatically through the use of *proof by reflection* [VS12] allowed by the `auto` instance method (*c.f.*, Appendix A.4), we instantiate the abstract module parameters of our development with concrete ones, namely the case where there is only an honest participant `A` and dishonest ones `B` and `M`.

```
data Participant : Type where
  A B M : Participant
unquoteDecl DecEqₚ = DERIVE DecEq [ quote Participant , DecEqₚ ]

Honest : List⁺ Participant
Honest = A ∷ []

open import BitML.Contracts Participant Honest public
```

Additionally, variables that refer to secrets `a`/`b`, time slots `t`/`t′`, and values `v`/`fee`, are implicitly instantiated with some arbitrary concrete values so that they do not block computation when appealing to a decision procedure.

Notation

Contracts. Instead of using the built-in `[]`/`∷`/`++` notation for constructing lists, we introduce some *syntactic sugar* that is closer to the paper's syntax.

First, we recover the choice operator `⊕` to combine contract branches whose operands can be either a contract or a single branch. To accommodate this *ad-hoc polymorphism* [WB89], our Prelude provides *implicit coercion* to lists in the form of the `List?` typeclass and its `toL` method (see Appendix A.8 for more details).

```
_⊕_ : ⦃ List? Branch X ⦄ → ⦃ List? Branch Y ⦄ → X → Y → Contract
x ⊕ y = toL x ++ toL y
```

The same applies for the contract bodies of `split` clauses, where we also want to accept a single branch instead of a contract, but we change the paper's $\rightarrow$ to `⊸` in order to hint at the linear usage of resources.

```
_⊸_ : Value → X → ⦃ List? Branch X ⦄ → VContracts
v ⊸ c = [ v , toL c ]
```

Moreover, we choose `⊗` instead of the original $|$ to separate `split` clauses:

```
_⊗_ = Op₂ VContracts ∋ _++_
```

This emphasises the duality between `⊕`-branches and `⊗`-clauses:

- ⊕ is a summing operator; only one of the branches will eventually execute.
- ⊗ is a multiplicative operator; all of the clauses will be executed.

Last, we do the same trick to provide smarter variants of the `put` synonyms we defined in Section 4.4 that employ implicit list conversion; the continuation arrow ⇒ is replaced with the paper's dot-notation (`put [ x ] ⇒ [ b ]` becomes `put x • b`).

Pay or refund. As a first simple example, consider the case where a buyer `A` wants to acquire a product or service from seller `B` with the option of requesting a refund in exceptional cases. The contract preconditions should naturally require `A` to have sufficient funds to pay (*i.e.*, have a *persistent deposit*), and the contract body covers the two branches of whether `A` actually finalises the payment or else the seller approves a refund. This can be succinctly formulated as the following BitML contract advertisement:

```
PayOrRefund : Contract
PayOrRefund = A : withdraw B
            ⊕ B : withdraw A

PayOrRefundAd : Ad
PayOrRefundAd =
 ⟨ A :! 10 at "𝔸" | B :! 0 at "𝔹" ⟩
 PayOrRefund
```

(The requirement that `B` holds an empty deposit might seem superfluous, but remember that the validity conditions mandate that *all* involved participants have a persistent deposit, even if it holds no funds, before any contract can be stipulated.)

As promised, we appeal to the decision procedure of contract validity to automatically decide whether the advertisements we define are valid or not.

```
_ = Valid PayOrRefundAd ∋ auto
```

Phew! This one is!!

Escrow. But what about the case where the two involved parties (*i.e.*, the buyer `A` and the seller `B`) are in disagreement? In the previous contract, the funds would just stay locked up. The contract can be extended with a third *escrow* party `M`, who acts as a mediator to handle such disputes at a cost of 10%. This is expressed as two additional branches allowing the buyer or seller to initiate a dispute to be resolved by the escrow. Resolution then splits the funds in the following way: the mediator takes out the fee (2₿ out of 20₿ in this case) for themselves, and transfers the remaining funds (18₿) to either `A` or `B`.

```
Resolve : Value → Value → Branch
Resolve v v′ =
 split $ v  ⊸ withdraw M
```

```
      ⊗ v′ ⊸ M : withdraw A
           ⊕ M : withdraw B

Escrow : Ad
Escrow =
  ⟨ A :! 20 at "𝔸" ∣ B :! 0 at "𝔹" ∣ M :! 0 at "𝕄" ⟩
      PayOrRefund
  ⊕   A : Resolve 2 18
  ⊕   B : Resolve 2 18

_ = Valid Escrow ∋ auto
```

Escrow with put. A slight variation on the previous example consists of having the participants provide *volatile deposits* to pay the escrow fee of 2₿. In BitML, this can be expressed using the put construct.

```
EscrowPut : Ad
EscrowPut =
  ⟨ A :! 20 at "𝔸" ∣ B :! 0 at "𝔹" ∣ M :! 0 at "𝕄"
  ∣ A :? 1  at "x" ∣ B :? 1 at "y"
  ⟩
    PayOrRefund
  ⊕ after t : withdraw B
  ⊕ put "x" .
         put "y" . Resolve 2 20
     ⊕   after t′ : withdraw A

_ = Valid EscrowPut ∋ auto
```

The additional time constraints make sure that these volatile deposits are submitted before a certain time (t for A, t′ for B) so that the dispute can be resolved; if any of the participants does not provide the (volatile) deposit in time, the other participant can withdraw all the remaining funds.

Variable escrow. One final variation of the escrow example is to have a variable percentage of how much each participant is refunded. The contract is parameterised by a list of possible percentage values ζ that indicate the amount ζ * v that A will get as a refund, where B takes the rest (1 - ζ * v). We recover the previous escrow examples that only covered full refunds to only one of the two parties by allowing ζ to be either 0% or 100%. If we further allow ζ to have the value of 50%, the dispute resolution can refund both parties equally.

Setting aside the mediate fee and implementation details of floating point arithmetic, the main difference now is that the contract has an additional branch for each possible value of ζ.

```
IsPercentage : Pred₀ Float
IsPercentage f = (0.0 ≤ f) × (f ≤ 1.0)

module _ (v : Value) (Z : List Float) {_ : All IsPercentage Z} where

  VariableResolve : Float → Branch
  VariableResolve ζ =
    split $ ζ * v       ⊸ withdraw A
          ⊗ (1- ζ) * v ⊸ withdraw B

  VariableEscrow : Contract
  VariableEscrow = PayOrRefund
                 ⊕ map (λ ζ → M : VariableResolve ζ) Z
```

Intermediated payment. The previous examples used a *trusted* escrow service, but sometimes the mediator is an *untrusted* entity. For instance, if A wants to make a payment of v₿ to B that first has to be authorised by an untrusted mediator M for some fee, we can use the following contract:

```
IntermediatedPayment : Ad
IntermediatedPayment =
 ⟨ A :! fee + v at "𝔸"
 | B :! 0       at "𝔹"
 | M :! 0       at "𝕄"
 ⟩
   M : split ( fee ⊸ withdraw M
             ⊗ v   ⊸ withdraw B )
 ⊕ after t : withdraw A

_ = Valid IntermediatedPayment ∋ auto
```

The contract presupposes that A puts up a persistent deposit that holds both the payment and the mediator's fee. Once stipulated, there are two possible scenarios:

- either the mediator is honest and authorises the payment to B while taking the mediator fee for themselves,
- or the mediator does not act as expected within a time limit, so A exercises the option of getting a full refund, effectively cancelling the transaction to B without any monetary losses.

Timed Commitment. A more interesting example involves participants committing to and revealing secrets. In the following *timed commitment* protocol, A promises to reveal a secret a (committed before stipulation) within a time frame, otherwise pays a penalty of v₿ to B.

```
TC : Ad
TC =
 ⟨ A :! v at "x" | A :secret a
 | B :! 0 at "y"
 ⟩
   reveal a ∙ withdraw A
 ⊕ after t  : withdraw B
```

We will revisit this example when we look at the formal semantics for executing BitML contracts in Section 4.8.

Mutual timed commitment. A more involved variation of the timed commitment protocol lets both participants commit to a secret *mutually*:

- A still has to reveal their secret a before time t, otherwise B can withdraw A's deposit of v ₿ (in addition to their own deposit of another v ₿).
- B also has to reveal b before time t′, otherwise A can withdraw B's deposit of v ₿. (in addition to their own deposit of another v ₿).
- If both A and B both revealed their secret in a timely fashion, the split command refunds their deposits accordingly and the contract execution is considered successful.

```
MutualTC : Ad
MutualTC =
 ⟨ A :! v at "x" | A :secret a
 | B :! v at "y" | B :secret b
 ⟩
   reveal a ∙ reveal b ∙ split (v ⊸ withdraw A ⊗ v ⊸ withdraw B)
                       ⊕ after t′ : withdraw A
            ⊕ after t′ : withdraw A
 ⊕ after t : withdraw B

_ = Valid MutualTC ∋ auto
```

Timed commitment protocols form the basis of more elaborate contracts that manipulate monetary resources based on exchanged secrets, most typically in the form of *games*.

OddsEvens game. As a first example game that clearly builds on the mutual timed commitment primitive, consider two participants playing a game of luck: each commits

a secret of a certain length with 2B collateral, and also deposits 1B to gamble whether their respective secret lengths are equal (in which case A wins) or not (B wins).

```
OddsEvens : Ad
OddsEvens =
  ⟨ A :! 3 at "x" ∣ B :! 3 at "y"
  ∣ A :secret a   ∣ B :secret b
  ⟩
  [ split $ 2 ⊸ reveal b if ` 0ℤ `≤ ∥ b ∥ `≤ ` 1ℤ ∙ withdraw B
                ⊕ after t : withdraw A
          ⊗ 2 ⊸ reveal a ∙ withdraw A
                ⊕ after t : withdraw B
          ⊗ 2 ⊸ reveal [ a ; b ] if ∥ a ∥ `= ∥ b ∥ ∙ withdraw A
                ⊕ reveal [ a ; b ] if ∥ a ∥ `≠ ∥ b ∥ ∙ withdraw B ]

_ = Valid OddsEvens ∋ auto
```

The extra condition in the first clause where B's secret is revealed is there to guarantee *fairness*, *i.e.*, both participants having 50% probability of winning the game after both secrets have been revealed.

Zero-collateral lottery. A simplification of the OddEvens game can be achieved by removing the extra collateral for the secrets (2B from each participant) altogether and using the bets themselves as collateral (2B in total). However, the secrets have to now be revealed in sequence: B first reveals their secret (of appropriate length and within the time limit), A then reveals their own secret (within a later time limit), and finally the lengths are checked to determine the winner.

```
ZeroCollateralLottery : Ad
ZeroCollateralLottery =
  ⟨ A :! 1 at "x" ∣ A :secret a
  ∣ B :! 1 at "y" ∣ B :secret b
  ⟩
    reveal b if ` 0ℤ `≤ ∥ b ∥ `≤ ` 1ℤ ∙
        reveal [ a ; b ] if ∥ a ∥ `= ∥ b ∥ ∙ withdraw A
      ⊕ reveal [ a ; b ] if ∥ a ∥ `≠ ∥ b ∥ ∙ withdraw B
      ⊕ after t′ : withdraw B
  ⊕ after t : withdraw A

_ = Valid ZeroCollateralLottery ∋ auto
```

Rock-paper-scissors. As a final example, let's consider a more intuitive game of broader interest: `rock-paper-scissors`, where two players choose one of the available shapes by committing to a secret of the corresponding length (0 for `rock`, 1 for `paper`, 2 for

`scissors`). It should now be clear why we have put emphasis on the operations of committing and revealing secrets and the minimal *timed commitment* protocol, since secrets are the means to enabling *participant choice*.

```
RockPaperScissors : Ad
RockPaperScissors =
 ⟨ A :! 3 at "x" ∣ A :secret a
 ∣ B :! 3 at "y" ∣ B :secret b
 ⟩
 [ split $ 2 ⊸ reveal b if ` 0ℤ `≤ ∥ b ∥ `≤ ` 2ℤ ∙ withdraw B
                ⊕ after t : withdraw A
         ⊗ 2 ⊸ reveal a if ` 0ℤ `≤ ∥ a ∥ `≤ ` 2ℤ ∙ withdraw A
                ⊕ after t : withdraw B
         ⊗ 2 ⊸ reveal [ a ; b ] if w ∥ a ∥ ∥ b ∥  ∙ withdraw A
                ⊕ reveal [ a ; b ] if w ∥ b ∥ ∥ a ∥  ∙ withdraw B
                ⊕ reveal [ a ; b ] if ∥ a ∥ `= ∥ b ∥ ∙ split ( 1 ⊸ withdraw A
                                                               ⊗ 1 ⊸ withdraw B ) ]
 where
  w : Arith → Arith → Predicate
  w n m  =  (n `= ` 0ℤ `∧ m `= ` 2ℤ)
        `∨ (n `= ` 2ℤ `∧ m `= ` 1ℤ)
        `∨ (n `= ` 1ℤ `∧ m `= ` 0ℤ)

_ = Valid RockPaperScissors ∋ auto
```

Following the same structure as the previous *OddEvens* game, we first have both `A` and `B` committing to their secrets in turn, ensuring that no more than 2 characters are used. Then, as soon as both secrets have been revealed, the winner is chosen based on `w` encoding the standard rules: `rock` beats `scissors`; `scissors` cut through `paper`; `paper` covers `rock`.

4.7 Operational semantics *(§A.2,Def.6 of [BZ18])*

At the core of our operational semantics lies the notion of states, henceforth called "configurations", of the labelled transition system. The execution of a contract is modelled as a transition between such configurations, dictated by the actual structure of the contract and, whenever there is branching involved, by the decisions made by the participants cooperating in the contract.

4.7.1 Actions *(§A.2,Fig.2 of [BZ18])*

Before we define what form these transition configurations take, we need to first specify what kind of actions a participant can perform.

```
data Action : Type where
  #▷_         : Ad → Action
  _▷ˢ_        : Id → Ad → Action
  _▷_         : Id → Branch → Action
  _↔_▷⟨_,_⟩   : Id → Id → Participant → Value → Action
  _▷⟨_,_,_⟩   : Id → Participant → Value → Value → Action
  _▷ᵈ_        : Id → Participant → Action
  _,_▷ᵈˢ_     : (xs : Ids) → Index xs → Id → Action

Actions = List Action
```

These include:

- #▷ commit secrets to stipulate a contract advertisement
- ▷ˢ spend a (persistent) deposit to stipulate a contract advertisement
- ▷ pick a branch of the contract that is currently executing
- ↔▷⟨ ⟩ join two deposits
- ▷⟨ , , ⟩ divide a deposit in two
- ▷ᵈ donate a deposit to another participant
- ▷ᵈˢ destroy a deposit, authorised by all involved parties

Here are some example actions, given arbitrary participants `A` and `B`, a contract advertisement `ad` and a contract `C` consisting of at least one branch.

```
module _ (A B : Participant) (ad : Ad) d ds (let c = d ∷ ds) where
  _ = Action
    ∋: #▷ ad                       -- commit secret
     : "x" ▷ˢ ad                   -- spend
     : "x" ▷ (c ‼ 0F)              -- take branch
     : "x" ↔ "y" ▷⟨ A , 10 ⟩       -- join
     : "x" ▷⟨ A , 33 , 67 ⟩        -- divide
     : "x" ▷ᵈ B                    -- donate
     : [ "x₀" ⨾ "x₁" ] , 1F ▷ᵈˢ "y" -- destroy
     :∅
```

We have to insist on `c` having at least one branch, otherwise the third example action of taking the *first* branch would be ill-formed and thus not *well-typed* in our system. A similar luxury that we afford due to our use of dependent types is showcased in the 'destroy' action, where it is impossible to make an *out-of-bounds* error.

Notation

Unit tests. The notation for unit tests used above is a generalisation of the standard library's `∋` to annotate multiple terms of a single type. Computationally speaking, the first one is returned so that the result type matches but this will never be used anyway if it acts as a unit test. See Appendix A.1 for the actual definition.

4.7.2 Configurations *(§4,Def.4 & §A.2,Def.5 of [BZ18])*

A configuration is initially empty (`∅ᶜ`) and gradually augmented (via `|`) with additional information of *base* configurations.

```
data Cfg : Type where
  ∅ᶜ          : Cfg
  _|_         : Cfg → Cfg → Cfg

  `_          : Ad → Cfg
  ⟨_,_⟩at_    : Contract → Value → Id → Cfg
  ⟨_has_⟩at_  : Participant → Value → Id → Cfg
  _auth[_]    : Participant → Action → Cfg
  ⟨_:_♯_⟩     : Participant → Secret → Maybe ℕ → Cfg
  _:_♯_       : Participant → Secret → ℕ → Cfg
```

Base configurations consist of:

- `` ` `` contract advertisements, which advertise a (well-formed) valid contract along with its preconditions to the public
- `⟨,⟩at` active contracts, *i.e.*, advertisements that have been stipulated and are now executing with a specified identifier and a current monetary balance
- `⟨has⟩at` deposits redeemable by a given participant
- `auth[_]` authorizations issued by a participant to perform a certain action (see Section 4.7.1 for a description of all the possible actions)
- `⟨:♯⟩` secrets committed by a participant, possibly along with their length (`just n` if honest, `nothing` otherwise)
- `:♯` revealed secrets, along with their length

Although the primitive binary composition `|` is our only means to extend an existing configuration, we will typically compose a list of configurations at once using the following auxiliary definition for *n*-ary composition that is derived from the binary one:

```
||_ : List Cfg → Cfg
|| []       = ∅ᶜ
|| (Γ ∷ []) = Γ
|| (Γ ∷ Γs) = Γ ∣ || Γs
```

Furthermore, our labelled transition will also operate in a timed manner by considering *timed configurations*, *i.e.*, a configuration at a particular point in time.

```
record Cfgᵗ : Type where
  constructor _at_
  field cfg  : Cfg
        time : Time

∅ᵗ : Cfgᵗ
∅ᵗ = ∅ᶜ at 0
```

An alternative presentation of configurations distinguishes *base* configurations from *composite* ones, which is easier to manipulate and reason about for some helpful lemmas we will prove later on.

```
data BaseCfg : Type where
  ``_          : Ad → BaseCfg
  `⟨_,_⟩at_    : Contract → Value → Id → BaseCfg
  `⟨_has_⟩at_  : Participant → Value → Id → BaseCfg
  _`auth[_]    : Participant → Action → BaseCfg
  `⟨_:_♯_⟩     : Participant → Secret → Maybe ℕ → BaseCfg
  _`:_♯_       : Participant → Secret → ℕ → BaseCfg

Cfg′ = List BaseCfg
pattern `∅ᶜ = []
```

We will be able to seamlessly translate between the two representations whenever we see fit, via the *(implicit) coercion* mechanism provided by the Prelude (see Appendix A.7 for more details). Notably, when both directions have been declared one automatically gets an isomorphism, merely through *instance search*.[1]

```
instance
  BaseCfg↝Cfg : BaseCfg ↝ Cfg
  BaseCfg↝Cfg .to = λ where
    (`` ad)              → ` ad
    (`⟨ c , v ⟩at x)     → ⟨ c , v ⟩at x
    (`⟨ A has v ⟩at x)   → ⟨ A has v ⟩at x
    ( A `auth[ a ])      → A auth[ a ]
```

[1] https://agda.readthedocs.io/en/v2.6.3/language/instance-arguments.html#instance-resolution

```
  `⟨ A : s ♯ mn ⟩     → ⟨ A : s ♯ mn ⟩
  (A `: s ♯ n)        → A : s ♯ n

Cfg′⇝Cfg : Cfg′ ⇝ Cfg
Cfg′⇝Cfg .to = ||_ ∘ map to

Cfg⇝Cfg′ : Cfg ⇝ Cfg′
Cfg⇝Cfg′ .to = λ where
  ∅ᶜ                 → `∅ᶜ
  (` ad)             → [ `` ad ]
  (⟨ c , v ⟩at x)    → [ `⟨ c , v ⟩at x ]
  (⟨ A has v ⟩at x) → [ `⟨ A has v ⟩at x ]
  (A auth[ a ])      → [ A `auth[ a ] ]
  ⟨ A : s ♯ mn ⟩     → [ `⟨ A : s ♯ mn ⟩ ]
  (A : s ♯ n)        → [ A `: s ♯ n ]
  (l | r)            → to l ++ to r
```

4.7.2.1 Configurations as a commutative monoid.

Let us immediately put the alternative representation of `Cfg′` to use: since we do not really care about the order that configurations appear in concrete values, we formulate what it means for two configurations to be *equivalent* modulo the order of the atomic elements. Concretely:

- lists of base configurations (`BaseCfg`) are related using the list permutation relation `_↭_`;

```
instance
  Setoid-Cfg′ : ISetoid Cfg′
  Setoid-Cfg′ = record {_≈_ = _↭_}

  SetoidLaws-Cfg′ : SetoidLaws Cfg′
  SetoidLaws-Cfg′ .isEquivalence = record
    {IsEquivalence L.Perm.↭-isEquivalence}
```

- the relation is then lifted to the original representation (`Cfg`) via the aforementioned coercion mechanism;

```
Setoid-Cfg : ISetoid Cfg
Setoid-Cfg = record {_≈_ = _≈_ on to[ Cfg′ ]}

SetoidLaws-Cfg : SetoidLaws Cfg
SetoidLaws-Cfg .isEquivalence = record
  {IsEquivalence L.Perm.↭-isEquivalence}
```

- we finally lift further to timed configurations by the obvious pairing construction on equivalences.

```
Setoid-Cfgᵗ : ISetoid Cfgᵗ
Setoid-Cfgᵗ = record {_≈_ = λ where (Γ at t) (Γ′ at t′) → (t ≡ t′) × (Γ ≈ Γ′)}

SetoidLaws-Cfgᵗ : SetoidLaws Cfgᵗ
SetoidLaws-Cfgᵗ .isEquivalence = record
  { refl = refl , ↭-refl
  ; sym  = λ where (t≡ , Γ≈) → sym t≡ , ↭-sym Γ≈
  ; trans = λ where (t≡ , Γ≈) (≡t , ≈Γ) → trans t≡ ≡t , ↭-trans Γ≈ ≈Γ
  }
```

4.7.3 Labels *(§A.2,Fig.3 of [BZ18])*

The type of labels recorded by the labelled transition system is given by the following datatype, where each label corresponds to a single transition rule of the operational semantics — as defined later in 4.7.5.

```
data Label : Type where
  auth-join⦅_,_↔_⦆       : Participant → Id → Id → Label
  join⦅_↔_⦆              : Id → Id → Label

  auth-divide⦅_,_▷_,_⦆  : Participant → Id → Value → Value → Label
  divide⦅_▷_,_⦆         : Id → Value → Value → Label

  auth-donate⦅_,_▷ᵈ_⦆   : Participant → Id → Participant → Label
  donate⦅_▷ᵈ_⦆          : Id → Participant → Label

  auth-destroy⦅_,_,_⦆   : Participant → (xs : Ids) → Index xs → Label
  destroy⦅_⦆            : Ids → Label

  advertise⦅_⦆          : Ad → Label

  auth-commit⦅_,_,_⦆    : Participant → Ad → List (Secret × Maybe ℕ) → Label
  auth-init⦅_,_,_⦆      : Participant → Ad → Id → Label
  init⦅_,_⦆             : Precondition → Contract → Label

  split⦅_⦆              : Id → Label
  auth-rev⦅_,_⦆         : Participant → Secret → Label
  put⦅_,_,_⦆            : Ids → Secrets → Id → Label
  withdraw⦅_,_,_⦆       : Participant → Value → Id → Label

  auth-control⦅_,_▷_⦆   : Participant → Id → Branch → Label

  delay⦅_⦆              : Time → Label
```

```
Labels = List Label
```

(We adopt a more robust naming scheme than the one used in the original paper, where we prefix each label with the name of the corresponding transition rule rather than opting for shorthands that rely on the particular shape of the arguments.)

A useful operation on labels is to record whether a contract has successfully taken a step, in which case the old contract identifier of the *evolved contract* is returned by the (partial) function `cv`:

```
cv : Label → Maybe Id
cv = λ where
  put⦅ _ , _ , y ⦆      → just y
  withdraw⦅ _ , _ , y ⦆ → just y
  split⦅ y ⦆            → just y
  _                     → nothing
```

4.7.4 Semantics of predicates *(§A.2,Fig.5 of [BZ18])*

Given a configuration, we can now define the semantics of expressions: arithmetic expressions evaluate to an integer and logical expressions evaluation to a boolean.

```
⟦_⟧ᵃʳ_ : Arith → Cfg → Maybe ℤ
⟦ ∥ a ∥   ⟧ᵃʳ Γ = go Γ
  where
    go : Cfg → Maybe ℤ
    go = λ where
      (_ : a′ ♯ N) → if a == a′ then ⁅ (+ N) ⁆ else nothing
      (l ∣ r)      → go l <|> go r
      _            → nothing
⟦ ` x     ⟧ᵃʳ _ = ⁅ x ⁆
⟦ e `+ e′ ⟧ᵃʳ Γ = ⁅ ⟦ e ⟧ᵃʳ Γ + ⟦ e′ ⟧ᵃʳ Γ ⁆
⟦ e `- e′ ⟧ᵃʳ Γ = ⁅ ⟦ e ⟧ᵃʳ Γ - ⟦ e′ ⟧ᵃʳ Γ ⁆

⟦_⟧ᵖ_ : Predicate → Cfg → Maybe Bool
⟦ `true   ⟧ᵖ Γ = ⁅ true ⁆
⟦ e `∧ e′ ⟧ᵖ Γ = ⁅ ⟦ e ⟧ᵖ Γ ∧ ⟦ e′ ⟧ᵖ Γ ⁆
⟦ `¬ e    ⟧ᵖ Γ = ⁅ not (⟦ e ⟧ᵖ Γ) ⁆
⟦ e `= e′ ⟧ᵖ Γ = ⁅ ⟦ e ⟧ᵃʳ Γ == ⟦ e′ ⟧ᵃʳ Γ ⁆
⟦ e `< e′ ⟧ᵖ Γ = ⁅ ⟦ e ⟧ᵃʳ Γ <ᵇ ⟦ e′ ⟧ᵃʳ Γ ⁆
```

Note that the semantics might fail when evaluating the length of a secret which has no corresponding configuration that reveals it. However, a predicate's semantics will

only be used in the [C-PutRev] rule in Section 4.7.5, where the configuration is always guaranteed to reveal these secrets by the rule's hypotheses, hence the predicate semantics are well-defined *by construction* in this case.

4.7.5 Inference rules *(§A.2,Fig3-5 of [BZ18])*

We can finally formulate the reduction rules for BitML contracts, classified into four categories:

1. rules for handling deposits
2. rules for activating contracts
3. rules for executing commands of active contracts
4. rules for handling time

These will be represented as constructors of an inductively-defined labelled relation; the first three categories will be relating *untimed* configurations (untimed layer), and the last category will be relating *timed* configurations (timed layer).

```
data _—[_]→_ : Cfg → Label → Cfg → Type where
```

Notation

Rule syntax. We will utilise *syntactic sugar* for inference rules, described in Appendix A.2, where function arrows have been replaced by horizontal lines (——) and bullet points (•) to resemble derivation rules in traditional form.

4.7.5.1 Handling deposits *(§A.2. Fig.3 of [BZ18])*

Most rules come in pairs: the first rule authorises a particular action, and then the second rules realises the actual effects of the authorised action. (This is not always a one-to-one correspondence, since there may be multiple participants which need to authorise a single action.)

This is precisely what happens to join a deposit, *i.e.,* merge two distinct deposits of the same participant to a single deposit holding the sum of the originals.

```
[DEP-AuthJoin] :
  ────────────────────────────────────────────────
  ⟨ A has v ⟩at x | ⟨ A has v′ ⟩at y | Γ
    —[ auth-join⦅ A , x ↔ y ⦆ ]→
```

```
⟨ A has v ⟩at x ∣ ⟨ A has v′ ⟩at y ∣ A auth[ x ↔ y ▷⟨ A , v + v′ ⟩ ] ∣ Γ
```

```
[DEP-Join] :

  z ∉ x ∷ y ∷ ids Γ
  ─────────────────────────────────────────────────────────────────────
  ⟨ A has v ⟩at x ∣ ⟨ A has v′ ⟩at y ∣ A auth[ x ↔ y ▷⟨ A , v + v′ ⟩ ] ∣ Γ
    —[ join⟪ x ↔ y ⟫ ]→
  ⟨ A has (v + v′) ⟩at z ∣ Γ
```

Notice how we focus on the relevant parts of the configurations *only*, by always placing them on the left of the composition. For the authorisation rule `[DEP-AuthJoin]`, we requite the *source* configuration to contain two distinct deposits owned by `A` and augment the configuration with the authorisation of `A` to join them in the *target*. For the execution rule `[DEP-Join]`, we require the source to contain the two deposits *and* the authorisation, all of which are replaced by a single merged deposit in the target.

Do not fret about the rules not applying for configurations that might not have the base configurations of interest at the "head"; when we later take the closure *up to setoid equivalence* (Section 4.7.5.5) we will be able to permute the configurations as needed, hence be able to pick out the relevant parts from the middle of a configuration. Also notice the *freshness* side-condition in the hypothesis of `[DEP-Join]`, assigning a *fresh* identifier for the newly created deposit at `z`.

A similar pattern plays out for doing the inverse action on deposits: dividing into two halves.

```
[DEP-AuthDivide] :

  ───────────────────────────────────────────────────────────
  ⟨ A has (v + v′) ⟩at x ∣ Γ
    —[ auth-divide⟪ A , x ▷ v , v′ ⟫ ]→
  ⟨ A has (v + v′) ⟩at x ∣ A auth[ x ▷⟨ A , v , v′ ⟩ ] ∣ Γ
```

```
[DEP-Divide] :

  All (_∉ x ∷ ids Γ) [ y ⨾ y′ ]
  ───────────────────────────────────────────────────────────
  ⟨ A has (v + v′) ⟩at x ∣ A auth[ x ▷⟨ A , v , v′ ⟩ ] ∣ Γ
    —[ divide⟪ x ▷ v , v′ ⟫ ]→
  ⟨ A has v ⟩at y ∣ ⟨ A has v′ ⟩at y′ ∣ Γ
```

(Both identifiers for the new deposits should be fresh.)

A participant might (authorise and) donate one of their deposit to another participant; the pattern should be familiar by now.

```
[DEP-AuthDonate] :

  ──────────────────────────────────
  ⟨ A has v ⟩at x | Γ
    —[ auth-donate⟦ A , x ▷ᵈ B ⟧ ]→
  ⟨ A has v ⟩at x | A auth[ x ▷ᵈ B ] | Γ

[DEP-Donate] :

  y ∉ x ∷ ids Γ
  ──────────────────────────────────
  ⟨ A has v ⟩at x | A auth[ x ▷ᵈ B ] | Γ
    —[ donate⟦ x ▷ᵈ B ⟧ ]→
  ⟨ B has v ⟩at y | Γ
```

Destroying a deposit cannot be done atomically like joining, dividing and donating. Rather, a set of n deposits is destroyed simultaneously, thereby requiring n authorisations from the involved participants/owners prior to executing the action that eventually destroys these deposits.

```
[DEP-AuthDestroy] :
  ∀ {ds : DepositRefs} {j : Index ds} (let xs = map select₃ ds) →

  let Aⱼ = (ds ‼ j) .proj₁
      j′ = ‼-map {xs = ds} j
      Δ = || map (uncurry₃ ⟨_has_⟩at_) ds
  in

  y ∉ ids Γ
  ──────────────────────────────────────────────────────
  Δ | Γ
    —[ auth-destroy⟦ Aⱼ , xs , j′ ⟧ ]→
  Δ | Aⱼ auth[ xs , j′ ▷ᵈˢ y ] | Γ

[DEP-Destroy] :
  ∀ {ds : DepositRefs} (let xs = map select₃ ds) →

  let Δ = || mapWithIndex ds λ (i , Aᵢ , vᵢ , xᵢ) →
            ⟨ Aᵢ has vᵢ ⟩at xᵢ | Aᵢ auth[ xs , ‼-map {xs = ds} i ▷ᵈˢ y ]
  in
  ──────────────────────────────────────────────────────────────────────
  Δ | Γ
    —[ destroy⟦ xs ⟧ ]→
  Γ
```

Concretely, `[DEP-Destroy]` makes sure each of the deposits to be destroyed actually exists in the configuration and moreover has a corresponding authorisation by the appropriate parties.

Notation

`let` **bindings.** As our rules become increasingly more complicated with many moving parts, we will need to grab hold of intermediate expressions. Some of them will be immediately computed from another (implicit) argument, in which case we use an argument-like `let` binding appearing right after the initial argument. Otherwise, we use a normal `let-in` binding that scopes over the whole rule (*i.e.*, both hypotheses and conclusions).

4.7.5.2 Activating contracts ('stipulation') *(§A.2. Fig.4 of [BZ18])*

Once all persistent deposits are in place, a contract can be advertised, provided that it is a *valid* advertisement (as defined in Section 4.5) and at least one of the involved participants is *honest*.

```
[C-Advertise] : let open ∣AD ad using (partG) in

  • ValidAd ad
  • Any (_∈ Hon) partG
  • deposits ad ⊆ deposits Γ
  ──────────────────────────────────────
  Γ —[ advertise⦅ ad ⦆ ]→ ` ad ∣ Γ
```

This has the effect of extending the current configuration with a contract advertisement `` ` ad``.

Notation

`let-open`**ing modules.** It is rather common to have multiple derived variables computed from a handful of given arguments, as in the case of the contract advertisement `ad` from which we derive the preconditions `G`, the contract `C` and the participants involved in the contract `partG`.

While we could manually introduce those via a `let` with multiple bindings, it quickly gets tedious to repeat these every time. Therefore, it is better practice to factorise this sort of repetitive computation into a reusable module (in the case of advertisements, `∣AD`); that way, we do not need to repeat the same `let` binding every time and just need to open the abbreviation module with a single `let-open`.

We will make another use of this pattern in Section 4.7.5.3 where we have a contract argument `c` focused on a particular branch `i` and a module `∣SELECT` will factorise the computation of the actual branch `d` and its version `d∗` where all authorisations have been removed (henceforth called 'stripped').

Then, participants can incrementally commit to secrets as promised in the contract's precondition, extending the configuration with the committed secret `⟨ A : a ♯ n ⟩` and recording that the current participant has completed their obligation for the contract's stipulation with `A auth[♯▷ ad ]`.

```
[C-AuthCommit] : let open ∣AD ad in
  ∀ {secrets : List (Secret × Maybe ℕ)} (let (as , ms) = unzip secrets) →

  let Δ = || map (uncurry ⟨ A :_♯_⟩) secrets in

  • as ≡ secretsOfᵖ A G
  • All (_∉ secretsOfᶜᶠ A Γ) as
  • (A ∈ Hon → All Is-just ms)
  ─────────────────────────────────────────────────
    ` ad ∣ Γ
      —[ auth-commit⟦ A , ad , secrets ⟧ ]→
    ` ad ∣ Γ ∣ Δ ∣ A auth[ ♯▷ ad ]
```

The additional hypotheses ensure that the current participant has not previously committed to the same secret and that honest participants commit to valid lengths (*i.e.*, containing an actual value).

Once all participants have committed their secrets, there is another round of incrementally spending each participant's persistent deposits, which is again recorded by an authorisation `A auth[ x ▷ˢ ad ]`.

```
[C-AuthInit] : let open ∣AD ad in

  • partG ⊆ committedParticipants ad Γ
  • (A , v , x) ∈ persistentDeposits G
  ─────────────────────────────────────────
    ` ad ∣ Γ
      —[ auth-init⟦ A , ad , x ⟧ ]→
    ` ad ∣ Γ ∣ A auth[ x ▷ˢ ad ]
```

Once all persistent deposits required by the contract's preconditions have been authorised for spending, `[C-Init]` finishes the job by actually consuming them as authorised by `[C-AuthInit]`, and replacing the contract advertisement `` ` ad`` with an active contract `⟨ C , v ⟩at x`.

```
[C-Init] : let open ∣AD ad in

  let toSpend = persistentDeposits G
      vs = map select₂ toSpend
      xs = map select₃ toSpend
  in
```

```
x ∉ xs ++ ids Γ
────────────────────────────────────────────────────────────
` ad
| Γ
| || map (λ (Aᵢ , vᵢ , xᵢ) → ⟨ Aᵢ has vᵢ ⟩at xᵢ | Aᵢ auth[ xᵢ ▷ˢ ad ]) toSpend
| || map _auth[ ♯▷ ad ] partG
  —[ init⟪ G , C ⟫ ]→
⟨ C , sum vs ⟩at x | Γ
```

Erratum

Correction on [BZ18]. In the original paper, `[C-Init]` also required an additional hypothesis to check that all participants have committed their secrets, but obviously this is already covered by the previous step `[C-AuthInit]`.

4.7.5.3 Executing active contracts

(§A.2. Fig.5 of [BZ18])

At this point, the active contract can start advancing via a separate reduction rule for each programming construct.

To reduce a `split` command that distributes the funds to n sub-contracts/continuations, the active contract is replaced by n active contracts in the configuration.

```
[C-Split] :
  ∀ {vcis : VIContracts} (let (vs , cs , ys) = unzip₃ vcis) →

  All (_∉ y ∷ ids Γ) ys
  ──────────────────────────────────────────
  ⟨ [ split (zip vs cs) ] , sum vs ⟩at y | Γ
    —[ split⟪ y ⟫ ]→
  || map (uncurry₃ $ flip ⟨_,_⟩at_) vcis | Γ
```

(We formulate the rule in such a way that the value is distributed correctly.)

Revealing a secret of an honest participant (thus always containing a numeric value) simply replaces a committed secret with a revealed one in the configuration.

```
[C-AuthRev] :

  ──────────────────────
  ⟨ A : a ♯ just n ⟩ | Γ
    —[ auth-rev⟪ A , a ⟫ ]→
  A : a ♯ n | Γ
```

Erratum

Correction on [BZ18]. In the original paper, [C-AuthRev] did not include the surrounding context Γ, rendering this rule impossible to apply within larger contexts.

Revealing these secrets is a prerequisite for executing a put command, along with the validity of the predicate in the if part and the existence of the volatile deposits xs.

```
[C-PutRev] :
 ∀ {ds : DepositRefs} (let (_ , vs , xs) = unzip₃ ds)
   {ss : List (Participant × Secret × ℕ)} (let (_ , as , _) = unzip₃ ss) →

 let Γ = || map (uncurry₃ ⟨_has_⟩at_) ds
     Δ = || map (uncurry₃ _:_♯_) ss
     ΔΓ′ = Δ | Γ′
 in

 • z ∉ y ∷ ids (Γ | ΔΓ′)
 • ⟦ p ⟧ᵖ Δ ≡ just true
   ──────────────────────────────────────────
   ⟨ [ put xs &reveal as if p ⇒ c ] , v ⟩at y | (Γ | ΔΓ′)
     —[ put⟪ xs , as , y ⟫ ]→
   ⟨ c , v + sum vs ⟩at z | ΔΓ′
```

To reduce the put command at the top we then replace it with its continuation c occurring after ⇒, while spending the volatile deposits xs and increasing its monetary value by the sum of these deposits.

The iterative execution of the contract comes to a halt with the withdraw command: the active contract is simply replaced by a new deposit holding the remaining funds of the contract.

```
[C-Withdraw] :

 x ∉ y ∷ ids Γ
 ──────────────────────────
 ⟨ [ withdraw A ] , v ⟩at y | Γ
   —[ withdraw⟪ A , v , y ⟫ ]→
 ⟨ A has v ⟩at x | Γ
```

The final pair of *untimed* rules take care of picking a branch of a contract, which might require certain participants to authorise it first. The authorisation pattern should be expected by now: first, an initial authorisation augments the current configuration with the information that a participant authorised a specific branch (A auth[x ▷ d]).

```
[C-AuthControl] :
 ∀ {i : Index c} (let open |SELECT c i) →
```

```
A ∈ authDecorations d
────────────────────────────────
⟨ c , v ⟩at x ∣ Γ
  —[ auth-control⟪ A , x ▷ d ⟫ ]→
⟨ c , v ⟩at x ∣ A auth[ x ▷ d ] ∣ Γ
```

(Recall that `|SELECT` binds the `i`-th branch `d` from the original contract `c`.)

Once all mandatory authorisations for a branch have been added to the running configuration, the corresponding contract branch is taken and a step is made according to the previous rules.

```
[C-Control] :
 ∀ {i : Index c} (let open |SELECT c i using (d; d∗)) →

  • ¬Null (authDecorations d) ⊎ (length c > 1)
  • Γ ≈ L
  • ⟨ [ d∗ ] , v ⟩at x ∣ L —[ α ]→ Γ′
  • cv α ≡ just x
  ────────────────────────────────────────────────────────────
  ⟨ c , v ⟩at x ∣ || map _auth[ x ▷ d ] (nub $ authDecorations d) ∣ Γ
    —[ α ]→
  Γ′
```

Given that we have to recursively call the same relation as a hypothesis, there is the unfortunate downside of having to permute the surrounding configuration manually ourselves using `≈`, which is normally handled automatically by the closure construction described in Section 4.7.5.5 and elaborated further in Appendix A.9. (The additional hypotheses make sure that the rule is fired on sensible cases only.)

4.7.5.4 Handling time *(§A.2. Fig.6 of [BZ18])*

It is high *time* we also covered the *timed* layer of BitML's reduction semantics: this now takes the form of a labelled relation between *timed* configurations.

```
data _—[_]→ₜ_ : Cfgᵗ → Label → Cfgᵗ → Type where

 [Delay] :

  δ > 0
  ────────────────────────────
  Γ at t —[ delay⟪ δ ⟫ ]→ₜ Γ at (t + δ)


 [Action] :
```

```
  • Γ —[ α ]→ Γ′
  • cv α ≡ nothing
  ────────────────────
  Γ at t —[ α ]→ₜ Γ′ at t

[Timeout] :
  ∀ {i : Index c} (let open |SELECT c i; As , ts = decorations d) →

  • Null As
  • All (_≤ t) ts
  • ⟨ [ d∗ ] , v ⟩at x | Γ —[ α ]→ Γ′
  • cv α ≡ just x
  ──────────────────────────────────────────
  (⟨ c , v ⟩at x | Γ) at t —[ α ]→ₜ Γ′ at t
```

[Delay] This rule makes it possible to proceed time by discrete intervals.

[Action] For labels that do not make progress on an active contract (as checked by the `cv` function defined in Section 4.7.3), we simply execute the untimed reduction semantics underneath the timed layer.

[Timeout] This is the only interesting timed rule, where we handle the advancement of an active contract. In order for this rule to fire, all the following conditions must be met:

- all authorisations (introduced with `A ⇛ ⋯`) should have already been handled by the untimed layer;
- all time constraints (introduced with `after t ⇛ ⋯`) should be satisfied, *i.e.*, the current time slot should be an upper bound on all such time *decorations*;
- removing all decorations and reducing the configuration in the untimed layer gives us the resulting state in the timed layer.

Erratum

Improvement on [BZ18]. In the original paper, the `[Timeout]` recursive call was using the decorated branch `d` instead of the stripped one `d∗` leading to an extraneous application of `[C-Control]`, which we have optimised away.

4.7.5.5 Closure of the reduction relation

Finally, we take the *reflexive* and *transitive* closure of the reduction relations (both untimed and timed versions) to enable reasoning about sequences of steps, *i.e., traces*, now indexed by a *list* of labels identifying such sequence. We will furthermore need a way to utilise the fact that the order of configurations conjoined using `_|_` does not matter, which lets us present the rules as in the original paper, *i.e.,* by having the sub-configurations of interest to the left and a remaining "context" to the right. Concretely, we enrich the inductive step of the reduction relation to also allow for converting between *equivalent* configurations, as defined in Section 4.7.2.1.

Instead of providing an ad-hoc implementation of this relation, we utilise a generic treatment of closures provided by the Prelude, which covers many different kinds of relations (*e.g.,* labelled or not, working *up to* setoid equivalence) and further defines boilerplate utility definitions (*e.g., left* versions of the step that focus on the *first* step of the trace rather than the *last*). We refer the reader to Appendix A.9 for more details on the exact inductive relations derived from the library.

```
open import Prelude.Closures
open UpToLabelledReflexiveTransitiveClosure _—[_]→_  public
open UpToLabelledReflexiveTransitiveClosure _—[_]→ₜ_ public
```

(To save space, we refrain from explicitly mentioning the renamed versions for relations/definitions associated to these closures; we simply adopt a naming convention where we use the double arrow ↠ for closures and refer to timed versions by appending a $_t$ suffix to the corresponding untimed construct.)

4.7.6 Decidability of inference rules

It turns out all of the rule hypotheses are *decidable* propositions, therefore admit a more convenient interface in the form of *smart constructors* that automatically discharge the proof obligations arising from these hypotheses, *e.g.,* to be readily used in examples with closed formulas as we will see in Section 4.8.

We simply need to repeat the constructors of the relation's inductive datatype, but now appealing to the decidability of its argument/hypothesis, namely that we can decide whether the corresponding proposition holds. The Prelude's infrastructure for working with decidable predicates gives us an effortless way to do just that: repeat the exact same but prefix each hypothesis *P* with an `auto:`; this triggers instance resolution to find the proof of decidability `P? : Dec P` associated to the current statement and then replace the hypothesis with `True P?`.

Naturally, rules with no hypotheses have no need for such a treatment, thus we only need to provide smart constructors for the rest of the rules.

As an example, we repeat the rule of [DEP-Join] with decidable hypotheses, where the above hypothesis using auto: is equivalent to p : True (z ∉? x ∷ y ∷ ids Γ). Given such a proof of validity, we can recover a proof of the original statement by using toWitness from the standard library.

```
DEP-Join :
  ∀ {p : auto: z ∉ x ∷ y ∷ ids Γ} →
  ────────────────────────────────────────────────────────────────
  ⟨ A has v ⟩at x ∣ ⟨ A has v′ ⟩at y ∣ A auth[ x ↔ y ▷⟨ A , v + v′ ⟩ ] ∣ Γ
    —[ join⟪ x ↔ y ⟫ ]→
  ⟨ A has (v + v′) ⟩at z ∣ Γ
DEP-Join {p = p} = [DEP-Join] (toWitness p)
```

The rest of the deposit-handling rules can be automated in a similar way:

```
DEP-Divide :
  ∀ {p : auto: All (_∉ x ∷ ids Γ) [ y ⨾ y′ ]} →
  ────────────────────────────────────────
  ⟨ A has (v + v′) ⟩at x ∣ A auth[ x ▷⟨ A , v , v′ ⟩ ] ∣ Γ
    —[ divide⟪ x ▷ v , v′ ⟫ ]→
  ⟨ A has v ⟩at y ∣ ⟨ A has v′ ⟩at y′ ∣ Γ
DEP-Divide {p = p} = [DEP-Divide] (toWitness p)

DEP-Donate :
  ∀ {p : auto: y ∉ x ∷ ids Γ} →
  ──────────────────────────────
  ⟨ A has v ⟩at x ∣ A auth[ x ▷ᵈ B ] ∣ Γ
    —[ donate⟪ x ▷ᵈ B ⟫ ]→
  ⟨ B has v ⟩at y ∣ Γ
DEP-Donate {p = p} = [DEP-Donate] (toWitness p)

DEP-AuthDestroy :
  ∀ {ds : DepositRefs} {j : Index ds} (let xs = map select₃ ds) →
  let Aⱼ = (ds ‼ j) .proj₁
      j′ = ‼-map {xs = ds} j
      Δ = || map (uncurry₃ ⟨_has_⟩at_) ds
  in
  ∀ {p : auto: y ∉ ids Γ} →
  ─────────────────────────────────
  Δ ∣ Γ
    —[ auth-destroy⟪ Aⱼ , xs , j′ ⟫ ]→
  Δ ∣ Aⱼ auth[ xs , j′ ▷ᵈˢ y ] ∣ Γ
DEP-AuthDestroy {p = p} = [DEP-AuthDestroy] (toWitness p)
```

The rest of the rules for both untimed and timed configurations follow the exact same pattern, so we omit them for the sake of brevity.

4.8 Example: the *timed commitment* protocol

Let us now see the rules in action by examining possible reductions of the *timed commitment* protocol TC introduced in Section 4.6.

We will once again instantiate the abstract module parameters of our development with concrete ones, namely the case where there is only an honest participant A and a dishonest one B. Furthermore, instead of using module parameters to treat example variables abstractly, we provide arbitrary concrete values so that our examples compute, *e.g.,* to decide some predicate.

As a reminder, the protocol consists of a participant A committing to a secret that they promise to reveal to another participant B before time t. A sets up a deposit of 1B as collateral; if they reveal the secret as promised they get it back, otherwise B is allowed to withdraw the collateral.

```
TC : Ad
TC = ⟨ A :! 1 at x ∣ A :secret a ∣ B :! 0 at y ⟩
       reveal a ⋯ withdraw A
     ⊕ after t : withdraw B
```

The first scenario is most concisely represented in the untimed layer:

```
TC-steps :
  ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y
    —[ advertise⦅ TC ⦆
     ∷ auth-commit⦅ A , TC , [ a , just N ] ⦆
     ∷ auth-commit⦅ B , TC , [] ⦆
     ∷ auth-init⦅ A , TC , x ⦆
     ∷ auth-init⦅ B , TC , y ⦆
     ∷ init⦅ TC .G , TC .C ⦆
     ∷ auth-rev⦅ A , a ⦆
     ∷ put⦅ [] , [ a ] , x₁ ⦆
     ∷ withdraw⦅ A , 1 , x₂ ⦆
     ∷ []
     ]↠
  ⟨ A has 1 ⟩at x₃ ∣ A : a ♯ N
```

The type already tells us the initial and final configuration, as well as the exact sequence of actions performed by the participants: the TC contract is advertised, A commits to a secret and spends the required deposit to stipulate the contract, then reveals the secret and successfully retrieves their deposit back.

To populate this type, one needs to use the reduction rules defined in Section 4.7.5; choosing the right ones is quite easy with the help of dependent types, as the labels already narrow the type checker to a single choice of constructor.

```
TC-steps =
 begin
  ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y
 —→⟨ C-Advertise {Γ = ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y} ⟩
  ` TC ∣ ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y
 —→⟨ C-AuthCommit {Γ = ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y} {secrets = [ a , just N ]} ⟩
  ` TC ∣ ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y ∣ ⟨ A : a ♯ just N ⟩ ∣ A auth[ ♯▷ TC ]
 —→⟨ C-AuthCommit {Γ = ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y ∣ ⟨ A : a ♯ just N ⟩
                   ∣ A auth[ ♯▷ TC ]} {secrets = []} ⟩
  ` TC ∣ ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y ∣ ⟨ A : a ♯ just N ⟩
      ∣ A auth[ ♯▷ TC ] ∣ B auth[ ♯▷ TC ]
 —→⟨ C-AuthInit {Γ = ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y ∣ ⟨ A : a ♯ just N ⟩
                  ∣ A auth[ ♯▷ TC ] ∣ B auth[ ♯▷ TC ]}
                {v = 1} ⟩
  ` TC ∣ ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y ∣ ⟨ A : a ♯ just N ⟩ ∣ A auth[ ♯▷ TC ]
      ∣ B auth[ ♯▷ TC ] ∣ A auth[ x ▷ˢ TC ]
 —→⟨ C-AuthInit {Γ = ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y ∣ ⟨ A : a ♯ just N ⟩
                  ∣ A auth[ ♯▷ TC ] ∣ B auth[ ♯▷ TC ] ∣ A auth[ x ▷ˢ TC ]}
                {v = 0} ⟩
  ` TC ∣ ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y ∣ ⟨ A : a ♯ just N ⟩ ∣ A auth[ ♯▷ TC ]
      ∣ B auth[ ♯▷ TC ] ∣ A auth[ x ▷ˢ TC ] ∣ B auth[ y ▷ˢ TC ]
 —→⟨ C-Init {x = x₁} {Γ = ⟨ A : a ♯ just N ⟩} ⟩
  ⟨ TC .C , 1 ⟩at x₁ ∣ ⟨ A : a ♯ just N ⟩
 —→⟨ [C-AuthRev] {n = N} {Γ = ⟨ TC .C , 1 ⟩at x₁} ⟩
  ⟨ TC .C , 1 ⟩at x₁ ∣ A : a ♯ N
 —→⟨ C-Control {c = TC .C}
                {Γ = A : a ♯ N}
                {Γ′ = ⟨ [ withdraw A ] , 1 ⟩at x₂ ∣ (A : a ♯ N ∣ ∅ᶜ)}
                {i = 0F}
  $ C-PutRev {ds = []} {ss = [ A , a , N ]} ⟩
  ⟨ [ withdraw A ] , 1 ⟩at x₂ ∣ A : a ♯ N
 —→⟨ C-Withdraw {x = x₃} {Γ = A : a ♯ N} ⟩
  ⟨ A has 1 ⟩at x₃ ∣ A : a ♯ N
 ∎
```

Notice how the use of the decidable smart constructors of Section 4.7.6 lets us proceed with the proof without having to manually prove the hypotheses of each rule. Alas, we still need to aid Agda's type inference by explicitly providing some of the implicit arguments, making our proofs more clunky than desired.

The second scenario, where A fails to keep their promise, has to be illustrated within the timed layer since the second branch has a timed guard.

```
TC-stepsₜ' :
  (⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y) at t
   —[ advertise⟪ TC ⟫
    ∷ auth-commit⟪ A , TC , [ a , just N ] ⟫
    ∷ auth-commit⟪ B , TC , [] ⟫
    ∷ auth-init⟪ A , TC , x ⟫
    ∷ auth-init⟪ B , TC , y ⟫
    ∷ init⟪ TC .G , TC .C ⟫
    ∷ delay⟪ 1 ⟫
    ∷ withdraw⟪ B , 1 , x₁ ⟫
    ∷ []
    ]↠ₜ
  (⟨ B has 1 ⟩at x₂ ∣ ⟨ A : a ♯ just N ⟩) at suc t
```

We perform the same exact actions as before to stipulate the contract, but now proceed execution on the second branch by inserting a time delay before making the final `withdraw` move.

```
TC-stepsₜ' =
  beginₜ
    (⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y) at t
  —→ₜ⟨ Act {t = t}
    $   C-Advertise {Γ = ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y} ⟩
    (` TC ∣ ⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y) at t

                  ⋮

—→ₜ⟨ Act {t = t}
  $   C-Init {x = x₁} {Γ = ⟨ A : a ♯ just N ⟩} ⟩
  (⟨ TC .C , 1 ⟩at x₁ ∣ ⟨ A : a ♯ just N ⟩) at t
—→ₜ⟨ Delay {Γ = ⟨ TC .C , 1 ⟩at x₁ ∣ ⟨ A : a ♯ just N ⟩} {t = t} ⟩
  (⟨ TC .C , 1 ⟩at x₁ ∣ ⟨ A : a ♯ just N ⟩) at suc t
—→ₜ⟨ Timeout {c = TC .C} {t = suc t} {v = 1} {i = 1F}
  $   C-Withdraw {x = x₂} {y = x₁} {Γ = ⟨ A : a ♯ just N ⟩} ⟩
  (⟨ B has 1 ⟩at x₂ ∣ ⟨ A : a ♯ just N ⟩) at suc t
∎ₜ
```

The derivation for this scenario looks very similar to the previous case, the differences being that it operates on the timed layer and therefore has to wrap an `[Act]` around all untimed rules, and the choice of branch is performed by the timed rule `[Timeout]` instead of the untimed `[C-Control]`.

Chapter 5

From Source To Target: The BitML Compiler

We are ready to define the translation of BitML contracts (as defined in Chapter 4) to Bitcoin transactions (as defined in Chapter 3). The translation will take the form of a compiler, taking as input BitML contract advertisements and outputting a set of Bitcoin transactions, each corresponding to a specific BitML programming construct appearing in the input contract.

Let us make clear that this does not yet cover the execution of BitML contracts, which would correspond to submitting (a subset of) the transactions generated by the BitML compiler to an existing blockchain in some specific order. We will return to the execution semantics when we later attempt to prove the compiler "correct" in some sense, in Chapter 6.

This chapter comprises a mechanisation of the corresponding material on the BitML compiler in the original paper, in particular §7 and Appendix A.5 with the definition of the compiler appearing in Fig.7:

> Massimo Bartoletti and Roberto Zunino. "BitML: a calculus for Bitcoin smart contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 83–100

We note that the authors have implemented an actual compiler in Racket,[1] along with useful tooling to develop BitML contracts [Atz+19] and verify interesting properties about them based on model checking [BZ19b]. The purpose of re-implementing it in a proof assistant rather than a common programming language is not to provide a replacement in yet another ecosystem, but rather to be able to formally reason about each model and the compilation between them, eventually establishing a *compilation correctness* result.

[1] https://bitml-lang.github.io/

Having said that, we conduct our reasoning in a *constructive* setting, which means that the compiler we define in this chapter will eventually be executable (if we are, of course, careful to not introduce any `postulate`s in our Agda code) and therefore an alternative choice for people that still want to compile their BitML programs but in a more rigorous, formalised fashion. In other words, once we also have a proof of compilation correctness, we will be able to provide an alternative **certified compiler**, which might be lacking in features but offers a much higher level of assurance. Due to our constructive approach, the proof of compilation correctness can readily be leveraged to give us a **certifying compiler** as well, *i.e.,* for a specific run of the compiler, the corresponding generated transactions are accompanied by a *proof certificate* telling us that the semantics of the input BitML contract and the output transactions somehow *coincide*.

The entirety of the Agda development is publicly accessible in HTML format here:

https://omelkonian.github.io/formal-bitml-to-bitcoin/

Chapter overview. Instead of defining the BitML compiler hastily in one go, we will first take some care to articulate an expressive type for it in §5.1, where we will precisely specify the type of arguments the compiler accepts as input (§5.1.2) and the output it generates (§5.1.3), *i.e.,* we will aim for an intrinsically-typed compiler.

Only then will we proceed to define the actual translation as a *well-founded* recursive computation in §5.2, and demonstrate how our specification is *executable* in §5.3, by showcasing the transactions generated by compiling the *timed commitment* protocol we introduced in Chapter 4.

5.1 Formulating the type of the compiler

Compilation is parameterised by several mappings that give us auxiliary information, either related to the contract advertisement `⟨ G ⟩ C` currently being compiled (where G is the precondition and C is the contract), or recording exchanged cryptographic key pairs between the participants involved in the contract precondition (`partG`).

- `txout`: maps each deposit name appearing in the precondition `G` to a transaction output on Bitcoin
- `sechash`: maps each secret in `G` to a committed hash
- `K`: gives the registered key pair of each participant in `partG` (used for all participants to sign the initial transaction)

- K²: returns a key pair for each pair of a subterm of C and a participant in partG (possibly used in subsequent transactions that require authorisation from some subset of the participants)

In a simply-typed setting we could just model these mappings with the regular function space and declare the type of the compiler to be a function from the aforementioned mappings and an input advertisement to a list of transactions:

```
  Ad
→ (sechash : Id → TxInput′)
→ (txout   : Secret → HashId)
→ (K       : Participant → KeyPair)
→ (K²      : Branch → Participant → KeyPair)
→ List ∃Tx
```

(Recall the formal definition ∃Tx of Bitcoin transactions from Section 3.4.)

But we can (and *should*) do much better in a dependently-typed language like Agda in the following respects:

1. Represent the mappings' finite domain more faithfully than simply using the unrestricted function arrow →; in fact all the functions we define and reason about in Agda must be *total* (*i.e.*, defined for all elements of the input domain), otherwise the underlying logic is rendered *unsound*.

2. Be more precise regarding the shape of the output transactions. The existential prefix of ∃Tx means that we are generating transactions with some unconstrained number of inputs and outputs, while these in fact closely follow the structure of the input contract. Why not use dependent types to reflect these dependencies as well?

5.1.1 A primer on mappings with a finite domain

To start with, let us examine how we can model mappings with a finite domain in a general setting.

Given some possibly infinite type A and a type family P indexed over A, we can form mappings from a finite collection of As as functions from proofs of membership in a List A to a result P, appropriately indexed by the element for which we currently hold a proof of membership.

```
_↦′_ : List A → (A → Type ℓ) → Type _
xs ↦′ P = ∀ {x} → x ∈ xs → P x
```

(Do not worry about the case where an element appears more than once, thus allowing for different values depending on the computational content of the membership proof; we will make sure only lists with distinct elements at use-site, rather than complicating these definitions further.)

In most cases, the result type will actually be the *constant* type family (*i.e.,* the return type is the same for all elements of our finite domain), therefore we will use the non-dependent version which we derive as a specialisation of the dependent case:

```
_↦_ : List A → Type ℓ → Type _
xs ↦ B = xs ↦′ const B
```

This provides a lightweight representation of mappings with a finite domain that is easy to work with and reason about.

Since we have lifted the input domain to the *type-level*, retrieving the domain itself is immediate, as well as computing the co-domain (otherwise known as *image*) by iterating the domain and applying our mapping:

```
dom : ∀ {xs : List A} → xs ↦′ P → List A
dom {xs = xs} _ = xs

codom : xs ↦ B → List B
codom = mapWith∈ _
```

One particularly helpful aspect is the reliance on list membership which comes with a lot of utilities and proven properties in the standard library, letting us succinctly derive our own properties about mappings. For instance, we can construct mapping inversion, weakening, permutation and sequential composition, all of which can be immediately derived from standard list operations and lemmas about membership.

```
codom-↦ : ∀ {xs : List A} → (f : xs ↦ B) → (codom f ↦ A)
codom-↦ {xs = x ∷ _} f = λ where
  (here _) → x
  (there x∈) → codom-↦ (f ∘ there) x∈

weaken-↦ : xs ↦′ P → ys ⊆ xs → ys ↦′ P
weaken-↦ f ys⊆xs = f ∘ ys⊆xs

permute-↦ : xs ↭ ys → xs ↦′ P → ys ↦′ P
permute-↦ xs↭ys xs↦ = xs↦ ∘ ∈-resp-↭ (↭-sym xs↭ys)

_++/↦_ : xs ↦′ P → ys ↦′ P → xs ++ ys ↦′ P
xs↦ ++/↦ ys↦ = ∈-++⁻ _ >≡> λ where
  (inj₁ x∈) → xs↦ x∈
  (inj₂ y∈) → ys↦ y∈

extend-↦ : zs ↭ xs ++ ys → xs ↦′ P → ys ↦′ P → zs ↦′ P
extend-↦ zs↭ xs↦ ys↦ = permute-↦ (↭-sym zs↭) (xs↦ ++/↦ ys↦)
```

5.1.2 Compilation parameters

Let us now apply the technique for finite mappings described in Section 5.1.1 to formulate the types of the compilation parameters.

The input domains will be drawn from *collections* of sub-components from larger structures, as introduced in Section 4.5 and elaborated upon in Appendix A.6. Specifically, we make use of the following new collections:

- `ids`: collects all deposit identifiers, derived from `names` by filtering out only the *right* summands (recall that `Name = Secret ⊎ Id`)
- `secrets`: returns all secret strings, taken from the *left* summands of `names`
- `subterms`: looks for all sub-branches in a larger structure, be it a branch, contract, list of `split` clauses, or advertisement, *e.g.,*

```
_ : subterms ( A : withdraw B
             ⊕ B : split ( v ⊸ withdraw A
                         ⊗ v ⊸ after t : withdraw B))
  ≡ [ A : withdraw B
    ; B : split (v ⊸ withdraw A ⊗ v ⊸ after t : withdraw B)
    ; withdraw A
    ; after t : withdraw B
    ]
_ = refl
```

First, consider the `txout` mapping from deposit names to Bitcoin transaction outputs. The type of such mappings will be a family indexed over elements of the type from which we collect the deposit names. Instead of fixing this underlying type to the specific case of preconditions G, we define such mappings generically for all types that contain names, as we will later find other uses of the same type of mappings for other structures (*e.g.,* execution traces in Chapter 6).

```
Txout : ⦃ X has Name ⦄ → X → Type
Txout x = ids x ↦ TxInput′
```

This definition takes the identifiers of the indexed element as the finite domain and maps each of them to (a reference to) a transaction output, consisting of a transaction and an index into its outputs, defined as `TxInput′` in Section 3.4.

A similar treatment applies to the `sechash` mapping from secrets to their corresponding hashes and `sechash` from secrets to committed hashes:

```
Sechash : ⦃ X has Name ⦄ → X → Type
Sechash x = secrets x ↦ HashId
```

Finally, we need to provide types for the mappings K and K² that correspond to the publicly exchanged key pairs between participants.

The K mapping will be used at the time of stipulation, where every participant will sign an initial transaction.

```
𝕂 : Precondition → Type
𝕂 g = participants g ↦ KeyPair
```

All the subsequent transactions will correspond to a subterm of the contract, and authorisation might again be required from any participant, thus the mapping structure is nested: for each subterm of contract c, give a 𝕂 mapping from the participants (appearing in the precondition) to a key pair.

```
𝕂²′ : Ad → Type
𝕂²′ (⟨ g ⟩ c) = subterms c ↦ 𝕂 g
```

We will generalise this to any type that contains advertisements, which now comprises a doubly-nested mapping with a dependent co-domain at the outer level.

```
𝕂² : ⦃ X has Ad ⦄ → X → Type
𝕂² x = advertisements x ↦′ 𝕂²′
```

At this point, we can precisely express the inputs of the compiler's type, although the return type still eludes us.

```
∀ {ad : Ad} → let ⟨ g ⟩ _ = ad in
  Valid ad
→ (sechash : Sechash g)
→ (txout   : Txout g)
→ (K       : 𝕂 g)
→ (K²      : 𝕂²′ ad)
→ {!!}
```

Also note the requirement that we only compile *valid* advertisements in the sense of Chapter 4.

5.1.3 The type of compilation results

As we mentioned already, the compiler generates one initial transaction that stipulates the contract, and then one transaction per subterm of the advertised contract. Furthermore, we know *a priori* the shape of these transactions (*i.e.*, their number of inputs and outputs), which we can reflect precisely at the result type of the BitML compiler through the power of dependent types.

The initial transaction will always have a single output, which we manifest at the type-level by fixing the output index o of the type of Bitcoin transactions Tx to 1

(recall that transactions of type `Tx i o` are indexed by the number of their inputs `i` and outputs `o`). The number of inputs, however, depends on the number of persistent deposits in the precondition of the contract advertisement we are currently compiling, hence the additional index on our type (family).

```
InitTx : Precondition → Type
InitTx g = Tx (length $ persistentIds g) 1
```

For the rest of the transactions, the shape will depend on the particular subterm we are currently compiling, which will have the form of a branch. Top-level decorations do not influence the shape in any way, so we need to strip them before performing case analysis on what kind of branch we have at hand. (The additional pattern matching on a lemma about `removeTopDecorations` help us omit the redundant cases for decoration constructors.)

```
BranchTx : Branch → Type
BranchTx d
 with _ ← decorations∘remove≡[] {d}
 with removeTopDecorations d
... | put xs &reveal _ if _ ⇒ c = Tx (suc $ length xs) 1
... | split vcs                 = Tx 1 (length vcs)
... | withdraw _                = Tx 1 1
```

We only have a plural number of inputs when handling a `put` command that introduced new volatile deposits, and multiple outputs in the case of a `split` command with several clauses.

The above specialisations of the transaction type are enough to let us specify the result type of our compiler:

- an initial transaction `T_init` with an input for each persistent deposit and a single output,
- and a transaction per subterm `d` of the contract, whose number of inputs and outputs is dictated by what kind of branch `d` is.

Integrating with the input types given in Section 5.1.2, we can formulate the type of the BitML compiler:

```
bitml-compiler : let ⟨ g ⟩ ds = ad in
 Valid ad
 → (sechash : Sechash g)
 → (txout   : Txout g)
 → (K       : 𝕂 g)
```

```
→ (K²        : 𝕂²′ ad)
→ InitTx g × (subterms⁺ ad ↦′ BranchTx)
```

The collection of `subterms⁺` is nearly identical to the implementation of `subterms`, but strips away all decorations and also includes the initial input itself, *e.g.,*

```
_ : subterms⁺ (put xs ∙ ( A : withdraw B
                        ⊕ B : split ( v ⊸ withdraw A
                                    ⊗ v ⊸ after t : withdraw B)))
  ≡ [ put xs ∙ (A : _ ⊕ B : _)
    ; withdraw B
    ; split (v ⊸ withdraw A ⊗ v ⊸ after t : withdraw B)
    ; withdraw A
    ; withdraw B
    ]
_ = refl
```

5.2 Defining the compilation procedure

The task is to now inhabit the type we formulated for the BitML compiler, *i.e.,* give the actual definition of the translation procedure described in Fig.7 of [BZ18].

There is, however, one last obstacle we need to overcome to give a well-defined Agda term as an implementation of the BitML compiler: the compiler's definition as given by the rules in the original paper is not *structurally recursive*, which is all that the termination checking implemented in Agda can statically check for us [Abe98]. Therefore, we need to define a *well-founded* recursion scheme for contracts, but before we do that let us take a detour and explain how Agda can accommodate well-founded recursion in general.

5.2.1 A primer on well-founded recursion

First of all, some motivation on the need for termination checking:[2] if non-terminating function were allowed, we would be able to produce a value of any type for free!

```
{-# TERMINATING #-}
deus-ex-machina : ∀ {A : Type ℓ} → A
deus-ex-machina = deus-ex-machina
```

Hence the need for termination, which can be syntactically checked if recursion happens only on strictly smaller arguments. Alas, it is not always obvious to Agda that arguments are indeed smaller, as the following case for natural numbers demonstrates.

[2] https://agda.readthedocs.io/en/v2.6.3/language/termination-checking.html#terminating-pragma

```
{-# TERMINATING #-}
f : ℕ → ℕ
f 0 = 0
f n = f ⌊ n /2⌋
```

Removing the compiler pragma that disables termination checking gives a type error indicating that Agda cannot automatically prove that repeatedly halving a non-zero number eventually terminates. If only we could prove the following termination property and instruct Agda to take it into consideration during termination checking.

```
/2-less : ∀ n → ⌊ n /2⌋ ≤ n
/2-less = λ where
  zero           → z≤n
  (suc zero)     → z≤n
  (suc (suc n)) → s≤s $ ≤-trans (/2-less n) (n≤1+n _)
```

Fortunately, there is a systematic way to turn any kind of well-founded recursion into structural recursion via *accessibility predicates*.

Given a type A with an arbitrary ordering relation ≲, we call an element of this type *accessible* only when all smaller elements are also accessible; *well-founded* relations are then the ones that render *all* elements of the corresponding type accessible.

```
module _ {A : Type} (_≲_ : Rel A 0ℓ) where
  data Acc (n : A) : Type where
    acc : (∀ m → m ≲ n → Acc m) → Acc n

  WellFounded : Type
  WellFounded = ∀ n → Acc n
```

Going back to the case of natural numbers, we can easily prove that the ‘less-than’ relation < is well-founded:

```
<-wf : WellFounded _<_
<-wf = acc ∘ go
  where
    go : ∀ n m → m < n → Acc _<_ m
    go (suc n) zero    _           = acc λ _ ()
    go (suc n) (suc m) (s≤s m<n) = acc λ o o<sm → go n o (<-transˡ o<sm m<n)
```

Now for the surprising bit: structurally recursing on such a *proof* gives us the means to perform well-founded recursion on the underlying (accessible) number!

Here is the same function f , but now provably terminating as witnessed by the additional proof obligations we have to provide at each recursive call:

```
f : ℕ → ℕ
f n = go _ (<-wf n)
```

```
  where
   go : ∀ n → Acc _<_ n → ℕ
   go zero    _       = 0
   go (suc n) (acc a) = go ⌊ n /2⌋ (a _ (s≤s $ /2-less _))
```

The magic lies in the unwrapping of the `acc` constructor, allowing us to actually recurse from our input argument `acc a` to a strictly smaller argument `a` (given that all defined datatypes are strictly positive[3]); this provides the technical explanation of how it is possible to get well-founded recursion via structural recursion.

The standard library already provides these fundamental definitions of accessibility and well-founded, as well as derivations of a recursion scheme embodied in the `go` helper function above. Here is how we can define the same example `f` in this more streamlined form:

```
f : ℕ → ℕ
f = <-rec _ λ where
  zero    _ → 0
  (suc n) r → r ⌊ n /2⌋ (s≤s $ /2-less _)
```

In the above, the standard library's modules for natural numbers contain a proof that `<` is well-founded (`<-wf`), from which a recursor `<-rec` is computed that let us state our fixpoint computation in a more concise way.

5.2.2 A well-founded recursion for BitML contracts

Let us immediately apply the technique for well-founded recursion we just outlined to the case of BitML contracts.

One small caveat is that we need some form of mutual recursion here between branches, contracts and lists of clauses, which creates the need for a *heterogeneous* ordering relation. To accommodate this, we bundle up these different kinds in a unified sum type `ℂ`, and then proceed as usual on a *homogeneous* relation `<`:

```
data ℂ : Type where
  D : Branch     → ℂ
  C : Contract   → ℂ
  V : VContracts → ℂ

data _<_ : Rel₀ ℂ where
  <-∈      : d ∈ c
           → D d   < C c
  <-∈ᵛ     : c ∈ map proj₂ vcs
           → C c   < V vcs
```

[3] https://agda.readthedocs.io/en/v2.6.3/language/positivity-checking.html

```
<-put   : C c   < D (put xs &reveal as if p ⇒ c)
<-auth  : D d   < D (A : d)
<-after : D d   < D (after t : d)
<-split : V vcs < D (split vcs)
```

Proving that the above relation is well-founded simply requires an induction on the list membership proof (in the case of branch choices or `split` clauses); all other cases are trivial.

```
<-wf : WellFounded _<_
<-wf = acc ∘ _>_
  where
    _>_ : ∀ c c′ → c′ < c → Acc _<_ c′
    (.(C (_ ∷ _)) > .(D _)) (<-∈ (here refl)) = acc (_ >_)
    (.(C (_ ∷ _)) > .(D _)) (<-∈ (there p))   = (_ > _) (<-∈ p)

    (.(V (_ ∷ _)) > .(C _)) (<-∈ᵛ {vcs = _ ∷ _} (here refl)) = acc (_ >_)
    (.(V (_ ∷ _)) > .(C _)) (<-∈ᵛ {vcs = _ ∷ _} (there p))   = (_ > _) (<-∈ᵛ p)

    (.(D (put _ &reveal _ if _ ⇒ _)) > .(C _)) <-put   = acc (_ >_)
    (.(D (_ : _))                    > .(D _)) <-auth  = acc (_ >_)
    (.(D (after _ : _))              > .(D _)) <-after = acc (_ >_)
    (.(D (split _))                  > .(V _)) <-split = acc (_ >_)
```

The standard library can take care of the rest for us and provides us with a recursor for the purpose of defining the BitML compiler.

```
<-rec : Recursor (WfRec _<_)
<-rec = wfRec <-wf 0ℓ
```

Normalising the succinct type of `<-rec` gives us the more readable

```
(P : ℂ → Type) → ((x : ℂ) → ((y : ℂ) → y < x → P y) → P x) → (x : ℂ) → P x
```

which lets us eliminate an input `x` into a final result type (indexed over said input) recursively: the higher-order argument is explicitly given to handle recursive calls on *strictly smaller* inputs.

5.2.3 The translation as a recursive function

We can finally state the translation procedure of the BitML compiler as a recursive function under the well-founded relation of Section 5.2.2.

```
bitml-compiler {ad = ⟨ G₀ ⟩ C₀} vad sechash₀ txout₀ K K² =
  Tᵢₙᵢₜ , (<-rec _ go) (ℂ.C C₀) s₀
  where
```

An initial transaction `Tinit` is immediately constructed, and then a well-founded recursion `go` is initiated on the initial contract `C₀` to construct the rest of the transactions (corresponding to subterms of the contract).

Note the additional argument `s₀` providing auxiliary information that will be threaded through sub-calls, rendering `go` *stateful*. The state keeps track of essential information that also appears in the logical presentation of [BZ18], namely the last transaction output produced (`T,o`) along with its value (`curV`), a subset of the participants (`P`) that need to sign the next transaction input, and the current time (`curT`).

```
record State (c : ℂ) : Type where
 constructor _&_&_&_&_&_&_&_&_&_&_
 pattern
 field
  T,o  : TxInput
  curV : Value
  P    : ∃ (_⊆ partG)
  curT : Time

  sechash : Sechash c
  txout   : Txout c
  part    : ids c ↦ ∃ (_∈ partG)
  val     : ids c ↦ Value

  p⊆ : participants c ⊆ partG
  s⊆ : subterms c ⊆ subterms C₀
  ∃s : case c of λ{ (ℂ.D _) → ∃ (_∈ subterms C₀) ; _ → ⊤}
```

In addition to these, the type of states is indexed over the current contract getting compiled, letting us require information specific to it, such as mappings from its secrets (`sechash`) or identifiers (`txout`, `part`, `val`). Finally, we retain proofs for some invariants we will need to produce the final proofs embedded in the compilation result; these all relate the current sub-contract to the initial contract `C₀` in some way or another.

The type of `go` therefore takes a second state argument, indexed over the existing contract argument `c`, and returns the `txout` mapping that we would to construct in the end.

```
Return : ℂ → Type
Return c = subterms⁺ c ↦′ BranchTx

go : ∀ c → (∀ c′ → c′ ≺ c → State c′ → Return c′) → State c → Return c
```

What remains for a complete definition of the compiler is:

- generating the appropriate validation script for each transaction ($\text{Ƀ}_{\textbf{out}}$),

- constructing the initial transaction `Tinit` and the initial state `s₀` (Ƀ_adv),
- and defining the recursive cases of `go` for atomic branches (Ƀ_D), list of contract choices (Ƀ_C), and parallel distribution funds across valued contracts (Ƀ_par).

5.2.3.1 Generating the validator script (Ƀ_out in [BZ18])

The first step is calculating the Bitcoin script that will lock the outputs of each generated transaction, *i.e.*, every subterm of the input contract `C₀`. One invariant across all cases is that all participants will need to provide their signatures, hence the context will be of size at least `ς = length partG`.

All the other checks will be introduced by `put` commands, where we make sure the predicates of the `put-if` part are satisfied and the revealed secrets have the hashes promised by the `sechash` mapping. (There is an additional check that the lengths of hashes are always above a certain threshold `η`, which is a security parameter for the compiler to be resistant to adversarial attacks; this can be safely ignored for now.)

```
Bₒᵤₜ : subterms C₀ ↦ (∃[ ctx ] Script ctx `𝔹)
Bₒᵤₜ {D} D∈ with removeTopDecorations D in D≡
... | put zs &reveal as if p ⇒ _
    = (ς + m)
    ,   versig (mapWith∈ partG $ K² D∈) (inject+ m <$> allFin ς)
     `∧ Bᵖʳ p p⊆as
     `∧ ⋀ (mapEnumWith∈ as $ λ i a a∈ → let bi = var (raise ς i) in
              hash bi `= ` (sechash₀ $ as⊆ a∈)
           `∧ ` (+ η) `< ∣ bi ∣)
  where
    m = length as
```

Formulating these checks naturally involves translating BitML predicates to Bitcoin ones (`Bᵖʳ`), which in turn requires a translation of arithmetic expressions (`Bᵃʳ`). Although the correspondence between constructs is obvious, calculating the size of a secret's hash has to refer to a script argument on the Bitcoin side, which requires some tedious proof plumbing incurred by the additional proof argument that all secrets (`secrets e`) have a matching revealed secret in `as`; `⊆as 0` then retrieves this secret.

```
Bᵃʳ : (e : Arith) → secrets e ⊆ as → Script (ς + m) `ℤ
Bᵃʳ = λ where
  (Arith.` x) _ →
   ` x
  (Arith.∥ s ∥) ⊆as →
   ∣ var $ raise ς $ index $ ⊆as 0 ∣ `- ` (+ η)
  (x Arith.`+ y) ⊆as →
     Bᵃʳ x (⊆as ∘ ∈-mapMaybe-++⁺ˡ isInj₁ {names x} {names y})
```

```
    `+ Bᵃʳ y (⊆as ∘ ∈-mapMaybe-++⁺ʳ isInj₁ (names x) {names y})
  (x Arith.`- y) ⊆as →
      Bᵃʳ x (⊆as ∘ ∈-mapMaybe-++⁺ˡ isInj₁ {names x} {names y})
    `- Bᵃʳ y (⊆as ∘ ∈-mapMaybe-++⁺ʳ isInj₁ (names x) {names y})
```

```
Bᵖʳ : (e : Predicate) → secrets e ⊆ as → Script (ς + m) `𝔹
Bᵖʳ = λ where
  Predicate.`true _ →
    `true
  (p Predicate.`∧ q) ⊆as →
      Bᵖʳ p (⊆as ∘ ∈-mapMaybe-++⁺ˡ isInj₁ {names p} {names q})
    `∧ Bᵖʳ q (⊆as ∘ ∈-mapMaybe-++⁺ʳ isInj₁ (names p) {names q})
  (Predicate.`¬ p) ⊆as →
    `not (Bᵖʳ p ⊆as)
  (x Predicate.`= y) ⊆as →
      Bᵃʳ x (⊆as ∘ ∈-mapMaybe-++⁺ˡ isInj₁ {names x} {names y})
    `= Bᵃʳ y (⊆as ∘ ∈-mapMaybe-++⁺ʳ isInj₁ (names x) {names y})
  (x Predicate.`< y) ⊆as →
      Bᵃʳ x (⊆as ∘ ∈-mapMaybe-++⁺ˡ isInj₁ {names x} {names y})
    `< Bᵃʳ y (⊆as ∘ ∈-mapMaybe-++⁺ʳ isInj₁ (names x) {names y})
```

(We omit the details of proving the two remaining lemmas and just show their type; these are easily proven by using previous lemmas already proven in the formalisation of BitML.)

```
as⊆  : as ⊆ secrets G₀
p⊆as : secrets p ⊆ as
```

In the case of non-`put` commands, the validator script is a simple signature check using the keys provided for this particular subterm via `K²`.

```
... | _ = ς , versig (mapWith∈ partG $ K² D∈) (allFin ς)
```

5.2.3.2 Constructing the initial transaction ($\B_{adv}$ in [BZ18])

With the inter-system translation of expressions at hand, constructing the initial transaction `Tᵢₙᵢₜ` is straightforward: the inputs are drawn from the `txout` mapping which points to an existing Bitcoin transaction output per persistent deposit in the BitML precondition, and a single output holding the sum of these deposits is locked by the script calculated by `Bₒᵤₜ`.

```
v₀ = sum $ (proj₁ ∘ proj₂) <$> persistentDeposits G₀
```

```
part₀ : ids G₀ ↣ ∃ (_∈ partG)
val₀  : ids G₀ ↣ Value
```

```
Tinit : InitTx G₀
Tinit = sig⋆ (fromList∘mapWith∈ xs K⋆)
 record
 { inputs  = fromList∘mapWith∈ xs (hashTxⁱ ∘ txout₀ ∘ xs⊆)
 ; wit     = wit⊥
 ; relLock = replicate 0
 ; outputs = [ -, v₀ locked-by ƛ ⋁ (mapWith∈ C₀ $ Bout ∘ subterms⊆ᶜ) .proj₂ ]
 ; absLock = 0 }
 where
  xs = persistentIds G₀

  xs⊆ : xs ⊆ ids G₀
  xs⊆ = persistentIds⊆ {G₀}

  K⋆ : xs ↦ List KeyPair
  K⋆ = [_] ∘ K ∘ proj₂ ∘ part₀ ∘ xs⊆
```

In addition to the mappings we have already introduced, we make use of auxiliary mappings `part` and `val` to get some more information about deposits. The helper definitions in the `where` clause help us construct the inputs and their witnesses from the existing `sechash` and `K` mappings.

Erratum

Correction on [BZ18]. In the original paper (§A.5, Fig.8), the inputs to T_{init} are not accompanied by any witnesses, contradicting the examples presented earlier in the very same paper (§7). This is rectified in our work by the application of `sig⋆` to the bare transaction.

Notation

Transaction hashes. We will often need to compute the hash of a transaction, but it is crucial we always pack the indices of the transaction along with it, *i.e.*, apply the hashing function `_♯` to a value of type `∃Tx` instead of an indexed value of type `Tx i o` or any other variant of this sort. We need to be consistent and hash transactions only in this bundled form, otherwise we will later be unable to appeal to the (postulated) injectivity of hashes.

Such existential plumbing is uninteresting, so we will omit these intermediate definitions and adopt a naming convention of always suffixing the transaction variable with `♯`, *e.g.*, variable T_{init}♯ holds the computed hash of transaction T_{init} which is computed as: `Tinit♯ = (∃Tx ∋ -, -, Tinit) ♯`.

The initial state of the recursion will set the *initial* parameters as the *current* ones, and provide some initialisation for the auxiliary mappings that record information about each deposit identifier in the contract.

```
s₀ : State (ℂ.C C₀)
s₀ = record
  { T,o      = Tᵢₙᵢₜ♯ at 0
  ; curV     = v₀
  ; P        = partG , ⊆-refl
  ; curT     = 0
  ; p⊆       = nub-⊆⁺ ∘ Valid⇒part⊆ vad
  ; s⊆       = id
  ; ∃s       = tt
  ; sechash = sechash₀ ∘ mapMaybe-⊆ isInj₁ names⊆
  ; txout   = txout₀   ∘ mapMaybe-⊆ isInj₂ names⊆
  ; part    = part₀    ∘ mapMaybe-⊆ isInj₂ names⊆
  ; val     = val₀     ∘ mapMaybe-⊆ isInj₂ names⊆ }
```

Since we will need to construct such states throughout all the recursive cases that will follow, we will avoid the verbose `record` syntax used above and use the ampersand constructor (`&`) to combine state components moving forward.

5.2.3.3 Handling branches (฿D in [BZ18])

We are ready to give the cases for translating singular branches, starting with the `withdraw` command that spends the current output and locks the remaining funds of the contract under the private key of the prescribed participant:

```
go (ℂ.D c) rec
   ( T,o & v & P , P⊆ & t
   & sechash & txout & part & val
   & p⊆ & s⊆ & ∃s@(Dₚ , Dₚ∈))
  with c
... | withdraw A = λ where
  (here refl) →
    sig⋆ [ mapWith∈ P (K² Dₚ∈ ∘ P⊆) ] record
      { inputs  = [ T,o ]
      ; wit     = wit⊥
      ; relLock = [ 0 ]
      ; outputs = [ 1 , v locked-by ƛ versig [ K {A} (p⊆ 0) ] [ 0 ] ]
      ; absLock = t }
```

(The pattern `here refl` matches the case where the element is the head of a non-empty list, which is the only case here as `withdraw` does not have any subterms within.)

The other two singular cases concern decorations which do not produce a transaction themselves, but simply change the threaded parameters: authorisations modify the current set of participants, while time decorations advance the current time.

```
... | A : d = rec (ℂ.D d) <-auth
  ( T,o & v & P \\ [ A ] , P⊆ ∘ \\-⊆ & t
  & sechash & txout & part & val
  & p⊆ ∘ there & s⊆ & ∃s)
... | after t′ : d = rec (ℂ.D d) <-after
  ( T,o & v & P , P⊆ & t ⊔ t′
  & sechash & txout & part & val
  & p⊆ & s⊆ & ∃s)
```

(The pattern `there` matches the cases where the subterms belongs to the tail of the list; here we use it as a term to weaken the domain of our mappings, *i.e.*, to disregard the top-level subterm.)

Notice how we made a recursive call with `rec` for the first time, appealing to the well-founded relation of Section 5.2.2 via its constructors `<-auth` and `<-after`.

5.2.3.4 Handling contract choices (฿C in [BZ18])

Moving on to the `put` command, we first generate a new transaction `Tᶜ` that spends the last transaction as well as the newly introduced *volatile* deposits `zs`.

Then, we get a list of branches as continuations, which we recursively compile to transactions that will spend the single output of `Tᶜ` (eventually only *one of them* will be executed though, that's why they represent alternative choices), making sure the state is updated to reflect the increased monetary value by the inclusion of the new deposits.

```
... | c@(put zs &reveal as if p ⇒ cs) = λ where
  (here refl) → Tᶜ
  (there x∈) → rec (ℂ.C cs) <-put
    ( (Tᶜ♯ at 0) & v′ & (partG , ⊆-refl) & 0
    & sechash ∘ mapMaybe-⊆ isInj₁ ns⊆ & txout ∘ mapMaybe-⊆ isInj₂ ns⊆
    & part    ∘ mapMaybe-⊆ isInj₂ ns⊆ & val   ∘ mapMaybe-⊆ isInj₂ ns⊆
    & p⊆ & s⊆ & tt
    ) x∈
    where
      cs⊆ : cs ⊆ subterms C₀
      zs⊆ : zs ⊆ ids c
      ns⊆ : names cs ⊆ names c

      v′ = v + sum (mapWith∈ zs $ val ∘ zs⊆)

      Tᶜ : BranchTx c
      Tᶜ = sig⋆ (mapWith∈ P (K² Dₚ∈ ∘ P⊆) ∷ fromList∘mapWith∈ zs K⋆) record
```

```
    { inputs  = T,o :: fromList∘mapWith∈ zs (hashTxⁱ ∘ txout ∘ zs⊆)
    ; wit     = wit⊥
    ; relLock = replicate 0
    ; outputs = [ -, v′ locked-by ƛ ⋁ (mapWith∈ cs $ Bₒᵤₜ ∘ cs⊆) .proj₂ ]
    ; absLock = t }
```

Ignoring some details regarding signatures and auxiliary lemmas that have been omitted (it is the same construction as for $\mathtt{T_{init}}$), the participants sign the first contract input (the last state of the active contract) using their subterm keys (K^2), but provide the witnesses (`wit`) for the volatile deposits using their public keys from K.

5.2.3.5 Handling split clauses (₿par in [BZ18])

The distribution of funds performed by `split` construct follows a very similar pattern to `put`, yet somewhat simpler due to the absence of new deposits that influence the current value.

A new transaction $\mathtt{T^c}$ is generated, now having a single input and multiple outputs, each of them getting the prescribed share of the funds $\mathtt{v_i}$. The subsequent transactions that will eventually spend these outputs (*all of them* will be executed in contrast to the branch choices in `put`) are compiled recursively in an appropriate state.

```
... | c@(split vcs) = λ where
  (here refl) → Tᶜ
  (there x∈)  → rec (ℂ.V vcs) <-split
    ( (Tᶜ♯ at 0) & v & (partG , ⊆-refl) & 0
    & sechash & txout & part & val
    & p⊆ & s⊆ & tt
    ) x∈
   where
    Tᶜ : BranchTx c
    Tᶜ = sig⋆ [ mapWith∈ P (K² Dₚ∈ ∘ P⊆) ] record
      { inputs  = [ T,o ]
      ; wit     = wit⊥
      ; relLock = replicate 0
      ; outputs = mapWith∈ (fromList vcs) λ{ {vᵢ , Cᵢ} x∈ →
          let eᵢ = mapWith∈ Cᵢ $ Bₒᵤₜ ∘ s⊆ ∘ subterms⊆ᵛ (fromList⁻ x∈)
          in -, vᵢ locked-by ƛ (⋁ eᵢ) .proj₂
        }
      ; absLock = t }
```

Erratum

Improvement on [BZ18]. In the original paper, the compilation case for `split` further required that the sum of values across the clauses does not exceed the current

value ($\sum_{i=1}^{k} v_i \leq v$). This is not necessary in our setting, where we work on contracts accompanied by a proof of validity (see Section 4.5), so this property invariably holds in our system *by construction*.

Do not worry about the incorrect index into the outputs of `Tᶜ` in the first argument of the updated state; this will be handled by the `ℂ.V` case which will recurse through the list of clauses and increment the index as needed. Apart from that, the other cases of `go` merely propagate the state accordingly to the recursive calls (`r`) with no interesting computational content (the boilerplate calls `mapMaybe-⊆` just weakens the associated mappings to fit the smaller domain of current resources as we recurse, without changing any values).

```
go (ℂ.C _) rec st = ↦-∈ λ {d} d∈ → rec (ℂ.D d) (<-∈ d∈) (↓ st d∈)
  where ↓ : State (ℂ.C ds) → ds ↦′ (State ∘ ℂ.D)

go (ℂ.V _) rec st = ↦-∈ᵛ λ {c} c∈ → rec (ℂ.C c) (<-∈ᵛ c∈) (↓ᵛ st c∈)
  where ↓ᵛ : State (ℂ.V vcs) → (proj₂ <$> vcs) ↦′ (State ∘ ℂ.C)
```

And we are done! We have completely defined the BitML compiler and done so using dependent types to ensure only valid inputs are accepted and the output has the expected form.

This is far from being a *correct-by-construction* compiler though, as we have neither related the high-level semantics of BitML with the low-level semantics of Bitcoin nor proven that the compiler respects those semantics in some sense. Thus, we still have a long way to go in order to establish *compilation correctness*, which will be the subject of the next chapter.

5.3 Compilation examples

By virtue of defining the compiler as a recursive function rather than a logical relation (as originally presented in [BZ18]), while at the same time staying within the confines of a constructive logic, it is possible to execute the compiler on specific examples and verify the results.

Before considering the more involved timed-commitment protocol, let's illustrate some basic examples of compiling each primitive BitML construct. (We assume an arbitrary security parameter `η`; it does not influence the examples in any way.)

Notation

Membership indices. When dealing with mappings that are defined as functions that take list membership proofs of the form `x ∈ xs` as input (Section 5.1.1), there

is a need to prove these by hand every time we construct a new mapping or query values from an existing one.

To avoid cluttering the code, one can treat such membership proofs as indices into the list xs via the following pattern synonyms:

```
pattern 𝟘 = here refl
pattern 𝟙 = there 𝟘
pattern 𝟚 = there 𝟙
```

5.3.1 Compiling primitives (§7 of [BZ18])

For a given advertisement with participants partG and underlying contract C, assume the key generators K and K² as given.

```
postulate
  K  : partG ↦ KeyPair
  K² : subterms C ↦ partG ↦ KeyPair
```

Withdraw. Let's start with a simple withdraw contract, where after both A and B deposit 1₿ each, B withdraws all remaining funds (*i.e.*, 2₿).

```
ex-ad : Ad
ex-ad = ⟨ A :! 1 ₿ at "x" ∣ B :! 1 ₿ at "y" ⟩ [ withdraw B ]
```

To compile this contract, first there need to exist some prior transaction outputs on Bitcoin that correspond to the BitML deposits required by the precondition.

```
T₀ : Tx 0 2
T₀ = record
  { inputs   = []
  ; wit      = wit⊥
  ; relLock  = V.replicate 0
  ; outputs  = [ (1 , 1 ₿ locked-by ƛ (versig [ K 𝟘 ] [ 0F ]))
               ⨾ (1 , 1 ₿ locked-by ƛ (versig [ K 𝟙 ] [ 0F ])) ]
  ; absLock  = 0 }

Tˣ Tʸ : TxInput′
Tˣ = (-, -, T₀) at 0F
Tʸ = (-, -, T₀) at 1F
```

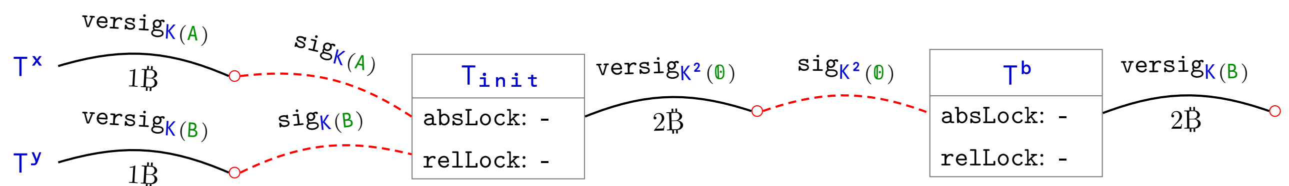
We can then construct the necessary mappings that the compiler requires and compile ex-ad into two transactions of the expected shape:

```
sechash : secrets G ↦ ℤ
sechash ()
```

```
txout : ids G ↦ TxInput′
txout = λ where
  {- "x" -} 𝟘 → Tˣ
  {- "y" -} 𝟙 → Tʸ

out : InitTx G × (subterms⁺ C ↦′ BranchTx)
out = bitml-compiler {ad = ex-ad} auto sechash txout K K²

outTxs : Tx 2 1 × Tx 1 1
outTxs = let t₀ , m = out in t₀ , m 𝟘
```

To test that the compiler produced the correct output, we can construct the expected transactions by hand (also illustrated below as a graph) and check that they are definitionally equal.

```
Tᵢₙᵢₜ : Tx 2 1
Tᵢₙᵢₜ = sig⋆ [ [ K 𝟘 ] ⨾ [ K 𝟙 ] ] record
  { inputs  = hashTxⁱ <$> [ Tˣ ⨾ Tʸ ]
  ; wit     = wit⊥
  ; relLock = V.replicate 0
  ; outputs = [ 2 , 2 ₿ locked-by ƛ versig (codom $ K² 𝟘) [ 0F ⨾ 1F ] ]
  ; absLock = 0 }


Tᵇ : Tx 1 1
Tᵇ = sig⋆ [ codom (K² 𝟘) ] record
  { inputs  = [ Tᵢₙᵢₜ♯ at 0 ]
  ; wit     = wit⊥
  ; relLock = V.replicate 0
  ; outputs = [ 1 , 2 ₿ locked-by ƛ versig [ K 𝟙 ] [ 0F ] ]
  ; absLock = 0 }

_ = outTxs ≡ (Tᵢₙᵢₜ , Tᵇ)
  ∋ refl
```



Everything seems to be in order: an initial transaction spends the (persistent) deposits, witnessed by the correct keys (the private key of A unlocks T^x and B unlocks T^y), and a subsequent transaction spends these to execute the `withdraw` command by locking the total value of 2₿ by the private key of B.

Split. Moving on to the splitting construct, here is a simple contract that expects 2₿ from A and 1₿ from B and exchanges their ownership:

```
ex-ad : Ad
ex-ad = ⟨ A :! 2 ₿ at "x" ∣ B :! 1 ₿ at "y" ⟩
        [ split (1 ₿ ⊸ withdraw A ⊗ 2 ₿ ⊸ withdraw B) ]
```

The compiler parameters and required mappings are constructed the exact same way as in the previous example, so there is no point in repeating them here.

```
out : InitTx G × (subterms⁺ C ↦′ BranchTx)
out = bitml-compiler {ad = ex-ad} auto sechash txout K K²
```

```
outTxs : Tx 2 1 × Tx 1 2 × Tx 1 1 × Tx 1 1
outTxs = let t₀ , m = out in t₀ , m 𝟘 , m 𝟙 , m 𝟚
```

The compiler produces four transactions for this contract. As always, there is the initial transaction spending the existing deposits and a second transaction that receives the funds and performs the current action, in this case splitting the funds in two separate outputs locked by the corresponding subterm-key:

```
Tᵢₙᵢₜ : Tx 2 1
Tᵢₙᵢₜ = sig⋆ [ [ K 𝟘 ] ⨾ [ K 𝟙 ] ] record
  { inputs  = hashTxⁱ <$> [ Tˣ ⨾ Tʸ ]
  ; wit     = wit⊥
  ; relLock = V.replicate 0
  ; outputs = [ 2 , 3 ₿ locked-by ƛ versig (codom $ K² 𝟘) [ 0F ⨾ 1F ] ]
  ; absLock = 0 }
```

```
Tₛₚₗᵢₜ : Tx 1 2
Tₛₚₗᵢₜ = sig⋆ [ codom (K² 𝟘) ] record
  { inputs  = [ Tᵢₙᵢₜ♯ at 0 ]
  ; wit     = wit⊥
  ; relLock = V.replicate 0
  ; outputs = [ 2 , 1 ₿ locked-by ƛ versig (codom $ K² 𝟙) [ 0F ⨾ 1F ]
              ⨾ 2 , 2 ₿ locked-by ƛ versig (codom $ K² 𝟚) [ 0F ⨾ 1F ] ]
  ; absLock = 0 }
```

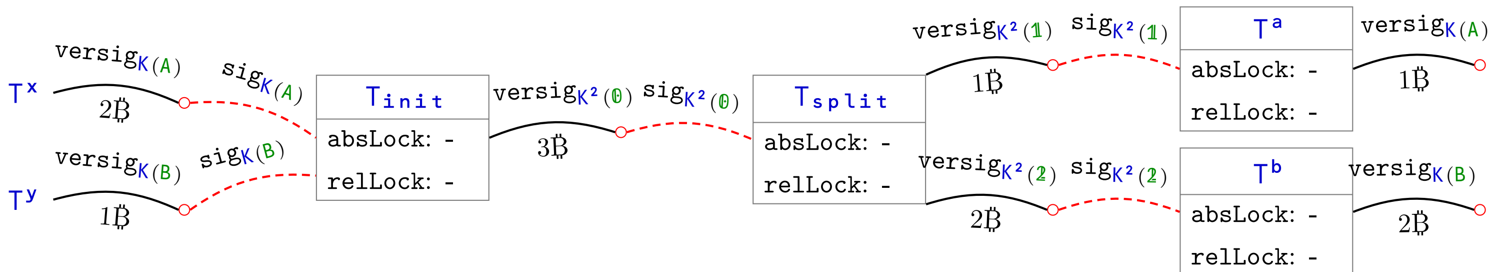
For each of the clauses, we now get another transaction that spends the matching output of `Tₛₚₗᵢₜ`, providing the correct witnesses for each case and performing the corresponding `withdraw`:

```
Tᵃ : Tx 1 1
Tᵃ = sig⋆ [ codom (K² 𝟙) ] record
  { inputs  = [ Tₛₚₗᵢₜ♯ at 0 ]
  ; wit     = wit⊥
```

```
  ; relLock = V.replicate 0
  ; outputs = [ 1 , 1 ₿ locked-by ƛ versig [ K 0 ] [ 0F ] ]
  ; absLock = 0 }

Tᵇ : Tx 1 1
Tᵇ = sig⋆ [ codom (K² 2) ] record
  { inputs  = [ Tsplit♯ at 1 ]
  ; wit     = wit⊥
  ; relLock = V.replicate 0
  ; outputs = [ 1 , 2 ₿ locked-by ƛ versig [ K 1 ] [ 0F ] ]
  ; absLock = 0 }
```

As a final step, we check that the compiler actually behaves as expected:

```
_ = outTxs ≡ (Tinit , Tsplit , Tᵃ , Tᵇ)
  ∋ refl
```



Put. What about *volatile* deposits? How do they come into the mix? There is no better way to demonstrate this than a stripped-down `put` that only provides a volatile deposit and nothing else:

```
ex-ad : Ad
ex-ad = ⟨ A :? 1 ₿ at "x" ∣ A :! 1 ₿ at "y" ∣ B :! 1 ₿ at "z" ⟩
        [ put "x" ∙ withdraw B ]
```

The difference with the previous examples is that there will now be *three* pre-existing deposits (the volatile Tˣ and the persistent Tʸ and Tᶻ), extending the `txout` mapping by one element:

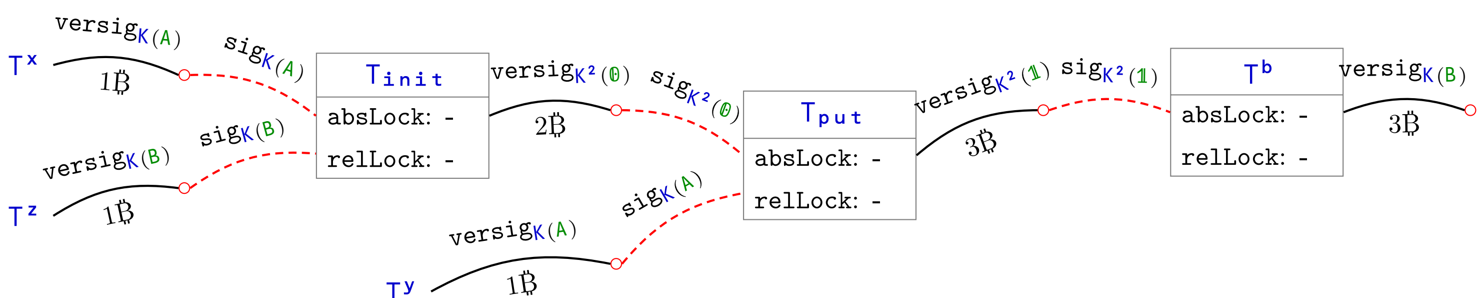
```
txout : ids G ↦ TxInput′
txout = λ where
  0 → Tˣ
  1 → Tʸ
  2 → Tᶻ
```

Apart from the initial transaction, the compiler should produce one transaction per subterm, *i.e.*, one for the `put` command and one for the final `withdraw`, and indeed it does:

```
out : InitTx G × (subterms⁺ C ↦′ BranchTx)
out = bitml-compiler {ad = ex-ad} auto sechash txout K K²

outTxs : Tx 2 1 × Tx 2 1 × Tx 1 1
outTxs = let t₀ , m = out in t₀ , m 𝟘 , m 𝟙
```

The shape and content of the produced transactions should be clear by now: the initial transaction gathers all persistent deposits into a single output of 2B̸, a second transaction spends them and performs the first `put` command that now additionally spends a volatile deposit and outputs a single output of 3B̸ for its continuation, which transfers ownership of the remaining funds to `B` using their private key.

```
Tᵢₙᵢₜ : Tx 2 1
Tᵢₙᵢₜ = sig⋆ [ [ K 𝟘 ] ⨾ [ K 𝟙 ] ] record
  { inputs  = hashTxⁱ <$> [ Tʸ ⨾ Tᶻ ]
  ; wit     = wit⊥
  ; relLock = V.replicate 0
  ; outputs = [ 2 , 2 B̸ locked-by ƛ versig (codom $ K² 𝟘) [ 0F ⨾ 1F ]
                                   `∧ `true `∧ `true ]
  ; absLock = 0 }

Tₚᵤₜ : Tx 2 1
Tₚᵤₜ = sig⋆ [ codom (K² 𝟘) ⨾ [ K 𝟘 ] ] record
  { inputs  = [ Tᵢₙᵢₜ♯ at 0 ⨾ hashTxⁱ Tˣ ]
  ; wit     = wit⊥
  ; relLock = V.replicate 0
  ; outputs = [ 2 , 3 B̸ locked-by ƛ versig (codom $ K² 𝟙) [ 0F ⨾ 1F ] ]
  ; absLock = 0 }

Tᵃ : Tx 1 1
Tᵃ = sig⋆ [ codom (K² 𝟙) ] record
   { inputs  = [ Tₚᵤₜ♯ at 0 ]
   ; wit     = wit⊥
   ; relLock = V.replicate 0
   ; outputs = [ 1 , 3 B̸ locked-by ƛ versig [ K 𝟙 ] [ 0F ] ]
   ; absLock = 0 }

_ = outTxs ≡ (Tᵢₙᵢₜ , Tₚᵤₜ , Tᵃ)
  ∋ refl
```



The only remaining primitive we have not seen in action is the `reveal` part of the `put` construct, which will be covered in the next and final example.

5.3.2 Timed-commitment protocol *(§A.5,Fig.8 of [BZ18])*

Let us illustrate the action of revealing secrets and picking branches out of several possible choices, by compiling the timed-commitment contract `TC` that we first encountered in Chapter 4, Section 4.6.

Assuming a secret `a` with hash `a♯` and the usual deposits, we need to compile the `TC` contract using a non-empty `sechash` mapping:

```
sechash : secrets G ↦ ℤ
sechash = λ where
  {- a -} 𝟘 → a♯

out : InitTx G × (subterms⁺ C ↦′ BranchTx)
out = bitml-compiler {ad = TC} auto sechash txout K K²

outTxs : Tx 2 1 × Tx 1 1 × Tx 1 1 × Tx 1 1
outTxs = let t₀ , m = out in t₀ , m 𝟘 , m 𝟙 , m 𝟚
```

We get four transactions as output, with the initial one doing the usual work of gathering the pre-existing deposits in a transaction output.

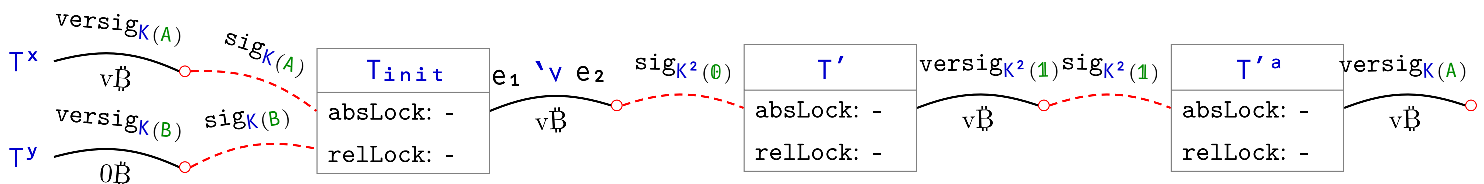
```
Tinit : Tx 2 1
Tinit = sig⋆ [ [ K 𝟘 ] ⨾ [ K 𝟙 ] ] record
  { inputs  = hashTxⁱ <$> [ Tᵃ ⨾ Tᵇ ]
  ; wit     = wit⊥
  ; relLock = V.replicate 0
  ; outputs = [ -, v locked-by ƛ (e₁ `∨ e₂) ]
  ; absLock = 0 }
  where
    e₁ : Script 3 `𝔹
    e₁ =  versig (codom $ K² 𝟘) [ 0F ⨾ 1F ]
       `∧ `true
       `∧ hash (var 2F) `= ` (sechash 𝟘)
       `∧ ` (+ η) `< ∣ var 2F ∣

    e₂ : Script 3 `𝔹
    e₂ = versig (codom $ K² 𝟚) [ 0F ⨾ 1F ]
```

The validation script takes the form of a logical disjunction, whose cases correspond to the two possible branches of the contract. Both require signatures from their respective subterm-keys, but in the first `reveal` case there is an additional requirement for a third script argument holding a revealed secret that has the promised hash and length.

Since there is a possible choice on which branch to execute next, the rest of the transactions can be conceptually divided into two sets that each corresponds to a choice.

Either the first branch is chosen using the next two produced transactions: T′ spends the initial output successfully verifying the first disjunctive clause (although A would actually have to add the secret a as the third witness when submitting T′ on-chain) and producing an output holding the deposit funds, which can then be withdrawn using T′ᵃ in the usual fashion.

```
T′ : Tx 1 1
T′ = sig⋆ [ codom (K² 0) ] record
  { inputs  = [ Tᵢₙᵢₜ♯ at 0 ]
  ; wit     = wit⊥
  ; relLock = V.replicate 0
  ; outputs = [ -, v locked-by ƛ e′ ]
  ; absLock = 0 }
  where
    e′ : Script 2 `𝔹
    e′ = versig (codom $ K² 1) [ 0F ; 1F ]

T′ᵃ : Tx 1 1
T′ᵃ = sig⋆ [ codom (K² 1) ] record
  { inputs  = [ T′♯ at 0 ]
  ; wit     = wit⊥
  ; relLock = V.replicate 0
  ; outputs = [ 1 , v locked-by ƛ versig [ K 0 ] [ 0F ] ]
  ; absLock = 0 }
```

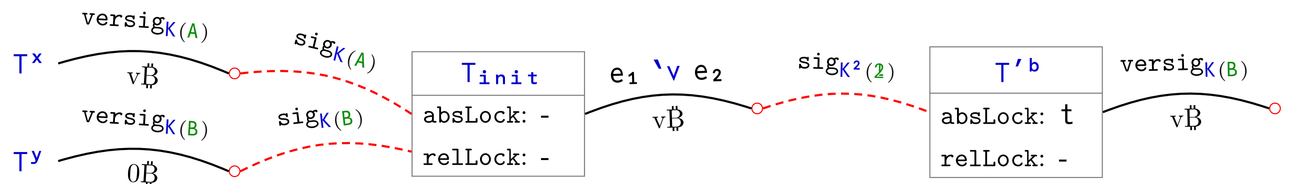


Notice how it is the first time where an input-output link is not simply a `versig-sig` pair, although if we actually unfold the union of the input (since we picked the first branch) it will turn out to have the form `versig` (along with the other 2 conditions on hashes).

Otherwise, the second branch is executed (corresponding to submitting T′ᵇ), where B receives A's deposit as penalty for not revealing their secret in time. Notice how the time delay stemming from `after` has manifested in the `absLock` field of the transaction, thus enforcing that this branch is only taken *after* time t.

```
T′ᵇ : Tx 1 1
T′ᵇ = sig⋆ [ codom (K² 2) ] record
```

```
{ inputs  = [ Tinit♯ at 0 ]
; wit     = wit⊥
; relLock = V.replicate 0
; outputs = [ 1 , v locked-by ƛ versig [ K 𝟙 ] [ 0F ] ]
; absLock = t }
```



Indeed these are the transactions generated from the BitML compiler:

```
_ = outTxs ≡ (Tinit , T′ , T′ᵃ , T′ᵇ)
  ∋ refl
```

Through these example, we demonstrated how the Bitcoin transactions generated by the BitML compiler for concrete inputs intuitively correspond to the constructs used in BitML contracts. But how can we make sure this holds in full generality and, more importantly, how do we turn this intuition into a precise formal specification that we can then prove?

Chapter 6

Relating Source To Target: Coherence

In order to prove the BitML compiler correct, we will establish a correspondence between *symbolic runs* of BitML contracts and *computational runs* of Bitcoin transactions.

This will take the form of an inductive *refinement* relation [DE98], henceforth called 'coherence', where the *base case* relates initial BitML configurations to an initial state of the blockchain, and the *inductive step* extends the relation step-wise by a single move on each side — essentially giving a Bitcoin implementation of each BitML construct — or solely extends the computational run with an *irrelevant* Bitcoin actions that do not influence or interact with the BitML contracts we are currently investigating.

Before presenting the coherence relation in Section 6.3 along with some example proofs of coherence in Section 6.3.4, we will first need to precisely define what we mean by the runs that coherence relates, both *computationally* in Section 6.1 (picking up from where we left off in Chapter 3) and *symbolically* in Section 6.2 (extending the traces defined in Chapter 4).

This chapter thus mechanises §8 and Appendix A.6 of the original paper, with the definition of the coherence relation appearing in Def.20, as well as the parts of §5 and A.3 that pertain to symbolic runs and the parts of §6 and A.4 regarding computational runs:

> Massimo Bartoletti and Roberto Zunino. "BitML: a calculus for Bitcoin smart contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 83–100

As always, the Agda code presented in this chapter is publicly accessible in HTML format here:

https://omelkonian.github.io/formal-bitml-to-bitcoin/

Chapter overview. We will start by precisely defining the low-level *computational* model of Bitcoin in §6.1, building on the previous definitions of Chapter 3, that will result in a notion of a *computational run* as a sequence of *computational steps*. This will involve some consideration about *serialising* data (§6.1.1), in particular how *computational contracts* arise as the serial format of their BitML counterparts (§6.1.2).

Similarly for the high-level *symbolic* model of BitML in §6.2, picking up from where we left off in Chapter 4, where *symbolic runs* will be defined as a sequence of *symbolic steps* respecting the operational semantics of Section 4.7. We will take care to provide lemmas that we prove crucial later on, such as properties of the mappings used in the compiler (§6.2.2), as well as some *trace properties* arising from the way the BitML rules are constructed (§6.2.3).

We will then finally be able to define the central contribution of this thesis: a mechanised definition of *coherence* will be presented as a *stratified* relation in §6.3, relating symbolic to computational runs via a *refinement* argument, *e.g.,* allowing us to prove the correspondence between the timed commitment BitML contract of Chapter 4 and the transactions we compiled from it at the end of Chapter 5, presented formally as coherence proofs in §6.3.4.

6.1 Computational model *(§6 & A.4 of [BZ18])*

Apart from a sequence of transactions, i.e. a blockchain, a computational run additionally includes communicated messages, either broadcast to all participants or exchanged between a participant and an *oracle*; the latter is how hashing is realised using the well-established *random oracle model* [BR93].

6.1.1 Serialization

Since messages will need to be transferred "on the wire", there must be some way to *serialize* the types of values that will be communicated as messages. Naturally, we will not go as far down as the level of bits and bytes, but rather postulate the necessary functionality, much like we did for the cryptographic primitives required so far. What we *do* however care about is the specification of serialization; what it means for a given type to be converted to bitstrings and what kind of properties should we expect from such a conversion.

First and foremost, messages will be of the same form as hashes,

```
Message = HashId
```

Hashes themselves have been concretely defined as integer numbers following the original paper [Atz+18b], but that is actually irrelevant; we will describe the encoding of values as messages with respect to an arbitrary type `DATA`:

```
record Serializable (A : Type) : Type where
 field
  encode : A → DATA
  encode-injective : Injective≡ encode

  decode : DATA → Maybe A
  encode-decode : ∀ m x →
   decode m ≡ just x
   ═══════════════
   m ≡ encode x
```

For a type to admit serialization, we need a way to encode any value — making sure that this encoding is *injective*, *i.e.*, no two different values are encoded to the same bitstring — as well as a way to decode some data into a value of the given type, possibly failing if the data is ill-formed to begin with.

The `encode-decode` property ensures that both directions of this translation *coincide*: decoding an encoded value always succeeds and returns the original value and, *vice versa*, a decoded value encodes to the original message.

The above give enough information to be able to decide, for an arbitrary message, whether it can be decoded to a value and, if so, provably establish that it is indeed the encoding of said value.

```
decode′ : ∀ m → Dec (∃ λ (x : A) → m ≡ encode x)
decode′ m
 with decode m in m≡
... | just x
 = yes (x , encode-decode m x .proj₁ m≡)
... | nothing
 = no λ (x , x≡) → case trans (sym $ encode-decode m x .proj₂ x≡) m≡ of λ ()
```

(The above utilises the recent addition of Agda's `with…in` syntax,[1] which lets us capture the relation between a term we are pattern matching on and its patterns in the `m≡` equality.)

Then, we can simply assume that such encoding/decoding procedures exist for the types we care about:

```
postulate instance
 Serializable-⊎ : ⦃ Serializable X ⦄ → ⦃ Serializable Y ⦄ → Serializable (X ⊎ Y)
```

[1] https://agda.readthedocs.io/en/v2.6.3/language/with-abstraction.html#with-abstraction-equality

```
Serializable-Σ : ⦃ Serializable X ⦄ → ⦃ ∀ {x} → Serializable (P x) ⦄ →
  Serializable (Σ X P)
Serializable-List : ⦃ Serializable X ⦄ → Serializable (List X)

Serialiazable-String : Serializable String
Serializable-ℕ      : Serializable ℕ
Serializable-Bool : Serializable Bool
Serializable-ℤ      : Serializable ℤ
Serialiazable-Tx  : Serializable (Tx i o)
```

(For instance, one could easily envisage how to encode the disjoint sum of two types using a single bit as a discrimination tag that tells us whether we are in the left or right case, but we refrain from exploring this further formally to focus on more important things.)

One aspect we cannot capture in the specification of serialization above concerns the encoding of different types and how they relate, namely that there is absolutely no way for encoding of values that inhabit different types to result in the same message (suggesting that we also encode the types themselves in the serial format). Thus, we have to postulate this requirement in general outside of the serialization interface:

```
postulate
  encode≢ : ⦃ _ : Serializable A ⦄ ⦃ _ : Serializable B ⦄ {a : A} {b : B} →
    ⦃ A ≢′ B ⦄ → encode a ≢ encode b
```

The inequality of types `A` and `B` is actually not provable within Agda itself; we welcome the reader to give this a try! That is the reason we use a home-grown primed inequality relation `≢′` whose inhabitants we postulate carefully, instantiated to only a handful of concrete types we will need to encode for the rest of the development.

Disclaimer

The `postulate` above should be handled with care, since a malicious user can easily derive `⊥` by providing implementations that do not respect this postulate (*e.g.*, encoding everything to the bitstring `0000`), thus compromising the soundness of our system. Therefore, the serialization part of the formalisation should morally be considered as part of the *trusted computed base*.

An adjacent assumption we will need later on is that such encodings do not clash with signatures (as defined in Chapter 3) and also signatures of different keys never collide:

```
postulate
  SIG≢ : ⦃ k ≢′ k′ ⦄ → SIG k x ≢ SIG k′ y

  SIG≢encode : ∀ ⦃ _ : Serializable B ⦄ {k} {x : A} {y : B} →
    SIG k x ≢ encode y
```

(Notice how we retain the use of the alternative inequality for keys as well; the reason is that it behaves more nicely than the standard implementation of inequality as negating propositional equality with a function to `⊥`, especially when combined with Agda's instance resolution because function arguments are silently introduced whereas our version wraps the above under a `record` to avoid these issues.)

6.1.2 Encoding contract advertisements *(§A.7 of [BZ18])*

There is some care to be taken when serializing BitML contracts though, as they contain arbitrary symbolic names as identifiers which precludes us from merely postulating an encoding procedure for them.

Instead, their serialized form will need to interpret these names *computationally*, *i.e.*, where names are substituted for actual Bitcoin transaction outputs. Formally, we repeat the definition of BitML contracts from Chapter 4, but now the identifiers used in the contract's preconditions and in `put` commands are replaced with explicit references to some Bitcoin output (`TxInput′`). Following the authors of the original paper (§A.7), we will refer to these entities as *computational contract advertisements*.

```
Idᶜ  = TxInput′
Idsᶜ = List Idᶜ

data Branchᶜ : Type
Contractᶜ   = List Branchᶜ
VContractsᶜ = List (Value × Contractᶜ)

data Branchᶜ where
  put_&reveal_if_⇒_ : Idsᶜ → Secrets → Predicate → Contractᶜ → Branchᶜ
  withdraw          : Participant → Branchᶜ
  split             : VContractsᶜ → Branchᶜ
  _:_               : Participant → Branchᶜ → Branchᶜ
  after_:_          : Time → Branchᶜ → Branchᶜ

data Preconditionᶜ : Type where
  _:?_at_   : Participant → Value → Idᶜ → Preconditionᶜ
  _:!_at_   : Participant → Value → Idᶜ → Preconditionᶜ
  _:secret_ : Participant → Secret → Preconditionᶜ
  _|_       : Preconditionᶜ → Preconditionᶜ → Preconditionᶜ

record Adᶜ : Type where
  constructor ⟨_⟩_
  field G : Preconditionᶜ
        C : Contractᶜ
```

(We could, of course, make the original definition of Chapter 4 parametric over the type of names to avoid so much code reuse, but this would widen the gap with the corresponding definition in the BitML paper.)

These can now be sensibly assumed to admit an encoding-decoding procedure:

```
postulate instance
  Serializable-Branchᶜ       : Serializable Branchᶜ
  Serializable-Preconditionᶜ : Serializable Preconditionᶜ
  Serializable-Adᶜ           : Serializable Adᶜ
```

In order to encode BitML contract advertisements, we will need to somehow bring them to this computational form first. But if we naively try to implement a translation function `Ad → Adᶜ`, we will quickly find out that it is impossible to define it; how are we going to come up with the transaction outputs that we need to refer to?

Instead, the correct formulation of encoding BitML contracts is to additionally provide a `Txout` mapping, as defined in Section 5.1.2; that way, we can query `txout` every time we encounter a symbolic name to retrieve the matching transaction output, which we can then substitute to get the computational translation:

```
reify : (∃ λ (ad : Ad) → Txout ad × Txout (ad .C)) → Adᶜ
reify = reifyᵃᵈ ∘ view

encodeAd : (ad : Ad) → Txout ad × Txout (ad .C) → HashId
encodeAd ad (txoutG , txoutC) = encode $ reify (ad , txoutG , txoutC)
```

In order to encode a BitML advertisement, we first *reify* it to a computational advertisement, and only then attempt to `encode` it.

Notice that the implementation of `reify` first *views* the contract along with its mappings as an intermediate representation, and then reifies that to get the final computational advertisement. Views are a technique for switching back and forth alternative representations of the same semantic entity, first introduced in functional programming [Wad87] and later adapted to dependent types [MM04], (*c.f.*, Appendix A.10 on how the Prelude provides a rustic framework to work with views and the associated laws that such translations should uphold.)

This intermediate form employs de Bruijn indices instead of identifiers to make our life a bit easier when implementing the translation, but we refer the reader to the full formalisation for more details on the exact translation and the associated views:

https://omelkonian.github.io/formal-bitml-to-bitcoin/Coherence.ComputationalContracts.html

In the other direction, decoding a computational advertisement does not simply return a symbolic advertisement, but also the associated mappings that gave rise to it (if decoding succeeds in the first place, that is).

```
decodeAd : HashId
         → Σ Ids (_↦ TxInput')
```

```
        → Maybe $ ∃ λ (ad : Ad) → Txout ad × Txout (ad .C)
decodeAd m (xs , txout)
 with decode {A = Adᶜ } m
... | nothing = nothing
... | just adᶜ
 with idsᶜ adᶜ ⊆? codom txout
... | no ins⊈ = nothing
... | yes ins⊆ = just $ abstractᶜ adᶜ (codom-↦ txout ∘ ins⊆)
```

6.1.3 Computational runs *(§6 & A.4 of [BZ18])*

For the rest of the development we will additionally assume that the abstract set of participants is *finite* (giving us a way to retrieve the set of `allParticipants`), as well as the existence of two key pairs `K` , `K̂` per participant, whose public parts will be broadcast as messages in the computational run. (Intuitively, `K̂` holds the private keys that participants use to authenticate themselves, while `K` are keys that have been generated off-chain just for the sake of executing the contract in question.)

```
 module ...
  (finPart : Finite Participant)
  (keyPairs : Participant → KeyPair × KeyPair)
  where
```

```
K K̂ : Participant → KeyPair
K = proj₁ ∘ keypairs
K̂ = proj₂ ∘ keypairs
```

Following the paper, we also define shorthands for retrieving the public or private parts of each key pair.

```
Kᵖ Kˢ K̂ᵖ K̂ˢ : Participant → HashId
Kᵖ = pub ∘ K
Kˢ = sec ∘ K
K̂ᵖ = pub ∘ K̂
K̂ˢ = sec ∘ K̂
```

We are now ready to define computational runs as a sequence of *computational labels*:

```
data Label : Type where
 _→∗:_ _→O:_ O→_:_ : Participant → Message → Label
 submit             : ∃Tx → Label
 delay              : Time → Label
```

```
Run = List Label
```

Participants can either broadcast messages throughout the Bitcoin network or exchange messages with the oracle to compute hashes. Apart from these three communication labels, we can `submit` a Bitcoin transaction to be appended to the current blockchain or advance time by performing a `delay`.

Recall that we defined a `Blockchain` to consist of a list of transactions paired with the time at which they occur. Summing up all the accumulated `delay` s of a computational run gives us a natural notion of 'current time', which can then be utilised to extract a concrete blockchain out of the transactions we have `submit`ted.

```
δʳ : Run → Time
δʳ = sum ∘ map λ where (delay t) → t; _ → 0

𝔹 : Run → Blockchain
𝔹 [] = []
𝔹 (l ∷ ls) = case l of λ where
  (submit tx) → tx at δʳ ls ∷ 𝔹 ls
  _           → 𝔹 ls
```

Not all sequences of such computational labels are valid though; valid sequences always start with a *coinbase transaction* having an output per participant that is locked with their private key `K̂`, followed by a sequence of *initial broadcasts* where participants share the public parts of their key pairs `Kᵖ` and `K̂ᵖ`.

```
Coinbase : Pred₀ ∃Tx
Coinbase (_ , _ , tx) =
  ∀ {A} → A ∈ allParticipants →
    (1 , ƛ (versig [ K̂ A ] [ 0F ])) ∈ (map₂′ validator <$> tx .outputs ∙toList)

initialBroadcasts : Run
initialBroadcasts = map go allParticipants
 module ∣initialBroadcasts∣ where
   go : Participant → Label
   go A = A →∗: encode (Kᵖ A , K̂ᵖ A)

instance
  Initial-Run : HasInitial Run
  Initial-Run .Initial R =
    ∃[ T₀ ] Coinbase T₀ × R ≡ submit T₀ ∷ initialBroadcasts
```

(We name the `where` module of `initialBroadcasts` so as to refer to the helper function when we prove lemmas about it later on.)

From now on, we will only deal with *valid* computational runs that have an initial run as a prefix, which can either be formulated *extrinsically* as a predicate on bare arbitrary

runs or *intrinsically* as an inductive type; we will choose the latter (`CRun`) as it matches more closely to the equivalent symbolic definition we will encounter in Section 6.2.1.

```
instance
  Valid-Run : Validable Run
  Valid-Run .Valid R = ∃[ R₀ ] (Initial R₀ × Suffix≡ R₀ R)

data CRun : Set where
  _█⊣_✓ : ∀ (R : Run) → Initial R → CRun
  _∷_✓  : Label → CRun → CRun
```

(All the predicates we defined above are decidable, but proving this, *i.e.,* coming up with the associated decision procedures, is quite straightforward, so we refrain from showing them here not to lose focus.)

6.2 Symbolic model *(§5 of [BZ18])*

6.2.1 Symbolic runs

Let us now turn our attention to *symbolic runs*, representing the moves performed on the BitML level. We have already seen how the operational semantics, presented as a binary reduction rule, naturally gives rise to reduction sequences via the reflexive transitive closure (*c.f.,* Section 4.7.5.5). If the source configuration is further constrained to a certain class of *initial* configurations, namely ones that occur at time $t = 0$ and only hold deposits, we arrive at a first definition of symbolic runs: traces stemming from the (closure of the) reduction relation of BitML's operational semantics.

```
Run = Trace _—↠ₜ_
```

(We refer the reader to Appendix A.9 for details on how the Prelude defines closures and traces.)

For instance, the example reduction of the timed commitment protocol we demonstrated in Section 4.8 can be readily viewed as a trace, since the source configuration we started with was indeed initial and *proof-by-reflection* is enough to prove this:

```
TC-run : Run
TC-run = record
  { start = (⟨ A has 1 ⟩at x ∣ ⟨ B has 0 ⟩at y) at 0
  ; init  = auto
  ; end   = (⟨ A has 1 ⟩at x₃ ∣ A : a ♯ N) at 0
  ; trace = -, TC-stepsₜ
  }
```

(Recall the auto construct, described in more detail in Appendix A.4, that use instance search to find the decision procedure of the current goal and then proves it by reflection/computation.)

One can now lift the reduction relation to apply to the most recent configuration in a run, or talk more generally about past transitions in the trace:

```
_——[_]→_ : Run → Label → Cfgᵗ → Type
R ——[ α ]→ tc′ = R .end —[ α ]→ₜ tc′

_≈⋯_ : Run → Cfgᵗ → Type
R ≈⋯ Γ at t = R .end ≈ Γ at t

_⋯∈_ : Cfg × Cfg → Run → Type
(Γ , Γ′) ⋯∈ R = (Γ , Γ′) ∈ allTransitions (R ∙trace′)

_⋯∈ᵗ_ : Cfgᵗ × Cfgᵗ → Run → Type
(Γₜ , Γₜ′) ⋯∈ᵗ R = (Γₜ , Γₜ′) ∈ allTransitionsᵗ (R ∙trace′)
```

Extending a run modifies the trace contained within, provided we give a proof that the source actually reduces to the target with respect to the BitML rules, as well as some ancillary proofs that take care of permuting the configurations as required by the context.

```
_⟨_⟩←——_⊣_ : ∀ Γₜ″ {Γₜ Γₜ′ : Cfgᵗ}
  → Γₜ —[ α ]→ₜ Γₜ′
  → (R : Run)
  → (Γₜ″ ≈ Γₜ′) × (R ≈⋯ Γₜ)
  ────────────────────────
    Run
Γₜ ⟨ Γ← ⟩←—— R@(record {trace = _ , Γ↞}) ⊣ eq =
  record R { end = Γₜ; trace = -, (Γₜ `⟨ Γ← ⟩←—ₜ eq ⊢ Γ↞) }
```

These permutation proofs will of course be automatically decidable when working with closed terms in examples later on, so a simpler notation is more appropriate in such cases:

```
_⟨_⟩←——_ : ∀ Γₜ″ {Γₜ Γₜ′}
  → Γₜ —[ α ]→ₜ Γₜ′
  → (R : Run)
  → {auto: Γₜ″ ≈ Γₜ′}
  → {auto: R ≈⋯ Γₜ} →
  ──────────────────
    Run
(Γₜ ⟨ Γ← ⟩←—— R) {p₁} {p₂} =
  Γₜ ⟨ Γ← ⟩←—— R ⊣ toWitness p₁ , toWitness p₂
```

Since there are a lot of moving parts that govern each step, we bundle them up in a single type 𝔸 (holding the next label, configuration, and permutation) to avoid cluttering

the definitions, and provide a base/step syntax akin to the one used for computational runs in Section 6.1.3.

```
𝔸 : Run → Cfgᵗ → Type
𝔸 R Γₜ =
  ∃ λ α → ∃ λ end′ → ∃ λ Γₜ′ →
    Σ (end′ —[ α ]→ₜ Γₜ′) λ Γ← →
      Γₜ ≈ Γₜ′ × R .end ≈ end′

_∎⊣_ : (Γₜ : Cfgᵗ) → Initial Γₜ → Run
Γₜ ∎⊣ init = record {start = Γₜ; end = Γₜ; trace = -, (Γₜ ∎ₜ); init = init}

_∷_⊣_ : (Γₜ : Cfgᵗ) (R : Run) → 𝔸 R Γₜ → Run
Γₜ ∷ R ⊣ (α , x , Γₜ′ , Γ← , eq) = _⟨_⟩←—_⊣_ {α} Γₜ {x}{Γₜ′} Γ← R eq
```

(The notation should resemble the traditional ‘nil-cons’ notation, but now augmented with proofs (the ⊣ parts) to suggest a list of some sort that is *well formed*.)

However, these symbolic runs contain arbitrary names for deposits, contracts, and secrets; these have to be accounted for, before coherence is able to relate them to their corresponding computational run.

Fortunately, this additional required information we need to provide takes the exact same form as the arguments to the BitML compiler we defined in Chapter 4, namely the `txout`, `sechash` and `κ` mappings which matched deposits to transaction outputs, secrets to their hashes, and participants to their key pairs, respectively. Hence the polymorphic definition of the mappings in Section 5.1.2, in order to later admit any type that might contain names and/or advertisements.

An alternative way to present the compiler arguments would be to *uncurry* the arguments into a well-defined advertisement, *i.e.*, one that is valid and is accompanied by the necessary mappings that are needed by compilation.

```
𝔾 : Pred₀ Ad
𝔾 ad = Valid ad × Txout (ad .G) × Sechash (ad .G) × 𝕂²′ ad
```

We explicate this construction in a generic family of *well-defined* values where such mappings are provided, and then instantiate this to configurations, timed or untimed, and eventually complete symbolic runs.

```
record 𝕎 {X : Type} ⦃ _ : X has Name ⦄ ⦃ _ : X has Ad ⦄ (x : X) : Type where
 constructor [txout:_∣sechash:_∣κ:_]
 field
  txout   : Txout x
  sechash : Sechash x
  κ       : 𝕂²     x
```

```
ℾ = Pred₀ Cfg ∋ 𝕎
module ℾ (𝕪 : ℾ Γ) where
  open 𝕎 𝕪 public renaming (txout to txoutΓ; sechash to sechashΓ; κ to κΓ)

ℾᵗ = Pred₀ Cfgᵗ ∋ 𝕎
module ℾᵗ (𝕪 : ℾᵗ Γₜ) where
  open 𝕎 𝕪 public renaming (txout to txoutΓ; sechash to sechashΓ; κ to κΓ)

ℝ = Pred₀ Run ∋ 𝕎
module ℝ (𝕣 : ℝ R) where
  open 𝕎 𝕣 public renaming (txout to txout′; sechash to sechash′; κ to κ′)
```

(We take care to name configuration mappings differently than run mappings, in order to avoid name clashes down the road.)

We are finally ready to give the definition of well-defined symbolic runs, starting from an initial (timed) configuration with the necessary mappings attached, and inductively extending the trace with subsequent configurations along with their new mappings, now updated to reflect changes in the names and advertisments occurring in the current run.

```
𝕃 : Run → Cfgᵗ → Type
𝕃 R Γₜ = Σ[ 𝕒 ∈ 𝔸 R Γₜ ] ℾᵗ (𝕒 .proj₂ .proj₂ .proj₁)

data ℝ∗ : Run → Type where
  _█⊣_✓ : ∀ {Γₜ} →

    • ℾᵗ Γₜ
    → (init : Initial Γₜ) →
    ─────────────────────
    ℝ∗ (Γₜ █⊣ init)

  _∷_⊣_✓ :
    ∀ Γₜ →
    • ℝ∗ R
    → (λˢ : 𝕃 R Γₜ) →
    ─────────────────────
    ℝ∗ (Γₜ ∷ R ⊣ λˢ .proj₁)

SRun : Type
SRun = ∃ ℝ∗
```

One might wonder why we need mappings for the whole history rather than simply the last configuration; it turns out that certain cases for the inductive step of coherence need to refer to past compiler invocations and therefore need access to the mappings used at that previous point in time.

Constructing these mappings as needed throughout the definition of coherence is far from straightforward, so we need to first need to take a brief detour in order to set up some infrastructure that will help us in the following respects:

- some general constructions on mappings that will prove indispensable when defining the inductive cases of coherence (Section 6.2.2),
- being able to retrieve a previously fired rule of BitML's operational semantics by observing the latest state of a symbolic run (Section 6.2.3),
- and finally utilising the information above to construct the necessary mappings that lets us invoke the BitML compiler in various scenarios (Section 6.2.4).

6.2.2 Properties of symbolic mappings

With all the permutations going on behind the scenes of almost every transition on the BitML level, we will often need to transfer mappings between configurations that are equivalent up to permutation. Thankfully, the Prelude already provides enough machinery and lemmas to express these concisely:

```
Txout≈    : _≈_ ⇛² _→⟨ Txout ⟩_
Txout≈    {Γ}{Γ′} = permute-↦ {P = const TxInput′} ∘ ≈⇒namesʳ↭ {Γ}{Γ′}

Sechash≈ : _≈_ ⇛² _→⟨ Sechash ⟩_
Sechash≈ {Γ}{Γ′} = permute-↦ ∘ ≈⇒namesˡ↭ {Γ}{Γ′}

𝕂²≈       : _≈_ ⇛² _→⟨ 𝕂² ⟩_
𝕂²≈       {Γ}{Γ′} = permute-↦ ∘ ≈⇒ads↭ {Γ}{Γ′}

ℾ≈ : Γ ≈ Γ′ → ℾ Γ → ℾ Γ′
ℾ≈ {Γ}{Γ′} Γ≈ ɣ =
  [txout:   Txout≈ {Γ}{Γ′} Γ≈ txoutΓ
  ;sechash: Sechash≈ {Γ}{Γ′} Γ≈ sechashΓ
  ;κ:       𝕂²≈      {Γ}{Γ′} Γ≈ κΓ
  ] where open ℾ ɣ
```

Notation

Collection transformers. The Prelude provides utilities for collections (*c.f.*, Appendix A.6), amongst which the syntax for transformers between (possibly) different types that hold the same kind of resources; we write `X →⟨ Txout ⟩ Y` to mean `Txout X → Txout Y`.

Furthermore, we typically work with mappings acting on different types, so we need to be able to transfer effortlessly between them. The definitions are not interesting enough to include here, but the type signatures will clarify things later on.

```
ℾᵗ⇒ℾ   : ℾᵗ (Γ at t) → ℾ Γ
ℾ⇒ℾᵗ   : ℾ Γ → ℾᵗ (Γ at t)
```

```
𝕃⇒𝔸    : 𝕃 R Γₜ → 𝔸 R Γₜ
𝕃⇒Γᵗ   : 𝕃 R Γₜ → Γᵗ Γₜ
𝕃⇒Γ    : 𝕃 R (Γ at t) → Γ Γ
ℝ∗⇒Γᵗ  : ℝ∗ R → Γᵗ (R .end)
ℝ∗⇒Γ   : ℝ∗ R → Γ (R •cfg)
ℝ∗⇒ℝ   : ℝ∗ R → ℝ R
```

All of the above either work on resources (*i.e.,* names and advertisements) between *identical* structures, *permutations* of them, or at most *immediately smaller* structures (*e.g.,* from a mappings for an entire run, extracting the sub-mapping of its latest configuration), although sometimes we need to get to an *intermediate* configuration.

```
Txout∈ : Txout R → Γ ∈ allCfgs R → Txout Γ
Txout∈ txout Γ∈
  = txout
  ∘ mapMaybe-⊆ isInj₂ (⊆-concat⁺ (L.Mem.∈-map⁺ collect Γ∈))
```

```
Sechash∈ : Sechash R → Γ ∈ allCfgs R → Sechash Γ
Sechash∈ sechash Γ∈
  = sechash
  ∘ mapMaybe-⊆ isInj₁ (⊆-concat⁺ (L.Mem.∈-map⁺ collect Γ∈))
```

More importantly, there will be eventual need for extending mappings to a *larger* structure, and we better do so in a *compositional* manner, *i.e.,* construct the mappings for a symbolic step out of the information we already have for the previous run and new mappings for the current configuration.

```
module _
  (Γ→ : Γₜ —[ α ]→ₜ Γₜ′)
  (eq : Γₜ″ ≈ Γₜ′ × R .end ≈ Γₜ) (let Γ≈ = eq .proj₁ .proj₂)
  where

  txout∷ : • Txout Γₜ′
           • Txout R
           ────────────────────────────
           Txout (Γₜ″ ⟨ Γ→ ⟩←—— R ⊣ eq)
  txout∷ txoutΓ′ txoutR
    = subst (_↦ TxInput′) (sym $ namesʳ-←—— {Γₜ = Γₜ″} {R = R} Γ→ eq)
    $ txoutR ++/↦ Txout≈ {x = cfg Γₜ′}{cfg Γₜ″} (↭-sym Γ≈) txoutΓ′

  sechash∷ : • Sechash Γₜ′
             • Sechash R
             ────────────────────────────
             Sechash (Γₜ″ ⟨ Γ→ ⟩←—— R ⊣ eq)
  sechash∷ sechashˡ sechashʳ
    rewrite namesˡ-←—— {Γₜ = Γₜ″} {R = R} Γ→ eq
```

```
    =       sechashʳ ++/↦ (Sechash≈ {x = cfg Γₜ′}{cfg Γₜ″} (↭-sym Γ≈) sechashˡ)
  κ:: : • 𝕂² Γₜ′
         • 𝕂² R
        ──────────────────────
         𝕂² (Γₜ″ ⟨ Γ→ ⟩←—— R ⊣ eq)
  κ:: κˡ κʳ
    rewrite ads-←—— {Γₜ = Γₜ″} {R = R} Γ→ eq
    =       κʳ ++/↦ (𝕂²≈ {x = cfg Γₜ′}{cfg Γₜ″} (↭-sym Γ≈) κˡ)
```

That way, we are able to inductively grow the mappings for symbolic runs:

```
ℝ-base : ∀ {init : Initial Γₜ} → ℾᵗ Γₜ → ℝ (Γₜ ■⊣ init)
ℝ-base {init = i} ℽ =
  [txout:    substʳ (_↦ TxInput′) (namesʳ-■ {init = i}) txoutΓ
  |sechash:  substʳ (_↦ ℤ)        (namesˡ-■ {init = i}) sechashΓ
  |κ:        substʳ (_↦′ 𝕂²′)     (ads-■ {init = i})    κΓ
  ] where open ℾᵗ ℽ

ℝ-step : ℝ R → (λˢ : 𝕃 R Γₜ) → ℝ (Γₜ ∷ R ⊣ λˢ .proj₁)
ℝ-step {R = R} 𝕣 ((_ , _ , _ , Γ→ , eq) , ℽ) =
  [txout:    txout::   {R = R} Γ→ eq txoutΓ   txout′
  |sechash:  sechash:: {R = R} Γ→ eq sechashΓ sechash′
  |κ:        κ::       {R = R} Γ→ eq κΓ       κ′
  ] where open ℝ 𝕣; open ℾᵗ ℽ
```

At the end of the day, even with all the aforementioned machinery in place, we will need to provide updated mappings for each possible transition in BitML's operational semantics. Each case will be different, involving different source and target configurations, that consequently means that the involved resources will need to be accounted for by the mappings. One simple abstraction that will come in handy in each of these cases, is to construct the updated mappings for the target configuration by providing *mapping transformers* from the source, henceforth called 'lifting'.

```
LIFTˢ : ∀ {R}{t}{t′} (r : ℝ R) Γ (R≈ : R ≈⋯ Γ at t) Γ′ →
  • Γ →⦅ Txout ⦆ Γ′
  • Γ →⦅ Sechash ⦆ Γ′
  • Γ →⦅ 𝕂² ⦆ Γ′
  ────────────────
  ℾᵗ (Γ′ at t′)
LIFTˢ {R} r Γ (_ , Γ≈) Γ′ txout↝ sechash↝ κ↝
  = [txout:    txout↝
               $ Txout≈   {R ∙cfg}{Γ} Γ≈ (weaken-↦ txout′   $ namesʳ⦅end⦆⊆ R)
    |sechash:  sechash↝
               $ Sechash≈ {R ∙cfg}{Γ} Γ≈ (weaken-↦ sechash′ $ namesˡ⦅end⦆⊆ R)
    |κ:        κ↝
```

```
            $ 𝕂²≈       {R •cfg}{Γ} Γ≈ (weaken-↦ κ′       $ ads⦅end⦆⊆ R)
  ] where open ℝ r
```

The woes of meta-properties. At some point we will need to reason about the concrete values returned by all the constructions on mappings we just described. This is surprisingly tricky, as we relied on lemmas (about list permutations, *etc.*) already provided by the standard library, that are *computationally relevant*, *i.e.,* we actually care about the results they compute. Therefore, we now need to prove higher-level properties about those level-1 properties (*e.g.,* how these lemmas commute with one another), which we can call "meta-properties". Alas, the standard library does not provide these (and for good reason, it is quite a rare occasion after all), so a fair chunk of the Prelude is dedicated to those (*c.f.,* Appendix A.13).

To save space and the reader's despair, we will not include these proofs here, but still find it informative to see one such example type signature, namely stating the fact that successively permuting a `Txout` mapping with two opposite permutations `Txout≈` leaves the values of the resulting mapping unchanged:

```
Txout≈∘Txout≈⁻¹ : (Γ≈ : Γ ≈ Γ′) (txout : Txout Γ) →
  (Txout≈ {Γ′}{Γ} (↭-sym Γ≈) ∘ Txout≈ {Γ}{Γ′} Γ≈) txout ≗↦ txout
```

6.2.3 BitML's trace properties

Apart from manipulating mappings by permuting, shrinking, and extending them, eventually we will need to populate their values, sometimes involving information that is not currently present in the current configuration, but rather depends on previous events that took place and ultimately stem from the order that the BitML rules fire by their very definition. In particular, we will need to detect all the following cases:

- When we see a contract advertisement (`` ` ad``) in the current configuration, trace back the previous point in the run where it got advertised via the rule `[C-Advertise]`.
- Similarly for authorisations performed by the participants: observing `A auth[ ♯▷ ad ]` means that the `C-AuthCommit` rule has occurred previously; observing `A auth[ x ▷ˢ ad ]` justifies a previous use of the `C-AuthInit` rule; and a configuration with `A auth[ x ▷ d ]` is the result of some previous `C-AuthControl` .
- If we come across an active contract (`⟨ C , v ⟩at x`), we wish to track its whole *lifetime*, consisting of all the above leading to an initial `C-Init` rule that stipulates

some *ancestor* contract `C′` and associated advertisement `⟨G⟩C′`. This is slightly more complicated than the previous cases, since we need to recurse on arbitrary sub-traces, rather than structurally smaller ones, hence the need for a well-founded induction on the trace length.

The proofs of the above are quite lengthy in contrast to simpler properties like *invariants*, which we could possibly prove using mere *rule induction*, *i.e.,* pattern matching on a single given transition, and then proving that the base cases satisfy the invariant and the inductive steps preserve it. Rule induction is not, however, adequate for such *trace properties* involving multiple points in the trace of a run and how they are inter-connected. (These are not to be confused with predicates on whole execution traces, as commonly employed in the field of Computational Security [Roy+10], or even *hyperproperties* that relate multiple traces together [CS08].) Instead, we need to follow a more intricate proof methodology, where we recurse to a specific *salient* point of a trace where a change that we care about occurs, and then employ rule induction as usual to derive the desired conclusion.

Instead of presenting the gory details of all of these cases (and bore the reader to death along the way), we will only demonstrate one of them in detail as they all share the same proof outline. The most sensible choice for that purpose is the last one, *i.e.,* tracing a contract's lifetime, as its reliance on the 'ancestor' relation make it more tricky (requiring well-founded recursion), but this is not explained in the original paper [BZ18] (where it is only mentioned briefly in passing without any explanation in A.5), so we deemed it useful to document somewhere.

Exceptionally, the code that follows resides in the same place as Chapter 4 on BitML:

`https://omelkonian.github.io/formal-bitml/`

That is morally the rightful place for such properties, as they can be thought of purely as part of the meta-theory of BitML, in no way depending on the concerns of compiling contracts to Bitcoin. Introducing them now, however, is a conscious choice on our part, as we did not want to overburden the reader too early on, and only unveil such complications when they would become absolutely necessary.

6.2.3.1 The general proof methodology

To first understand the whole process, let us consider the simplest case of tracing an initial advertisement and just see the proof outline without any distracting details.

Our ulterior motive will be to retrieve the hypotheses of the `[C-Advertise]` rule, given an existing advertisement in the current configuration. To this end, we have to

define a relation externalising said hypotheses, which can be trivially constructed if we have a transition with the appropriate label:

```
ℍ[C-Advertise]⦅_⇝_⦆⦅_⦆ : Cfg → Cfg → Ad → Type
ℍ[C-Advertise]⦅ Γ ⇝ Γ′ ⦆⦅ ad ⦆ = let open ∣AD ad in
   (Γ′ ≡ ` ad ∣ Γ)
  × ValidAd ad
  × Any (_∈ Hon) partG
  × deposits ad ⊆ deposits Γ

advertise⇒ :
  Γ —[ advertise⦅ ad ⦆ ]→ Γ′
  ─────────────────────────────
  ℍ[C-Advertise]⦅ Γ ⇝ Γ′ ⦆⦅ ad ⦆
advertise⇒ ([C-Advertise] vad hon⁺ d⊆) = refl , vad , hon⁺ , d⊆
```

Now the task becomes to search a given trace for a single transition that occurs somewhere in the middle, carrying the desired label. Of course, the existence of such a label will be predicated on some conditions on the source and target configuration of the run, in this case that the target contains an advertisement `` ` ad`` that did not exist in the source:

```
traceAd :
  • ` ad ∉ᶜ Γ₀
  • ` ad ∈ᶜ Γ
  • Γ₀ at t —[ αs ]↠ₜ Γ at t′
    ─────────────────────────
    advertise⦅ ad ⦆ ∈ αs
```

The proof of the above will recurse (structurally) back through the trace's history to find the single transition that gave rise to the label, *i.e.*, one where the conditions hold for the source and target configuration (within a single step). At this point, we can complete the proof by *rule induction*; pattern matching on the transition we found and excluding all other transition because they are labelled differently, as well as `[C-Advertise]` transitions that involve a different advertisement. To save space, we omit the proofs and only show the type of the helper lemma that performs the rule induction:

```
traceAd ad∉ ad∈ (_ ∎ₜ) = ⊥-elim $ ad∉ ad∈
traceAd {ad}{Γ₀}{Γ}{t}{α ∷ αs}{t′} ad∉ ad∈
  (_—→ₜ⟨_⟩_⊢_ .(Γ₀ at t) {Γ₀′ at _}{M at _}{M′ at _}{Γ at t′}
                Γ₀→M ((refl , Γ₀≈) , (refl , M≈)) M↠)
  = case ¿ ` ad ∈ᶜ M′ ¿ of λ where
      (yes ad∈M′) → here $ sym $ go (∉ᶜ-resp-≈ {Γ₀}{Γ₀′} Γ₀≈ ad∉) ad∈M′ Γ₀→M
      (no ad∉M′) → there $ traceAd (ad∉M′ ∘ ∈ᶜ-resp-≈ {M}{M′} M≈) ad∈ M↠
  where
  go :
    • ` ad ∉ᶜ Γ
```

```
  • ` ad ∈ᶜ Γ′
  • Γ at t —[ α ]→ₜ Γ′ at t′
  ───────────────────────────
    α ≡ advertise⟪ ad ⟫
```

(See Appendix A.4 for an explanation of the ¿ quotations to retrieve a decision procedure of a given type.)

This is all that is needed to give the final tracing theorem for advertising: observing an advertisement in the current configuration of a trace that starts from an initial configuration, lets us retrieve a specific [C-Advertise] transition between adjacent configurations in the middle of the trace and recover its corresponding hypotheses.

```
traceAd∗ :
  Initial Γ₀ →
  ` ad ∈ᶜ Γ →
  (tr : Γ₀ at t —[ αs ]↠ₜ Γ at t′) →
  ─────────────────────────────────────
  ∃[ tr ∋ ℍ[C-Advertise]⦅_↝_⦆⟪ ad ⟫ ]
traceAd∗ init ad∈ Γ↠ = advertise⇒∗ Γ↠ $ traceAd (Initial⇒∉ init) ad∈ Γ↠
  where
    advertise⇒∗ :
      ∀ (tr : Γₜ —[ αs ]↠ₜ Γₜ′) →
      • advertise⟪ ad ⟫ ∈ αs
      ─────────────────────────────────────
        ∃[ tr ∋ ℍ[C-Advertise]⦅_↝_⦆⟪ ad ⟫ ]
    advertise⇒∗ Γ↠ α∈
      with _ , _ , _ , _ , xy∈ , ((_ , x≈) , (_ , y≈)) , [Action] Γ→ refl ← zoom Γ↠ α∈
        = -, -, -, -, ∈-map⁺ (map₁₂ cfg) xy∈ , (x≈ , y≈) , advertise⇒ Γ→
```

Notation

Predicates on adjacent configurations. The syntax `∃[ tr ∋ P ]` checks that, somewhere in the middle of the transition sequence `tr`, we find two adjacent configurations `Γᵢ` and `Γᵢ₊₁` that are related under `P : Cfg → Cfg → Type`, possibly after permuting them appropriately, *i.e.*, `tr` is of the form `Γ₀→⋯→Γᵢ→Γᵢ₊₁⋯→`.

A similar notation will later be used for traces (sequences that start from an initial configuration) in Section 6.2.3.3.

6.2.3.2 Tracing active contracts throughout their lifetime

The case of tracing active contracts is more complicated, as it is not enough to identify a single transition in the previous trace, but rather the full 'lifetime' of a contract, tracking how the contract has evolved over time.

To start with, instead of simply externalising the hypotheses of a single rule, we need to define a relation that captures the most recent evolution of the contract, which might arise due to several reasons:

- A [C-Init] rule stipulated the current contract.
- The current contract is one of the branches/clauses of a put/split command, arising from a [C-PutRev] or [C-Split] rule, respectively.
- The current contract is one of the branches chosen from a [C-Control] rule (or equivalently a [C-Timeout] rule in the timed layer).

```
data ℍ[Contract]⦅_—[_]↝_⦆⦅_⦆ : Cfg → Label → Cfg → Contract → Type where

 base :
  • ` (⟨ g ⟩ c) ∈ᶜ Γ
  • Γ —[ init⦅ g , c ⦆ ]→ Γ′
   ──────────────────────────────────────
   ℍ[Contract]⦅ Γ —[ init⦅ g , c ⦆ ]↝ Γ′ ⦆⦅ c ⦆

 step-put :
  ──────────────────────────────────────────
  ℍ[Contract]⦅
   ⟨ [ put xs &reveal as if p ⇒ c ] , v ⟩at y ∣ Γ —[ put⦅ xs , as , y ⦆ ]↝ Γ′
  ⦆⦅ c ⦆

 step-split :
  c ∈ map proj₂ vcs
  ────────────────────────────────────────────────────────────
  ℍ[Contract]⦅ ⟨ [ split vcs ] , v ⟩at y ∣ Γ —[ split⦅ y ⦆ ]↝ Γ′ ⦆⦅ c ⦆

 step-control : ∀ {i : Index c′} → let open ∣SELECT c′ i in
  • Γ ≈ L
  • cv α ≡ just x
  • ℍ[Contract]⦅ ⟨ [ d∗ ] , v ⟩at x ∣ L —[ α ]↝ Γ′ ⦆⦅ c ⦆
   ──────────────────────────────────────────────────────────────
   ℍ[Contract]⦅
    ⟨ c′ , v ⟩at x ∣ || map _auth[ x ▷ d ] (nub $ authDecorations d) ∣ Γ —[ α ]↝ Γ′
   ⦆⦅ c ⦆

 step-timeout : ∀ {i : Index c′} → let open ∣SELECT c′ i in
  • cv α ≡ just x
  • ℍ[Contract]⦅ ⟨ [ d∗ ] , v ⟩at x ∣ Γ —[ α ]↝ Γ′ ⦆⦅ c ⦆
   ──────────────────────────────────────────────
   ℍ[Contract]⦅ ⟨ c′ , v ⟩at x ∣ Γ —[ α ]↝ Γ′ ⦆⦅ c ⦆
```

We then proceed as before to find the transition that justifies the creation of the active contract in question:

```
traceContractₜ :
  • ⟨ c , v ⟩at y ∉ᶜ Γ₀
  → ⟨ c , v ⟩at y ∈ᶜ Γ
  → ∀ (tr : Γ₀ at t —[ αs ]↠ₜ Γ at t′) →
    ────────────────────────────────────────
    ∃[ tr ∋ ∃ℍ[Contract]⦅_⇝_⦆⦅ c ⦆ ]
```

This is done by recursing through the trace and applying rule induction; we have to have each possible transition and dismiss all irrelevant cases separately by appealing to the form the source and target configurations.

Try not to get overwhelmed by the size of the following statements and focus on the structure of rule induction we just mentioned, but since we promised to show the case of active contracts in excruciating detail, here is the complete proof (up to definitions of local helper lemmas):

```
private
  ¬AuthControl : • ⟨ c , v ⟩at x ∉ᶜ Γ
                 • Γ —[ auth-control⦅ A , x′ ▷ d ⦆ ]→ Γ′
                 ──────────────────────────────────────────
                 ⟨ c , v ⟩at x ∉ᶜ Γ′
  ¬AuthRev : • ⟨ c , v ⟩at x ∉ᶜ Γ
             • Γ —[ auth-rev⦅ A , a ⦆ ]→ Γ′
             ──────────────────────────────
             ⟨ c , v ⟩at x ∉ᶜ Γ′
  ¬AuthJoin : • ⟨ c , v ⟩at y ∉ᶜ Γ
              • Γ —[ auth-join⦅ B , x ↔ y′ ⦆ ]→ Γ′
              ────────────────────────────────────
              ⟨ c , v ⟩at y ∉ᶜ Γ′
  ¬Join : • ⟨ c , v ⟩at x ∉ᶜ Γ
          • Γ —[ join⦅ x′ ↔ y ⦆ ]→ Γ′
          ───────────────────────────
          ⟨ c , v ⟩at x ∉ᶜ Γ′
  ¬AuthDivide : • ⟨ c , v ⟩at x ∉ᶜ Γ
                • Γ —[ auth-divide⦅ A , x′ ▷ v″ , v′ ⦆ ]→ Γ′
                ────────────────────────────────────────────
                ⟨ c , v ⟩at x ∉ᶜ Γ′
  ¬Divide : • ⟨ c , v ⟩at x ∉ᶜ Γ
            • Γ —[ divide⦅ x′ ▷ v″ , v′ ⦆ ]→ Γ′
            ───────────────────────────────────
            ⟨ c , v ⟩at x ∉ᶜ Γ′
  ¬AuthDonate : • ⟨ c , v ⟩at x ∉ᶜ Γ
                • Γ —[ auth-donate⦅ A , x′ ▷ᵈ B ⦆ ]→ Γ′
```

```
                 ────────────────────────────────────────
                 ⟨ c , v ⟩at x ∉ᶜ Γ′
¬Donate : • ⟨ c , v ⟩at x ∉ᶜ Γ
          • Γ —[ donate⦅ x′ ▷ᵈ B ⦆ ]→ Γ′
             ──────────────────────────────
             ⟨ c , v ⟩at x ∉ᶜ Γ′
¬AuthDestroy : ∀ {j′ : Index xs} →
  • ⟨ c , v ⟩at x ∉ᶜ Γ
  • Γ —[ auth-destroy⦅ A , xs , j′ ⦆ ]→ Γ′
    ─────────────────────────────────────────
    ⟨ c , v ⟩at x ∉ᶜ Γ′
¬Destroy : • ⟨ c , v ⟩at x ∉ᶜ Γ
           • Γ —[ destroy⦅ xs ⦆ ]→ Γ′
              ──────────────────────────
              ⟨ c , v ⟩at x ∉ᶜ Γ′
¬Advertise : • ⟨ c , v ⟩at x ∉ᶜ Γ
             • Γ —[ advertise⦅ ad ⦆ ]→ Γ′
                ────────────────────────────
                ⟨ c , v ⟩at x ∉ᶜ Γ′
¬AuthCommit : ∀ {secrets} →
  • ⟨ c , v ⟩at x ∉ᶜ Γ
  • Γ —[ auth-commit⦅ A , ad , secrets ⦆ ]→ Γ′
    ─────────────────────────────────────────────
    ⟨ c , v ⟩at x ∉ᶜ Γ′
¬AuthInit : • ⟨ c , v ⟩at x ∉ᶜ Γ
            • Γ —[ auth-init⦅ A , ad , x′ ⦆ ]→ Γ′
               ──────────────────────────────────────
               ⟨ c , v ⟩at x ∉ᶜ Γ′
¬Init : • ⟨ c , v ⟩at x ∉ᶜ Γ
        • c ≢ c′
        • Γ —[ init⦅ g , c′ ⦆ ]→ Γ′
           ───────────────────────────
           ⟨ c , v ⟩at x ∉ᶜ Γ′
¬Withdraw : • ⟨ c , v ⟩at x ∉ᶜ Γ
            • Γ —[ withdraw⦅ A , v′ , y ⦆ ]→ Γ′
               ─────────────────────────────────────
               ⟨ c , v ⟩at x ∉ᶜ Γ′
¬Withdraw c∉ ([C-Withdraw] {x}{y}{Γ}{A}{v} _) =
  ∈-++⁻ [ ⟨ A has v ⟩at x ] >≡>
  Sum.[ (λ{ (here ()) })
      , c∉ ∘ ∈-++⁺ʳ [ ⟨ [ withdraw A ] , v ⟩at y ]
      ]
c∉→ : let ⟨C,v⟩ₓ = ⟨ [ d ] , v ⟩at x in
  • ⟨C,v⟩ₓ ∉ᶜ L
```

```
        • cv α ≡ just x
        • ⟨C,v⟩ₓ ∣ L —[ α ]→ Γ′
          ────────────────────────
          ⟨C,v⟩ₓ ∉ᶜ Γ′
h : • ⟨ c , v ⟩at y ∉ᶜ Γ
    • ⟨ c , v ⟩at y ∈ᶜ Γ′
    • Γ —[ α ]→ Γ′
      ──────────────────────────────
      ℍ[Contract]⦅ Γ —[ α ]↝ Γ′ ⦆⦅ c ⦆
h {c₀}{v₀}{y₀}{Γ}{Γ′}{α = α} c∉ c∈ step
  with α | step
... | advertise⦅ _ ⦆                | st = ⊥-elim $ ¬Advertise c∉ st c∈
... | auth-join⦅ _ , _ ↔ _ ⦆        | st = ⊥-elim $ ¬AuthJoin c∉ st c∈
... | join⦅ _ ↔ _ ⦆                 | st = ⊥-elim $ ¬Join c∉ st c∈
... | auth-divide⦅ _ , _ ▷ _ , _ ⦆  | st = ⊥-elim $ ¬AuthDivide c∉ st c∈
... | divide⦅ _ ▷ _ , _ ⦆           | st = ⊥-elim $ ¬Divide c∉ st c∈
... | auth-donate⦅ _ , _ ▷ᵈ _ ⦆     | st = ⊥-elim $ ¬AuthDonate c∉ st c∈
... | donate⦅ _ ▷ᵈ _ ⦆              | st = ⊥-elim $ ¬Donate c∉ st c∈
... | auth-destroy⦅ _ , _ , _ ⦆     | st = ⊥-elim $ ¬AuthDestroy c∉ st c∈
... | destroy⦅ _ ⦆                  | st = ⊥-elim $ ¬Destroy c∉ st c∈
... | auth-commit⦅ _ , _ , _ ⦆      | st = ⊥-elim $ ¬AuthCommit c∉ st c∈
... | auth-init⦅ _ , _ , _ ⦆        | st = ⊥-elim $ ¬AuthInit c∉ st c∈
... | auth-rev⦅ _ , _ ⦆             | st = ⊥-elim $ ¬AuthRev c∉ st c∈
... | withdraw⦅ _ , _ , _ ⦆         | st = ⊥-elim $ ¬Withdraw c∉ st c∈
... | auth-control⦅ _ , _ ▷ _ ⦆     | st = ⊥-elim $ ¬AuthControl c∉ st c∈
... | delay⦅ _ ⦆                    | st = ⊥-elim $ ¬Delay st
... | put⦅ xs , as , y ⦆ | [C-PutRev] {Γ′}{z}{.y}{p}{c′}{v′} {ds = ds}{ss} fresh-z _
    = case c∈ of λ where
    (here refl) → step-put
    (there c∈′) → let (_ , vs , xs) = unzip₃ ds
                      (_ , as , _)  = unzip₃ ss
                      Γ    = || map (uncurry₃ ⟨_has_⟩at_) ds
                      Δ    = || map (uncurry₃ _:_♯_) ss
                      ΔΓ′ = Δ ∣ Γ′
                  in ⊥-elim $ c∉ $ there $′ ∈ᶜ-++⁺ʳ Γ ΔΓ′ c∈′
... | split⦅ y ⦆ | [C-Split] {.y}{Γ}{vcis} _
    = let (vs , cs , _) = unzip₃ vcis
          vcs = zip vs cs
      in case ∈ᶜ-++⁻ (|| map (uncurry₃ $ flip ⟨_,_⟩at_) vcis) Γ c∈ of λ where
          (inj₁ c∈Δ) → step-split {vcs = vcs} (c∈vcis⇒′ {vcis = vcis} c∈Δ)
          (inj₂ c∈Γ) → ⊥-elim $ c∉ $ there $ c∈Γ
... | _ | [C-Control] {c}{Γ}{L}{v}{x}{α}{Γ′}{i} p Γ≈ Γ→ cv≡
    = let d = c ‼ i; d∗ = removeTopDecorations d; As = nub $ authDecorations d
        c∉L : ⟨ c₀ , v₀ ⟩at y₀ ∉ᶜ L
```

```
          c∉L = c∉ ∘ ∈ᶜ-++⁺ʳ (⟨ c , v ⟩at x ∣ || map _auth[ x ▷ d ] As) Γ
                ∘ ∈ᶜ-resp-≈ {L}{Γ} (↭-sym Γ≈)
      in step-control Γ≈ cv≡ $ h (∈ᶜ-++⁻ (⟨ [ d∗ ] , v ⟩at x) L >≡>
          Sum.[ (λ{ (here refl) → c∉→ c∉L cv≡ Γ→ c∈ }) , c∉L ]) c∈ Γ→
 ... | init⦅ g , c′ ⦆ | st@([C-Init] _)
     = case ¿ c₀ ≡ c′ ¿ of λ where
        (no ¬eq)  → ⊥-elim $ ¬Init c∉ ¬eq st c∈
        (yes refl) → base (here refl) st

 hᵗ :  • ⟨ c , v ⟩at y ∉ᶜ Γ
       • ⟨ c , v ⟩at y ∈ᶜ Γ′
       • Γ at t —[ α ]→ₜ Γ′ at t′
       ────────────────────────────────
       ℍ[Contract]⦅ Γ —[ α ]↝ Γ′ ⦆⦅ c ⦆
 hᵗ c∉ c∈ ([Action] Γ→ _) = h c∉ c∈ Γ→
 hᵗ {Γ = Γ} c∉ c∈ ([Delay] _) = ⊥-elim $ c∉ c∈
 hᵗ {c₀}{v₀}{x₀} c∉ c∈ ([Timeout] {c}{t}{v}{x}{Γ}{α}{Γ′}{i} _ _ Γ→ cv≡) =
   step-timeout cv≡ $ h c∉′ c∈ Γ→
   where open ∣SELECT c i
         c∉Γ : ⟨ c₀ , v₀ ⟩at x₀ ∉ᶜ Γ
         c∉Γ = c∉ ∘ ∈ᶜ-++⁺ʳ (⟨ c , v ⟩at x) Γ
         c∉′ : ⟨ c₀ , v₀ ⟩at x₀ ∉ᶜ ⟨ [ d∗ ] , v ⟩at x ∣ Γ
         c∉′ = λ where (here refl) → c∉→ c∉Γ cv≡ Γ→ c∈
                       (there c∈Γ) → c∉Γ c∈Γ
traceContractₜ {Γ₀ = Γ} c∉ c∈ (_ ∎ₜ) = ⊥-elim $ c∉ c∈
traceContractₜ {c}{v}{y}{Γ₀}{Γ}{t}{α ∷ αs}{t′} c∉ c∈
  (_—→ₜ⟨_⟩_⊢_ _ {Γ₀′ at _}{M at _}{M′ at _} Γ₀→M ((refl , Γ₀≈) , (refl , M≈)) M↠)
  with ¿ ⟨ c , v ⟩at y ∈ᶜ M′ ¿
... | no c∉M′ =
  let _ , _ , _ , _ , x∈ , H = traceContractₜ (∉ᶜ-resp-≈ {M′}{M} (↭-sym M≈) c∉M′) c∈ M↠
  in -, -, -, -, there x∈ , H
... | yes c∈M′
    = Γ₀ , Γ₀′ , M , M′ , here refl , (Γ₀≈ , M≈)
    , α , hᵗ (∉ᶜ-resp-≈ {Γ₀}{Γ₀′} Γ₀≈ c∉) c∈M′ Γ₀→M
```

This is the point where we would be finished in all other cases of tracing, but in this particular instance we care about the full lifetime of a contract, not just its latest evolution. Therefore, what is further required is an **ancestor** relation with respect to the initial advertisement that gave rise to the current contract (possibly with many intermediate steps in between), whose inhabitants gives us enough information to reconstruct the contract's evolution, *i.e.*, its 'lifetime'.

```
Ancestor⦅_↝_⦆ : Ad → Contract → Rel₀ Cfg
Ancestor⦅ ad ↝ c ⦆ Γ Γ′ =
  ℍ[C-Init]⦅ Γ ↝ Γ′ ⦆⦅ ad ⦆
  × (ad •↝∗ c)
```

The actual lifetime relation `•⇝*` is not that interesting; it essentially identifies the branching points of a contract (`put` and `split` commands) and chains them together to represent the entire evolution of a contract. We refer the reader to the full formalisation here:

https://omelkonian.github.io/formal-bitml/BitML.Properties.Lifetime.html

We note in passing that this construction greatly resembles the notion of 'provenance' explored in the context of EUTxO meta-theory involving native custom tokens [Cha+20b], where one can track any on-chain transaction output back to its origin (*i.e.*, the transaction where the currency was initially minted).

The ancestor relationship can now be established by recursing through the trace and repeatedly applying the aforementioned tracing of the 'latest-evolution' until we encounter the base case of a `[C-Init]` move, at which point we are able to reconstruct the full history of the contract's evolution.

```
traceContract* : Initial Γ₀
               → ⟨ c , v ⟩at y ∈ᶜ Γ
               → (tr : Γ₀ at t₀ —[ αs ]↠ₜ Γ at t)
               → ∃ λ ad → ∃[ tr ∋ Ancestor⟪ ad ⇝ c ⟫ ]
traceContract* {Γ₀ = Γ₀} {t₀ = t₀} init c∈₀ tr₀ =
 go _ (<-wf ∣ tr₀ ∣) c∈₀ tr₀ refl
 where
  go  : ∀ n → Acc _<_ n
     → ⟨ c , v ⟩at y ∈ᶜ Γ
     → (tr : Γ₀ at t₀ —[ αs ]↠ₜ Γ at t)
     → n ≡ ∣ tr ∣
     → ∃ λ ad → ∃[ tr ∋ Ancestor⟪ ad ⇝ c ⟫ ]
  go {c₀}{v₀}{y₀}{Γ′}{αs}{t′} _ (acc rec) c∈ Γ↠ refl
   with x , x′ , y , y′ , xy∈ , xy≈ , α , H ← traceContractₜ (Initial⇒∉ init) c∈ Γ↠
   with tᵢ , _ , xy∈ᵗ ← ×∈⇒×∈ᵗ Γ↠ xy∈
   with Γ<  ← <-splitTraceˡ Γ↠ xy∈ᵗ
   with ∃↑  ← (λ ad {x}{x′} → ∃-splitTraceˡ {P = Ancestor⟪ ad ⇝_⟫} Γ↠ xy∈ᵗ {x}{x′})
   with Γ↠ᵢ  ← splitTraceˡ {Γ₀ at t₀}{αs}{Γ′ at t′}{x at tᵢ} Γ↠ xy∈ᵗ
   with ∈ᶜ↑_ ← (λ {z} → ∈ᶜ-resp-≈ {x′}{x}{z} (↭-sym $ xy≈ .proj₁))
   with H
  ... | base {g}{c} _ Γ→
     = ((⟨ g ⟩ c) , x , x′ , y , y′ , xy∈ , xy≈ , init⇒ Γ→ , base
  ... | step-put
     = let ad , H = go _ (rec _ Γ<) (∈ᶜ↑ here refl) Γ↠ᵢ refl
       in ad , ∃↑ ad (λ (H , ⊆ad) → H , step˘ ⊆ad (put⇝ {i = 0F} refl)) H
  ... | step-split {vcs = vcs} c∈
     = let ad , H = go _ (rec _ Γ<) (∈ᶜ↑ here refl) Γ↠ᵢ refl
       in ad , ∃↑ ad (λ (H , ⊆ad) → H , step˘ ⊆ad (split⇝ {i = 0F} refl c∈)) H
```

```
... | step-control {c′} _ _ ⊩
    = let ad , H = go _ (rec _ Γ<) (∈ᶜ↑ here refl) Γ↠ᵢ refl
      in ad , ∃↑ ad (λ (H , ⊆ad) → H , step˘ ⊆ad (⊩⇒Lifetime ⊩)) H
... | step-timeout {c′} _ ⊩
    = let ad , H = go _ (rec _ Γ<) (∈ᶜ↑ here refl) Γ↠ᵢ refl
      in ad , ∃↑ ad (λ (H , ⊆ad) → H , step˘ ⊆ad (⊩⇒Lifetime ⊩)) H
```

In contrast to how we demonstrated well-founded induction over contracts to define the BitML compiler in Section 5.2.2, where we rolled out our own custom strict order, we now employ a *measure*-based approach on the length of a trace and re-use the existing well-founded recursion on natural numbers (*c.f.,* Appendix A.11 for a brief explanation of how the Prelude provides facilities for termination based on a notion of 'measure').

For the remaining tracing proofs, consult the complete formalisation:

https://omelkonian.github.io/formal-bitml/BitML.Properties.html

6.2.3.3 Constructing mappings out of trace properties

Back to the level of symbolic runs, we can lift the previous results on reduction sequences from Section 6.2.3 to observe the latest configuration of a run and trace their origin back to an intermediate transition.

Tracing an advertisement uses the previous definition `traceAd∗` from Section 6.2.3.1 to derive a similar result for a symbolic run:

```
ad∈≈⇒⊩ : • R ≈⋯ Γ at t
         • ` ad ∈ᶜ Γ
           ─────────────────────────────
           ∃[ R ∋ʳ ⊩[C-Advertise]⦅_↝_⦆⦅ ad ⦆ ]
ad∈≈⇒⊩ {R@record {init = i , _; trace = _ , tr}}{Γ} (_ , Γ≈) ad∈ =
  traceAd∗ i (∈ᶜ-resp-≈ {Γ}{cfg $ R .end} (↭-sym Γ≈) ad∈) tr
```

Similarly for tracing authorisation, for which we did not include their respective tracing lemmas (`traceAuthCommit∗` , `traceAuthInit∗` , and `traceAuthControl∗`) in the previous section:

```
auth-commit∈≈⇒⊩ : • R ≈⋯ Γ at t
                  • A auth[ ♯▷ ad ] ∈ᶜ Γ
                    ──────────────────────────────────────────
                    ∃ λ Δ → ∃[ R ∋ʳ ⊩[C-AuthCommit]⦅_↝_⦆⦅ ad , A , Δ ⦆ ]
auth-commit∈≈⇒⊩ {R@record {init = i , _; trace = _ , tr}}{Γ} (_ , Γ≈) auth∈ =
  traceAuthCommit∗ i (∈ᶜ-resp-≈ {Γ}{cfg $ R .end} (↭-sym Γ≈) auth∈) tr

auth-init∈≈⇒⊩ : • R ≈⋯ Γ at t
                • A auth[ z ▷ˢ ad ] ∈ᶜ Γ
```

```
                          ────────────────────────────────────────
                          ∃[ R ∋ʳ ℍ[C-AuthInit]⦅_↝_⦆⦅ A , ad , z ⦆ ]
auth-init∈≈⇒ℍ {R@record {init = i , _; trace = _ , tr}}{Γ} (_ , Γ≈) auth∈ =
  traceAuthInit∗ i (∈ᶜ-resp-≈ {Γ}{cfg $ R .end} (↭-sym Γ≈) auth∈) tr

auth-control∈≈⇒ℍ : • R ≈⋯ Γ at t
                   • A auth[ z ▷ d ] ∈ᶜ Γ
                     ────────────────────────────────────────
                     ∃[ R ∋ʳ ℍ[C-AuthControl]⦅_↝_⦆⦅ A , z , d ⦆ ]
auth-control∈≈⇒ℍ {R@record {init = i , _; trace = _ , tr}}{Γ} (_ , Γ≈) auth∈ =
  traceAuthControl∗ i (∈ᶜ-resp-≈ {Γ}{cfg $ R .end} (↭-sym Γ≈) auth∈) tr
```

And, finally, observing a contract in the current run to establishe its 'ancestor', by appealing to the tracing lemma for active contracts of Section 6.2.3.2:

```
c∈≈⇒Ancestor : • R ≈⋯ Γ at t
               • ⟨ c , v ⟩at x ∈ᶜ Γ
               ──────────────────────────────────────
               ∃ λ ad → ∃[ R ∋ʳ Ancestor⦅ ad ↝ c ⦆ ]
c∈≈⇒Ancestor {R@record {init = i , _; trace = _ , tr}}{Γ}{t}{c} (_ , Γ≈) c∈ =
  traceContract∗ i (∈ᶜ-resp-≈ {Γ}{cfg $ R .end} (↭-sym Γ≈) c∈) tr
```

In the case of active contracts, we can go a step further and combine multiple trace properties to gain some extra guarantees, namely that the ancestor advertisement is valid and appeared in some previous configuration, as well as the fact that the descendant contract we currently have in our hands is indeed one of the advertisement's subterms.

```
ANCESTOR : • R ≈⋯ Γ at t
           • ⟨ c , v ⟩at x ∈ᶜ Γ
           ───────────────────────────────────────
           ∃ λ ad → Valid ad
                  × ad ∈ advertisements R
                  × c ⊆ subterms ad
                  × ∃[ R ∋ʳ Ancestor⦅ ad ↝ c ⦆ ]
ANCESTOR {R = R@(record {trace = _ , tr})} {Γ = Γ} R≈ c∈ =
  let ad , ∃H@(_ , _ , _ , _ , _ , _ , _ , ad↝c) = c∈≈⇒Ancestor {R}{Γ} R≈ c∈
      _ , vad , ad∈ = ℍ[C-Init]⇒∃ℍ[C-AuthInit] (R .init) tr (∃-weakenP tr proj₁ ∃H)
  in  ad , vad , ad∈ , h-sub∙↝∗ {ad} ad↝c , ∃H
```

Notice how the hypotheses of `[C-Init]` we get from tracing the active contract (`traceContract∗`) can now be utilised to trace a `[C-AuthInit]` move further back into history, effectively chaining together the results of multiple tracing properties.

The whole point of retrieving previous hypotheses was to be able to construct suitable mappings for invoking the BitML compiler: `[C-Advertise]` gives us the means to construct the `Txout` mappings and a collection of `[C-AuthCommit]` moves from all involved participants lets us construct the `Sechash` mapping:

```
ℍ[C-Advertise]⇒Txout :
  • ℍ[C-Advertise]⦅ Γ ↝ Γ′ ⦆⦅ ad ⦆
  • Txout Γ
    ─────────────────────────────
    Txout ad × Txout (ad .C)
ℍ[C-Advertise]⇒Txout {Γ = Γ} {ad = ad} (_ , vad , _ , d⊆) txout =
  let txoutG = weaken-↦ txout (deposits⊆⇒namesʳ⊆ {ad}{Γ} d⊆)
  in txoutG , weaken-↦ txoutG (mapMaybe-⊆ isInj₂ $ vad ∙names-⊆)

ℍ[C-AuthCommit]∗⇒SechashG :
  (∀ {p} → p ∈ nub-participants ad →
    ∃ λ Γ → ∃ λ Γ′ → ∃ λ secrets →
        ℍ[C-AuthCommit]⦅ Γ ↝ Γ′ ⦆⦅ ad , p , secrets ⦆
     × Sechash Γ′)
  ────────────────────────────────────────────────────
  Sechash (ad .G)
ℍ[C-AuthCommit]∗⇒SechashG {ad} ∀p {s} s∈ =
  let partG = nub-participants ad; ⟨ G ⟩ _ = ad
      pₛ , pₛ∈ = namesˡ⇒part {g = G} s∈
      _ , Γₛ , secrets , (Γ₁ , _ , Γₛ≡ , as≡ , _) , SechashΓₛ = ∀p pₛ∈
      (as , ms) = unzip secrets; Δ = || map (uncurry ⟨ pₛ :_♯_⟩) secrets

      s∈Δ : s ∈ namesˡ Δ
      s∈Δ = namesʳ-∥map-authCommit {A = pₛ} {secrets = secrets}
              (⟪ s L.Mem.∈_ ⟫ as≡ ~: names⊆secretsOf {g = G} s∈)

      n⊆ : namesˡ Δ ⊆ namesˡ (` ad ∣ Γ₁ ∣ Δ ∣ pₛ auth[ ♯▷ ad ])
      n⊆ = mapMaybe-⊆ isInj₁
         $ ∈-collect-++⁺ˡ (` ad ∣ Γ₁ ∣ Δ) (pₛ auth[ ♯▷ ad ])
         ∘ ∈-collect-++⁺ʳ (` ad ∣ Γ₁) Δ

      s∈′ : s ∈ namesˡ Γₛ
      s∈′ = ⟪ (λ ◆ → s ∈ namesˡ ◆) ⟫ Γₛ≡ ~: n⊆ s∈Δ
  in SechashΓₛ {s} s∈′
```

As a final step, we combine all the previous results to calculate the mappings required by observing certain parts of the current configuration in the run:

```
ad∈⇒Txout :
  • ` ad ∈ᶜ Γ
  • R ≈⋯ Γ at t
  • Txout R
    ────────────────────────
    Txout ad × Txout (ad .C)
ad∈⇒Txout {ad}{Γ}{R@(record {trace = _ , tr})} ad∈ R≈ txout =
  let Γᵢ′ , Γᵢ , _ , _ , xy∈ , (x≈ , _) , ℍ = ad∈≈⇒ℍ {R}{Γ} R≈ ad∈
```

```
    Γᵢ∈ , _ = ∈-allTransitions⁻ tr xy∈
    txoutΓᵢ = Txout≈ {Γᵢ′}{Γᵢ} x≈
              $ Txout∈ {R = R} txout Γᵢ∈
  in ℍ[C-Advertise]⇒Txout {Γ = Γᵢ}{ad = ad} ℍ txoutΓᵢ

committed⇒ℍ[C-AuthCommit]∗ :
  • R ≈⋯ Γ₀ at t
  • nub-participants ad ⊆ committedParticipants ad Γ₀
  • Sechash R
    ────────────────────────────────────────────────
    (∀ {p} → p ∈ nub-participants ad →
      ∃ λ Γ → ∃ λ Γ′ → ∃ λ secrets →
        ℍ[C-AuthCommit]⦅ Γ ↝ Γ′ ⦆⦅ ad , p , secrets ⦆
      × Sechash Γ′)
committed⇒ℍ[C-AuthCommit]∗ {R}{Γ₀}{t}{ad} R≈ committedA sechash′ {p} p∈ =
  let
    authCommit∈′ : p auth[ ♯▷ ad ] ∈ᶜ Γ₀
    authCommit∈′ = committed⇒authCommit {Γ = Γ₀} $ committedA p∈

    Δ , x , x′ , y , y′ , xy∈ , (_ , y≈) , ℍ = auth-commit∈≈⇒ℍ {R}{Γ₀} R≈ authCommit∈′
    _ , y∈ = ∈-allTransitions⁻ (R .trace .proj₂) xy∈

    sechash-y : Sechash y′
    sechash-y = Sechash≈ {x = y}{y′} y≈
              $ Sechash∈ {R = R} sechash′ y∈
  in
    x′ , y′ , Δ , ℍ , sechash-y
```

(These are not complete in any sense, as we could observe later configurations arising from these that in turn lead us to the base cases, *e.g.*, seeing an `[C-AuthCommit]` means that a previous `[C-Advertise]` has necessarily occurred, but we chose to show only the fundamental results here to understand the construction rather than trying to be complete at the expense of readability.)

6.2.4 Invoking the BitML compiler

It is high time we put all the properties we proved in Section 6.2.3 to use, namely to construct the mappings required to call the BitML compiler. This amounts to providing mappings for the advertisement to be compiled, which are constructed by observing the advertisement in the current configuration and *tracing* it back in history to recover all required information.

Much like we lifted mappings from the current configuration to the next in Section 6.2.2 using $\texttt{LIFT}^{s}$, we can now use the properties of mappings and traces from

the last two sections to lift the mappings of the current run to a 𝔾 mapping for compiling an advertisement.

There are two possible scenarios: either we find the initial contract advertisement in the current configuration (LIFT⁰), or hold an active contract that has already been stipulated (LIFTᶜ).

In the former case, we have to appeal to the tracing lemma for advertisements (traceAd∗) in order to retrieve the Txout mapping from the hypotheses of [C-Advertise] and the Sechash mapping from the subsequent [C-AuthCommit] moves:

```
LIFT⁰ : ∀ (r : ℝ R) (t : Time) Γ (R≈ : R ≈⋯ Γ at t) ad →
  • ` ad ∈ᶜ Γ
  • nub-participants ad ⊆ committedParticipants ad Γ
    ─────────────────────────────────────────────────
    𝔾 ad
LIFT⁰ {R} r t Γ R≈@(_ , Γ≈) ad ad∈ committedA = vad , txout₀ , sechash₀ , κ₀
  where
  module _
    (let Γᵢ′ , Γᵢ , _ , _ , xy∈ , (x≈ , _) , ℍ = ad∈≈⇒ℍ {R}{Γ} R≈ ad∈)
    (let _ , $vad , honG , _ = ℍ)
    where
    open ℝ r

    vad : Valid ad
    vad = $vad

    txout₀ : Txout (ad .G)
    txout₀ =
     let
      Γᵢ∈ , _ = ∈-allTransitions⁻ (R ∙trace′) xy∈

      txoutΓᵢ : Txout Γᵢ
      txoutΓᵢ = Txout≈ {Γᵢ′}{Γᵢ} x≈
              $ Txout∈ {R = R} txout′ Γᵢ∈
     in
      ℍ[C-Advertise]⇒TxoutG {Γ = Γᵢ}{ad = ad} ℍ txoutΓᵢ

    sechash₀ : Sechash (ad .G)
    sechash₀ = ℍ[C-AuthCommit]∗⇒SechashG {ad = ad}
             $ committed⇒ℍ[C-AuthCommit]∗ {R}{Γ}{t}{ad} R≈ committedA sechash′

    κ₀ : 𝕂²′ ad
    κ₀ =
     let
      ad∈Hon : ad ∈ authorizedHonAds Γ
```

```
    ad∈Hon = committed⇒authAd (L.Any.lookup-result honG) {Γ = Γ}
           $ committedA (L.Mem.∈-lookup $ L.Any.index honG)
  in
    weaken-↦ κ′ (ads⦅end⦆⊆ R ∘ ∈ads-resp-≈ _ {Γ}{R ∙cfg} (↭-sym $ R≈ .proj₂))
     $ ∈-collect-++⁺ʳ (` ad) Γ ad∈Hon
```

In the latter case of an active contract amidst execution, we need to track the contract's lifetime back when its ancestor advertisement was stipulated using `[C-Init]` , extract the initial prefix of the trace up to that point, and only then proceed as the former case using `LIFT⁰`.

```
LIFTᶜ : ∀ (𝕣 : ℝ Rˢ) {ad c} →
  • ∃[ Rˢ ∋ʳ Ancestor⦅ ad ↝ c ⦆ ]
    ──────────────────────────────
    𝔾 ad
LIFTᶜ {R} 𝕣 {ad} ∃H =
  let
    open ∣AD ad

    ∃R : ∃[ R ∋ʳ ∃ℍ[C-AuthInit]⦅_↝_⦆⦅ ad ⦆ ]
    ∃R = proj₁
      $ ℍ[C-Init]⇒∃ℍ[C-AuthInit] (R .init) (R ∙trace′)
      $ ∃-weakenP (R ∙trace′) proj₁ ∃H

    x , x′ , _ , _ , xy∈ , (x≈ , _) , _ , _ , _ , _ , Γ≡ , _ , p⊆′ , _ = ∃R
    tr = R ∙trace′
    tᵢ , _ , xy∈ᵗ = ×∈⇒×∈ᵗ tr xy∈
    tr′ = splitTraceˡ tr xy∈ᵗ
    R′  = splitRunˡ R xy∈ᵗ

    𝕣′ : ℝ R′
    𝕣′ = ℝ⊆ xy∈ᵗ 𝕣

    R≈′ : R′ ≈⋯ x at tᵢ
    R≈′ = splitRunˡ-≈⋯ R xy∈ᵗ

    ad∈ : ` ad ∈ᶜ x
    ad∈ = ∈ᶜ-resp-≈ {x′}{x} (↭-sym x≈) $ subst (` ad ∈ᶜ_) (sym Γ≡) (here refl)

    p⊆ : partG ⊆ committedParticipants ad x
    p⊆ = L.Perm.∈-resp-↭
          (collectFromList↭ (∣committedParticipants∣.go ad .collect) (↭-sym x≈))
       ∘ p⊆′
  in
    LIFT⁰ 𝕣′ tᵢ x R≈′ ad ad∈ p⊆
```

Last, we provide convenient wrappers for invoking the compiler using such a `𝔾` mapping (`COMPILE`) and provide a shorthand for just retrieving the initial transaction (`COMPILE-INIT`) for the cases where the rest are irrelevant:

```
COMPILE : 𝔾 ad
          ──────────────────────────────────────────────────
          InitTx (ad .G) × (subterms ad ↦′ BranchTx ∘ _∗)
COMPILE {ad = ad} (vad , txout₀ , sechash₀ , κ₀) =
 let
  K : 𝕂 (ad .G)
  K {p} _ = K̂ p

  T , ∀d = bitml-compiler {ad = ad} vad sechash₀ txout₀ K κ₀
 in
  T , weaken-sub {ad} ∀d

COMPILE-INIT : 𝔾 ad
               ───────────────
               InitTx (ad .G)
COMPILE-INIT = proj₁ ∘ COMPILE
```

In the final scenario of an active contract, we employ the ancestor relationship to retrieve the initial compilation parameters, and then only extract the corresponding generated transaction and associated key pairs for signing it:

```
COMPILE-ANCESTOR : ∀ {i : Index c} (open ∣SELECT c i) →
  • R ≈⋯ Γ at t
  • ⟨ c , v ⟩at x ∈ᶜ Γ
  • ℝ R
  ──────────────────────────────────────────────────
  BranchTx (d ∗) × (authDecorations d ↦ KeyPair)
COMPILE-ANCESTOR {c}{R}{Γ}{t}{v}{x}{i} R≈ c∈ 𝕣 =
 let
  ⟨G⟩C , vad , ad∈ , c⊆ , anc = ANCESTOR {R = R} {Γ = Γ} R≈ c∈
  open ∣AD ⟨G⟩C; open ∣SELECT c i; open ℝ 𝕣

  d∈ : d ∈ subterms ⟨G⟩C
  d∈ = c⊆ (L.Mem.∈-lookup i)

  A∈ : authDecorations d ⊆ partG
  A∈ = ∈-nub⁺ ∘ subterms-part⊆ᵃ vad d∈ ∘ auth⊆part {d = d}

  _ , ∀d∗ = COMPILE (LIFTᶜ 𝕣 anc)
 in
  ∀d∗ d∈ , κ′ ad∈ d∈ ∘ A∈
```

We now have established the whole infrastructure we are going to need for defining all cases of coherence in the next section, so let us get right down to it!

6.3 The coherence relation (§8 & A.6 of [BZ18])

The coherence relation will match symbolic runs (`SRun`, as defined in Section 6.2.1) to the corresponding computational runs (`CRun`, as defined in Section 6.1.3) that are expected to behave the same. Coherence will be defined inductively as a type family `SRun → CRun → Type`, where each constructor corresponds to a single case out of a total of 22 cases (1 base case, 18 relevant inductive cases, 3 irrelevant inductive cases) as appearing in Def.20 of the original paper (§A.6).

We would love to give this definition immediately, but there are three significant hindrances that get in the way:

1. Some cases read as "If no other case applies, ...", which would formally translate into a hypothesis that negates the coherence relation currently being defined. Alas, this is not possible in type theory in general, since we would end up with a datatype that is not *strictly positive*, which is not allowed by Agda's positivity checker and is necessary to ensure soundness of the underlying logic (*i.e.,* we could derive `⊥` otherwise).[2]

   However, we can take inspiration from the rich history of Logic Programming where negation-as-failure presents similar problems; the technique of *stratification* seems to be the go-to solution for such issues across a diverse class of logics [ABW88; Gel88; Prz88; RS]. In our case, this would entail separating the coherence relation into two distinct *strata* (∼ levels), and placing the offending cases that use negation strictly on the second level. (Note that a recent proposal to add negation to datatypes [Atk22] independently turned to the semantics of negation in Logic Programming as well, so we must be doing something right!)

2. The formulation of each case is highly non-trivial and requires complicated reasoning, which is expected to appear as the type signature of each constructor of the coherence relation. Sadly, Agda only allows `let`-expressions to appear on the type signatures of datatype constructors, so we have to offload these heavy calculations to some prior location living outside the datatype definition, if we ever hope to use advanced features such as `rewrite` or `abstract`.

3. Indexing types by intricate calls to predefined functions is guaranteed to lead to an explosion of difficult unification problems for the type-checker, a phenomenon widely known as *green slime* in the dependent-types world [McB14]. As an entertaining fact, type-checking even a trivial case analysis on a given proof of

[2] https://agda.readthedocs.io/en/v2.6.3/language/data-types.html#strict-positivity

coherence that merely eliminates to ⊤ never seems to terminate! Therefore, we need to somehow bundle all this computation in a separate container that will let us control unfolding at a finer level of granularity.

To remedy all the above, we will proceed in incremental steps: in Section 6.3.1 define each case in isolation, only to bring them together in a stratified fashion in Section 6.3.2. Along the way, we will have to make sure that things are set up in a way such that unification does not get stuck when we later prove properties about coherence.

6.3.1 The relevant inductive cases (§A.6,Def.20 of [BZ18])

For each sub-case i of 'Inductive case 1', we will define a separate relation `_~ℍ[i]~_` that offloads expensive Agda computation to a corresponding module `Hᵢ`, both sharing a common context bundled up in a record `Hᵢ-args`.

In an attempt to reduce unwanted unification problems, we will exploit the fact that all of these cases are inductive steps, and specialise their type to reflect the individual components, in contrast to the more general `SRun → CRun → Type`.

```
StepRel : Type₁
StepRel = (Γₜ : Cfgᵗ) {Rˢ : S.Run}
         → ℝ∗ Rˢ
         → 𝕃 Rˢ Γₜ
         → CLabel
         → CRun
         → Type
```

(This is due some annoying issues with type-checking performance that we will not go through, but intuitively we are delaying unification by constructing the combined run as late as possible.)

Moreover, each case will make use of a specific transition from the BitML rules of Section 4.7.5, so we provide shorthands to specify such transitions and their constituents while enforcing the same naming convention across cases.

```
record Transition : Type where
  constructor _⨾_—_—→_⨾_⊣_
  field Γ t α Γ′ t′ : _
  Γₜ  = Cfgᵗ ∋ Γ at t
  Γₜ′ = Cfgᵗ ∋ Γ′ at t′
  field Γ→ : Γₜ —[ α ]→ₜ Γₜ′
```

(Notice how we are not required to provide the types for the first few fields, as they can be inferred by how they are used later in the record definition.)

Similarly, the context will always carry the so-far constructed runs, along with the current mappings and the permuted intermediate configuration that acts as the most recent end of the symbolic run.

```
record ℍ-Run {Γₜ α Γₜ′} (Γ→ : Γₜ —[ α ]→ₜ Γₜ′) : Type where
  constructor _⨾_⨾_⊣_≈_⊣_
  private Γ′ = Γₜ′ .cfg; t′ = Γₜ′ .time
  field Rᶜ : CRun  ; Rˢ : S.Run
        𝕣∗ : ℝ∗ Rˢ ; R≈ : Rˢ ≈⋯ Γₜ
        Γ″ : Cfg   ; Γ≈ : Γ″ ≈ Γ′
  Γₜ″ = Cfgᵗ      ∋ Γ″ at t′
  𝕣   = ℝ Rˢ      ∋ ℝ∗⇒ℝ 𝕣∗
  𝕒   = 𝔸 Rˢ Γₜ″ ∋ α , Γₜ , Γₜ′ , Γ→ , (refl , Γ≈) , R≈
  Rˢ′ = Run       ∋ Γₜ″ ∷ Rˢ ⊣ 𝕒
  open ℝ 𝕣 public
```

As this is indexed by the reduction proof Γ→ (obtained from some previous `Transition`), we can already compute the next run `Rˢ′` and only need to further provide the updated mappings for each case.

6.3.1.1 Case (1)

The first case corresponds to the `[C-Advertise]` transition, hence the context will contain all the free variables associated to this particular rule and its hypotheses:

```
record H₁-args : Type where
  constructor mk
  field {ad Γ₀ t} : _
  open ∣AD ad public
  field -- Hypotheses from [C-Advertise]
        vad : ValidAd ad
        hon : Any (_∈ Hon) partG
        d⊆  : ad ⊆⟪ deposits ⟫ Γ₀
  open Transition ( Γ₀ ⨾ t —— advertise⦅ ad ⦆ —→ ` ad ∣ Γ₀ ⨾ t
                  ⊣ Act ([C-Advertise] vad hon d⊆)
                  ) public

  field 𝕙r : ℍ-Run Γ→
  open ℍ-Run 𝕙r public
```

(All context records will end in the same manner after specifying the `Transition` : assuming the bundled information about the current run in `𝕙r` and `open`ing it as a last step; we will omit these last two lines from now on.)

The helper module will typically take care of updating the mappings for the next configuration (and invoking the BitML compiler in other cases later on). In this case, the mappings remain identical between the source and target configuration, so we lift using the identity mapping transformers:

```
module H₁ (h : H₁-args) (open H₁-args h) where
  -- txout′ = txout, sechash′ = sechash, κ′ = κ
  λˢ : ℾᵗ Γₜ′
  λˢ = LIFTˢ ɾ Γ R≈ Γ′ id id id
```

Notation

Correspondence with the paper text. We will often include prose (possibly paraphrased) from the original paper (Def.20) as comments inlined across the code we display, so that it is easier to match it with our formalisation.

The case is then formulated as an inductive datatype with a single constructor, where the context `H₁-args` is passed as an implicit argument that we immediately `open` and also apply to the helper module `H₁` to retrieve the mappings for the next configuration `λˢ`.

```
data _⨾_⨾_~ℍ[1]~_⨾_ : StepRel where
  mkℍ : ∀ {h : H₁-args} {A}
    → let open H₁-args h renaming (ad to ⟨G⟩C)
          open H₁ h using (λˢ)
          -- C is obtained by encoding {G}C as a bitstring
          txoutΓ = Txout Γ ∋ Txout≈ {Rˢ •cfg}{Γ} (R≈ .proj₂) (ɾ •txoutEnd_)
          txoutG = Txout G ∋ weaken-↦ txoutΓ (deposits⊆⇒namesʳ⊆ {⟨G⟩C}{Γ} d⊆)
          txoutC = Txout C ∋ weaken-↦ txoutG (mapMaybe-⊆ isInj₂ $ vad •names-⊆)

          λᶜ = A →∗: encodeAd ⟨G⟩C (txoutG , txoutC)
      in
      ──────────────────────────────────────────────────────────
      Γₜ″ ⨾ ɾ∗ ⨾ (𝕒 , λˢ) ~ℍ[1]~ λᶜ ⨾ Rᶜ
```

To complete the correspondence, we define the next computational label `λᶜ` as the broadcast of the advertisement `⟨G⟩C` from some participant `A`, using the serialisation procedure from Section 6.1.2.

The helper module `H₁` might seem spurious, since we could simply inline the updated mappings (embodied in `λˢ`) directly in the constructor's type signature, but we prefer to retain a consistent structure across all cases rather than take ad-hoc shortcuts for a handful of simple cases.

6.3.1.2 Case (2)

The second case covers the `[C-AuthInit]` moves where participants commit to their secrets (as required by the contract's preconditions).

In addition to the free variables necessary to express the hypotheses, here the context record also defines some intermediate computations out of the context arguments, so that there is no need to repeat them both in the datatype and its helper module when we need to use them afterwards.

```
record H₂-args : Type where
  constructor mk
  field {ad Γ₀ t A} : _; k⃗ : 𝕂²′ ad; Δ×h̅ : List (Secret × Maybe ℕ × ℤ)
  open ∣AD ad public hiding (C)
  h̅  = List HashId ∋ map (proj₂ ∘ proj₂) Δ×h̅
  Δ  = List (Secret × Maybe ℕ) ∋ map drop₃ Δ×h̅
  Δᶜ = || map (uncurry ⟨ A :_♯_⟩) Δ
  as = unzip Δ .proj₁
  ms = unzip Δ .proj₂
  k̅ = concatMap (map pub ∘ codom) $ codom k⃗
  sechash⁺ : as ↦ HashId
  sechash⁺ a∈ = let _ , a×m∈ , _    = ∈-unzip⁻ˡ Δ a∈
                    (_ , _ , z) , _ = ∈-map⁻ drop₃ a×m∈
                in z
  field -- Hypotheses from [C-AuthCommit]
        as≡  : as ≡ secretsOfᵖ A G
        All∉ : All (_∉ secretsOfᶜᶠ A Γ₀) as
        Hon⇒ : A ∈ Hon → All Is-just ms
  open Transition (  (` ad ∣ Γ₀)                         ⨾ t
                   —— auth-commit⦅ A , ad , Δ ⦆ —→
                     (` ad ∣ Γ₀ ∣ Δᶜ ∣ A auth[ ♯▷ ad ]) ⨾ t
                   ⊣ Act ([C-AuthCommit] as≡ All∉ Hon⇒)
                   ) public
```

Since the target configuration now introduces new resources, lifting the mappings will require some work, namely accounting for the new secrets `as` using the mapping extension `sechash⁺` and the new keypairs required by the advertisement's authorisations using `k⃗`
.

First, we will prove some auxiliary lemmas that relate the resources between source and target destination, as the way that the mapping domains change is not immediately obvious to Agda:

```
module H₂ (h : H₂-args) (open H₂-args h) where
  private
    open ≡-Reasoning
```

```
mkRev : List (Secret × Maybe ℕ) → Cfg
mkRev = ||_ ∘ map (uncurry ⟨ A :_♯_⟩)

ids≡ : Γ′ ≡⟨ ids ⟩ Γ
ids≡ =
  begin
    ids Γ′
  ≡⟨⟩
    ids (Γ ∣ mkRev Δ ∣ A auth[ ♯▷ ad ])
  ≡⟨ mapMaybe∘collectFromBase-++ isInj₂ (Γ ∣ mkRev Δ) (A auth[ ♯▷ ad ]) ⟩
    ids (Γ ∣ mkRev Δ) ++ ids (A auth[ ♯▷ ad ])
  ≡⟨⟩
    ids (Γ ∣ mkRev Δ) ++ []
  ≡⟨ L.++-identityʳ _ ⟩
    ids (Γ ∣ mkRev Δ)
  ≡⟨ mapMaybe∘collectFromBase-++ isInj₂ Γ (mkRev Δ) ⟩
    ids Γ ++ ids (mkRev Δ)
  ≡⟨ cong (ids Γ ++_) (hʳ Δ) ⟩
    ids Γ ++ []
  ≡⟨ L.++-identityʳ _ ⟩
    ids Γ
  ∎ where
    hʳ : ∀ Δ → Null $ ids (mkRev Δ)
    hʳ [] = refl
    hʳ (_ ∷ []) = refl
    hʳ (_ ∷ Δ@(_ ∷ _)) rewrite hʳ Δ = L.++-identityʳ _

secrets≡ : secrets Γ′ ≡ secrets Γ ++ as
secrets≡ =
  begin
    secrets Γ′
  ≡⟨ mapMaybe∘collectFromBase-++ isInj₁ (Γ ∣ mkRev Δ) (A auth[ ♯▷ ad ]) ⟩
    secrets (Γ ∣ mkRev Δ) ++ []
  ≡⟨ L.++-identityʳ _ ⟩
    secrets (Γ ∣ mkRev Δ)
  ≡⟨ mapMaybe∘collectFromBase-++ isInj₁ Γ (mkRev Δ) ⟩
    secrets Γ ++ secrets (mkRev Δ)
  ≡⟨ cong (secrets Γ ++_) (hˡ Δ) ⟩
    secrets Γ ++ as
  ∎ where
    hˡ : ∀ Δ → secrets (mkRev Δ) ≡ proj₁ (unzip Δ)
    hˡ [] = refl
    hˡ (_ ∷ []) = refl
    hˡ ((s , m) ∷ Δ@(_ ∷ _)) =
```

```
    begin
      secrets (⟨ A : s ♯ m ⟩ ∣ mkRev Δ)
    ≡⟨ mapMaybe∘collectFromBase-++ isInj₁ ⟨ A : s ♯ m ⟩ (mkRev Δ) ⟩
      secrets ⟨ A : s ♯ m ⟩ ++ secrets (mkRev Δ)
    ≡⟨⟩
      s ∷ secrets (mkRev Δ)
    ≡⟨ cong (s ∷_) (hˡ Δ) ⟩
      s ∷ proj₁ (unzip Δ)
    ∎

hᵃ : ∀ Δ → Null $ advertisements (mkRev Δ)
hᵃ [] = refl
hᵃ (_ ∷ []) = refl
hᵃ (_ ∷ Δ@(_ ∷ _)) rewrite hᵃ Δ = L.++-identityʳ _

ads≡ : advertisements Γ′ ≡ advertisements Γ ++ advertisements (A auth[ ♯▷ ad ])
ads≡ rewrite collectFromBase-++ {X = Ad} (Γ ∣ mkRev Δ) (A auth[ ♯▷ ad ])
           | collectFromBase-++ {X = Ad} Γ (mkRev Δ)
           | hᵃ Δ
           | L.++-identityʳ (advertisements Γ)
           = refl
```

The long equational chains above establish how the resulting resources are obtained from the previous ones, as propositional equality. Crucially, these use advanced features like `rewrite` for equalities and `where` clauses, rendering them incompatible expressions for immediate use in a constructor, hence the need for the helper module H_2.

It is then possible to define the mapping transformers for the next configuration. The only tricky part that needs some attention is the calculation of new keys to record in `κ↝`, where we conditionally add the provided keypairs $\vec{k}$ only in the case where `A` is an honest participant. The fact that this code typechecks without having to provide keypairs for dishonest participant reveals the fact that the method for collecting the advertisements does not simply gather all mentioned advertisements from each base configuration, but retains only those appearing under an authorisation by an honest participant.

```
txout↝ : Γ →⦅ Txout ⦆ Γ′
txout↝ = lift Γ —⟨ ids ⟩— Γ′ ⊣ ids≡

sechash↝ : Γ →⦅ Sechash ⦆ Γ′
sechash↝ sechash′ = extend-↦ (↭-reflexive secrets≡) sechash′ sechash⁺

κ↝ : Γ →⦅ 𝕂² ⦆ Γ′
κ↝ κ′ = extend-↦ (↭-reflexive ads≡) κ′ κ″
  where κ″ : advertisements (A auth[ ♯▷ ad ]) ↦′ 𝕂²′
        κ″ x∈ with does (A ∈? Hon) | x∈
        ... | true  | 𝟘 = k⃗
```

```
        ... | false | ()
 -- (v) txout = txout′ (vi) extend sechash′ (vii) extend κ′
 λˢ : ℾᵗ Γₜ′
 λˢ = LIFTˢ ɼ Γ R≈ Γ′ txout↝ sechash↝ κ↝
```

The rest of the conditions of case (2) are formulated in the datatype itself, where the next computational move is defined as A signing the advertisement and broadcasting it along with the hashes and keypairs from the context.

```
data _⨾_⨾_~ℍ[2]~_⨾_ : StepRel where
 mkℍ : ∀ {h : H₂-args} {B}
  → let open H₂-args h renaming (ad to ⟨G⟩C)
        C = encodeAd ⟨G⟩C (ad∈⇒Txout {⟨G⟩C}{Γ}{Rˢ} ⊕ R≈ txout′)
        m = SIG (K A) $ encode (C , h̄ , k̄)
        -- (ii) broadcast message in Rᶜ
        λᶜ = B →∗: m
        open H₂ h using (λˢ)
    in
    -- (i) ⟨G⟩C has been previously advertised in Rᶜ
  ∀ (∃B : ∃ λ B → (B →∗: C) ∈ toList Rᶜ) →
    -- ∘ it is the first occurrence of such a broadcast in Rᶜ
  • All (λ l → ∀ X → l ≢ X →∗: C) (Any-tail $ ∃B .proj₂)
    -- ∘ hashes respect security parameter η
  • All (λ hᵢ → ∣ hᵢ ∣ᵐ ≡ η) h̄
    -- ∘ make sure that λᶜ is the first occurrence of such a message after C
  • All (λ l → ∀ X → l ≢ X →∗: m) (Any-front $ ∃B .proj₂)
    -- (iii) each hᵢ is obtained by querying the oracle,
    --       otherwise we have a dishonestly chosen secret
  • CheckOracleInteractions Rᶜ Δ×h̄
    -- (iv) no hash is reused
  • Unique h̄
  • Disjoint h̄ (codom sechash′)
    ─────────────────────────────────────────────────────
    Γₜ″ ⨾ ɼ∗ ⨾ (𝕒 , λˢ) ~ℍ[2]~ λᶜ ⨾ Rᶜ
```

Most conditions closely resembles the formulation in the pen-and-paper version, except for condition (iii) which relies on a helper definition for checking participant interactions with the oracle and making sure the hashes have been picked as expected:

```
CheckInteractions : List OracleInteraction → Pred₀ (Secret × Maybe ℕ × HashId)
CheckInteractions os = λ where
  (_ , just Nᵢ , hᵢ) →
   ∃ λ B → ∃ λ mᵢ → ((B , mᵢ , hᵢ) ∈ os) × (∣ mᵢ ∣ᵐ ≡ η + Nᵢ)
  (_ , nothing , hᵢ) →
   hᵢ ∉ map select₃ (filter ((η ≤?_) ∘ ∣_∣ᵐ ∘ select₂) os)
```

```
CheckOracleInteractions : CRun → List (Secret × Maybe ℕ × HashId) → Type
CheckOracleInteractions Rᶜ = All (CheckInteractions $ oracleInteractionsᶜ Rᶜ)
```

6.3.1.3 Case (3)

The case for `[C-AuthInit]`, which authorises the persistent deposits at the time of stipulation, is somewhat simpler as the mapping domains remain the same, albeit not definitionally, so we still need to prove some minor lemmas:

```
record H₃-args : Type where
  constructor mk
  field {ad Γ₀ t A x v} : _
  open ∣AD ad public
  field -- Hypotheses from [C-AuthInit]
        committedA : partG ⊆ committedParticipants ad Γ₀
        A∈per : (A , v , x) ∈ persistentDeposits G
  open Transition (  (` ad ∣ Γ₀)                    ; t
                  —— auth-init⦅ A , ad , x ⦆ —→
                     (` ad ∣ Γ₀ ∣ A auth[ x ▷ˢ ad ]) ; t
                  ⊣ Act ([C-AuthInit] committedA A∈per)
                  ) public


module H₃ (h : H₃-args) (open H₃-args h) where
  private
    names≡ : Γ′ ≡⦅ names ⦆ Γ
    names≡ rewrite collectFromBase-++ {X = Name} Γ (A auth[ x ▷ˢ ad ])
           = L.++-identityʳ _

    ids≡ : Γ′ ≡⦅ ids ⦆ Γ
    ids≡ = cong filter₂ names≡

    secrets≡ : Γ′ ≡⦅ secrets ⦆ Γ
    secrets≡ = cong filter₁ names≡

    ads≡ : Γ′ ≡⦅ advertisements ⦆ Γ
    ads≡ rewrite collectFromBase-++ {X = Ad} Γ (A auth[ x ▷ˢ ad ])
         = L.++-identityʳ _

    txout↝ : Γ →⦅ Txout ⦆ Γ′
    txout↝ txout′ rewrite ids≡ = txout′

    sechash↝ : Γ →⦅ Sechash ⦆ Γ′
    sechash↝ sechash′ rewrite secrets≡ = sechash′
```

```
  κ⇝ : Γ →⦅ 𝕂² ⦆ Γ′
  κ⇝ κ′ rewrite collectFromBase-++ {X = Ad} Γ (A auth[ x ▷ˢ ad ])
        | L.++-identityʳ (advertisements Γ)
        = κ′

 λˢ : 𝕣ᵗ Γₜ′
 λˢ = LIFTˢ 𝕣 Γ R≈ Γ′ txout⇝ sechash⇝ κ⇝
```

However, this case needs to get a hold of the initial transaction generated by the BitML compiler, which is where the lifting of Section 6.2.4 comes in handy and takes care of populating the appropriate compiler parameters.

```
private
 𝕘 : 𝔾 ad
 𝕘 = LIFT⁰ 𝕣 t Γ R≈ ad 𝟘 committedA

T : ∃Tx
T = -, -, COMPILE-INIT 𝕘
```

We then proceed as usual by placing the conditions as hypotheses in the datatype constructor, noting that the helper module H_3 is now also providing the compiled transaction T in addition to the updated symbolic mappings λ^s.

```
data _⨾_⨾_~ℍ[3]~_⨾_ : StepRel where
 mkℍ : ∀ {h : H₃-args} {B}
  → let open H₃-args h
      -- (iv) txout = txout′, sechash = sechash′, κ = κ′
      open H₃ h using (λˢ; T)
      -- (i) broadcast Tᵢₙᵢₜ , signed with A's private key
      m = SIG (K̂ A) T
      λᶜ = B →∗∶ m
    in
    -- (ii) Tᵢₙᵢₜ occurs as a message in Rᶜ
  ∀ (∃B : ∃ λ B → B →∗∶ encode (T .proj₂ .proj₂) ∈ toList Rᶜ) →
    -- (iii) broadcast message in Rᶜ
    -- ∘ λᶜ is the first occurrence of such a message after Tinit in Rᶜ
  • All (λ l → ∀ X → l ≢ X →∗∶ m) (Any-front $ ∃B .proj₂)
    ────────────────────────────────────────────
    Γₜ″ ⨾ 𝕣∗ ⨾ (𝕒 , λˢ) ~ℍ[3]~ λᶜ ⨾ Rᶜ
```

6.3.1.4 Case (4)

As usual, the context is drawn from the associated BitML rule for [C-Init]. This case corresponds to the stipulation of the contract: at the symbolic level the advertisement is

evolved into an active contract, while at the computational level the initial transaction is submitted to the blockchain.

```
record H₄-args : Type where
  constructor mk
  field {ad Γ₀ t z} : _
  open ∣AD ad public
  ds = persistentDeposits G
  vs = map select₂ ds
  xs = map select₃ ds
  v  = sum vs
  field -- Hypotheses from [C-Init]
        fresh-z : z ∉ xs ++ ids Γ₀
  Γ₁ = Cfg ∋ ` ad ∣ Γ₀
  Γ₂ = Cfg ∋ || map (λ{ (Aᵢ , vᵢ , xᵢ) → ⟨ Aᵢ has vᵢ ⟩at xᵢ ∣ Aᵢ auth[ xᵢ ▷ˢ ad ] }) ds
  Γ₃ = Cfg ∋ || map (_auth[ ♯▷ ad ]) partG
  -- (i) consume {G}C and its persistent deposits from Rˢ to produce ⟨C,v⟩z
  open Transition (  (Γ₁ ∣ Γ₂ ∣ Γ₃)       ⨾ t
                  —— init⟪ G , C ⟫ —→
                     (⟨ C , v ⟩at z ∣ Γ₀) ⨾ t
                  ⊣ Act ([C-Init] fresh-z)
                  ) public
```

Since the assumption that all participants have committed their secret is no longer immediately available from the context, we need to prove it manually (it holds necessarily due to the previous `[C-AuthInit]` rule) in order to retrieve the compiler parameters:

```
module H₄ (h : H₄-args) (open H₄-args h) where
  private
    committedA : partG ⊆ committedParticipants ad Γ
    committedA {p} p∈ =
      ∈-collect-++⁺ʳ (Γ₁ ∣ Γ₂) Γ₃ ⦃ ∣committedParticipants∣.go ad ⦄ p∈′
      where p∈′ : p ∈ committedParticipants ad Γ₃
            p∈′ rewrite committedPartG≡ {ad} partG = p∈

    𝕘 : 𝔾 ad
    𝕘 = LIFT⁰ 𝕣 t Γ R≈ ad 𝟘 committedA

  T : InitTx G
  T = COMPILE-INIT 𝕘
```

As to the mappings of the next configuration, we need to account for the new active contract by matching it to the first transaction output of the initial transaction `T`.

To save space for the rest of this section, we will only display the types of lemmas proven about resources and omit their lengthy proofs which should be familiar by now.

```
 private
  ids≡      : ids Γ ≡ ids Γ₀ ++ map select₃ ds
  secrets≡ : Γ′ ≡⟪ secrets ⟫ Γ
  ads⊆      : Γ′ ⊆⟪ advertisements ⟫ Γ

  sechash↝ : Γ →⟪ Sechash ⟫ Γ′
  sechash↝ = lift Γ —⟨ secrets ⟩— Γ′ ⊣ secrets≡

  κ↝ : Γ →⟪ 𝕂² ⟫ Γ′
  κ↝ κ′ = weaken-↦ κ′ ads⊆

  txout↝ : Γ →⟪ Txout ⟫ Γ′
  txout↝ txout′ rewrite ids≡ = cons-↦ z ((-, -, T) at 0)
                                $ weaken-↦ txout′ ∈-++⁺ˡ
 -- (iii) sechash = sechash′, κ = κ′, txout extends txout′ with {z↦Tᵢₙᵢₜ}
 λˢ : 𝕣ᵗ Γₜ′
 λˢ = LIFTˢ 𝕣 Γ R≈ Γ′ txout↝ sechash↝ κ↝
```

There are no additional conditions in this case, so we simply declare the next computational move that submits the transaction to the blockchain.

```
data _⨾_⨾_~ℍ[4]~_⨾_ : StepRel where
 mkℍ : ∀ {h : H₄-args} (open H₄-args h)
  → let open H₄ h using (λˢ; T)
        -- (ii) append Tᵢₙᵢₜ to the blockchain
        λᶜ = submit (-, -, T)
    in
    ────────────────────────────────────────────────
    Γₜ″ ⨾ 𝕣∗ ⨾ (𝕒 , λˢ) ~ℍ[4]~ λᶜ ⨾ Rᶜ
```

6.3.1.5 Case (5)

This case corresponds to the `[C-AuthControl]` rule, where participants agree on picking one of the branches of the contract:

```
record H₅-args : Type where
 constructor mk
 field {c v x Γ₀ t A} : _⨾ {i} : Index c
 open ∣SELECT c i public
 field -- Hypotheses from [C-AuthControl]
       d≡ : A ∈ authDecorations d -- ...for some D = A : D′
 -- (i) Rˢ contains ⟨C,v⟩ₓ with C = D + ∑ᵢ Dᵢ
 open Transition ( (⟨ c , v ⟩at x ∣ Γ₀)                ⨾ t
                  —— auth-control⟦ A , x ▷ d ⟧ —→
                    (⟨ c , v ⟩at x ∣ A auth[ x ▷ d ] ∣ Γ₀) ⨾ t
```

```
                    ⊣ Act ([C-AuthControl] d≡)
                    ) public
```

The mappings stay unchanged, so no work is required for constructing `λˢ`:

```
module H₅ (h : H₅-args) (open H₅-args h) where
  -- (iv) txout = txout′, sechash = sechash′, κ = κ′
  λˢ : ℾᵗ Γₜ′
  λˢ = LIFTˢ ɾ Γ R≈ Γ′ id id id
```

This is also the first case where we need to track the **ancestor** advertisement of the current active contract, which has already been handled in Section 6.2.4. Apart from the transaction corresponding to the current subterm, the compiler wrapper also provides the keypair to sign the transaction broadcast to other participants (drawn from the `κ′` mapping):

```
private
  -- (ii) {G}C is the ancestor of ⟨C,v⟩ₓ in Rˢ
  T×K = COMPILE-ANCESTOR {Γ = Γ} {i = i} R≈ 𝟘 ɾ
  T   = BranchTx d∗ ∋ T×K .proj₁
  Kᵈ  = KeyPair     ∋ T×K .proj₂ d≡
  ∃T  = ∃Tx         ∋ Inᵈ d∗ , Outᵈ d∗ , T
abstract
  m ⌞T⌟ : Message
  m    = SIG Kᵈ ∃T
  ⌞T⌟ = encode T

  m≡ : m ≡ SIG Kᵈ ∃T
  m≡ = refl

  T≡ : ⌞T⌟ ≡ encode T
  T≡ = refl
```

Unfortunately, we cannot simply expose the transaction and key to be immediately used in the constructor — like we previously did in cases (3) and (4) — since the plethora of "green slime" (*i.e.,* computations in type indices) becomes unbearable for the poor Agda type checker. Hence the use of `abstract` to control unfolding by just providing the equational constraints on the exposed variables for the user to employ strictly when needed. (Note that this is another reason for using a helper module; `abstract` definitions are not allowed in mere `let`-expressions.)

As an aside, the next version of Agda implements recent ideas that allow finer control over unfolding [Gra+22], through a new `opaque` keyword to stand in for `abstract`.[3]

[3] https://agda.readthedocs.io/en/latest/language/opaque-definitions.html

If we were to use these new facilities, there would be no need for the exposed equations above.

On the computational side, a participant B broadcasts the transaction signed with the corresponding key, with the extra condition that the (encoding of the) transaction has previously been broadcast and we are emitting the first such signature:

```
data _⨾_⨾_~ℍ[5]~_⨾_ : StepRel where
  mkℍ : ∀ {h : H₅-args} (open H₅-args h) {B : Participant}
    → let open H₅ h using (λˢ; m; ⌞T⌟)
          -- (iii) B broadcasts the signed transaction
          λᶜ = B →∗∶ m
      in
    -- (v) Rᶜ contains B→∗:T for some B...
    ∀ (∃B : ∃ λ B → B →∗∶ ⌞T⌟ ∈ toList Rᶜ) →
    -- ... and m is the first signature of T after that
    • All (λ l → ∀ B → l ≢ B →∗∶ m) (Any-front $ ∃B .proj₂)
    ────────────────────────────────────────────────────────────
    Γₜ″ ⨾ ℝ∗ ⨾ (𝕒 , λˢ) ~ℍ[5]~ λᶜ ⨾ Rᶜ
```

6.3.1.6 Case (6)

Moving on to the case of a (timed) `put-reveal` command, the context becomes slightly more involved due to the fact that we have authorisations and time decorations in front of the branch, requiring some additional work to define the transition to begin with.

```
record H₆-args : Type where
  constructor mk
  field {c v y c′ y′ Γ₀ t p} : _
        {ds} : DepositRefs; {ss} : List (Participant × Secret × ℕ)
        {i}  : Index c
  vs  = unzip₃ ds .proj₂ .proj₁
  xs  = unzip₃ ds .proj₂ .proj₂ -- (i) xs = x₁⋯xₖ
  as  = unzip₃ ss .proj₂ .proj₁
  Γ₁  = || map (uncurry₃ ⟨_has_⟩at_) ds
  Δ   = || map (uncurry₃ _∶_♯_) ss
  Γ₂  = Cfg ∋ Δ | Γ₀
  Γ₁₂ = Cfg ∋ Γ₁ | Γ₂
  open ∣SELECT c i public
  As = decorations d .proj₁
  ts = decorations d .proj₂
  field
    t≡ : t ≡ maximum t ts                    -- let t be the maximum deadline
    d≡ : d ≡⋯∶ put xs &reveal as if p ⇒ c′ -- where D = ⋯ : put⋯reveal⋯.C′
    -- Hypotheses from [C-PutRev]
```

```
  fresh-y′ : y′ ∉ y L.∷ ids Γ₁₂
  p⟦Δ⟧≡ : ⟦ p ⟧ᵖ Δ ≡ just true
  -- Hypotheses from [Timeout]
  As≡∅ : Null As
 private
  α  = put⟪ xs , as , y ⟫
  Γ′ = ⟨ c′ , v + sum vs ⟩at y′ ∣ Γ₂

  ∀≤t : All (_≤ t) ts
  ∀≤t = ⟪ (λ ◆ → All (_≤ ◆) ts) ⟫ t≡ ~: ∀≤max t ts

  put→ : ⟨ [ d∗ ] , v ⟩at y ∣ Γ₁₂ —[ α ]→ Γ′
  put→ = ⟪ (λ ◆ → (⟨ [ ◆ ] , v ⟩at y ∣ (Γ₁ ∣ Γ₂) —[ α ]→ Γ′)) ⟫ d≡
        ~: [C-PutRev] {ds = ds} {ss = ss} fresh-y′ p⟦Δ⟧≡
 -- ii) in Rˢ, α consumes ⟨D+C,v⟩y and deposits ⟨Aᵢ,vᵢ⟩ₓᵢ to produce ⟨C′,v′⟩y′
 open Transition ( (⟨ c , v ⟩at y ∣ Γ₁₂) ⨾ t —— α —→ Γ′ ⨾ t
                 ⊣ [Timeout] As≡∅ ∀≤t put→ refl
                 ) public
```

Notation

Substitution syntax.. The standard `subst` function that rewrites an equality in a type is always awkward to use because it is difficult to remember which way the equality should be directed. To remedy that, our Prelude provides an alternative syntax that is more intuitive as regards to orientation, as evidenced in terms `∀≤t` and `put→` above (*c.f.*, Appendix A.12 for more details).

Similarly to case (5), we invoke the compiler by tracking down the ancestor advertisement in the trace history, but fortunately there is longer need to abstract away the results to aid Agda's type-checker, since we get more concrete input/output indices from the specific shape of the branch we are dealing with and we no longer need the subterm-keys to sign the transaction (handled by a previous `[C-AuthControl]`).

```
module H₆ (h : H₆-args) (open H₆-args h) where
 -- (ii) {G}C is the ancestor of ⟨C,v⟩ₓ in Rˢ
 T : Tx (suc $ length xs) 1
 T = COMPILE-ANCESTOR {Γ = Γ} {i = i} R≈ 0 𝕣 .proj₁ :~ d≡ ⟪ BranchTx ⟫
```

Regarding symbolic mappings, we simply need to extend the `txout` mapping to match the most recent identifier `y` to the first (and sole) output of `T`:

```
-- (v) extend txout′ with {y′↦(T,0)}, sechash = sechash′, κ = κ′
txout↝ : Γ →⦅ Txout ⦆ Γ′
txout↝ txout′ = cons-↦ y′ ((-, -, T) at 0) $ weaken-↦ txout′ ids⊆
```

```
sechash⤳ : Γ →⦅ Sechash ⦆ Γ′
sechash⤳ = lift Γ —⟨ namesˡ ⟩— Γ′ ⊣ secrets≡

κ⤳ : Γ →⦅ 𝕂² ⦆ Γ′
κ⤳ = lift Γ —⟨ advertisements ⟩— Γ′ ⊣ ads≡

  λˢ : ℾᵗ Γₜ′
  λˢ = LIFTˢ ɼ Γ R≈ Γ′ txout⤳ sechash⤳ κ⤳
```

The computational move consists of submitting the transaction corresponding to the `put` command and returned by the compiler in `H₆`.

```
data _⨾_⨾_~ℍ[6]~_⨾_ : StepRel where
  mkℍ : ∀ {h : H₆-args} (open H₆-args h) →
    let open H₆ h using (λˢ; T)
        λᶜ = submit (-, -, T)
    in
    ──────────────────────────────────────
    Γₜ″ ⨾ ɼ∗ ⨾ (𝕒 , λˢ) ~ℍ[6]~ λᶜ ⨾ Rᶜ
```

6.3.1.7 Case (7)

The next case corresponds to a participant revealing one of their secrets using `[C-AuthRev]`:

```
record H₇-args : Type where
  constructor mk
  field {ad A a n Γ₀ t} : _; k⃗ : 𝕂²′ ad; Δ×h̅ : List (Secret × Maybe ℕ × ℤ)
  open ∣AD ad public
  Δ = map drop₃ Δ×h̅
  h̅ = map select₃ Δ×h̅
  k̅ = concatMap (map pub ∘ codom) (codom k⃗)
  open Transition (  (⟨ A : a ♯ just n ⟩ ∣ Γ₀) ⨾ t
                   —— auth-rev⦅ A , a ⦆ —→
                     A : a ♯ n ∣ Γ₀           ⨾ t
                   ⊣ Act [C-AuthRev]
                   ) public
```

The resources remain unchanged in the new configuration, so the helper module performs the trivial identity lifting, in addition to retrieving the secret's hash from the `sechash` mapping.

```
module H₇ (h : H₇-args) (open H₇-args h) where
  -- (iii) txout = txout′, sechash = sechash′, κ = κ′
  λˢ : ℾᵗ Γₜ′
```

```
  λˢ = LIFTˢ r Γ R≈ Γ′ id id id
 private
  a∈Γ : a ∈ secrets (Rˢ •cfg)
  a∈Γ = ∈namesˡ-resp-≈ a {Γ}{Rˢ •cfg} (↭-sym $ R≈ .proj₂) 𝟘

 a♯ : HashId
 a♯ = sechashΓ {a} a∈Γ
  where open ℾ (ℝ∗⇒Γ r∗) using (sechashΓ)
```

The constructor's side-conditions ensure that the secret is of suitable size, has been computed using the oracle, and its owner A has previously authorised to reveal it.

```
data _;_;_~ℍ[7]~_;_ : StepRel where
 mkℍ : ∀ {h : H₇-args} {B} {mˢ : String} (let m = encode mˢ)
  → let open H₇-args h renaming (ad to ⟨G⟩C)
        open H₇ h using (λˢ; a♯)
        -- (i) some participant B broadcasts message m
        λᶜ = B →∗: m
    in
    -- ... with a corresponding broadcast of m′=(C,h̄,k̄) in Rᶜ
  ∀ (∃λ : ∃ λ B → ∃ λ txoutᶜ →
        let m′ = encode (encodeAd ⟨G⟩C txoutᶜ , h̄ , k̄)
        in B →∗: SIG (K B) m′ ∈ toList Rᶜ) →
  • a ∈ secrets G
  • ∣ m ∣ᵐ ≥ η
    -- (ii) in Rᶜ we find ⋯ (B → O : m) (O → B : sechash′(a)) for some B ⋯
  • (∃ λ B → (B , m , a♯) ∈ oracleInteractionsᶜ Rᶜ)
    -- (iv) in Rˢ, we find an A:{G}C,Δ action, with a in G
  • auth-commit⦅ A , ⟨G⟩C , Δ ⦆ ∈ labelsᴿ R
    -- (v) λᶜ is the first broadcast of m after m′
  • All (λ l → ∀ X → l ≢ X →∗: m) (Any-front $ ∃λ .proj₂ .proj₂)
    ────────────────────────────────────────────────────────────
    Γₜ″ ; r∗ ; (𝕒 , λˢ) ~ℍ[7]~ λᶜ ; Rᶜ
```

6.3.1.8 Cases (8) and (9)

The following two cases correspond to the `split` and `withdraw` commands, respectively. As these share the same structure with case (6) for `put` commands, we omit their definition here and refer the reader to the complete formalisation:

https://omelkonian.github.io/formal-bitml-to-bitcoin/Coherence/Hypotheses.html

6.3.1.9 Cases (10)-(15)

Cases (10)-(15) (as well as (16) and (17) which will be covered later) concern deposit manipulation, thus do not require any compiler invocations, although there is still need for updating mappings accordingly.

Rules for the possible actions of joining, dividing, donating, or destroying a deposit, come in pairs: first a participant authorises the action without changing the configuration, and only then the effects are enacted by a subsequent move. For the sake of brevity, we only cover cases (10) and (11) for joining two deposits, as well as cases (16) and (17) for destroying deposits that stand out, and refer the reader to the full formalisation for the other actions (12)-(15) that follow a very similar structure.

Authorising for two deposits to be joined is realised using the `[DEP-AuthJoin]` rule and does not require any mapping transformation since the resources remain constant:

```
record H₁₀-args : Type where
  constructor mk
  field {A v x v′ x′ Γ₀ t} : _
  open Transition (  (⟨ A has v ⟩at x ∣ ⟨ A has v′ ⟩at x′ ∣ Γ₀) ; t
                  —— auth-join⦅ A , x ↔ x′ ⦆ —→
                     ( ⟨ A has v ⟩at x ∣ ⟨ A has v′ ⟩at x′
                     ∣ A auth[ x ↔ x′ ▷⟨ A , v + v′ ⟩ ] ∣ Γ₀)    ; t
                  ⊣ Act [DEP-AuthJoin]
                  ) public


module H₁₀ (h : H₁₀-args) (open H₁₀-args h) where
  -- (v) txout = txout′, sechash = sechash′, κ = κ′
  λˢ : 𝕣ᵗ Γₜ′
  λˢ = LIFTˢ 𝕣 Γ R≈ Γ′ id id id
```

The rest of the side-conditions read very closely to the paper, except for retrieving the Bitcoin outputs that correspond to the deposits being joined, which imposes a mild *name resolution* overhead that is offloaded to the n⊆ lemma:

```
data _;_;_~ℍ[10]~_;_ : StepRel where
  mkℍ : ∀ {h : H₁₀-args} (open H₁₀-args h) {B}
    → let n⊆ : Γ ⊆⦅ ids ⦆ Rˢ
          n⊆ = namesʳ⦅end⦆⊆ Rˢ
             ∘ ∈namesʳ-resp-≈ _ {Γ}{Rˢ ∙cfg} (↭-sym $ R≈ .proj₂)
      in
      -- (ii) In Rᶜ we find B→∗:T for some B,T...
    ∀ (∃B : ∃ λ B → ∃ λ (T : Tx 2 1) →
      (B →∗: encode T ∈ toList Rᶜ)
       -- ...where T has as its two inputs txout′(x) and txout′(x′)...
```

```
    × (T .inputs ≡ (hashTxⁱ <$> [ txout′ {x} (n⊆ 0) ; txout′ {x′} (n⊆ 1) ]))
    -- ...and a single output of value v+v′ redeemable with K̂(A)
    × (T .outputs ≡ [ (v + v′) redeemable-by K̂ A ]))
  → let
    -- (iii) broadcast transaction T, signed by A
    _ , T , B∈ , _ = ∃B
    m′ = SIG (K̂ A) (∃Tx ∋ -, -, T)
    λᶜ = B →∗: m′
    open H₁₀ h using (λˢ)
   in
   -- (iv) λᶜ is the first broadcast of m′ in Rᶜ after T
  • All (λ l → ∀ B → l ≢ B →∗: m′) (Any-front B∈)
   ────────────────────────────────────────────────────────────────
   Γₜ″ ; ℝ∗ ; (𝕒 , λˢ) ~ℍ[10]~ λᶜ ; Rᶜ
```

Notice how the intrinsically-types formulation of Bitcoin transactions gives us a more precise way to talk about its inputs and outputs; the type-checker ensures we never invoke this case for an ill-formed transaction.

The corresponding rule `[C-Join]` for enacting the above authorisation `[C-AuthJoin]` follows a similar pattern, although we now have to account for the changes in the configuration's resources, namely recording the newly created deposit in `txout`:

```
record H₁₁-args : Type where
 constructor mk
 field {A v x v′ x′ y Γ₀ t} : _
     -- Hypotheses from [DEP-Join]
     fresh-y : y ∉ x L.∷ x′ ∷ ids Γ₀
 open Transition (  (⟨ A has v ⟩at x ∣ ⟨ A has v′ ⟩at x′
                   ∣ A auth[ x ↔ x′ ▷⟨ A , v + v′ ⟩ ] ∣ Γ₀) ; t
                ―― join⟪ x ↔ x′ ⟫ ―→
                   (⟨ A has (v + v′) ⟩at y ∣ Γ₀)          ; t
                ⊣ Act ([DEP-Join] fresh-y)
                ) public


module H₁₁ (h : H₁₁-args) (open H₁₁-args h) (tx : TxInput′) where
 private
  -- (iii) extend txout′ with y↦T₀ (removing {x↦_;x′↦_}),
  --        sechash = sechash′, κ = κ′
  txout↝ : Γ →⦅ Txout ⦆ Γ′
  txout↝ txout′ = cons-↦ y tx $ weaken-↦ txout′ (λ x∈ → there (there x∈))

 λˢ : 𝕣ᵗ Γₜ′
 λˢ = LIFTˢ 𝕣 Γ R≈ Γ′ txout↝ id id
```

Notice that the helper module H₁₁ now requires an additional argument to construct the mappings, namely a transaction output to match to the newly created deposit, which will be provided by the computational part of the constructor below. The corresponding computational move submits a transaction that spends the previous two outputs/deposits to create a single new output that holds the sum of their values:

```
data _⨾_⨾_~ℍ[11]~_⨾_ : StepRel where
 mkℍ : ∀ {h : H₁₁-args}
   → let open H₁₁-args h

         n⊆ : Γ ⊆⟪ ids ⟫ Rˢ
         n⊆ = namesᶦ⟪end⟫⊆ Rˢ
            ∘ ∈namesᶦ-resp-≈ _ {Γ}{Rˢ •cfg} (↭-sym $ R≈ .proj₂)

         -- (ii) submit transaction T
         T : ∃Tx
         T = 2 , 1 , sig⋆ (V.replicate [ K̂ A ]) record
           { inputs = hashTxⁱ <$> [ txout′ {x} (n⊆ 0) ⨾ txout′ {x′} (n⊆ 1) ]
           ; wit     = wit⊥
           ; relLock = V.replicate 0
           ; outputs = [ (v + v′) redeemable-by K̂ A ]
           ; absLock = 0 }
         λᶜ = submit T

         open H₁₁ h (T at 0) using (λˢ)
     in
     ──────────────────────────────────────────────────────────
     Γₜ″ ⨾ 𝕣∗ ⨾ (𝕒 , λˢ) ~ℍ[11]~ λᶜ ⨾ Rᶜ
```

6.3.1.10 Cases (16) and (17)

The next couple of cases, albeit similar in structure with the aforementioned deposit handling cases, deserve special attention as they accommodate computational moves that that *cannot* be expressed in BitML but nonetheless interact with the resources we track in the `txout` mapping (*i.e.,* they spend outputs/deposits that previous *relevant* moves have created).

These two cases furthermore require a non-trivial *name resolution* procedure, and, in contrast to the simplistic formulation of the paper where the mappings remain unchanged, our more precise formulation mandates that we *weaken* the mappings accordingly to account for the *destroyed* resources.

First comes the authorisation to destroy a list of deposits via [DEP-AuthDestroy] :

```
record H₁₆-args : Type where
 constructor mk
 field {y Γ₀ t} : _; {ds} : DepositRefs; j : Index ds
       -- Hypotheses from [DEP-AuthDestroy]
       fresh-y : y ∉ ids Γ₀
 k  = length ds
 A  = (ds ‼ j) .proj₁
 xs = map (proj₂ ∘ proj₂) ds
 Δ  = || map (uncurry₃ ⟨_has_⟩at_) ds
 j′ = Index xs ∋ ‼-map {xs = ds} j
 -- (ii) in Rˢ we find ⟨Bᵢ,vᵢ⟩yᵢ for i ∈ 1..k
 open Transition (  (Δ | Γ₀)                          ; t
                 —— auth-destroy⟪ A , xs , j′ ⟫ —→
                    (Δ | A auth[ xs , j′ ▷ᵈˢ y ] | Γ₀) ; t
                 ⊣ Act ([DEP-AuthDestroy] fresh-y)
                 ) public
```

In order to retrieve the matching transaction outputs of the deposits we wish to destroy, we need to prove that all of their associated identifiers belong to the current `txout` mapping:

```
module H₁₆ (h : H₁₆-args) (open H₁₆-args h) where
 abstract
  xs↦ : xs ↦ TxInput′
  xs↦ = txout′ ∘ xs⊆
   where
    xs⊆ : xs ⊆ ids R
    xs⊆ = begin
      xs           ⊆⟨ namesʳ-∥map-authDestroy ds ⟩
      ids Δ        ⊆⟨ mapMaybe-⊆ isInj₂ $ ∈-collect-++⁺ˡ Δ Γ₀ ⟩
      ids Γ        ⊆⟨ ∈namesʳ-resp-≈ _ {Γ}{R ∙cfg} (↭-sym $ proj₂ R≈) ⟩
      ids (R .end) ⊆⟨ namesʳ⦅end⦆⊆ R ⟩
      ids R        ∎ where open ⊆-Reasoning Secret
```

As most authorisations, no resources are actually tampered with, so we lift the mappings as is:

```
 private
  txout↝ : Γ →⦅ Txout ⦆ Γ′
  txout↝ = lift Γ —⟨ ids ⟩— Γ′ ⊣ ids≡

  sechash↝ : Γ →⦅ Sechash ⦆ Γ′
  sechash↝ = lift Γ —⟨ secrets ⟩— Γ′ ⊣ secrets≡

  κ↝ : Γ →⦅ 𝕂² ⦆ Γ′
  κ↝ = lift Γ —⟨ advertisements ⟩— Γ′ ⊣ ads≡
```

```
-- (vii) txout = txout′, sechash = sechash′, κ = κ′
λˢ : 𝕣ᵗ Γₜ′
λˢ = LIFTˢ 𝕣 Γ R≈ Γ′ txout↝ sechash↝ κ↝
```

As for the computational counterpart: a participant signs the transaction that will later destroy said deposits, probably amongst other unrelated effects.

```
data _⨾_⨾_~ℍ[16]~_⨾_ : StepRel where
 mkℍ : ∀ {h : H₁₆-args} (open H₁₆-args h) {B}
   → let open H₁₆ h using (λˢ; xs↦) in
     -- (iii) in Rᶜ we find B → ∗ : T
     --       for some T having txout′(yᵢ) as inputs (+ possibly others)
   ∀ (∃B : ∃ λ B → ∃ λ T →
      (B →∗: encode (T .proj₂ .proj₂) ∈ toList Rᶜ)
      × ((hashTxⁱ <$> codom xs↦) ⊆ V.toList (∃inputs T)))
   → let -- (iv) broadcast transaction T, signed by A
         --      (corresponding to the j-th input)
         _ , T , B∈ , _ = ∃B
         m = SIG (K̂ A) T
         λᶜ = B →∗: m
     in
     -- (v) λᶜ is the first broadcast of m in Rᶜ after T
   • All (λ l → ∀ B → l ≢ B →∗: m) (Any-front B∈)
     -- (vi) λᶜ does not correspond to any *other* symbolic move
     -- handled afterwards by relation ~₁₂~
     ─────────────────────────────────────────────────
     Γₜ″ ⨾ 𝕣∗ ⨾ (𝕒 , λˢ) ~ℍ[16]~ λᶜ ⨾ Rᶜ
```

We are not yet equipped to handle condition (iv) that ascertains that no other symbolic move is possible; this will come later when we unite all these individual relations into the final coherence relation in Section 6.3.2 (the constraints for both cases (17) and (18) will be handled by the second stratum of the relation `_~₁₂_` in one go).

Once all participants have authorised the destruction of their deposits using the previous case, a `[DEP-Destroy]` move finally removes them from the current configuration:

```
record H₁₇-args : Type where
 constructor mk
 field {Γ₀ y t} : _; {ds} : DepositRefs; j : Index ds
 xs = map (proj₂ ∘ proj₂) ds -- (i) x⃗ = x₁⋯xₖ
 Δ = || mapWithIndex ds λ (i , Aᵢ , vᵢ , xᵢ) →
       ⟨ Aᵢ has vᵢ ⟩at xᵢ | Aᵢ auth[ xs , ‼-map {xs = ds} i ▷ᵈˢ y ]
 -- (ii) in Rˢ, α consumes ⟨Aᵢ,vᵢ⟩xᵢ to obtain ∅
 open Transition ( (Δ | Γ₀) ⨾ t —— destroy⟪ xs ⟫ —→ Γ₀ ⨾ t
```

```
                       ⊣ Act [DEP-Destroy]
                       ) public
```

As promised, we are being honest with the exact resources that we record, so have to weaken all mappings to reflect that:

```
private
  txout↝ : Γ →⦅ Txout ⦆ Γ′
  txout↝ txout′ rewrite ids≡ = weaken-↦ txout′ (∈-++⁺ʳ _)

  sechash↝ : Γ →⦅ Sechash ⦆ Γ′
  sechash↝ sechash′ rewrite secrets≡ = weaken-↦ sechash′ (∈-++⁺ʳ _)

  κ↝ : Γ →⦅ 𝕂² ⦆ Γ′
  κ↝ κ′ = weaken-↦ κ′ (∈-collect-++⁺ʳ Δ Γ₀)
-- (v) × txout = txout′, sechash = sechash′, κ = κ′
--      ✓ remove {⋯ xᵢ ↦ (Tᵢ,j) ⋯} from txout′
λˢ : 𝕣ᵗ Γₜ′
λˢ = LIFTˢ 𝕣 Γ R≈ Γ′ txout↝ sechash↝ κ↝
```

The corresponding computational move is to submit a transaction that consumes the destroyed outputs/deposits:

```
data _⨾_⨾_~ℍ[17]~_⨾_ : StepRel where
  mkℍ : ∀ {h : H₁₇-args} (open H₁₇-args h)
    → let open H₁₇ h using (λˢ; xs↦) in
      -- (iii) submit T, having as some of its inputs txout′(x₁)⋯txout′(xₖ)
    ∀ {i : ℕ} (T : Tx i 0) →
    • (hashTxⁱ <$> codom xs↦) ⊆ V.toList (T .inputs)
    → let λᶜ = submit (_ , _ , T) in
      -- (iv) λᶜ does not correspond to any *other* symbolic move
      -- handled afterwards by relation ~₁₂~
      ────────────────────────────────────────────────────────
      Γₜ″ ⨾ 𝕣* ⨾ (ⓐ , λˢ) ~ℍ[17]~ λᶜ ⨾ Rᶜ
```

6.3.1.11 Case (18)

We finish this section's epic with, ironically, the simplest case that lets us progress time by `[Delay]`: the configuration remain unchanged, so there is no need for updating the symbolic mappings:

```
record H₁₈-args : Type where
  constructor mk
  field {Γ t δ} : _; δ>0 : δ > 0
  open Transition ( Γ ⨾ t —— delay⦅ δ ⦆ —→ Γ ⨾ (t + δ)
                       ⊣ [Delay] δ>0
                       ) public
```

Decidability. Most of the propositions that appear as hypotheses are actually decidable, so we can replay the same construction as the one for BitML's inference rules (*c.f.,* Section 4.7.6): systematically define alternative datatype constructors that use proof-by-reflection to automatically discharge the proof obligations, something quite useful for when reasoning about closed terms without any free variables.

Unfortunately, as we will soon see in Section 6.3.4, the amount of inference Agda has to take on her shoulders is just too much for any practical purposes, either due to scale and performance issues, or even due to `postulate`s and non-definitional equalities blocking computation, which is what drives proof-by-reflection; another name for it is proof-by-computation after all.

Nonetheless, let us show, for the sake of completeness, how decidability of case (2) can be established:

1. Define the decidable counterpart of `ℍ-Run`, which takes care of the current run and its mappings, while permuting the last configuration:

```
open import Prelude.Init renaming (toWitness to _¡)

record ℍ-Run? {Γₜ α Γₜ′} (Γ→ : Γₜ —[ α ]→ₜ Γₜ′) : Type where
  private Γ′ = Γₜ′ .cfg; t′ = Γₜ′ .time
  field {Rᶜ} : CRun  ; {Rˢ} : Run
        {𝕣∗} : ℝ∗ Rˢ ; {R≈?} : auto: Rˢ ≈⋯ Γₜ
        {Γ″} : Cfg   ; {Γ≈?} : auto: Γ″ ≈ Γ′
  R≈ = R≈? ¡; Γ≈ = Γ≈? ¡

auto-ℍ-Run : ∀ {Γ→ : Γₜ —[ α ]→ₜ Γₜ′} → {h : ℍ-Run? Γ→} → ℍ-Run Γ→
auto-ℍ-Run {h = h} = record {ℍ-Run? h}
```

Notation

Records-as-modules syntax. The proof that our decision procedure is complete, *i.e.,* that we can always reconstruct `ℍ-Run` from its decidable counterpart `ℍ-Run?`, is given here by constructing a record by opening a module inside it[a] that happens to have all the field names defined beforehand (hence the definitions of `R≈` and `Γ≈` inside `ℍ-Run?`).

[a] https://agda.readthedocs.io/en/v2.6.3/language/record-types.html#building-records-from-modules

2. Once the permutation business has been taken care of, repeat the definition of the context record of the case we are interested in, replacing every explicit proof argument with its implicit `auto:` version:

```
record H₂-args? : Type where
  field {ad Γ₀ t A} : _; {k⃗} : 𝕂²′ ad; {Δ×h̄} : List (Secret × Maybe ℕ × ℤ)
  Δ  = List (Secret × Maybe ℕ) ∋ map drop₃ Δ×h̄
  Δᶜ = || map (uncurry ⟨ A :_♯_⟩) Δ
  as = unzip Δ .proj₁
  ms = unzip Δ .proj₂
  field {as≡? } : auto: as ≡ secretsOfᵖ A (ad .G)
        {All∉? } : auto: All (_∉ secretsOfᶜᶠ A Γ₀) as
        {Hon⇒? } : True (A ∈Hon⇒? ms)
  as≡ = as≡? ¡; All∉ = All∉? ¡; Hon⇒ = Hon⇒? ¡
  open Transition (  (` ad | Γ₀)                          ; t
                  —— auth-commit⟪ A , ad , Δ ⟫ —→
                     (` ad | Γ₀ | Δᶜ | A auth[ ♯▷ ad ]) ; t
                  ⊣ Act ([C-AuthCommit] as≡ All∉ Hon⇒)
                  )
  field {ℍr? } : ℍ-Run? Γ→
  ℍr = auto-ℍ-Run {h = ℍr? }

auto-H₂ : {H₂-args? } → H₂-args
auto-H₂ {h?} = record {H₂-args? h?}
```

3. Last, do the same for the corresponding relation's datatype constructor, replacing the explicit proof terms as before:

```
mkℍ₂ : ∀ {h? : H₂-args? } {B : Participant} →
  let open H₂-args (auto-H₂ {h?}) renaming (ad to ⟨G⟩C)
      C = encodeAd ⟨G⟩C (ad⊆⇒Txout {⟨G⟩C}{Γ}{Rˢ} 𝟘 R≈ txout′)
      m = SIG (K A) $ encode (C , h̄ , k̄)
  in
  ∀ {∃B? : auto: ∃ λ B → (B →∗: C) ∈ toList Rᶜ} (let ∃B = ∃B? ¡)
  → {auto: All (λ l → ∀ X → l ≢ X →∗: C) (Any-tail $ ∃B .proj₂)}
  → {auto: All (λ hᵢ → ∣ hᵢ ∣ᵐ ≡ η) h̄}
  → {auto: All (λ l → ∀ X → l ≢ X →∗: m) (Any-front $ ∃B .proj₂)}
  → {auto: CheckOracleInteractions Rᶜ Δ×h̄}
  → {auto: Unique h̄}
  → {auto: Disjoint h̄ (codom sechash′)}
  → _ ; _ ; _ ~ℍ[2]~ _ ; _
mkℍ₂ {h?}{B}{p₁}{p₂}{p₃}{p₄}{p₅}{p₆}{p₇} =
  mkℍ {auto-H₂ {h?}}{B} (p₁ ¡) (p₂ ¡) (p₃ ¡) (p₄ ¡) (p₅ ¡) (p₆ ¡) (p₇ ¡)
```

6.3.2 The Stratified relation *(§8 & §A.6,Def.20 of [BZ18])*

It is finally time for the coherence relation to take form, first by combining all of the cases defined in Section 6.3.1 into a relation $_\sim_1_$ that captures the first 'Inductive case' of

relevant steps (*i.e.*, where a 1-to-1 correspondence between symbolic and computational moves is established). However, the negative position in the hypotheses of cases (16) and (17) oblige us to to *stratify* the relation into two separate relations `_~₁₁_` and `_~₁₂_`:

```
data _⨾_⨾_~₁₁_⨾_ : StepRel where

  [1] : ∀ {ᵣ* : ℝ* Rˢ} {λˢ : 𝕃 Rˢ Γₜ} →
    Γₜ ⨾ ᵣ* ⨾ λˢ ~ℍ[1]~ λᶜ ⨾ Rᶜ
    ────────────────────────────────────────
    Γₜ ⨾ ᵣ* ⨾ λˢ ~₁₁ λᶜ ⨾ Rᶜ
                    ⋮
  [15] : ∀ {ᵣ* : ℝ* Rˢ} {λˢ : 𝕃 Rˢ Γₜ} →
    Γₜ ⨾ ᵣ* ⨾ λˢ ~ℍ[15]~ λᶜ ⨾ Rᶜ
    ────────────────────────────────────────
    Γₜ ⨾ ᵣ* ⨾ λˢ ~₁₁ λᶜ ⨾ Rᶜ

  [18] : ∀ {ᵣ* : ℝ* Rˢ} {λˢ : 𝕃 Rˢ Γₜ} →
    Γₜ ⨾ ᵣ* ⨾ λˢ ~ℍ[18]~ λᶜ ⨾ Rᶜ
    ────────────────────────────────────────
    Γₜ ⨾ ᵣ* ⨾ λˢ ~₁₁ λᶜ ⨾ Rᶜ

data _⨾_⨾_~₁₂_⨾_ : StepRel where

  [16] : ∀ {ᵣ* : ℝ* Rˢ} {λˢ : 𝕃 Rˢ Γₜ} →
    Γₜ ⨾ ᵣ* ⨾ λˢ ~ℍ[16]~ λᶜ ⨾ Rᶜ
    ────────────────────────────────────────
    Γₜ ⨾ ᵣ* ⨾ λˢ ~₁₂ λᶜ ⨾ Rᶜ

  [17] : ∀ {ᵣ* : ℝ* Rˢ} {λˢ : 𝕃 Rˢ Γₜ} →
    Γₜ ⨾ ᵣ* ⨾ λˢ ~ℍ[17]~ λᶜ ⨾ Rᶜ
    ────────────────────────────────────────
    Γₜ ⨾ ᵣ* ⨾ λˢ ~₁₂ λᶜ ⨾ Rᶜ

_⨾_⨾_≁₁₁_⨾_ : StepRel
Γₜ ⨾ ᵣ* ⨾ λˢ ≁₁₁ λᶜ ⨾ Rᶜ = ¬ (Γₜ ⨾ ᵣ* ⨾ λˢ ~₁₁ λᶜ ⨾ Rᶜ)

data _⨾_⨾_~₁_⨾_ : StepRel where

  [L]_ : ∀ {ᵣ* : ℝ* Rˢ} {λˢ : 𝕃 Rˢ Γₜ} →
    Γₜ ⨾ ᵣ* ⨾ λˢ ~₁₁ λᶜ ⨾ Rᶜ
    ────────────────────────────────────────
    Γₜ ⨾ ᵣ* ⨾ λˢ ~₁ λᶜ ⨾ Rᶜ

  [R] : ∀ {ᵣ* : ℝ* Rˢ} {λˢ : 𝕃 Rˢ Γₜ} →
    • Γₜ ⨾ ᵣ* ⨾ λˢ ≁₁₁ λᶜ ⨾ Rᶜ
    • Γₜ ⨾ ᵣ* ⨾ λˢ ~₁₂ λᶜ ⨾ Rᶜ
```

```
  ────────────────────────────────
  Γₜ ⨾ ℝ∗ ⨾ λˢ ~₁ λᶜ ⨾ Rᶜ
```

Since case (3) of 'Inductive case 2' is also predicated on the negation of all other cases applying, the same stratification is required one level higher: we define an intermediate `_~_` relation to hold the irrelevant cases of 'Inductive case 2' (*i.e.,* ones where only the computational model is advanced with messages/transactions that not relevant for BitML):

```
data _~₂_::ʳ_ : SRun → CRun → C.Label → Type where

 [1] : ∀ {Rˢ} {T : ∃Tx} (let ℝ = ℝ∗⇒ℝ (Rˢ .proj₂); open ℝ ℝ) →
  -- no input of T belongs to ran txout
  T .proj₂ .proj₂ .inputs ♯ (hashTxⁱ <$> codom txout′)
  ──────────────────────────────────────────────
  Rˢ ~₂ Rᶜ ::ʳ submit T

 [2] : ∀ {Rˢ} →
  (λᶜ ≡ A →O: m) ⊎ (λᶜ ≡ O→ A : m)
  ──────────────────────────────
  Rˢ ~₂ Rᶜ ::ʳ λᶜ

 [2]′ : ∀ {Rˢ} →
  ─────────────
  Rˢ ~₂ Rᶜ ::ʳ delay 0

 [3] : ∀ {ℝ∗ : ℝ∗ Rˢ} →
  let λᶜ = A →∗: m in
  -- λᶜ does not correspond to any symbolic move
  (∀ Γₜ λˢ → Γₜ ⨾ ℝ∗ ⨾ λˢ ≁₁ λᶜ ⨾ Rᶜ)
  ──────────────────────────────
  (-, ℝ∗) ~₂ Rᶜ ::ʳ λᶜ
```

The first case (`[1]`) submits transactions that have no overlap with the current `txout` mapping; `[2]` covers all oracle interactions (no symbolic counterpart to that); and the last case acts as a "catch-all" clause for all other computational moves.

For the final act, we tie the recursive knot by providing the base case and merging the two inductive cases in `_~′_`, and hide the mapping information to expose the final coherence relation `_~_`.

```
data _~′_ : SRun → CRun → Type where
 base :
  ∀ {ℽ : ℾᵗ Γₜ₀} (let open ℾᵗ ℽ; Γ₀ = Γₜ₀ .cfg)
   -- (i) Rˢ = Γ₀ ∣ 0, with Γ₀ initial
   (init : Initial Γₜ₀)
   -- (ii) Rᶜ = T₀ ⋯ initial
   (cinit : Initial Rᶜ) →
```

```
    -- (iii) generation of public keys according to Def.13
    -- already covered by initiality of Rᶜ
    -- (iv) ⟨A,v⟩ₓ ∈ Γ₀ ⇒ txout{ x ↦ (v₿ spendable with K̂(A)(rₐ)) ∈ T₀ }
  • (∀ {A v x} (d∈ : ⟨ A has v ⟩at x ∈ᶜ Γ₀) →
     let ∃T₀ , _ = cinit; _ , o , T₀ = ∃T₀ in
     ∃ λ oᵢ → (txoutΓ (deposit∈Γ⇒namesʳ {Γ = Γ₀} d∈) ≡ ∃T₀ at oᵢ)
           × (T₀ ‼ᵒ oᵢ ≡ v redeemable-by K̂ A))
    -- (v)  dom sechash = ∅
    -- (vi) dom κ       = ∅
    -- by definition of Initial/ℝ
    ─────────────────────────────────────────────────────────
    (-, ℽ ■⊣ init ✓) ~′ Rᶜ

  step₁ : ∀ {Rˢ} {𝕣∗ : ℝ∗ Rˢ} {λˢ : 𝕃 Rˢ Γₜ} →
  • (-, 𝕣∗) ~′ Rᶜ
  • Γₜ ⨾ 𝕣∗ ⨾ λˢ ~₁ λᶜ ⨾ Rᶜ
    ────────────────────────────
    (-, Γₜ ∷ 𝕣∗ ⊣ λˢ ✓) ~′ (λᶜ ∷ Rᶜ ✓)

  step₂ : ∀ {Rˢ} →
  • Rˢ ~′ Rᶜ
  • Rˢ ~₂ Rᶜ ∷ʳ λᶜ
    ──────────────
    Rˢ ~′ (λᶜ ∷ Rᶜ ✓)

_~_ : S.Run → CRun → Type
Rˢ ~ Rᶜ = ∃ λ (𝕣∗ : ℝ∗ Rˢ) → (-, 𝕣∗) ~′ Rᶜ
```

The base case starts us off on initial symbolic and computational runs and makes sure the initial `txout` mapping is set up properly: *i.e.*, values match between the symbolic deposits in the current configuration and the Bitcoin outputs they point to, and they are also locked by the private key of their owner.

Notation

Abbreviated constructors. While the coherence relation is stratified, we can define shorthands for doing case analysis as if it was a flat non-stratified relation:

```
pattern [L1]_ h = [L] [1] h
                    ⋮
pattern [L18]_ h = [L] [18] h
pattern [R16⊣_]_ ¬coh h = [R] ¬coh ([16] h)
pattern [R17⊣_]_ ¬coh h = [R] ¬coh ([17] h)
```

Reasoning syntax for proofs of coherence. If we stick to constructing Agda terms using the above constructors, constructing proofs of coherence will not be as intuitive. Instead, we want to be able to express successive steps in a linear chain of reasoning (akin to equational reasoning), where we start at an initial pair of symbolic and computational states, then extend this incrementally either on both sides (relevant steps) or only computationally (irrelevant steps), while at the same time discharging proof obligations that might arise at each step (an alternative would be to devise a method for delaying the proofs until the very end, which has been done in Agda before using a *proof delay applicative functor* [OCo19], but we refrain from complicating our definitions at this point).

```
█_⊣_,_~_⊣_⊣_ :
  ∀ Γₜ₀ (init : Initial Γₜ₀) (ɣ₀ : ℾᵗ Γₜ₀) →
  ∀ (rᶜ : C.Run) (cinit : Initial rᶜ) →
  let open ℾᵗ ɣ₀; Γ₀ = Γₜ₀ .cfg in
  (∀ {A v x} (d∈ : ⟨ A has v ⟩at x ∈ᶜ Γ₀) →
      let ∃T₀ , _ = cinit; _ , o , T₀ = ∃T₀ in
      ∃ λ oᵢ → (txoutΓ (deposit∈Γ⇒namesʳ {Γ = Γ₀} d∈) ≡ ∃T₀ at oᵢ)
          × (T₀ ‼ᵒ oᵢ ≡ v redeemable-by K̂ A))
  → (Γₜ₀ █⊣ init) ~ (rᶜ █⊣ cinit ✓)
█ Γₜ ⊣ init , ɣ₀ ~ Rᶜ ⊣ cinit ⊣ txout≈ =
  -, base {ɣ = ɣ₀} init cinit txout≈

_—→_⊣_~_⊣_ : ∀ {Rˢ Rᶜ} (coh : Rˢ ~ Rᶜ) Γₜ (λˢ : 𝕃 Rˢ Γₜ) λᶜ →
  Γₜ ⨾ coh .proj₁ ⨾ λˢ ~₁ λᶜ ⨾ Rᶜ
  ────────────────────────────────────────────
  (Γₜ ∷ Rˢ ⊣ λˢ .proj₁) ~ (λᶜ ∷ Rᶜ ✓)
(ɪr∗ , coh) —→ Γₜ ⊣ λˢ ~ λᶜ ⊣ p =
  Γₜ ∷ ɪr∗ ⊣ λˢ ✓ , step₁ {λˢ = λˢ} coh p

_—→ᵋ_⊣_ : ∀ {Rˢ Rᶜ} (coh : Rˢ ~ Rᶜ) λᶜ →
  (-, coh .proj₁) ~₂ Rᶜ ∷ʳ λᶜ
  ──────────────────────────────
  Rˢ ~ (λᶜ ∷ Rᶜ ✓)
(ɪr∗ , coh) —→ᵋ λᶜ ⊣ p
  = ɪr∗ , step₂ coh p
```

These essentially comprise an alternative 'mixfix' notation to the constructor of the coherence relation. Do not worry if you cannot discern the syntax from these definitions just yet; it will become clear when we examine concrete examples in Section 6.3.4.

6.3.3 An example property of coherence

Let us put the humongous relation we just defined to the test, by proving a minor meta-theoretical property of coherence that we will later need for the examples in

Section 6.3.4, namely that it cannot be the case that we emit a computational label of the form `encode T` from the first inductive case (`_~₁_`) of relevant computational steps.

```
module _ {Rˢ Γₜ Rᶜ} {ℝ* : ℝ* Rˢ} {λˢ : 𝕃 Rˢ Γₜ} where
  get-λᶜ : Γₜ ⨾ ℝ* ⨾ λˢ ~₁ λᶜ ⨾ Rᶜ → C.Label
  get-λᶜ {λᶜ = λᶜ} _ = λᶜ

  get-λᶜ-correct : (coh : Γₜ ⨾ ℝ* ⨾ λˢ ~₁ λᶜ ⨾ Rᶜ)
                 → get-λᶜ coh ≡ λᶜ
  get-λᶜ-correct _ = refl

  module _ {A} (T : Tx i o) where abstract
    λᶜ≢encodeT : (coh : Γₜ ⨾ ℝ* ⨾ λˢ ~₁ λᶜ ⨾ Rᶜ)
               → get-λᶜ coh ≢ A →*: encode T
    λᶜ≢encodeT coh with coh
    ... | [L1] mkℍ = label≢ encode≢
    ... | [L2] mkℍ _ _ _ _ _ _ _ _ = label≢ (SIG≢encode {y = T})
    ... | [L3] mkℍ _ _ = label≢ (SIG≢encode {y = T})
    ... | [L4] mkℍ = λ ()
    ... | [L5] mkℍ {h} _ _ = label≢
                          $ subst (_≢ encode T) (sym $ H₅.m≡ h)
                          $ SIG≢encode {y = T}
    ... | [L6] mkℍ = λ ()
    ... | [L7] mkℍ _ _ _ _ _ _ _ = label≢ encode≢
    ... | [L8] mkℍ = λ ()
    ... | [L9] mkℍ = λ ()
    ... | [L10] mkℍ (_ , _ , _) _ = label≢ (SIG≢encode {y = T})
    ... | [L11] mkℍ = λ ()
    ... | [L12] mkℍ (_ , _ , _) _ = label≢ (SIG≢encode {y = T})
    ... | [L13] mkℍ = λ ()
    ... | [L14] mkℍ (_ , _ , _) _ = label≢ (SIG≢encode {y = T})
    ... | [L15] mkℍ = λ ()
    ... | [R16-⊣ _ ] mkℍ (_ , _ , _) _ = label≢ (SIG≢encode {y = T})
    ... | [R17-⊣ _ ] mkℍ _ _ = λ ()
    ... | [L18] mkℍ = λ ()

    ≁₁-encodeT : Γₜ ⨾ ℝ* ⨾ λˢ ≁₁ A →*: encode T ⨾ Rᶜ
    ≁₁-encodeT coh = λᶜ≢encodeT coh $ get-λᶜ-correct coh
```

Notice the complication arising from the use of `abstract` in the helper module `H₅`, where we need to use the exposed equation `m≡` in contrast to all the other cases where unification does the work for us. It might be entertaining (or heart-breaking, depending where you stand) to find out that if we instead `rewrite` with the same equation that we are `subst`ituting, we quickly bring Agda down to her knees and have an unresponsive type-checker in our hands.

6.3.4 Example coherence proof: timed commitment protocol

6.3.4.1 Detailed version (type-checked)

At this point, we expect the reader to feel somewhat overwhelmed by the sheer technical complexity of the coherence relation. Let us make things clearer by demonstrating an example proof of coherence on some concrete BitML contract. And what better example to choose than the timed commitment protocol that has served as the running example throughout the thesis (BitML syntax in Section 4.6; BitML's operational semantics in Section 4.8; compiling to Bitcoin transactions in Section 5.3).

Given the compiled transactions T_{init}, T', and T'^{a} (as well as the "genesis" transaction T_0 which bootstraps the whole thing) from the compilation example in Section 5.3, we will work in a step-wise fashion to incrementally grow a symbolic run corresponding to the case where A's secret is revealed in time, in tandem with a list of computational moves that involve broadcast messages and submitting the aforementioned transactions that we compiled to the blockchain.

Each step `i` will be contained in its own module $Step_i$, so that we can retain a consistent naming convention for intermediate variables throughout the example (*e.g.,* source and target symbolic configuration). Moreover, each step will refer to the previous step-module as `«`, *e.g.,* to get a hold of the so-far constructed proof of coherence and extend it with a single step.

The base case starts from an initial state where:

- symbolically, the configuration only holds the persistent deposits of participants A and B;
- computationally, transaction T_0 has been submitted on-chain and the participants have exchanged the public parts of both their keys K and $\hat{K}$.

```
module Step₁ where
  Γ₀ = Cfg ∋ ⟨ A has 1 ₿ ⟩at x | ⟨ B has 0 ⟩at y

  γ₀ : Γᵗ (Γ₀ at 0)
  γ₀ = [txout: (λ where 𝟘 → Tᵃ ; 𝟙 → Tᵇ) |sechash: (λ ()) |κ: (λ ()) ]

  Rᶜ₀ : C.Labels
  Rᶜ₀ = [ submit (-, -, T₀)
        ⨾ (A →∗: encode (Kᵖ A , K̂ᵖ A))
        ⨾ (B →∗: encode (Kᵖ B , K̂ᵖ B))
        ]
```

```
cinit : Initial Rᶜ₀
cinit = (-, -, T₀) , (λ where 𝟘 → 𝟘; 𝟙 → 𝟙) , refl

Rˢ = (Γ₀ at 0) ∎⊣ auto
Rᶜ = CRun ∋ Rᶜ₀ ∎⊣ cinit ✓

coh : Rˢ ~ Rᶜ
coh = -, base {𝕪 = 𝕪₀} auto cinit
  λ where 𝟘 → 0F , refl , refl; 𝟙 → 1F , refl , refl
𝕣∗ = coh .proj₁
```

Establishing the base case simply involves proving initiality for both the symbolic and the computational run. We expose the new mappings `𝕣∗` of the resulting symbolic run, so that the subsequent steps can extend them.

For the rest of the example, assume that the required key(-pairs) `K` and `K²` have been generated (direct consequence of the `keyPairs` module parameter we introduced in Section 6.1.3).

```
k̄ = List ℤ ∋ concatMap (map pub ∘ codom) (codom K²)
```

The next step advertises the `TC` contract, resulting in a new configuration `Γ′` which extends the previous run `«.Rˢ` to `Rˢ`. Since this is a symbolic `[C-Advertise]` run, we establish coherence from the `[L1]` case by broadcasting the encoding of the advertisement `GC` (hence the need to use the `txout` mapping from the previous step).

```
module Step₂ (let module « = Step₁ ; 𝕣 = ℝ∗⇒ℝ «.𝕣∗) where
  α  = advertise⦅ TC ⦆
  Γ′ = Cfg ∋ ` TC ∣ ⟨ A has 1 ₿ ⟩at x ∣ ⟨ B has 0 ₿ ⟩at y
  Rˢ = (Γ′ at 0) ⟨ Act {t = 0} $ C-Advertise {ad = TC} {Γ = «.Γ₀} ⟩←—— «.Rˢ

  vad = ValidAd TC ∋ auto
  d⊆  = TC ⊆⦅ deposits ⦆ «.Γ₀ ∋ auto .unmk⊆
  txoutΓ = Txout Γ′ ∋ Txout≈ {«.Rˢ ∙cfg}{«.Γ₀} auto (𝕣 ∙txoutEnd_)
  txoutG = Txout TC ∋ weaken-↦ txoutΓ (deposits⊆⇒namesʳ⊆ {TC}{«.Γ₀} d⊆)
  txoutC = Txout C  ∋ weaken-↦ txoutG (mapMaybe-⊆ isInj₂ $ vad ∙names-⊆)

  GC = encodeAd TC (txoutG , txoutC)
  λᶜ = A →∗: GC
  Rᶜ = λᶜ ∷ «.Rᶜ ✓

  coh : Rˢ ~ Rᶜ
  coh = -, step₁ («.coh .proj₂) ([L1] mkℍ)
  𝕣∗ = coh .proj₁
```

Since `A` will reveal their secret in an honest way, they need to interact with the oracle to calculate the secret's hash. The next two steps realise this bidirectional communication;

the associated coherence proofs are trivial as this is an *irrelevant* computational move without any symbolic counterpart.

```
module Step₃ (let module ≪ = Step₂) where
  Rˢ = ≪.Rˢ
  λᶜ = A →O: encode a
  Rᶜ = λᶜ ∷ ≪.Rᶜ ✓

  coh : Rˢ ~ Rᶜ
  coh = -, step₂ (≪.coh .proj₂) ([2] (inj₁ refl))
  𝕣∗ = coh .proj₁
module Step₄ (let module ≪ = Step₃) where
  Rˢ = ≪.Rˢ
  λᶜ = O→ A : a♯
  Rᶜ = λᶜ ∷ ≪.Rᶜ ✓

  coh : Rˢ ~ Rᶜ
  coh = -, step₂ (≪.coh .proj₂) ([2] (inj₂ refl))
  𝕣∗ = coh .proj₁
```

A then goes on to commit their secret for the stipulation of the TC contract; this again involves encoding the contract into its bitstring version GC using the `txout` mapping provided by the trace properties of Section 6.2.3.

```
module Step₅ (let module ≪≪ = Step₂; module ≪ = Step₄; open ℝ (ℝ∗⇒ℝ ≪.𝕣∗)) where
  Γ₀ Γ Γ′ : Cfg
  Γ₀ = ⟨ A has 1 ₿ ⟩at x ∣ ⟨ B has 0 ₿ ⟩at y
  Γ  = ` TC ∣ Γ₀
  Γ′ = ` TC ∣ Γ₀ ∣ ⟨ A : a ♯ just N ⟩ ∣ A auth[ ♯▷ TC ]

  Γ→ : Γ at 0 —[ auth-commit⦅ A , TC , [ a , just N ] ⦆ ]→ₜ Γ′ at 0
  Γ→ = Act $ [C-AuthCommit] refl auto (λ _ → auto)

  Rˢ = (Γ′ at 0) ⟨ Γ→ ⟩←—— ≪.Rˢ

  txoutGC = ad∈⇒Txout {TC}{≪≪.Γ′}{≪.Rˢ}{0} (here refl) auto txout′
  txoutG  = txoutGC .proj₁; txoutC = txoutGC .proj₂
  GC      = encodeAd TC txoutGC

  m  = SIG (Kᵖᵘᵇ A) $ encode (GC , [ a♯ ] , k̄)
  λᶜ = A →∗: m
  Rᶜ = λᶜ ∷ ≪.Rᶜ ✓

  coh : Rˢ ~ Rᶜ
  coh = -, step₁ (≪.coh .proj₂)
    ([L2] mkℍ {h = h} ∃B first-∃B [ h≡ ] first-λᶜ h∈O auto (λ ()))
    where
```

```
    h : H₂-args
    h = mk {TC}{Γ₀}{0}{A} K² [ a , just N , a♯ ] auto auto (λ _ → auto)
           («.Rᶜ ⨾ «.Rˢ ⨾ «.𝕣∗ ⊣ auto ≈ Γ′ ⊣ auto)
```

In contrast to the previous steps, Agda cannot fully infer the context arguments, so we construct them explicitly in `h`.

As for the proof obligations needed for case `[L2]`, we need to first argue that the encoding is the same as the one used in `Step₂` (*i.e.*, the values of the `txout` mappings used agree), as well as postulate the cryptographic constraints regarding the `η` security parameter.

```
C≡ : GC ≡ ««.GC
C≡ = encode-txout≡ TC txoutG ««.txoutG txoutC ««.txoutC
       (λ where 𝟘 → refl; 𝟙 → refl) λ ()

∃B : ∃ λ B → (B →∗: GC) ∈ toList «.Rᶜ
∃B rewrite C≡ = A , 𝟚

first-∃B : All (λ l → ∀ X → l ≢ X →∗: GC) (Any-tail $ ∃B .proj₂)
first-∃B rewrite C≡ =
  [ (λ _ ()) ⨾ (λ _ → label≢ encode≢) ⨾ (λ _ → label≢ encode≢) ]

first-λᶜ : All (λ l → ∀ X → l ≢ X →∗: m) (Any-front $ ∃B .proj₂)
first-λᶜ rewrite C≡ = [ (λ _ ()) ⨾ (λ _ ()) ]

postulate h≡  : ∣ a♯ ∣ᵐ        ≡ η
          ∣a∣ : ∣ encode a ∣ᵐ ≡ η + N

h∈O : CheckOracleInteractions «.Rᶜ _
h∈O = [ (A , encode a , 𝟘 , ∣a∣) ]

𝕣∗ = coh .proj₁
```

Since the actual serialization procedure has been postulated rather than implemented (*c.f.*, Section 6.1.2), we need to further postulate that reifying advertisements with pointwise-equal mappings results in the same bitstring, from which we derive the `encode-txout≡` lemma:

```
module _ ad (txoutG txoutG′ : Txout ad) (txoutC txoutC′ : Txout (ad .C)) where
  postulate reify-txout≡ : • txoutG ≗↦ txoutG′
                           • txoutC ≗↦ txoutC′
                             ────────────────────────
                             reify (ad , txoutG , txoutC)
                           ≡ reify (ad , txoutG′ , txoutC′)

  encode-txout≡ : • txoutG ≗↦ txoutG′
                  • txoutC ≗↦ txoutC′
```

```
                        ─────────────────────────────────
                        encodeAd ad (txoutG  , txoutC)
                      ≡ encodeAd ad (txoutG′ , txoutC′)
 encode-txout≡ = cong encode ∘₂ reify-txout≡
```

The corresponding [C-AuthCommit] step for participant B is a bit simpler, since there are no secrets/hashes involved:

```
module Step₆ (let module ⋘ = Step₂ ; module ⋖ = Step₅ ; open ℝ (ℝ∗⇒ℝ ⋖.𝕣∗)) where
 Γ₀ Γ Γ′ Γ″ : Cfg
 Γ₀ = ⟨ A has 1 ฿ ⟩at x ∣ ⟨ B has 0 ฿ ⟩at y
    ∣ ⟨ A : a ♯ just N ⟩ ∣ A auth[ ♯▷ TC ]
 Γ  = ` TC ∣ Γ₀
 Γ′ = ` TC ∣ Γ₀ ∣ ∅ᶜ ∣ B auth[ ♯▷ TC ]
 Γ″ = ` TC ∣ Γ₀ ∣ B auth[ ♯▷ TC ]

 Γ→ : Γ at 0 —[ auth-commit⦅ B , TC , [] ⦆ ]→ₜ Γ′ at 0
 Γ→ = Act $ [C-AuthCommit] refl [] (λ _ → [])

 Rˢ = (Γ″ at 0) ⟨ Γ→ ⟩←—— ⋖.Rˢ

 txoutGC = ad∈⇒Txout {TC}{⋖.Γ′}{⋖.Rˢ}{0} (here refl) auto txout′
 txoutG  = txoutGC .proj₁ ; txoutC = txoutGC .proj₂
 GC      = encodeAd TC txoutGC

 m  = SIG (Kᵖᵘᵇ B) $ encode (GC , (List HashId ∋ []) , k̄)
 λᶜ = B →∗∶ m
 Rᶜ = λᶜ ∷ ⋖.Rᶜ ✓

 coh : Rˢ ~ Rᶜ
 coh = -, step₁ (⋖.coh .proj₂)
  ([L2] mkℍ {h = h} ∃B first-∃B [] first-λᶜ [] [] (λ ()))
  where
   h : H₂-args
   h = mk {TC}{Γ₀}{0}{B} K² [] refl [] (λ _ → [])
          (⋖.Rᶜ ⨾ ⋖.Rˢ ⨾ ⋖.𝕣∗ ⊣ auto ≈ Γ″ ⊣ auto)

   C≡ : GC ≡ ⋘.GC
   C≡ = encode-txout≡ TC txoutG ⋘.txoutG txoutC ⋘.txoutC
         (λ where 𝟘 → refl; 𝟙 → refl) λ ()

   ∃B : ∃ λ B → (B →∗∶ GC) ∈ toList ⋖.Rᶜ
   ∃B rewrite C≡ = A , 𝟛

   first-∃B : All (λ l → ∀ X → l ≢ X →∗∶ GC) (Any-tail $ ∃B .proj₂)
   first-∃B rewrite C≡ =
```

```
      [ (λ _ ())
      ; (λ _ → label≢ encode≢)
      ; (λ _ → label≢ encode≢)
      ]

    first-λᶜ : All (λ l → ∀ X → l ≢ X →∗: m) (Any-front $ ∃B .proj₂)
    first-λᶜ rewrite C≡ = [ (λ _ → label≢ SIG≢) ; (λ _ ()) ; (λ _ ()) ]
  ℝ∗ = coh .proj₁
```

Before participants spends their persistent deposits to stipulate the contract, the initial transaction `Tinit` has to be first advertised (*c.f.*, the hypotheses for case `[L2]` of coherence in Section 6.3.1, corresponding to a symbolic `[C-AuthInit]`). We do that by an irrelevant step that requires no symbolic counterpart, using the property we proved in Section 6.3.3 stating that no symbolic move emits such a computational message:

```
module Step₇ (let module ≪ = Step₆) where
  Rˢ = ≪.Rˢ
  λᶜ = A →∗: encode Tinit
  Rᶜ = λᶜ ∷ ≪.Rᶜ ✓

  coh : Rˢ ~ Rᶜ
  coh = -, step₂ (≪.coh .proj₂) ([3] {A = A} (≁₁-encodeT Tinit))
  ℝ∗ = coh .proj₁
```

Both participants can now authorise the expenditure of their persistent deposits (`[C-AuthInit]`), corresponding computationally to broadcasting the initial transaction `Tinit` signed with their private keys:

```
module Step₈ (let module ≪ = Step₇) where
  Γ₀ Γ Γ′ : Cfg
  Γ₀ = ⟨ A has 1 ₿ ⟩at x ∣ ⟨ B has 0 ₿ ⟩at y
     ∣ ⟨ A : a ♯ just N ⟩
     ∣ A auth[ ♯▷ TC ] ∣ B auth[ ♯▷ TC ]
  Γ  = ` TC ∣ Γ₀
  Γ′ = ` TC ∣ Γ₀ ∣ A auth[ x ▷ˢ TC ]

  Γ→ : Γ at 0 —[ auth-init⦅ A , TC , x ⦆ ]→ₜ Γ′ at 0
  Γ→ = Act $ C-AuthInit {Γ = Γ₀} {v = 1 ₿}

  Rˢ = (Γ′ at 0) ⟨ Γ→ ⟩←—— ≪.Rˢ

  λᶜ = A →∗: SIG (Kᵖʳⁱᵛ A) (∃Tx ∋ -, -, Tinit)
  Rᶜ = λᶜ ∷ ≪.Rᶜ ✓

  coh : Rˢ ~ Rᶜ
  coh = -, step₁ (≪.coh .proj₂) ([L3] mkℍ {h = h} (A , 𝟘) [])
   where h : H₃-args
```

```
          h = mk {TC}{Γ₀}{0} (auto .unmk⊆) 0
                 («.Rᶜ ; «.Rˢ ; «.ℝ∗ ⊣ auto ≈ Γ′ ⊣ auto)
  ℝ∗ = coh .proj₁

module Step₉ (let module « = Step₈) where
  Γ₀ Γ Γ′ : Cfg
  Γ₀ = ⟨ A has 1 ₿ ⟩at x | ⟨ B has 0 ₿ ⟩at y
     | ⟨ A : a ♯ just N ⟩
     | A auth[ ♯▷ TC ] | B auth[ ♯▷ TC ]
     | A auth[ x ▷ˢ TC ]
  Γ  = ` TC | Γ₀
  Γ′ = ` TC | Γ₀ | B auth[ y ▷ˢ TC ]

  Γ→ : Γ at 0 —[ auth-init⦅ B , TC , y ⦆ ]→ₜ Γ′ at 0
  Γ→ = Act $ C-AuthInit {Γ = Γ₀} {v = 0 ₿}

  Rˢ = (Γ′ at 0) ⟨ Γ→ ⟩←—— «.Rˢ

  λᶜ = B →∗: SIG (Kᵖʳⁱᵛ B) (∃Tx ∋ -, -, Tᵢₙᵢₜ)
  Rᶜ = λᶜ ∷ «.Rᶜ ✓

  coh : Rˢ ~ Rᶜ
  coh = -, step₁ («.coh .proj₂)
    ([L3] mkℍ {h = h} (A , 𝟙) [ (λ _ → label≢ SIG≢) ] )
    where h : H₃-args
          h = mk {TC}{Γ₀}{0}{B}{y}{0 ₿} (auto .unmk⊆) 𝟙
                 («.Rᶜ ; «.Rˢ ; «.ℝ∗ ⊣ auto ≈ Γ′ ⊣ auto)
  ℝ∗ = coh .proj₁
```

The contract can finally be stipulated using the relevant `[L4]` step of coherence; this corresponds to a symbolic `[C-Init]`, while on the computational level we submit T_{init} to the blockchain:

```
module Step₁₀ (let module « = Step₉) where
  Γ Γ′ : Cfg
  Γ  = ` TC
     | ⟨ A : a ♯ just N ⟩
     | (((⟨ A has 1 ₿ ⟩at x | A auth[ x ▷ˢ TC ])
     | (⟨ B has 0 ₿ ⟩at y | B auth[ y ▷ˢ TC ]))
     | (A auth[ ♯▷ TC ] | B auth[ ♯▷ TC ])
  Γ′ = ⟨ C , 1 ₿ ⟩at x₁ | ⟨ A : a ♯ just N ⟩

  fresh-x₁ : x₁ ∉ [ x ; y ] ++ ids ⟨ A : a ♯ just N ⟩
  fresh-x₁ = auto

  Γ→ : Γ at 0 —[ init⦅ G , C ⦆ ]→ₜ Γ′ at 0
  Γ→ = Act $ [C-Init] {x = x₁} {Γ = ⟨ A : a ♯ just N ⟩} fresh-x₁
```

```
 Rˢ = (Γ′ at 0) ⟨ Γ→ ⟩←—— «.Rˢ

 λᶜ = submit (-, -, Tᵢₙᵢₜ)
 Rᶜ = λᶜ ∷ «.Rᶜ ✓

 coh : Rˢ ~ Rᶜ
 coh = -, step₁ («.coh .proj₂) ([L4] mkℍ {h = h})
  where h : H₄-args
        h = mk {TC}{⟨ A : a ♯ just N ⟩}{0}{x₁} fresh-x₁
             («.Rᶜ ⨾ «.Rˢ ⨾ «.𝕣∗ ⊣ auto ≈ Γ′ ⊣ auto)
 𝕣∗ = coh .proj₁
```

(In order to aid Agda's unification procedure, we capture the freshness condition in a top-level definition and propagate that as proof for the transition `Γ→` and context `h`.)

The preconditions have now been satisfied and the current configuration holds the active timed-commitment contract that consists of two branches: either `A` reveals their secret in time, or `B` can withdraw `A`'s deposits after the deadline.

Here we want to demonstrate the first branch, therefore `A` has to first reveal secret `a` in the configuration, which computationally translates to broadcasting the (encoding of the) secret:

```
module Step₁₁ (let module «« = Step₆ ; module « = Step₁₀) where
 Γ Γ′ : Cfg
 Γ  = ⟨ A : a ♯ just N ⟩ | ⟨ C , 1 ฿ ⟩at x₁
 Γ′ = A : a ♯ N | ⟨ C , 1 ฿ ⟩at x₁

 Γ→ : Γ at 0 —[ auth-rev⟪ A , a ⟫ ]→ₜ Γ′ at 0
 Γ→ = Act [C-AuthRev]

 Rˢ = (Γ′ at 0) ⟨ Γ→ ⟩←—— «.Rˢ

 m  = encode a
 λᶜ = A →∗∶ m
 Rᶜ = λᶜ ∷ «.Rᶜ ✓

 coh : _ ~ Rᶜ
 coh = -, step₁ («.coh .proj₂)
   ([L7] mkℍ {h}{A}{a} (A , ««.txoutGC , 𝟝) 𝟘 m≥ (A , 𝟘) 𝟙
    [ (λ _ ()) ⨾ (λ _ → label≢ SIG≢encode) ⨾ (λ _ → label≢ SIG≢encode)
    ⨾ (λ _ → label≢ encode≢) ⨾ (λ _ → label≢ SIG≢encode) ])
  where h : H₇-args
        h = mk {TC}{A}{a}{N}{⟨ C , 1 ฿ ⟩at x₁}{0} K² [ a , just N , a♯ ]
             («.Rᶜ ⨾ «.Rˢ ⨾ «.𝕣∗ ⊣ auto ≈ Γ′ ⊣ auto)
        postulate m≥ : ∣ encode a ∣ᵐ ≥ η
 𝕣∗ = coh .proj₁
```

As before, we postulate the conditions on the size of bitstrings related to the security parameter η.

(Readers with a keen eye might have noticed that we replaced the resulting symbolic run Rˢ in coh's type signature with a "don't care" identifier _; this again has to do with improving type-checking performance by lifting some of the heavy weight that Agda's unifier has to carry, letting the index to be automatically inferred instead.)

The first branch can now be selected using the [Timeout] rule of BitML's timed layer and then a symbolic [C-PutRev] rule that makes sure a has been revealed:

```
module Step₁₂ (let module ≪ = Step₁₁) where
  Γ Γ′ : Cfg
  Γ  = ⟨ C , 1 ₿ ⟩at x₁ ∣ (∅ᶜ ∣ (A : a ♯ N ∣ ∅ᶜ))
  Γ′ = ⟨ [ withdraw A ] , 1 ₿ ⟩at x₂ ∣ (A : a ♯ N ∣ ∅ᶜ)

  Γ→ : Γ at 0 —[ put⟪ [] , [ a ] , x₁ ⟫ ]→ₜ Γ′ at 0
  Γ→ = Timeout {c = C} {t = 0} {v = 1} {i = 0F}
     $ C-PutRev {Γ′ = ∅ᶜ} {z = x₂} {ds = []} {ss = [ A , a , N ]}

  Rˢ = (Γ′ at 0) ⟨ Γ→ ⟩←—— ≪.Rˢ

  λᶜ = submit (-, -, T′)
  Rᶜ = λᶜ ∷ ≪.Rᶜ ✓

  coh : _ ~ _
  coh = -, step₁ (≪.coh .proj₂) ([L6] mkℍ {h = h})
   where h : H₆-args
         h = mk {C}{1 ₿}{x₁}{[ withdraw A ]}{x₂}{∅ᶜ}{0}
                {ds = []} {ss = [ A , a , N ]} {i = 0F}
                refl refl auto refl refl
                (≪.Rᶜ ⨾ _ ⨾ ≪.𝕣∗ ⊣ auto ≈ Γ′ ⊣ auto)
  𝕣∗ = coh .proj₁
```

As promised in the compilation example of Section 5.3.2, the computational equivalent of picking the branch is to submit transaction T′ to the blockchain, which we have formally established above.

The withdraw continuation of the first branch can now be executed, after which the contract successfully terminates with A having revealed their secret and received their deposit back, which corresponds to submitting transaction T′ᵃ on the computational level:

```
module Step₁₃ (let module ≪ = Step₁₂) where
  Γ Γ′ : Cfg
  Γ  = ⟨ [ withdraw A ] , 1 ₿ ⟩at x₂ ∣ A : a ♯ N
  Γ′ = ⟨ A has 1 ₿ ⟩at x₃ ∣ A : a ♯ N
```

```
Γ→ : Γ at 0 —[ withdraw⟪ A , 1 ₿ , x₂ ⟫ ]→ₜ Γ′ at 0
Γ→ = Timeout {i = 0} C-Withdraw

Rˢ = (Γ′ at 0) ⟨ Γ→ ⟩←—— «.Rˢ

λᶜ = submit (-, -, T′ᵃ)
Rᶜ = λᶜ ∷ «.Rᶜ ✓

coh : _ ~ _
coh = -, step₁ («.coh .proj₂) ([L9] mkℍ {h = h})
  where h : H₉-args
        h = mk {[ withdraw A ]}{1 ₿}{x₂}{A : a ♯ N}{A}{x₃}{0}{i = 0F}
               refl auto refl []
               («.Rᶜ ⨾ _ ⨾ «.ℾ∗ ⊣ auto ≈ Γ′ ⊣ auto)
```

6.3.4.2 Shorthand version (scope-checked)

The ideal presentation that more closely matches the way we would write the same proof on paper would be to use the syntax introduced at the end of Section 6.3.2, which gives a more intuitive way to present each of the 13 steps we just saw as a derivation chain. Sadly, the burden imposed upon the Agda type checker to unify all involved variables and infer dozens more, proves too much to handle and renders the following code impossible to type-check within reasonable time. Nevertheless, we find it informative to still display it here to provide a more high-level view of the proof, even if it has only been scope-checked; the carefully written down example of Section 6.3.4.1 can serve as assurance that the type checking would eventually terminate successfully for this version as well.

```
_ : _ ~ _
_ =
  ∎  ((⟨ A has 1 ₿ ⟩at x ∣ ⟨ B has 0 ⟩at y) at0
     ⊣ auto , [txout: (λ where 𝟘 → Tᵃ; 𝟙 → Tᵇ) ∣sechash: (λ ()) ∣κ: (λ ()) ]
   ~ [ submit (-, -, T₀)
     ⨾ (A →∗: encode (Kᵖ A , K̂ᵖ A))
     ⨾ (B →∗: encode (Kᵖ B , K̂ᵖ B))
     ] ⊣ ((-, -, T₀) , (λ where 𝟘 → 𝟘; 𝟙 → 𝟙) , refl)
   ⊣ (λ where 𝟘 → 0F , refl , refl; 𝟙 → 1F , refl , refl)

  —→ (` TC ∣ ⟨ A has 1 ₿ ⟩at x ∣ ⟨ B has 0 ⟩at y) at0
     ⊣ ((advertise⟪ TC ⟫ , _) , _)
   ~ (A →∗: TC′)
   ⊣ [L1] mkℍ {h = mk {TC}{⟨ A has 1 ₿ ⟩at x ∣ ⟨ B has 0 ₿ ⟩at y}
```

```
            auto auto (auto .unmk⊆) _}

—→ᵋ (A →O: encode a)
  ⊣ [2] (inj₁ refl)

—→ᵋ (O→ A : a♯)
  ⊣ [2] (inj₂ refl)

—→ (` TC | ⟨ A has 1 ฿ ⟩at x | ⟨ B has 0 ⟩at y
          | ⟨ A : a ♯ just N ⟩ | A auth[ ♯▷ TC ]) at0
   ⊣ ((auth-commit⟪ A , TC , [ a , just N ] ⟫ , _) , _)
  ~ (A →∗: SIG (Kᵖᵘᵇ A) C,h̄,k̄)
  ⊣ [L2] mkℍ {h = h₂}
            (A , 𝟚)
            [ (λ _ ()) ⨾ (λ _ → label≢ encode≢) ⨾ (λ _ → label≢ encode≢) ]
            [ h≡ ] [ (λ _ ()) ⨾ (λ _ ()) ] [ (A , encode a , 𝟘 , |a|) ] auto (λ ())

—→ (` TC | (⟨ A has 1 ฿ ⟩at x | ⟨ B has 0 ⟩at y
          | ⟨ A : a ♯ just N ⟩ | A auth[ ♯▷ TC ])
          | B auth[ ♯▷ TC ]) at0
          ⊣ ((auth-commit⟪ B , TC , [] ⟫ , _) , _)
  ~ (B →∗: SIG (Kᵖᵘᵇ B) C,h̄,k̄)
  ⊣ [L2] mkℍ {h = h₂′}
            (A , 𝟛)
            [ (λ _ ()) ⨾ (λ _ → label≢ encode≢) ⨾ (λ _ → label≢ encode≢) ]
            [] [ (λ _ → label≢ SIG≢) ⨾ (λ _ ()) ⨾ (λ _ ()) ] [] [] (λ ())

—→ᵋ (A →∗: encode Tᵢₙᵢₜ)
  ⊣ [3] {A = A} (≁₁-encodeT Tᵢₙᵢₜ)

—→ (` TC | (⟨ A has 1 ฿ ⟩at x | ⟨ B has 0 ⟩at y | ⟨ A : a ♯ just N ⟩
          | A auth[ ♯▷ TC ] | B auth[ ♯▷ TC ])
          | A auth[ x ▷ˢ TC ]) at0
          ⊣ ((auth-init⟪ B , TC , y ⟫ , _) , _)
  ~ (A →∗: SIG (Kᵖʳⁱᵛ A) (∃Tx ∋ -, -, Tᵢₙᵢₜ))
  ⊣ [L3] mkℍ {h = h₃} (A , 𝟘) []

—→ (` TC | (⟨ A has 1 ฿ ⟩at x | ⟨ B has 0 ⟩at y | ⟨ A : a ♯ just N ⟩
          | A auth[ ♯▷ TC ] | B auth[ ♯▷ TC ]
          | A auth[ x ▷ˢ TC ]) | B auth[ y ▷ˢ TC ]) at0
          ⊣ ((auth-init⟪ B , TC , y ⟫ , _) , _)
  ~ (B →∗: SIG (Kᵖʳⁱᵛ B) (∃Tx ∋ -, -, Tᵢₙᵢₜ))
  ⊣ [L3] mkℍ {h = h₃′} (A , 𝟙) [ (λ _ → label≢ SIG≢) ]

—→ (⟨ C , 1 ⟩at x₁ | ⟨ A : a ♯ just N ⟩) at0
   ⊣ ((init⟪ G , C ⟫ , _) , _)
```

```
  ~ submit (-, -, Tᵢₙᵢₜ)
  ⊣ [L4] mkℍ {h = h₄}

—→ (⟨ C , 1 ⟩at x₁ ∣ A : a ♯ N) at0
    ⊣ ((auth-rev⦅ A , a ⦆ , _) , _)
  ~ (A →∗∶ encode a)
  ⊣ [L7] mkℍ {h = h₇} (A , txoutTC , 𝟝) 𝟘 m≥ (A , 𝟘)
      [ (λ _ ())
      ; (λ _ → label≢ SIG≢encode)
      ; (λ _ → label≢ SIG≢encode)
      ; (λ _ → label≢ encode≢)
      ; (λ _ → label≢ SIG≢encode)
      ]

—→ (⟨ [ withdraw A ] , 1 ⟩at x₂ ∣ A : a ♯ N) at0
    ⊣ ((put⦅ [] , [ a ] , x₁ ⦆ , _) , _)
  ~ submit (-, -, T′)
  ⊣ [L6] mkℍ {h = h₆}

—→ (⟨ A has 1 ₿ ⟩at x₃ ∣ A : a ♯ N) at0
    ⊣ ((withdraw⦅ A , 1 ₿ , x₂ ⦆ , _) , _)
  ~ submit (-, -, T′ᵃ)
  ⊣ [L9] mkℍ {h = h₉}
  where

   ⋮
```

We still need to provide the context `Hᵢ-args` for each case under the `where` clause, but these remain exactly the same as before so there is no need to repeat them here.

Chapter 7

Towards Correct Compilation

So far we have formalised the low-level semantics of Bitcoin in Chapter 3 and its computational model in Section 6.1, similarly for the high-level semantics of the BitML calculus in Chapter 4 and its symbolic model in Section 6.2, and then formulated their correspondence by means of a coherence relation in Chapter 6, which internally made use of the BitML compiler we defined in Chapter 5.

This brings us to the final part of the thesis: proving the compiler correct with respect to the coherence relation, what the authors call a *computational soundness* result in the original paper (drawing terminology from the field of Cryptography, especially in the cases where both a computational and a symbolic model are jointly studied [AR02]).

Before we can state the final theorem formally, we need to touch upon the *game-theoretic* aspects of both the computational and the symbolic model, which we have deliberately avoided in the previous chapters as they did not pertain to any of the definitions and results we saw.

In particular, we will introduce *symbolic strategies* in Section 7.1, the construction will be game-theoretic in flavour, talking about moves between players, introducing a notion of what it means for such a strategy to be valid, with different restrictions applied to honest versus dishonest participants.

In a similar vein, we will define *computational strategies* in Section 7.2; differentiating between the set of strategies of honest participants and the single adversarial strategy.

The final piece of the puzzle before the theorem computational soundness can be stated is a translation from symbolic to computational strategies, what we will henceforth refer to as *back-translation* (drawing terminology from fully-abstract compilation for λ-calculi [DPP16b]) for reasons that will become clearer in Section 7.3.

Section 7.4 will then be the final act of the thesis, stating the theorem of computational soundness theorem, alas in an anticlimactic way: we have not mechanised the entirety of the proof yet.

However, by sieving out the game-theoretic aspects of the theorem, we are able to separate the theorem into two halves:

1. first, a provably coherent translation between runs (henceforth called 'parsing') which we have partially mechanised by handling only a couple of demonstrative cases,

2. second, the game-theoretic aspect of computational soundness that translates symbolic to computational strategies, whose proof entails that it respects coherence factored through the run translation of item (1). Our formalisation gives a partial outline of the definitions involved, but postulates the proof as appearing in 'Proof of Theorem 2' of the original paper.

Disclaimer

The formulation of computational soundness in the original paper is actually a *probabilistic* statement that holds with overwhelming probability, given a constraint on the *algorithmic complexity* of participant strategies.

These aspects are especially hard to formalise in a type-theoretic proof assistant and only serve to ensure that cryptographic primitives do not break, so we have simplified matters by presenting a non-probabilistic version with no notion of computational complexity; this is, after all, a natural extension to our treatment of the cryptography and serialisation aspects so far, which we postulated as much as possible to put more emphasis on the Programming-Languages perspective of BitML.

This chapter thus (partially) mechanises §9 and Appendices A.7 and A.8 of the original paper (as well as the game-theoretic definitions from §5 and §6), with computational soundness appearing in Thm.2:

> Massimo Bartoletti and Roberto Zunino. "BitML: a calculus for Bitcoin smart contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 2018, pp. 83–100

The Agda formalisation presented in this chapter lives together with the previous couple of chapters:

`https://omelkonian.github.io/formal-bitml-to-bitcoin/`

Chapter overview. In §7.1, the game-theoretic aspects of the symbolic model will be accounted for, by means of a set of *symbolic strategies*: one for each honest participant and a separate *adversarial strategy*. We then repeat a very similar construction for the computational model in §7.2, giving rise to *computational strategies*.

A translation from symbolic to computational strategies is then described in §7.3, which we indicatively dub "back-translation", given that it internally involves translating in the other direction: "parsing" from computational runs to their symbolic counterparts.

All the above culminate in the statement of the final theorem of *computational soundness* in §7.4, or *compilation correctness* as we prefer to call it. A partial attempt at its proof then closes our thesis, arguably in an anticlimactic way, especially considering that it discards the game-theoretic half of the proof completely.

7.1 Symbolic strategies *(§5 & A.3 of [BZ18])*

Given a current symbolic run, an honest participant's strategy determines a set of moves to play next, which take the form of semantic labels as used in the inference rules of BitML's operational semantics in Section 4.7:

```
record ParticipantStrategy (A : Participant) : Type where
  field Σ : Run → Labels
```

(The datatype index A is actually unused in the definition; we simply use it as a *phantom* parameter to track which participant has what strategy on the type level.)

Participants will typically have limited information about each run, not having access to sensitive information such as secrets. This is captured in the action of 'stripping', which we define polymorphically for several semantic entities:

```
record Strippable (X : Type) : Type where
  field _∗ : X → X
```

We use a 'postfix' operator to closely follow the BitML paper, we can now denote a stripped run R as R ∗.

Configurations are stripped of their secret lengths, which is equivalent to treating all reveals as dishonest.

```
instance
  ∗ᶜ : Strippable Cfg
  ∗ᶜ ._∗ = λ where
    (⟨ A : a ♯ _ ⟩) → ⟨ A : a ♯ nothing ⟩
    (l ∣ r)         → l ∗ ∣ r ∗
    c               → c
```

```
*ᵗ = Strippable Cfgᵗ ∋ λ where ._* (Γ at t) → (Γ *) at t
```

Semantic labels for the `[C-AuthCommit]` also carry secrets in their payload, so we strip away that information as well:

```
*ˡ = Strippable Label ∋ λ where ._* → λ where
  (auth-commit⟪ p , ad , _ ⟫) → auth-commit⟪ p , ad , [] ⟫
  l                            → l
```

We then recursively apply all the above to a whole run:

```
*ʳ = Strippable Run ∋ λ where ._* → mapRun _* _*
```

Not all strategies are considered valid though; we impose a set of restrictions on the moves emitted by a strategy to arrive at the final definition of a *symbolic participant strategy* as appearing in §A.3-Def.10 of the original paper. (As usual, we will insert comments to make it easy to cross-check the formal statements with the original text.)

```
instance
 Valid-Strategy : Validable (ParticipantStrategy A)
 Valid-Strategy {A = A} .Valid (record {Σ = Σ}) =
    -- participant is honest
    A ∈ Hon
    -- (1) only moves enabled by the semantics
  × (∀ {R α} → α ∈ Σ (R *)
               ─────────────────
               ∃ (R ——[ α ]→_))
    -- (2) only self-authorizations
  × (∀ {R α} → α ∈ Σ (R *)
               ──────────────────────────────────
               All (_≡ A) (authDecoration α))
    -- (3) coherent secret lengths
  × (∀ {R ad Δ Δ′} → • auth-commit⟪ A , ad , Δ ⟫ ∈ Σ (R *)
                     • auth-commit⟪ A , ad , Δ′ ⟫ ∈ Σ (R *)
                     ────────────────────────────────────────
                     Δ ≡ Δ′)
    -- (4) persistence
  × (∀ {R Ṙ : Run} {α : Label} → • α ∈ Σ (R *)
                                 • ∃[ α₁ ] ∃[ Γₜ ]
                                   ( (R ——[ α₁ ]→ Γₜ ≈ Ṙ)
                                   × (Ṙ ——[ α ]→ Γₜ))
                                 ─────────────────────────
                                 α ∈ Σ (Ṙ *))
```

Condition (1) ensures that honest participants always pick a move that agrees with BitML's semantics; (2) makes sure a participant's authorisation cannot be impersonated

by others; condition (3) disallows a participant to present two different lengths for the same secret in different points in time; finally, (4) assumes that all honest participants are *persistent* in their strategy, *i.e.*, they do not regret any move chosen in the past (for a prefix of the current run) that still remains valid. In other words, past moves that are still valid with respect to the current run *persist* in the set of proposed moves that the participant now proposes.

We package all honest strategies together and derive the stripped versions of the moves by stripping sensitive information from the semantic labels:

```
HonestStrategies : Type
HonestStrategies = ∀ {A} → A ∈ Hon → ParticipantStrategy A

HonestMoves : Type
HonestMoves = List (Participant × Labels)


instance ∗ᵐ = Strippable HonestMoves ∋ λ where ._∗ → map (map₂ (map _∗))
```

On the adversarial side, the dishonest participants (embodied in the single participant 'adversary') follow strategies a bit differently: given a current run *and* the set of available moves picked by the honest participants, decide on a single move to advance the game:

```
module AdvM (Adv : Participant) (Adv∉ : Adv ∉ Hon) where

  record AdversaryStrategy : Type where
    field Σₐ : Run → HonestMoves → Label


  instance
    Valid-AdversaryStrategy : Validable AdversaryStrategy
    Valid-AdversaryStrategy .Valid (record {Σₐ = Σₐ}) =
      ∀ {R moves} →
        let α = Σₐ (R ∗) (moves ∗) in
        ( -- (1) pick from honest moves
          ∃[ A ] ( A ∈ Hon
                 × authDecoration α ≡ just A
                 × α ∈ concatMap proj₂ moves)
          -- (2) independent move
        ⊎ ( authDecoration α ≡ nothing
          × (∀ δ → α ≢ delay⟦ δ ⟧)
          × ∃[ R′ ] (R ——[ α ]→ R′))
          -- (3) move from dishonest participant
        ⊎ ∃[ B ] ( authDecoration α ≡ just B
                 × B ∉ Hon
                 × (∀ a → α ≢ auth-rev⟦ B , a ⟧)
                 × ∃[ R′ ] (R ——[ α ]→ R′))
```

```
    -- (4) delay
  ⊎ ∃[ δ ] ( (α ≡ delay⟪ δ ⟫)
          × ∀[ Λ ∈ proj₂ <$> moves ]
              (Null Λ ∧ ⊎ Any (λ where delay⟪ δ′ ⟫ → δ′ ≥ δ; _ → ⊥) Λ))
    -- (5) dishonest participant reveals secret
  ⊎ ∃[ B ] ∃[ a ]
      ( (α ≡ auth-rev⟪ B , a ⟫)
      × (B ∉ Hon)
      × (⟨ B : a ♯ nothing ⟩ ∈ᶜ (R ∗) .end .cfg)
      × (∃[ R∗ ] ∃[ Δ ] ∃[ ad ] ( R∗ ∈ prefixRuns (R ∗)
                                  × Σₐ R∗ [] ≡ auth-commit⟪ B , ad , Δ ⟫
                                  × Any (λ (s , m) → (s ≡ a) × Is-just m) Δ)))
  )
  × -- secrets committed by adversaries must not depend on honest moves
    (∀ {B ad Δ} → • B ∉ Hon
                  • α ≡ auth-commit⟪ B , ad , Δ ⟫
                  ─────────────────────────────
                  α ≡ Σₐ (R ∗) [])
```

The formal presentation above closely matches the original description in the paper: the adversary either lets one of the honest moves proceed (1), or proposes a valid move that requires no authorisation (2) or authorisation from a dishonest participant that does not reveal secrets (3), or simply delaying the game if there are no better honest moves (4). Dishonest participants can only authorise revealing their secret under the condition that they had previously (*i.e.*, in some prefix of the run) made an honest commitment for the same secret.

The set of honest strategies along with the adversarial strategy can be thought of as comprising a *game*, which we can advance by iterating the (finite) set of honest participants to generate all possible next (honest) moves, and after that let the adversary make the final choice that determines how the game advances:

```
Strategies : Type
Strategies = AdversaryStrategy × HonestStrategies


runHonestAll : Run → HonestStrategies → HonestMoves
runHonestAll R S = mapWith∈ Hon (λ {A} A∈ → A , Σ (S A∈) (R ∗))

runAdversary : Strategies → Run → Label
runAdversary (S† , S) R = Σₐ S† (R ∗) (runHonestAll R S)
```

As far as computational soundness is concerned, we will only talk about symbolic runs that *conform* to such a set of strategies, *i.e.*, where each transition is dictated by the final adversarial move:

```
data _~ˢ_ : Run → Strategies → Type where

  base :
    ∀ (init : Initial Γ) →
     ──────────────────────────────
     (Γ at 0) ■⊣ (init , refl) ~ˢ SS

  step : let α = runAdversary SS R in
    • R ~ˢ SS
    → (R→ : R ──[ α ]→ Γₜ) →
     ──────────────────────
     (Γₜ ⟨ R→ ⟩←── R ⊣≡) ~ˢ SS
```

7.2 Computational strategies *(§6 & A.4 of [BZ18])*

Computational strategies follow a very similar definition to their symbolic counterparts from Section 7.1, but now operating on computational runs instead:

```
record ParticipantStrategy (A : Participant) : Type where
  field Σ : CRun → Labels
```

In order to define the subclass of strategies that are valid, we need to augment the consistent update of a blockchain (defined in Section 3.5) with additional constraints to situate it in the context of a computational run where certain components of the transaction have been made public, as dictated by condition (2) in the paper which restricts consistent updates only to those that are preceded by broadcasts of the correspondingg transaction and all witnesses:

```
_▷ʳ_ : CRun → ∃Tx → Type
CR ▷ʳ ∃tx@(_ , _ , tx) = let R = CR ∙toList in
    𝔹 R ▷ tx , δʳ R
  × ∃[ B ] (B →∗: (∃tx ♯) ∈ R)
  × ∀[ i ∈ tx .inputs ] ∃[ tx′ ]
     ( submit tx′ ∈ R
     × tx′ ♯ ≡ txId i)
  × ∀[ w ∈ tx .wit ] ∃[ B ]
     (B →∗: encode (w .proj₂ ∙toList) ∈ R)
```

A computational strategy is only valid when the honest participant *persistently* picks one of the following moves:

- submit a transaction to *consistently* update the current blockchain;
- broadcast a message or send a query to the oracle;

- perform a `delay`.

```
instance
 Valid-Strategy : ∀ {A} → Validable (ParticipantStrategy A)
 Valid-Strategy {A = A} .Valid (record {Σ = Σ}) =
    -- participant is honest
    A ∈ Hon
    -- only valid computational labels
  × (∀ {R α} → (let R∗ = stripᶜ A R) → α ∈ Σ R∗ →
      ( -- (1) message from A
        ∃[ m ]  ((α ≡ A →∗: m) ⊎ (α ≡ A →O: m))
        -- (2) new transaction
      ⊎ ∃[ tx ] ((α ≡ submit tx) × (R∗ ▷ʳ tx))
        -- (3) delay
      ⊎ ∃[ δ ]  (α ≡ delay δ)))
    -- persistence
  × (∀ {R λᶜ} → (let R∗ = stripᶜ A R; R⋆ = R∗ ∙toList
                       Λ  = Σ R∗
                       R′ = λᶜ ∷ R∗ ✓
                       Λ′ = Σ R′) →
      • λᶜ ∈ Λ
      • ConsistentBlockchain (𝔹 R⋆)
        ────────────────────────────────────────────
        (∀ {tx} → • submit tx ∈ Λ →
                  • 𝔹 R⋆ ▷ (tx .proj₂ .proj₂) , δʳ R⋆
                  ──────────────────────────────
                  submit tx ∈ Λ′)
      × (∀ {m} → • (A →∗: m) ∈ Λ
                 • (A →∗: m) ≢ λᶜ
                 ─────────────
                 (A →∗: m) ∈ Λ′)
      × (∀ {m} → • (A →O: m) ∈ Λ
                 • (A →O: m) ≢ λᶜ
                 ─────────────
                 (A →O: m) ∈ Λ′))
```

The adversarial strategy closely matches the previous symbolic definition from Section 7.1; the adversary either picks one of the honest moves or impersonates another participant:

```
module AdvM (Adv : Participant) (Adv∉ : Adv ∉ Hon) where

 record AdversaryStrategy : Type where
  field
   Σₐ : CRun → HonestMoves → Label
```

```
valid : ∀ {R moves} →
  (let R∗ = stripᶜ Adv R
       α  = Σₐ R∗ moves)
  → -- (1) impersonate another participant
    ∃[ m ]  ((∃[ A ] α ≡ A →∗: m) ⊎ (α ≡ Adv →O: m))
    -- (2) consistently update the blockchain
  ⊎ ∃[ tx ] ((α ≡ submit tx) × (R∗ ▷ʳ tx))
    -- (3) delay, if all honest participants agree
  ⊎ ∃[ δ ]  ( (α ≡ delay δ)
            × ∀[ Λ ∈ proj₂ <$> moves ]
                (Null Λ ⊎ Any (λ where (delay δ′) → δ′ ≥ δ; _ → ⊥) Λ))
```

Like in the symbolic case, we consider a game to be composed of an adversarial strategy along with a set of honest strategies; honest moves are picked first, stripped appropriately, and only then does the adversary make the final move that decides the next state of the game.

```
runHonestAll : CRun → HonestStrategies → HonestMoves
runHonestAll R S = mapWith∈ Hon (λ {A} A∈ → A , Σ (S A∈) (stripᶜ A R))

runAdversary : Strategies → CRun → Label
runAdversary (S† , S) R = Σₐ S† (stripᶜ Adv R) (runHonestAll R S)
```

We say that a computational run conforms (~ᶜ) to given strategies, only if it is a prefix of another run that 'pre-conforms' (⋯~ᶜ) to these strategies.

```
_~ᶜ_ : CRun → Strategies → Type
R ~ᶜ SS = ∃[ R′ ] (Prefix≡ (toList R) (toList R′) × (R′ ⋯~ᶜ SS))

data _⋯~ᶜ_ : CRun → Strategies → Type where
  -- (1)
  base : ∀ {R} →
    Initial R
    ──────────
    R ⋯~ᶜ SS
  -- (2)
  step : ∀ {R} →
    let (S† , S) = SS
        moves = runHonestAll R S
        Λ = map proj₂ moves
        α = Σₐ S† (stripᶜ Adv R) moves
    in
    • R ⋯~ᶜ SS
    • Null (oracleMessages [ α ])
    • Null (concatMap oracleMessages Λ)
```

```
    ─────────────────────────────────────
    α ∷ R ✓ ⋯↝ᶜ SS
-- (3)
oracle-adv : ∀ {R} {m hm : Message} →
  let (S† , S) = SS
      moves = runHonestAll R S
      Λ = map proj₂ moves
      α = Σₐ S† (stripᶜ Adv R) moves
  in
  • R ⋯↝ᶜ SS
  • α ≡ Adv →O: m
  • Null (concatMap oracleMessages Λ)
  • (∀ {hm′} → (Adv →O: m , O→ Adv : hm′) ∈ oracleRequests Adv (R ∙toList)
               ──────────────────────────────────────────────────────────
               hm ≡ hm′)
    ──────────────────────────────────────────────
    (Adv →O: m) ∷ (O→ Adv : hm) ∷ R ✓ ✓ ⋯↝ᶜ SS
-- (4)
oracle-hon : ∀ {R} {A} {A∈ : A ∈ Hon} {m hm : Message} →
  let (_ , S) = SS
      Λ = Σ (S A∈) (stripᶜ A R)
  in
  • R ⋯↝ᶜ SS
  • L.head (oracleMessages Λ) ≡ just (Adv →O: m)
  • (∀ {hm′} → (A →O: m , O→ A : hm′ ) ∈ oracleRequests A (R ∙toList)
               ─────────────────────────────────────────────────────
               hm ≡ hm′)
    ──────────────────────────────────────────
    (A →O: m) ∷ (O→ A : hm) ∷ R ✓ ✓ ⋯↝ᶜ SS
```

The four constructors correspond to any of the 4 scenarios in which pre-conformance can be established: either the run is initial (`base`); or an honest move decided by the strategies is taken (`step`); or an adversary interacts with the oracle while respecting past oracle responses (`oracle-adv`); or we have an honest oracle interaction which is also consistent (`oracle-hon`).

7.3 Translating strategies *(§A.7, Def.22 of [BZ18])*

The last remaining piece of infrastructure that we need before computational soundness can be phrased, is the translation of symbolic strategies to their computational counterpart.

Given a (valid) *symbolic* strategy for an honest participant A, the translation proceeds to construct a *computational* strategy, *i.e.*, a function determining a set of computational labels Λᶜ by observing the current computational run Rᶜ.

```
ℵ : 𝕍 (S.ParticipantStrategy A) → C.ParticipantStrategy A
ℵ (Σˢ , 𝕧ˢ@(_ , rule-abiding , _)) .C.Σ Rᶜ =
  let
    Rᶜ∗ : CRun
    Rᶜ∗ = Rᶜ ∗
    -- (1) parse Rᶜ∗ to obtain a corresponding symbolic run Rˢ∗
    Rˢ , Rˢ~Rᶜ = parseRun~ Rᶜ∗
    Rˢ∗ = Rˢ ∗
    -- × (2) we do not consider random sequences
    -- (3) evaluate Λˢ = Σˢ(Rˢ∗)
    Λˢ : S.Labels
    Λˢ = Σˢ .S.Σ Rˢ∗

    Λˢ′ : List (∃ λ α → ∃ (Rˢ ——[ α ]→_))
    Λˢ′ = mapWith∈ Λˢ (-,_ ∘ rule-abiding {R = Rˢ})
    -- (4) convert symbolic actions Λˢ into computational actions Λᶜ
    Λᶜ : C.Labels
    Λᶜ = unparseMoves Rˢ~Rᶜ Λˢ′
  in
    Λᶜ
```

The procedure of computing these computational labels can be conceptually divided into 3 steps:

1. *parsing* Rᶜ into a symbolic run Rˢ;
2. after stripping the run from step (1), run the symbolic strategy to get a list of symbolic labels Λˢ (by virtue of the strategy being *valid*, we know that these moves are all admitted by the operational semantics);
3. translate the labels from step (2) to their computational counterpart via a process we dub *unparsing*.

It might seem strange at first that the essence of the translation between symbolic and computational strategies in fact lies in the opposite direction: *parsing* a computational run into a symbolic run in step (1), which one could also refer to as *back-translation* from a more semantic point of view. At the same time, there is still a minor translation in the original direction, namely that of labels in step (3). This stems from the fact that strategies are defined as functions taking runs as arguments and returning labels; we

say that the runs occur in a *contravariant* position but labels in a *covariant* one, which clarifies how the direction of the corresponding translations is determined.

What remains is the definition of these two translations, for runs (Section 7.3.1) and labels (Section 7.3.2) respectively, before we finally get to state the theorem of computational soundness in Section 7.4.

7.3.1 Parsing computational runs

Our goal is to parse computational runs into symbolic ones:

```
parseRun : CRun → S.Run
```

Thinking ahead, we will instead define a dependently-typed version of parsing that also builds up a proof of coherence between the input computational run and the resulting symbolic run:

```
parseRun~ : (Rᶜ : CRun) → ∃ (_~ Rᶜ)
```

In other words, we will define a *coherent-by-construction* back-translation from computational to symbolic runs; yet another instance of our preference for *intrinsic* verification, as opposed to an *extrinsic* approach where we would first define `parseRun` as a simply-typed function and only prove that it preserves coherence *after the fact*.

Dismissing the proof part lets us regain the simply-typed parser:

```
parseRun = proj₁ ∘ parseRun~
```

The base case is trivial; we start from the empty symbolic run:

```
parseRun~ (Rᶜ ■⊣ init ✓) = ∅ˢ , ℝ∗-∅ˢ , base auto init (λ ())
```

For the inductive step, we will analyse all possible cases for the most recent computational label, given the inductive hypothesis that the rest of the computational run (`Rᶜ`) already has a corresponding symbolic run (`Rˢ`) that is coherent with it (`Rˢ Rᶜ`).

```
parseRun~ (λᶜ :: Rᶜ ✓)
  with Rˢ , ℝ∗ , Rˢ~Rᶜ ← parseRun~ Rᶜ
  with λᶜ
```

Examining the form of the computational label, we are tasked with handling the following four categories:

1. **Oracle interactions** can be immediately added without touching the symbolic run:

   ```
   parseRun~ _ | A →O: m =
     Rˢ , ℝ∗ , step₂ Rˢ~Rᶜ ([2] $′ inj₁ refl)
   parseRun~ _ | O→ A : m =
     Rˢ , ℝ∗ , step₂ Rˢ~Rᶜ ([2] $′ inj₂ refl)
   ```

2. **Delays** must not be vacuous ($=0$), otherwise case `[L18]` of coherence would not be derivable:

```
parseRun~ _ | delay 0 =
  Rˢ , ℾ* , step₂ Rˢ~Rᶜ [2]′
parseRun~ (_ ∷ Rᶜ ✓) | delay (suc _) =
  -, -, step₁ Rˢ~Rᶜ ([L18] mkℍ {h})
  where h : H₁₈-args
        h = mk {Γ = Rˢ ∙cfg} auto (ℾ* ;≈ Rˢ ∙cfg)
```

Erratum

Omission in [BZ18]. The cases of 'Inductive case 2' in the original paper do not cover delays of $\delta = 0$, since case (1) only matches transactions, case (2) takes care of oracle interactions, and (3) message broadcasts.

Therefore, either computational delays are assumed to never be vacuous but it is never explicitly mentioned in the text, or coherence can be extended to cover this edge case as well. We opted for the latter (by adding another constructor `[2]′` on the `_~₂_` relation) mostly because it practically led to less refactoring of the formalisation (labels are one of the most central definitions), but it is also conceptually more inclusive (more computational runs are considered).

3. Message **broadcasts**, in contrast to the previous simple cases, may correspond to many different symbolic steps; we have to decipher the message to *decide* which of the relevant steps of coherence applies and if none applies, use the irrelevant case (3) of coherence to mitigate the need for constructing a symbolic step:

```
parseRun~ (_ ∷ Rᶜ ✓) | A₀ →∗: m₀
  with dec-H₁ Rˢ ℾ* Rᶜ A₀ m₀
... | yes (_ , _ , ✓1) = -, -, step₁ Rˢ~Rᶜ ([L1] ✓1)
... | no ¬1
  with dec-H₂ Rˢ ℾ* Rᶜ A₀ m₀
... | yes (_ , _ , ✓2) = -, -, step₁ Rˢ~Rᶜ ([L2] ✓2)
... | no ¬2
  with dec-H₃ Rˢ ℾ* Rᶜ A₀ m₀
... | yes (_ , _ , ✓3) = -, -, step₁ Rˢ~Rᶜ ([L3] ✓3)
... | no ¬3
  with dec-H₅ Rˢ ℾ* Rᶜ A₀ m₀
... | yes (_ , _ , ✓5) = -, -, step₁ Rˢ~Rᶜ ([L5] ✓5)
... | no ¬5
  with dec-H₇ Rˢ ℾ* Rᶜ A₀ m₀
... | yes (_ , _ , ✓7) = -, -, step₁ Rˢ~Rᶜ ([L7] ✓7)
```

```
... | no ¬7
 with dec-H₁₀ Rˢ ℾ* Rᶜ A₀ m₀
... | yes (_ , _ , ✓10) = -, -, step₁ Rˢ~Rᶜ ([L10] ✓10)
... | no ¬10
 with dec-H₁₂ Rˢ ℾ* Rᶜ A₀ m₀
... | yes (_ , _ , ✓12) = -, -, step₁ Rˢ~Rᶜ ([L12] ✓12)
... | no ¬12
   with dec-H₁₄ Rˢ ℾ* Rᶜ A₀ m₀
... | yes (_ , _ , ✓14) = -, -, step₁ Rˢ~Rᶜ ([L14] ✓14)
... | no ¬14
 with dec-H₁₆ Rˢ ℾ* Rᶜ A₀ m₀
... | yes (_ , _ , ✓16) = -, -, step₁ Rˢ~Rᶜ
  ([R16⊣ (λ where ([1] h)  → ¬1  (-, -, h)
                  ([2] h)  → ¬2  (-, -, h)
                  ([3] h)  → ¬3  (-, -, h)
                  ([5] h)  → ¬5  (-, -, h)
                  ([7] h)  → ¬7  (-, -, h)
                  ([10] h) → ¬10 (-, -, h)
                  ([12] h) → ¬12 (-, -, h)
                  ([14] h) → ¬14 (-, -, h)) ] ✓16)
... | no ¬16 = -, -, step₂ Rˢ~Rᶜ
  ([3] λ where _ _ ([L1] h)      → ¬1  (-, -, h)
               _ _ ([L2] h)      → ¬2  (-, -, h)
               _ _ ([L3] h)      → ¬3  (-, -, h)
               _ _ ([L5] h)      → ¬5  (-, -, h)
               _ _ ([L7] h)      → ¬7  (-, -, h)
               _ _ ([L10] h)     → ¬10 (-, -, h)
               _ _ ([L12] h)     → ¬12 (-, -, h)
               _ _ ([L14] h)     → ¬14 (-, -, h)
               _ _ ([R16⊣ _ ] h) → ¬16 (-, -, h))
```

Notice that case `[L16]` only applies when cases `[L1]`-`[L14]` do not, so we need to provide a proof that this is the case using the previous chain of decisions. Similarly for the last case where not even a `[C-AuthDestroy]` action is possible, letting us pick the irrelevant case `[3]` to advance the computational run without any symbolic change.

We have factored out all decision procedures outside the function; sadly we have not mechanised most of them due to time constraints, but we can demonstrate a partial mechanisation of the first case `dec-H₁` if we at least `postulate` boring helper lemmas that are just tedious to prove but have no conceptual importance.

The goal of `dec-H₁` is to decide whether there is a coherent symbolic step of advertising a contract using `[C-Advertise]`, hence the following type signature:

```
dec-H₁ : Dec $ ∃ λ Γₜ″ → ∃ λ λˢ → Γₜ″ ⨾ ℾ∗ ⨾ λˢ ~ℍ[1]~ λᶜ ⨾ Rᶜ
```

This strongly resembles the decidability of the inductive hypotheses of coherence we established at the end of Section 6.3.1, but this is slightly stronger: we now have to come up with all the free variables of the hypotheses, *i.e.*, materialise symbolic entities such as advertisements, names, and mappings, out of just a bitstring m_0.

As an intermediate step, we can decide whether we can advertise a specific advertisement by universally quantifying the free variable ad outside the decision procedure (a form of *skolemization*) and refining the relation to reflect that (via the prefix ⟪ ad ⟫). In other words, we solve the easier problem of deciding the ℍ[1] relation for a *fixed* advertisement:

```
∀dec-H₁ : ∀ ad → Dec $ ∃ λ Γₜ″ → ∃ λ λˢ → ⟪ ad ⟫ Γₜ″ ⨾ ℾ∗ ⨾ λˢ ~ℍ[1]~ λᶜ ⨾ Rᶜ
∀dec-H₁ ad@(⟨ G ⟩ C)
 with ¿ Valid ad ¿ | ¿ Any (_∈ Hon) (G ∙partG) ¿ | ¿ ad ⊆⟪ deposits ⟫ Rˢ ∙cfg ¿
... | no ¬vad | _ | _
  = no λ where (_ , _ , mkℍ {h = mk vad _ _ _} , refl) → ¬vad vad
... | _ | no ¬hon | _
  = no λ where (_ , _ , mkℍ {h = mk _ hon _ _} , refl) → ¬hon hon
... | _ | _ | no ¬d⊆
  = no λ where
    (_ , _ , mkℍ {h = mk {Γ₀ = Γ₀} _ _ d⊆ (_ ⨾ _ ⨾ _ ⊣ (_ , R≈) ≈ _ ⊣ _)} , refl) →
     ¬d⊆ (L.Perm.∈-resp-↭ (≈⇒deposits↭ {Γ₀}{Rˢ ∙cfg} $ ↭-sym R≈) ∘ d⊆)
... | yes vad | yes hon | yes d⊆
  = ≫
  where
  t  = Rˢ .end .time
  Γ₀ = Rˢ ∙cfg
  Γ′ = ` ad ∣ Γ₀

  h : H₁-args
  h = mk {Γ₀ = Γ₀} vad hon d⊆ (ℾ∗ ⨾≈ Γ′)

  open H₁-args h using (Γ; 𝕒; R≈; ℾ; Γₜ″)
  open H₁       h using (λˢ)
  txoutΓ = Txout Γ ∋ Txout≈ {Rˢ ∙cfg}{Γ} (R≈ .proj₂) (ℾ ∙txoutEnd_)
  txoutG = Txout G ∋ weaken-↦ txoutΓ (deposits⊆⇒namesʳ⊆ {ad}{Γ} d⊆)
  txoutC = Txout C ∋ weaken-↦ txoutG (mapMaybe-⊆ isInj₂ $ vad ∙names-⊆)
  m      = encodeAd ad (txoutG , txoutC)

  𝕙 : ⟪ ad ⟫ Γₜ″ ⨾ ℾ∗ ⨾ (𝕒 , λˢ) ~ℍ[1]~ A₀ →∗∶ m ⨾ Rᶜ
  𝕙 = mkℍ {h} , refl

  ≫ : Dec $ ∃ λ Γₜ″ → ∃ λ λˢ → ⟪ ad ⟫ Γₜ″ ⨾ ℾ∗ ⨾ λˢ ~ℍ[1]~ λᶜ ⨾ Rᶜ
  ≫ with m ≟ m₀
```

```
    ... | no  m≢ = no  $ T≢⇒¬H₁ ℍ m≢
    ... | yes m≡ = yes $ -, -, QED
      where
      QED : ⟪ ad ⟫ Γₜ″ ; ℝ∗ ; (𝕒 , λˢ) ~ℍ[1]~ λᶜ ; Rᶜ
      QED = ℍ :~ m≡ ⟪ (λ ◆ → ⟪ ad ⟫ Γₜ″ ; ℝ∗ ; (𝕒 , λˢ) ~ℍ[1]~ A₀ →∗: ◆ ; Rᶜ) ⟫
```

The decision procedure above is unsurprising: we first check that the advertisement satisfied the usual (decidable) conditions drawn from the hypotheses of `[C-Advertise]`; if everything checks out, we can construct a proof that relates the runs appealing to the fact that the message being decoded (`m₀`) must necessarily be the one we construct in the case of coherence.

We can then draw candidate advertisements from the current run (these are always finite), and recover the more general decidability result we initially wanted by arguing that we cannot manufacture another advertisement out of thin air that would still apply to this case of coherence:

```
dec-H₁ with any? ∀dec-H₁ (allAdsᴵ Rˢ)
... | yes p∈ =
  let ad , _ , _ , p , _ = L.Any.satisfied p∈
  in  yes (-, -, p)
... | no p∉ = no $ λ (_ , _ , p) →
  let p∈ = L.Any.map (λ where refl → -, -, p , refl) (getAd∈allAds₁ p)
  in p∉ p∈
```

4. Similarly for **transactions**, we decide whether any of the cases (4)-(15) apply and, if not, a `[DEP-Destroy]` is tried out and, if that also do not apply, submit an irrelevant transaction that uses resources outside `txout`:

```
parseRun~ (_ ∷ Rᶜ ✓) | submit T₀
  with dec-H₄ Rˢ ℝ∗ Rᶜ T₀
... | yes (_ , _ , ✓4) = -, -, step₁ Rˢ~Rᶜ ([L4] ✓4)
... | no ¬4
  with dec-H₆ Rˢ ℝ∗ Rᶜ T₀
... | yes (_ , _ , ✓6) = -, -, step₁ Rˢ~Rᶜ ([L6] ✓6)
... | no ¬6
  with dec-H₈ Rˢ ℝ∗ Rᶜ T₀
... | yes (_ , _ , ✓8) = -, -, step₁ Rˢ~Rᶜ ([L8] ✓8)
... | no ¬8
  with dec-H₉ Rˢ ℝ∗ Rᶜ T₀
... | yes (_ , _ , ✓9) = -, -, step₁ Rˢ~Rᶜ ([L9] ✓9)
... | no ¬9
  with dec-H₁₁ Rˢ ℝ∗ Rᶜ T₀
... | yes (_ , _ , ✓11) = -, -, step₁ Rˢ~Rᶜ ([L11] ✓11)
```

```
... | no ¬11
 with dec-H₁₃ Rˢ 𝕣∗ Rᶜ T₀
... | yes (_ , _ , ✓13) = -, -, step₁ Rˢ~Rᶜ ([L13] ✓13)
... | no ¬13
 with dec-H₁₅ Rˢ 𝕣∗ Rᶜ T₀
... | yes (_ , _ , ✓15) = -, -, step₁ Rˢ~Rᶜ ([L15] ✓15)
... | no ¬15
 with dec-H₁₇ Rˢ 𝕣∗ Rᶜ T₀
... | yes (_ , _ , ✓17) = -, -, step₁ Rˢ~Rᶜ
 ([R17⊣ (λ where ([4] h)  → ¬4  (-, -, h)
                 ([6] h)  → ¬6  (-, -, h)
                 ([8] h)  → ¬8  (-, -, h)
                 ([9] h)  → ¬9  (-, -, h)
                 ([11] h) → ¬11 (-, -, h)
                 ([13] h) → ¬13 (-, -, h)
                 ([15] h) → ¬15 (-, -, h)) ] ✓17)
... | no ¬17
 = -, 𝕣∗ , step₂ Rˢ~Rᶜ ([1] ins♯)
 where open ℝ (ℝ∗⇒ℝ 𝕣∗)
       ins♯ : ∃inputs T₀ ♯ (hashTxⁱ <$> codom txout′)
       ins♯ = ¬H17⇒♯ Rˢ 𝕣∗ Rᶜ T₀ ¬17
```

Again, we have only mechanised the first case, but only postulated the rest which are left as an exercise (not to the reader though, that would be inhumane!). The argument for the last case appeals to an invariant satisfied by the coherence relation, namely that transactions submitted in the computational run only ever refer to names from the `txout′` mapping.

The goal of `dec-H₄` is to decide whether there is a coherent symbolic step of stipulating an advertisement using `[C-Init]`, hence the following type signature:

```
dec-H₄ : Dec $ ∃ λ Γₜ″ → ∃ λ λˢ → Γₜ″ ⨾ 𝕣∗ ⨾ λˢ ~ℍ[4]~ λᶜ ⨾ Rᶜ
```

We repeat the same trick as for `dec-H₁`, *i.e.*, prove decidability of the relation for a fixed advertisement instead, with the only special aspect here being that we generate a *fresh* identifier for the contract's continuation:

```
∀dec-H₄ : ∀ ad → Dec $ ∃ λ Γₜ″ → ∃ λ λˢ → ⟪ ad ⟫ Γₜ″ ⨾ 𝕣∗ ⨾ λˢ ~ℍ[4]~ λᶜ ⨾ Rᶜ
∀dec-H₄ ad@(⟨ G ⟩ C)
 with dec-R≈₄ Rˢ ad
... | no R≉ = no ¬QED
 where
 ¬QED : ¬_ $ ∃ λ Γₜ″ → ∃ λ λˢ → ⟪ ad ⟫ Γₜ″ ⨾ 𝕣∗ ⨾ λˢ ~ℍ[4]~ λᶜ ⨾ Rᶜ
 ¬QED (_ , _ , mkℍ {h} , refl) = let open H₄-args h in R≉ (Γ₀ , t , R≈)
... | yes (Γ₀ , t , R≈) = ≫
```

```
where
ds = persistentDeposits G
vs = map select₂ ds
xs = map select₃ ds
v  = sum vs
∃fresh-z = fresh ⦃ Enum∞-String ⦄ (xs ++ ids Γ₀)
z         = ∃fresh-z .proj₁
fresh-z : z ∉ xs ++ ids Γ₀
fresh-z = ∃fresh-z .proj₂
Γ′ = Cfg ∋ ⟨ C , v ⟩at z ∣ Γ₀

h : H₄-args
h = mk {Γ₀ = Γ₀} fresh-z (_ ⨾ _ ⨾ 𝕣∗ ⊣ R≈ ≈ Γ′ ⊣ ↭-refl)

open H₄-args h hiding (𝕣∗; Rᶜ; ad)
open H₄      h using (λˢ; T)

abstract
 ∃T : ∃Tx
 ∃T = -, -, T

 ∃T≡ : ∃T ≡ (-, -, T)
 ∃T≡ = refl

𝕙 : ⟪ ad ⟫ Γₜ″ ⨾ 𝕣∗ ⨾ (𝕒 , λˢ) ~ℍ[4]~ submit (-, -, T) ⨾ Rᶜ
𝕙 = mkℍ {h} , refl

» : Dec $ ∃ λ Γₜ″ → ∃ λ λˢ → ⟪ ad ⟫ Γₜ″ ⨾ 𝕣∗ ⨾ λˢ ~ℍ[4]~ λᶜ ⨾ Rᶜ
» with T₀ ≟ ∃T
... | no T≢ = no ¬QED
 where
 ¬QED : ¬_ $ ∃ λ Γₜ″ → ∃ λ λˢ → ⟪ ad ⟫ Γₜ″ ⨾ 𝕣∗ ⨾ λˢ ~ℍ[4]~ λᶜ ⨾ Rᶜ
 ¬QED = T≢⇒¬H₄′ 𝕙 ∃T≡ (≢-sym T≢)
... | yes T≡ = yes $ -, -, QED
 where
 QED : ⟪ ad ⟫ Γₜ″ ⨾ 𝕣∗ ⨾ (𝕒 , λˢ) ~ℍ[4]~ λᶜ ⨾ Rᶜ
 QED = ⟪ (λ ◆ → ⟪ ad ⟫ Γₜ″ ⨾ 𝕣∗ ⨾ (𝕒 , λˢ) ~ℍ[4]~ submit ◆ ⨾ Rᶜ) ⟫
        trans T≡ ∃T≡ ~: 𝕙
```

Notice how we need to `abstract` the compiled transaction `T`, otherwise the equality test with T_0 would get stuck at type-checking time!

Again, we lift the previous decidability for fixed advertisements to the whole (finite) set of advertisements that have appeared in the run so far:

```
dec-H₄ with any? ∀dec-H₄ (allAdsʳ Rˢ)
... | yes p∈ =
```

```
    let ad , _ , _ , p , _ = L.Any.satisfied p∈
    in yes (-, -, p)
  ... | no p∉ = no ¬QED
    where
    ¬QED : ¬_ $ ∃ λ Γₜ″ → ∃ λ λˢ → Γₜ″ ; ℝ* ; λˢ ~ℍ[4]~ λᶜ ; Rᶜ
    ¬QED (_ , _ , p) = p∉ p∈
      where
      ad = getAd p
      ad∈ : ad ∈ allAdsᴵ Rˢ
      ad∈ = getAd∈allAds p

      p∈ : Any (λ ad → ∃ λ Γₜ″ → ∃ λ λˢ → ⟪ ad ⟫ Γₜ″ ; ℝ* ; λˢ ~ℍ[4]~ λᶜ ; Rᶜ)
               (allAdsᴵ Rˢ)
      p∈ = L.Any.map (λ where refl → -, -, p , refl) ad∈
```

The fact that this strongly resembles the only other case we mechanised (that of parsing a [C-Advertise] move from a message broadcast) suggests that a systematic treatment of all the cases in a principled way is possible, which consequently gives us enough confidence that the rest of the cases can sensibly be postulated.

7.3.2 Interpreting symbolic moves

On the inverse direction of the translation, from symbolic to computational, step (4) of translating strategies entailed interpreting labels, *i.e.*, produce the computational counterpart of a symbolic step in a given run (already coherent with a computational run):

```
unparseMove :
  • Rˢ ~ Rᶜ
  • Rˢ ——[ α ]→ Γₜ
  ─────────────────────────────
  ∃ λ λᶜ → ∃ λ (𝕒 : 𝔸 Rˢ Γₜ) →
    (Γₜ ∷ Rˢ ⊣ 𝕒) ~ (λᶜ ∷ Rᶜ ✓)
```

We know *a-priori* that the coherent step will be a relevant one, *i.e.*, cases (1)-(18) of ‘Inductive case’ in the original paper, since we have to account for the symbolic step assumed as given in the hypothesis.

We proceed by case analysis on the given symbolic step, handling each of the 18 possible rules. For each case, given the semantic information from the hypotheses of the associated BitML rule, as well as the computational run that is coherent with the previous symbolic run, we are tasked with reconstructing the proofs needed by the corresponding step of coherence.

Many of the hypotheses are propagated verbatim as requirements for the coherent step, such as the cases of [C-Advertise], [C-Init], and executing put /split /withdraw commands:

```
unparseMove
  {Rˢ  = record {end = Γ at _}}
  {α   = advertise⟪ ⟨G⟩C ⟫}
  {Γₜ  = Γ′@(` .⟨G⟩C ∣ Γ) at _}
  (ℝ∗ , Rˢ~Rᶜ)
  ([Action] ([C-Advertise] vad hon d⊆) refl)
  = -, -, -, step₁ Rˢ~Rᶜ
      ([L1] mkℍ {mk {⟨G⟩C}{Γ} vad hon d⊆ (ℝ∗ ;≈ Γ′)}
                  {A = A₀})
unparseMove
  {Rˢ  = record {end = (.(` (⟨ G ⟩ C)) ∣ Γ₀ ∣ _ ∣ _) at _}}
  {α   = init⟪ G , C ⟫}
  {Γₜ  = Γ′ at _}
  (ℝ∗ , Rˢ~Rᶜ)
  ([Action] ([C-Init] fresh-z) refl)
  = -, -, -, step₁ Rˢ~Rᶜ
      ([L4] mkℍ {mk {⟨ G ⟩ C}{Γ₀} fresh-z (ℝ∗ ;≈ Γ′)})
unparseMove
  {α   = put⟪ _ , _ , _ ⟫}
  {Γₜ  = Γ′ at t}
  (ℝ∗ , Rˢ~Rᶜ)
  stepₜ@([Timeout] As≡∅ ∀≤t _ refl)
  with ds , ss , _ , _ , _ , Γ₀ , _ , d≡
    ,  refl , refl , refl , refl , refl , refl
    ,  fresh-z , p≡ ← match-putₜ stepₜ tt
  = -, -, -, step₁ Rˢ~Rᶜ
       ([L6] mkℍ {mk {Γ₀ = Γ₀} {ds = ds} {ss = ss}
                     (∀≤⇒≡max ∀≤t) d≡ fresh-z p≡ As≡∅
                     (ℝ∗ ;≈ Γ′)})
unparseMove
  {Rˢ  = record {end = Γ@(⟨ c , v ⟩at .y ∣ Γ₀) at _}}
  {α   = split⟪ y ⟫}
  {Γₜ  = Γ′ at _}
  (ℝ∗ , Rˢ~Rᶜ)
  stepₜ@([Timeout] As≡∅ ∀≤t _ refl)
  with vcis , _ , _ , d≡
    ,  refl , refl , refl , refl
    ,  fresh-xs ← match-splitₜ stepₜ tt
  = -, -, -, step₁ Rˢ~Rᶜ
      ([L8] mkℍ {mk {Γ₀ = Γ₀} {vcis = vcis}
```

```
                            (∀≤⇒≡max ∀≤t) d≡ fresh-xs As≡∅
                            (⊩∗ ;≈ Γ′)})
unparseMove
  {α = withdraw⟪ _ , _ , _ ⟫}
  {Γₜ = Γ′ at _}
  (⊩∗ , Rˢ~Rᶜ)
  stepₜ@([Timeout] As≡∅ ∀≤t _ refl)
  with Γ₀ , x , d≡ , refl , refl , refl , refl , fresh-x ← match-withdrawₜ stepₜ tt
  = -, -, -, step₁ Rˢ~Rᶜ
     ([L9] mk⊪ {mk {Γ₀ = Γ₀} d≡ fresh-x As≡∅ ∀≤t (⊩∗ ;≈ Γ′)})
```

Notice the invocation to “matching” lemmas that extract useful information out of a transition with a specific label, which are already provided by the BitML formalisation of Chapter 4; *c.f.*, the online formalisation for more details:

https://omelkonian.github.io/formal-bitml/BitML.Semantics.RuleMatching.html

Other cases are much harder to establish, since they require proofs on the computational side as well, which cannot in any way be inferred solely from the symbolic hypotheses. To prove these, we have to utilise the coherent computational run and, in particular, prove invariants as to to which messages must necessarily occur in the previous computational run, and we also expect some tracing properties as the ones we saw for BitML’s semantics in Section 6.2.3 to be required at some point.

All these have not been mechanised yet, so we leave them as proof holes in the code to follow. Furthermore, we have not yet mechanised the *stipulation protocol* of Def.21, particularly items (4) and (5) for communicating the keypairs and querying the oracle for hashes, so we postulate it much like we did for all cryptographic operations thus far; this precludes us from defining the complete cases for `[C-AuthCommit]` and `[C-AuthInit]`, since we need to emit silent (*i.e.*, via the irrelevant case `[3]` of coherence) computational labels (for querying the oracle, etc.), before we eventually pick the corresponding coherent cases `[L2]` and `[L3]`.

For the rest of the cases, we provide a partial mechanisation with the aforementioned proof holes whenever computational information is needed, *e.g.*, for the case of authorising a deposit merge:

```
unparseMove
  {Rˢ = record {end = Γ@(⟨ .A has v ⟩at .x ∣ ⟨ .A has v′ ⟩at .x′ ∣ Γ₀) at _}}
  {α = auth-join⟪ A , x ↔ x′ ⟫}
  {Γₜ = Γ′ at _}
  (⊩∗ , Rˢ~Rᶜ)
  ([Action] [DEP-AuthJoin] refl)
  = -, -, -, step₁ Rˢ~Rᶜ
```

```
([L10] mk⊪ {mk {Γ₀ = Γ₀} (⊪* ;≈ Γ′)}
          {B = A₀}
          {!!} {!!})
```

Last, we iterate over the whole set of symbolic steps (arising from the honest symbolic strategies for each participant) and iteratively unparse them into a set of computational labels:

```
unparseMoves : Rˢ ~ Rᶜ → List (∃ λ α → ∃ (Rˢ ——[ α ]→_)) → C.Labels
unparseMoves Rˢ~Rᶜ = map λ where
  (α , Γₜ , R→) → unparseMove Rˢ~Rᶜ R→ .proj₁
```

7.4 Computational soundness *(§9 & A.8 of [BZ18])*

Given the (partial) mechanisations of *parsing* in Section 7.3.1 and *unparsing* in Section 7.3.2, we can finally state the theorem of *computational soundness*.

First, let us set up the context of the theorem:

- a single adversary Adv, who does not belong to the set of honest participants;
- a set of (valid) symbolic strategies Σˢ for all honest participants;
- an arbitrary computational strategy for the adversary Σᶜₐ.

```
module _
  Adv (Adv∉ : Adv ∉ Hon)
  (open S.AdvM Adv Adv∉ renaming (AdversaryStrategy to AdvStrategyˢ))
  (open C.AdvM Adv Adv∉ renaming (AdversaryStrategy to AdvStrategyᶜ))
  (Σˢ : S.HonestStrategies)
  (⊨Σˢ : ∀ {A} (A∈ : A ∈ Hon) → Valid (Σˢ A∈))
  (Σᶜₐ : AdvStrategyᶜ)
  where
```

Furthermore, we translate the given symbolic strategies using the *back-translation* of Section 7.3 to acquire a set of computational strategies for the honest participants:

```
  Σᶜ : C.HonestStrategies
  Σᶜ A∈ = ℵ A∈ (Σˢ A∈ , ⊨Σˢ A∈)
```

Erratum

Omission in [BZ18]. We also have to translate the adversarial strategy $\Sigma^c{}_a$ to talk about symbolic conformance, but this is not mentioned in the original paper. Therefore, the only way for the theorem to make sense — without any structural changes — is to further assume an adversarial strategy translation $\aleph_a$ acting in the other direction; from symbolic to computational:

```
postulate ℵa : AdvStrategyᶜ → AdvStrategyˢ
Σˢa = ℵa Σᶜa
```

The change of direction is rather unfortunate, as we would have to now implement parsing of labels and unparsing of runs, respectively, which is a lot of work.
(An alternative formulation with much less overhead would be to assume a adversarial strategy back-translation `AdvStrategyˢ → AdvStrategyᶜ` and use *that* to get the computational adversarial strategy $\Sigma^c{}_a$, but we do not pursue that here).

Without much further ado, computational soundness can finally be phrased as:

```
Theorem2 : ∀ Rᶜ →
  Rᶜ ~ᶜ Σᶜa , Σᶜ
  ──────────────────────────────────────
  ∃ λ Rˢ → (Rˢ ~ˢ Σˢa , Σˢ) × (Rˢ ~ Rᶜ)
```

Any computational run conforming to the computational strategies has a coherent symbolic run that conforms to the initial symbolic strategies.

Intuitively, such a result means that it is fine to reason about attacks purely on the symbolic level, which is more high-level thus easier to reason about, since any attack possible on the Bitcoin level can be demonstrated in the corresponding (coherent) symbolic run. Phrased in the negative, if we prove that an attack is *not* possible on the symbolic level, it is not possible on Bitcoin as well.

An *alternative* way of presenting the theorem is in two steps, separating out the game-theoretic aspects that we have not proven yet. And while we are at it, let us adopt different terminology that is more focused on the logical perspective rather than the cryptographic. Instead of calling it *computational soundness*, we can also argue that it is a proof about the correctness of the BitML compiler, *i.e., compilation correctness.*

The theorem can be separated into the following two halves:

1. **Completeness:** every computational run has a *coherent* symbolic run;
2. **Game-theoretic Soundness:** the back-translation under item (1) preserves strategy *conformance* through $\aleph$.

```
Theorem2 Rᶜ R~ᶜ =
  let Rˢ , Rˢ~Rᶜ = ⟪1⟫completeness Rᶜ
  in  Rˢ , ⟪2⟫soundness Rᶜ R~ᶜ , Rˢ~Rᶜ
  where
   ⟪1⟫completeness : ∀ Rᶜ → ∃ (_~ Rᶜ)
   ⟪1⟫completeness = parseRun~

   ⟪2⟫soundness : ∀ Rᶜ →
    Rᶜ ~ᶜ Σᶜₐ , Σᶜ
    ─────────────────────────
    parseRun Rᶜ ~ˢ Σˢₐ , Σˢ
   ⟪2⟫soundness = {!!}
```

Completeness is immediately derived from our *coherent-by-construction* parsing from Section 7.3.1; long live intrinsic typing!

Sadly, we also reach the *anticlimax* of the thesis; we have no results regarding **game-theoretic soundness** to ensure that the strategy translation ℵ (defined in Section 7.3 and using both parsing of runs and unparsing of labels) preserves the notion of conformance (defined symbolically in Section 7.1 and computationally in Section 7.2). This is due to both conceptual and technical reasons.

The proof as presented in §A.8 of the original paper is *classical*, employing *proof-by-contradiction*, which is not a valid proof methodology in *constructive* logic where the *law of the excluded middle* (LEM) does not hold. Of course, we could postulate the *double negation elimination* principle (consequence of LEM), but that would render our proof *non-constructive* and we would lose the ability to exploit any computational content from it. It might seem harmless in the case of a conformance proof, but is it really necessary?

On the technical side, the problems of scalability and performance mentioned in Chapter 6 have come to haunt us; we are floating in a sea of *green slime*, myriads of meta-variables and unification problems still unresolved, every step reminding us of the myth of Sisyphus, and interactivity — one of the essential ingredients of dependently-typed theorem proving — has now been lost. Should we rethink everything from the start and try a drastically different approach to encoding things? Or is it that Agda's type-checker could be optimised to handle proofs of larger scale and/or provide utilities for a finer level of control?

Chapter 8

Conclusion

8.1 Reflection

The Good. We have come a long way, reading between the lines of the original BitML paper to unveil a whole new world of detail that I hope to have conveyed in a rigorous enough manner.

More importantly, the results have been presented in a *compositional* fashion: the Bitcoin formalisation can be studied in its own right (*e.g.,* to transfer my parallel results in UTxO meta-theory [Cha+20a; Cha+20c; Cha+20b; MSC23] to Bitcoin), the BitML formalisation can be separated to study other compilation targets or even purely its meta-theory. Even the compilation correctness artefact can provide inspiration for other refinement/bisimulation results; I am certain that the challenges faced throughout the definition of the 'coherence' relation, for instance, apply more broadly than just the BitML use case. In fact, the aforementioned parallel work on UTxO meta-theory was not completely independent to my BitML work; each informed the other.

The resulting model is also *executable*, giving us concrete computational tools to utilise and extend in various ways, possibly enabling exciting future research and tools to emerge. More broadly, I hope to have set the tone for more formal verification research to come to fruition at the confluence of Blockchain and PL.

Another aspect that should not be overlooked is the ample evidence in favour of *intrinsic* over *extrinsic* verification scattered throughout the thesis. There are many uses of dependent types that led to a satisfying state where the information carried by the types leave little to no room for error. What first might have appeared as mere notational convenience quickly evolved into a very efficient framework to formulate any construction we encountered, with the types guiding us along the way. I hope I managed to let you in on this process and how it unfolded, despite the inherent dichotomy between

the interactive experience of a graphical interface and the static content rendered on the pages of this document.

Last but not least, given the sheer size and technical complexity of our formalisation endeavour, this thesis can also be utilised as an extensive Agda tutorial for medium-to-large verification projects, which is practically non-existent, but presents unique challenges that you would not encounter in a smaller scale.

The Bad. Alas, the correctness proof for BitML has not yet been fully mechanised, so one could say that the initial goal of the thesis has not been achieved:

Develop a verified compiler from BitML contracts to Bitcoin transactions.

On the other hand, there are not that many proof holes remaining, but it might be harder to make progress at the scale the project has reached and every step forward gets harder and harder.

The Ugly. It is unfortunate that a great deal of our woes originated as artefacts of the formalisation itself that did not pertain to the subject matter in an essential manner: infrastructure that was not already available in the standard library; reconciling the gap between formal and paper encodings; driving Agda to its limits, given it is a far from mainstream development most often used for small-size research projects and prototypes.

The worst moment, I believe, was when we had to resort to alternative encodings that hindered understanding just to gain type-checking speed; this seems more like an engineering problem, of no scientific interest in itself. (Actually, I find the problem of better tooling (*e.g.,* refactoring) for dependent types incredibly interesting in itself, but this lies outside the scope of this thesis.)

8.2 Future Work

I wish to conclude on a more positive note, listing various exciting new directions that this work can grow into.

Targeting EUTxO. Given my parallel work on the *Extended UTxO* model [Cha+20a; Cha+20b; MSC23] it is natural that I would suggest an alternative compilation target for BitML: the Cardano blockchain, where the validation scripts can be much more expressive and thus allows for a more direct translation overall, in which case one should probably call it *AdaML* (credits to Victor Miraldo for suggesting the name to me).

Furthermore, the added expressivity would mean that less parts of the BitML protocol would happen off-chain, which I believe to be one of the major sources of complexity in the BitML mechanisation; it is no coincidence after all that these were the moments (*e.g.,* the stipulation protocol) when we had to resort to postulates, compromising the rigour of our proofs/definitions.

Marlowe as the source language. It so happens that there already exists a *domain specific language* (DSL) for developing high-level contracts in Cardano: *Marlowe* [ST18]. I believe there is strong resemblance with BitML here, hence it would wise to try to transfer the results and the lessons we learned in this thesis to Marlowe with the ulterior motive of having a verified compiler from Marlowe to Cardano transactions.

In fact, Marlowe already comes with a straightforward EUTxO implementation that is executed purely on-chain, and has been specified in detail on paper, so it is only natural to follow the same methodology and see if this greatly simplifies things by not having to take care of the off-chain shenanigan.

Extracting to Haskell. Another aspect that I have not touched upon so far is how the resulting artefact of our formalisation can be used afterwards; after all, there are not many Agda experts, especially in the Blockchain/PL space.

A fantastic opportunity here is to compile Agda into Haskell using the `agda2hs` [Coc+22] backend to produce readable Haskell code with all the proofs erased. This would allow us to communicate our results with programmers that are not necessarily experts in formal methods, *e.g.,* the developers of the production code of the actual blockchain.

It will not be effortless though, as this will need a drastic rethink of the whole code base; even the Prelude would have to be written from the start with *irrelevance* in mind, a recent (experimental) feature of Agda that allows marking proofs as erased and thus not to appear in the extracted Haskell code.

Compositional reasoning. We have talked a lot about the formal definitions of our models, and how to prove that translating between the two preserves the corresponding semantics, thus allowing one to reason about Bitcoin *symbolically* in the higher level of abstraction afforded by BitML.

But what about the concrete proofs we eventually do afterwards, on the BitML level? Actual use cases will involve multiple contracts interacting, how will the proofs that we do in Agda scale accordingly?

One possible research direction is to adapt parallel work of mine on compositional reasoning of UTxO-based ledgers [MSC23] via a novel separation logic to the more concrete setting of BitML and Bitcoin.

Further into BitML. I have left the most obvious continuation of this work for last, namely discharging the remaining proof holes and extending the results to subsequent work on BitML:

- formalise the property of *liquidity* [BZ19b] for BitML contracts: funds never get "frozen", *i.e.,* they can always be spent thus cannot indefinitely remain locked.
- incorporate recursive BitML contracts as introduced in [Bar+20]; we expect this to complicate matters throughout the formalisation, but connects especially well with the previous paragraph on EUTxO.

8.3 Epigraph

In conclusion, despair not about the lack of proofs for a full-blown verified compiler and that the goal of this thesis has not been accomplished to its full degree. Let us break this teleological stance, itself a symptom of the broader socio-political environment that is driven by profit rather than curiosity, and adopt a more *dialectical* perspective to appreciate the exploratory process [Fey93] and rejoice in the lessons we learned along the way.

And with that, I leave you with the first and last stanzas of a much-celebrated Greek poem centred around the same motif, but now in the context of Homer's *Odyssey*.

Σα βγεις στον πηγαιμό για την Ιθάκη,
να εύχεσαι να'ναι μακρύς ο δρόμος,
γεματος περιπέτειες, γεμάτος γνώσεις.
⋮
Κι αν πτωχική την βρεις, η Ιθάκη δεν σε γέλασε.
Έτσι σοφός που έγινες, με τόση πείρα,
ήδη θα το κατάλαβες οι Ιθάκες τί σημαίνουν.

Κ. Π. Καβάφης
Ιθάκη

As you set out for Ithaka,
hope the voyage is a long one,
full of adventure, full of discovery.
⋮
And if you find her poor, Ithaka won't have fooled you.
Wise as you will have become, so full of experience,
you'll have understood by then what these Ithakas mean.

C. P. Cavafy
Ithaka

Bibliography

[Mea55] George H Mealy. "A method for synthesizing sequential circuits". In: *The Bell System Technical Journal* 34.5 (1955), pp. 1045–1079.

[MP67] John McCarthy and James Painter. "Correctness of a compiler for arithmetic expressions". In: *Mathematical aspects of computer science* 1 (1967).

[Sco70] Dana Scott. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group Oxford, 1970.

[Dij75] Edsger W Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs". In: *Communications of the ACM* 18.8 (1975), pp. 453–457.

[Mil77] Robin Milner. "Fully abstract models of typed λ-calculi". In: *Theoretical Computer Science* 4.1 (1977), pp. 1–22.

[Plo77] Gordon D Plotkin. "LCF considered as a programming language". In: *Theoretical computer science* 5.3 (1977), pp. 223–255.

[Hoa78] Charles Antony Richard Hoare. "Communicating sequential processes". In: *Communications of the ACM* 21.8 (1978), pp. 666–677.

[MM79] George Milne and Robin Milner. "Concurrent processes and their syntax". In: *Journal of the ACM (JACM)* 26.2 (1979), pp. 302–321.

[Mil79a] Robin Milner. "An algebraic theory for synchronization". In: *Theoretical Computer Science 4th GI Conference*. Springer. 1979, pp. 27–35.

[Mil79b] Robin Milner. "Flowgraphs and flow algebras". In: *Journal of the ACM (JACM)* 26.4 (1979), pp. 794–818.

[Hoa80] Charles Antony Richard Hoare. "A model for communicating sequential process". In: (1980).

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980. ISBN: 3-540-10235-3. DOI: 10.1007/3-540-10235-3. URL: https://doi.org/10.1007/3-540-10235-3.

[HBR81] Charles Antony Richard Hoare, Stephen D Brookes, and Andrew William Roscoe. *A theory of communicating sequential processes*. Oxford University Computing Laboratory, Programming Research Group, 1981.

[Plo81] Gordon D Plotkin. “A structural approach to operational semantics”. In: (1981).

[DY83] Danny Dolev and Andrew Yao. “On the security of public key protocols”. In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208.

[BK87] Jan A Bergstra and Jan Willem Klop. “ACP τ a universal axiom system for process specification”. In: *Workshop on Algebraic Methods*. Springer. 1987, pp. 445–463.

[Wad87] Philip Wadler. “Views: A Way for Pattern Matching to Cohabit with Data Abstraction”. In: *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 1987, pp. 307–313. DOI: 10.1145/41625.41653. URL: https://doi.org/10.1145/41625.41653.

[AL88] Martín Abadi and Leslie Lamport. “The existence of refinement mappings”. In: (1988).

[ABW88] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. “Towards a Theory of Declarative Knowledge”. In: *Foundations of Deductive Databases and Logic Programming*. Ed. by Jack Minker. Morgan Kaufmann, 1988, pp. 89–148. DOI: 10.1016/b978-0-934613-40-8.50006-3. URL: https://doi.org/10.1016/b978-0-934613-40-8.50006-3.

[Gel88] Allen Van Gelder. “Negation as Failure Using Tight Derivations for General Logic Programs”. In: *Foundations of Deductive Databases and Logic Programming*. Ed. by Jack Minker. Morgan Kaufmann, 1988, pp. 149–176. DOI: 10.1016/b978-0-934613-40-8.50007-5. URL: https://doi.org/10.1016/b978-0-934613-40-8.50007-5.

[Prz88] Teodor C. Przymusinski. “On the Declarative Semantics of Deductive Databases and Logic Programs”. In: *Foundations of Deductive Databases and Logic Programming*. Ed. by Jack Minker. Morgan Kaufmann, 1988,

pp. 193–216. DOI: 10.1016/b978-0-934613-40-8.50009-9. URL: https://doi.org/10.1016/b978-0-934613-40-8.50009-9.

[Mil89] Robin Milner. "A calculus of mobile process". In: *LFCS Report, Dept. of Computer Science, Univ. of Edinbough* (1989).

[WB89] Philip Wadler and Stephen 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. DOI: 10.1145/75277.75283. URL: https://doi.org/10.1145/75277.75283.

[Fel91] Matthias Felleisen. "On the expressive power of programming languages". In: *Science of computer programming* 17.1-3 (1991), pp. 35–75.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems". In: *Journal of the ACM (JACM)* 38.3 (1991), pp. 690–728.

[Sha91] Ehud Shapiro. "Separating concurrent languages with categories of language embeddings". In: *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. 1991, pp. 198–208.

[BR93] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*. Ed. by Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby. ACM, 1993, pp. 62–73. DOI: 10.1145/168588.168596. URL: https://doi.org/10.1145/168588.168596.

[Fey93] P.K. Feyerabend. *Against Method*. Verso, 1993. ISBN: 9780860914815. URL: https://books.google.co.uk/books?id=HGG3QgAACAAJ.

[Mit93] John C Mitchell. "On abstraction and the expressive power of programming languages". In: *Science of Computer Programming* 21.2 (1993), pp. 141–163.

[Rie93] Jon G Riecke. "Fully abstract translations between functional languages". In: *Mathematical Structures in Computer Science* 3.4 (1993), pp. 387–415.

[OR95] Peter W Ohearn and Jon G Riecke. "Kripke logical relations and PCF". In: *Information and Computation* 120.1 (1995), pp. 107–116.

[San96] Davide Sangiorgi. "A theory of bisimulation for the π-calculus". In: *Acta informatica* 33.1 (1996), pp. 69–97.

[Abr97] Samson Abramsky. “Game semantics for programming languages”. In: *MFCS*. Vol. 97. 1997, pp. 25–29.

[Bar+97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. “The Coq proof assistant reference manual: Version 6.1”. PhD thesis. Inria, 1997.

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. “UPPAAL in a nutshell”. In: *International journal on software tools for technology transfer* 1.1-2 (1997), pp. 134–152.

[Pie97] Benjamin C Pierce. “Foundational Calculi for Programming Languages.” In: *The Computer Science and Engineering Handbook* 1997 (1997), pp. 2190–2207.

[Abe98] Andreas Abel. “foetus–termination checker for simple functional programs”. In: *Programming Lab Report* 474 (1998). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.3494.

[DE98] W-P De Roever and Kai Engelhardt. *Data refinement: model-oriented proof methods and their comparison*. 47. Cambridge University Press, 1998.

[NL98] George C Necula and Peter Lee. “The design and implementation of a certifying compiler”. In: *ACM SIGPLAN Notices* 33.5 (1998), pp. 333–344.

[Aba99] Martín Abadi. “Protection in programming-language translations”. In: *Secure Internet programming*. Springer, 1999, pp. 19–34.

[Mor+99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. “From System F to typed assembly language”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.3 (1999), pp. 527–568.

[LP00] John R Longley and Gordon Plotkin. “Logical full abstraction and PCF”. In: (2000).

[AR02] Martín Abadi and Phillip Rogaway. “Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)”. In: *J. Cryptol.* 15.2 (2002), pp. 103–127. DOI: 10.1007/s00145-001-0014-7. URL: https://doi.org/10.1007/s00145-001-0014-7.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media, 2002.

[Coh03] Bram Cohen. "Incentives build robustness in BitTorrent". In: *Workshop on Economics of Peer-to-Peer systems*. Vol. 6. 2003, pp. 68–72.

[MM04] Conor McBride and James McKinna. "The view from the left". In: *J. Funct. Program.* 14.1 (2004), pp. 69–111. DOI: 10.1017/S0956796803004829. URL: https://doi.org/10.1017/S0956796803004829.

[Bae05] Jos CM Baeten. "A brief history of process algebra". In: *Theoretical Computer Science* 335.2-3 (2005), pp. 131–146.

[Ler06] Xavier Leroy. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Ed. by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, 2006, pp. 42–54. DOI: 10.1145/1111037.1111042. URL: https://doi.org/10.1145/1111037.1111042.

[Cur07] Pierre-Louis Curien. "Definability and full abstraction". In: *Electronic Notes in Theoretical Computer Science* 172 (2007), pp. 301–310.

[Lai07] James Laird. "A fully abstract trace semantics for general references". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2007, pp. 667–679.

[BRS08] Marcello M Bonsangue, Jan Rutten, and Alexandra Silva. "Coalgebraic logic and synthesis of Mealy machines". In: *International Conference on Foundations of Software Science and Computational Structures*. Springer. 2008, pp. 231–245.

[CS08] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*. IEEE Computer Society, 2008, pp. 51–65. DOI: 10.1109/CSF.2008.7. URL: https://doi.org/10.1109/CSF.2008.7.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[MP08] Conor McBride and Ross Paterson. “Applicative programming with effects”. In: *J. Funct. Program.* 18.1 (2008), pp. 1–13. DOI: 10 . 1017 / S0956796807006326. URL: https : / / doi . org / 10 . 1017 / S0956796807006326.

[Nak08] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. https://bitcoin.org/en/bitcoin-paper. Oct. 2008.

[Nor08] Ulf Norell. “Dependently typed programming in Agda”. In: *International School on Advanced Functional Programming*. Springer. 2008, pp. 230–266.

[BH09] Nick Benton and Chung-Kil Hur. “Biorthogonality, step-indexing and compiler correctness”. In: *ACM Sigplan Notices* 44.9 (2009), pp. 97–108.

[LQ09] Shuvendu K Lahiri and Shaz Qadeer. “Complexity and algorithms for monomial and clausal predicate abstraction”. In: *International Conference on Automated Deduction*. Springer. 2009, pp. 214–229.

[Ler09] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115.

[BHP10] Nick Benton, Chung-Kil Hur, and Cambridge Paris. “Realizability and compositional compiler correctness for a polymorphic language”. In: *Technical Report MSR-TR-2010–62, Microsoft Research, Tech. Rep.* (2010).

[Roy+10] Arnab Roy, Anupam Datta, Ante Derek, and John C. Mitchell. “Inductive trace properties for computational security”. In: *J. Comput. Secur.* 18.6 (2010), pp. 1035–1073. DOI: 10.3233/JCS-2009-389. URL: https://doi.org/10.3233/JCS-2009-389.

[SSW10] Tim Sheard, Aaron Stump, and Stephanie Weirich. “Language-based verification will change the world”. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. 2010, pp. 343–348.

[HD11] Chung-Kil Hur and Derek Dreyer. “A Kripke logical relation between ML and assembly”. In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2011, pp. 133–146.

[San11] Davide Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[Swa+11] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. “Secure distributed programming with value-dependent types”. In: *ACM SIGPLAN Notices* 46.9 (2011), pp. 266–278.

[Hur+12] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. "The marriage of bisimulations and Kripke logical relations". In: *ACM SIGPLAN Notices* 47.1 (2012), pp. 59–72.

[VS12] Paul Van Der Walt and Wouter Swierstra. "Engineering proof by reflection in Agda". In: *Symposium on Implementation and Application of Functional Languages*. Springer. 2012, pp. 157–173.

[AJM13] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. "Full abstraction for PCF". In: *arXiv preprint arXiv:1311.6125* (2013).

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. ISBN: 978-0-262-02665-9. URL: http://mitpress.mit.edu/books/certified-programming-dependent-types.

[Fou+13] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. "Fully abstract compilation to JavaScript". In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2013, pp. 371–384.

[And+14a] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. "Secure multiparty computations on bitcoin". In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE. 2014, pp. 443–458.

[And+14b] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. "Modeling bitcoin contracts by timed automata". In: *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer. 2014, pp. 7–22.

[BK14] Iddo Bentov and Ranjit Kumaresan. "How to use bitcoin to design fair protocols". In: *International Cryptology Conference*. Springer. 2014, pp. 421–439.

[But14] Vitalik Buterin. *A next-generation smart contract and decentralized application platform (white paper)*. https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf. 2014.

[Goo14] L.M Goodman. *Tezos—a self-amending crypto-ledger (white paper)*. https://www.tezos.com/whitepaper.pdf. 2014.

[McB14] Conor Thomas McBride. “How to keep your neighbours in order”. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, 2014, pp. 297–309. DOI: 10.1145/2628136.2628163. URL: https://doi.org/10.1145/2628136.2628163.

[Nan+14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. “Communicating State Transition Systems for Fine-Grained Concurrent Resources”. In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 290–310. DOI: 10.1007/978-3-642-54833-8_16. URL: https://doi.org/10.1007/978-3-642-54833-8_16.

[Bon+15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. “Sok: Research perspectives and challenges for bitcoin and cryptocurrencies”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 104–121.

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. “The bitcoin backbone protocol: Analysis and applications”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2015, pp. 281–310.

[Nar+15] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. “Bitcoin and cryptocurrency technologies”. In: *Curso elaborado pela* (2015).

[Nei+15] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. “Pilsner: a compositionally verified compiler for a higher-order imperative language”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 2015, pp. 166–178.

[CB16] David R. Christiansen and Edwin C. Brady. “Elaborator reflection: extending Idris in Idris”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue, Gabriele Keller,

and Eijiro Sumii. ACM, 2016, pp. 284–297. DOI: 10.1145/2951913.2951932. URL: https://doi.org/10.1145/2951913.2951932.

[DPP16a] Dominique Devriese, Marco Patrignani, and Frank Piessens. "Fully-abstract compilation by approximate back-translation". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 164–177.

[DPP16b] Dominique Devriese, Marco Patrignani, and Frank Piessens. "Fully-abstract compilation by approximate back-translation". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 164–177. DOI: 10.1145/2837614.2837618. URL: https://doi.org/10.1145/2837614.2837618.

[GN16] Daniele Gorla and Uwe Nestmann. "Full abstraction for expressiveness: History, myths and facts". In: *Mathematical Structures in Computer Science* 26.4 (2016), pp. 639–654.

[Lau16] Johnson Lau. *Upgrading Bitcoin: Segregated Witness*. https://www.bitcoin.org.hk/media/presentations/2016-03-16/2016-03-16-Segregated_Witness.pdf. 2016.

[Luu+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. "Making smart contracts smarter". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 254–269.

[Par16] Joachim Parrow. "General conditions for full abstraction". In: *Mathematical Structures in Computer Science* 26.4 (2016), pp. 655–657.

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A survey of attacks on ethereum smart contracts (sok)". In: *International conference on principles of security and trust*. Springer. 2017, pp. 164–186.

[Bad+17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. "Bitcoin as a transaction ledger: A composable treatment". In: *Annual International Cryptology Conference*. Springer. 2017, pp. 324–356.

[Cac+17] Christian Cachin, Angelo De Caro, Pedro Moreno-Sanchez, Björn Tackmann, and Marko Vukolic. "The Transaction Graph for Modeling Blockchain Semantics." In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 1070.

[HP17] Joseph Y Halpern and Rafael Pass. "A knowledge-based analysis of the blockchain protocol". In: *arXiv preprint arXiv:1707.08751* (2017).

[SH17] Ilya Sergey and Aquinas Hobor. "A concurrent perspective on smart contracts". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, pp. 478–493.

[AR18] Leonardo Alt and Christian Reitwießner. "Smt-based verification of solidity smart contracts". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 376–388.

[Atz+18a] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, and Roberto Zunino. "SoK: Unraveling Bitcoin Smart Contracts". In: *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Lujo Bauer and Ralf Küsters. Vol. 10804. Lecture Notes in Computer Science. Springer, 2018, pp. 217–242. DOI: 10.1007/978-3-319-89722-6_9. URL: https://doi.org/10.1007/978-3-319-89722-6_9.

[Atz+18b] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. "A Formal Model of Bitcoin Transactions". In: *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*. Ed. by Sarah Meiklejohn and Kazue Sako. Vol. 10957. Lecture Notes in Computer Science. Springer, 2018, pp. 541–560. DOI: 10.1007/978-3-662-58387-6_29. URL: https://doi.org/10.1007/978-3-662-58387-6_29.

[BCZ18] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. "Fun with Bitcoin smart contracts". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 432–449.

[BZ18] Massimo Bartoletti and Roberto Zunino. "BitML: a calculus for Bitcoin smart contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 83–100.

[CPR18] Xiaohong Chen, Daejun Park, and Grigore Roşu. "A language-independent approach to smart contract verification". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 405–413.

[CEP18] Christian Colombo, Joshua Ellul, and Gordon J Pace. "Contracts over smart contracts: Recovering from violations dynamically". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 300–315.

[GK18] Juan A Garay and Aggelos Kiayias. "SoK: A Consensus Taxonomy in the Blockchain Era." In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 754.

[Gre+18] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. "Madmax: Surviving out-of-gas conditions in ethereum smart contracts". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–27.

[HK18] Dominik Harz and William Knottenbelt. "Towards safer smart contracts: A survey of languages and verification methods". In: *arXiv preprint arXiv:1809.09805* (2018).

[Hir18] Yoichi Hirai. "Blockchains as Kripke Models: An Analysis of Atomic Cross-Chain Swap". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 389–404.

[ML18] Anastasia Mavridou and Aron Laszka. "Designing secure ethereum smart contracts: A finite state machine based approach". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2018, pp. 523–540.

[MCJ18] Andrew Miller, Zhicheng Cai, and Somesh Jha. "Smart contracts and opportunities for formal methods". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 280–299.

[ST18] Pablo Lamela Seijas and Simon J. Thompson. "Marlowe: Financial Contracts on Blockchain". In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 11247. Lecture Notes in Computer Science. Springer, 2018, pp. 356–375. DOI: 10.1007/978-3-030-03427-6_27. URL: https://doi.org/10.1007/978-3-030-03427-6_27.

[SKH18] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. “Scilla: a smart contract intermediate-level language”. In: *arXiv preprint arXiv:1801.00687* (2018).

[Set18] Anton Setzer. “Modelling Bitcoin in Agda”. In: *CoRR* abs/1804.06398 (2018). arXiv: 1804.06398. URL: http://arxiv.org/abs/1804.06398.

[Tsa18] Petar Tsankov. “Security analysis of smart contracts in datalog”. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 316–322.

[Wan+18] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. “Formal specification and verification of smart contracts for Azure blockchain”. In: *arXiv preprint arXiv:1812.08829* (2018).

[Zah18] Joachim Zahnentferner. “An Abstract Model of UTxO-based Cryptocurrencies with Scripts”. In: *IACR Cryptol. ePrint Arch.* (2018), p. 469. URL: https://eprint.iacr.org/2018/469.

[Aba+19] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. “Journey beyond full abstraction: Exploring robust property preservation for secure compilation”. In: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE. 2019, pp. 256–25615.

[AS19] Danil Annenkov and Bas Spitters. “Towards a smart contract verification framework in Coq”. In: *arXiv preprint arXiv:1907.10674* (2019).

[Atz+19] Nicola Atzei, Massimo Bartoletti, Stefano Lande, Nobuko Yoshida, and Roberto Zunino. “Developing secure bitcoin contracts with BitML”. In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo. ACM, 2019, pp. 1124–1128. DOI: 10.1145/3338906.3341173. URL: https://doi.org/10.1145/3338906.3341173.

[BZ19a] Massimo Bartoletti and Roberto Zunino. “Formal Models of Bitcoin Contracts: A Survey”. In: *Frontiers Blockchain* 2 (2019), p. 8. DOI: 10.3389/fbloc.2019.00008. URL: https://doi.org/10.3389/fbloc.2019.00008.

[BZ19b] Massimo Bartoletti and Roberto Zunino. “Verifying liquidity of Bitcoin contracts”. In: *International Conference on Principles of Security and Trust*. Springer. 2019, pp. 222–247.

[Ber+19] Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. “Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts”. In: *arXiv preprint arXiv:1909.08671* (2019).

[Coc19] Jesper Cockx. “Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules”. In: *25th International Conference on Types for Proofs and Programs, TYPES 2019, June 11-14, 2019, Oslo, Norway.* Ed. by Marc Bezem and Assia Mahboubi. Vol. 175. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 2:1–2:27. DOI: 10.4230/LIPIcs.TYPES.2019.2. URL: https://doi.org/10.4230/LIPIcs.TYPES.2019.2.

[Her19] Maurice Herlihy. “Blockchains from a distributed computing perspective”. In: *Communications of the ACM* 62.2 (2019), pp. 78–85.

[KMP19] Wen Kokke, Fabrizio Montesi, and Marco Peressotti. “Better late than never: a fully-abstract semantics for classical processes”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[Mar19] Diego Marmsoler. “Towards Verified Blockchain Architectures: A Case Study on Interactive Architecture Verification”. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems.* Springer. 2019, pp. 204–223.

[NS19] Jakob Botsch Nielsen and Bas Spitters. “Smart Contract Interactions in Coq”. In: *arXiv preprint arXiv:1911.04732* (2019).

[OCo19] Liam O’Connor. “Deferring the details and deriving programs”. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2019, Berlin, Germany, August 18, 2019.* Ed. by David Darais and Jeremy Gibbons. ACM, 2019, pp. 27–39. DOI: 10.1145/3331554.3342605. URL: https://doi.org/10.1145/3331554.3342605.

[PAC19] Marco Patrignani, Amal Ahmed, and Dave Clarke. “Formal approaches to secure compilation: A survey of fully abstract compilation and related work”. In: *ACM Computing Surveys (CSUR)* 51.6 (2019), pp. 1–36.

[Ser+19] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. “Safer smart contract programming with Scilla”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 185:1–185:30. DOI: 10.1145/3360611. URL: https://doi.org/10.1145/3360611.

[ANS20] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. "ConCert: a smart contract certification framework in Coq". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2020, pp. 215–228.

[Bar+20] Massimo Bartoletti, Stefano Lande, Maurizio Murgia, and Roberto Zunino. "Verification of recursive Bitcoin contracts". In: *CoRR* abs/2011.14165 (2020). arXiv: 2011.14165. URL: https://arxiv.org/abs/2011.14165.

[Cha+20a] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. "The Extended UTXO Model". In: *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*. Ed. by Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala. Vol. 12063. Lecture Notes in Computer Science. Springer, 2020, pp. 525–539. DOI: 10.1007/978-3-030-54455-3_37. URL: https://doi.org/10.1007/978-3-030-54455-3_37.

[Cha+20b] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. "Native Custom Tokens in the Extended UTXO Model". In: *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12478. Lecture Notes in Computer Science. Springer, 2020, pp. 89–111. DOI: 10.1007/978-3-030-61467-6_7. URL: https://doi.org/10.1007/978-3-030-61467-6_7.

[Cha+20c] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnentferner. "$\text{UTXO}_{\mathsf{ma}}$: UTXO with Multi-asset Support". In: *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12478. Lecture Notes in Computer Science. Springer, 2020,

pp. 112–130. DOI: 10.1007/978-3-030-61467-6_9. URL: https://doi.org/10.1007/978-3-030-61467-6_8.

[Zah20] Joachim Zahnentferner. "Multi-Currency Ledgers". In: *IACR Cryptol. ePrint Arch.* (2020), p. 895. URL: https://eprint.iacr.org/2020/895.

[CTW21] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. "The taming of the rew: a type theory with computational assumptions". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: 10.1145/3434341. URL: https://doi.org/10.1145/3434341.

[Atk22] Robert Atkey. "Data Types with Negation (talk only)". In: *9th Workshop on Mathematically Structured Functional Programming (MSFP)*. 2022. URL: https://msfp-workshop.github.io/msfp2022/atkey-abstract.pdf.

[Coc+22] Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell. "Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs". In: *Haskell '22: 15th ACM SIGPLAN International Haskell Symposium, Ljubljana, Slovenia, September 15 - 16, 2022*. Ed. by Nadia Polikarpova. ACM, 2022, pp. 108–122. DOI: 10.1145/3546189.3549920. URL: https://doi.org/10.1145/3546189.3549920.

[Gra+22] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. "Controlling unfolding in type theory". In: *CoRR* abs/2210.05420 (2022). DOI: 10.48550/arXiv.2210.05420. arXiv: 2210.05420. URL: https://doi.org/10.48550/arXiv.2210.05420.

[MSC23] Orestis Melkonian, Wouter Swierstra, and James Chapman. "Program logics for ledgers". 2023. URL: https://omelkonian.github.io/data/publications/hoare-ledgers.pdf.

[RS] Panos Rondogiannis and Ioanna Symeonidou. "Stratified disjunctive logic programs and the infinite-valued semantics". In: *18th International Workshop on Non-monotonic Reasoning (NMR)*, pp. 140–149. URL: https://openportal.isti.cnr.it/data/2020/433160/2020_433160.published.pdf#page=140.

Appendix A

`formal-prelude`: Infrastructure

More often than not, we have resorted to utilities provided by the Prelude rather than formalising every little detail we came across. These mostly include notational conveniences and gathering collections of types into *typeclasses* which is not typically done at the level of the standard library; this helps us avoid inventing different names for each specialisation we might need and definitely standarises notation.

This appendix serves the purpose of filling in some gaps in the main body of the thesis, where it might be difficult to understand some concept without any access to the Prelude utilities that are used. Here we include only those parts of the Prelude, but refer the reader to the full library here:

https://omelkonian.github.io/formal-prelude/

A.1 Syntax for unit tests

Providing examples in Agda is typically done using a "don't-care" identifier _, *e.g.,* to write some example proofs of number inequality:

```
_ : ∃ (_≤ 3)
_ = 3 , ≤-refl

_ : ∃ (_≤ 3)
_ = 2 , ≤-step ≤-refl

_ : ∃ (_≤ 3)
_ = 1 , ≤-step (≤-step ≤-refl)

_ : ∃ (_≤ 3)
_ = 0 , ≤-step (≤-step (≤-step ≤-refl))
```

When the values/unit-tests happen to share their types, this admittedly gets too repetitive and verbose like above. The type annotations from the standard library can definitely help, but we still repeat the type $n-1$ too many times:

```
_ = ∃ (_≤ 3) ∋ 3 , ≤-refl
_ = ∃ (_≤ 3) ∋ 2 , ≤-step ≤-refl
_ = ∃ (_≤ 3) ∋ 1 , ≤-step (≤-step ≤-refl)
_ = ∃ (_≤ 3) ∋ 0 , ≤-step (≤-step (≤-step ≤-refl))
```

The Prelude provides a notation for abbreviating this exact construction, letting us write multiple values for a single type:

```
_∋⋮_ : (A : Type ℓ) → List⁺ A → A
_∋⋮_ _ (x :: _) = x
pattern _⋮_ x xs = x L.NE.:: xs
```

```
pattern _⁝_ x xs = x ∷ xs
pattern _⁝∅ x = x ∷ []

_ =  ∃ (_≤ 3)
  ∋: 3 , ≤-refl
   ⁝ 2 , ≤-step ≤-refl
   ⁝ 1 , ≤-step (≤-step ≤-refl)
   ⁝ 0 , ≤-step (≤-step (≤-step ≤-refl))
   ⁝∅
```

Do not worry about the computational behaviour of `∋:` (it just returns the first item), since we cannot use it anyway as there is no way to reference the "don't care" identifier in Agda.

A.2 Syntax for inference rules

We will be working with inference rules a lot throughout the thesis, so we deem it worthy to spend some time to define a readable syntax as an alternative to Agda's built-in notation for function spaces, *e.g.,* :

```
data Y : Pred₀ Level where
 ruleY : ℓ ≢ 1ℓ
       → ℓ ≢ 2ℓ
         ------
       → Y ℓ
```

One very simple solution (credits to Andre Knispel) is to replace the last arrow → with a horizontal line consisting of fat unicode dashes, the intermediate arrows with a fat 'dot' and also add the option to have nullary hypotheses and an initial 'dot':

```
infix -50 _———————_
_———————_ : Type ℓ → Type ℓ′ → Type (ℓ ⊔ₗ ℓ′)
A ——————— B = A → B

infix -100 ———————_
———————_ : Type ℓ → Type ℓ
——————— A = A

infixr -100 _•_
_•_ : Type ℓ → Type ℓ′ → Type (ℓ ⊔ₗ ℓ′)
A • B = A → B

infix -150 •_
•_ : Type ℓ → Type ℓ
• A = A
```

(We need to repeat all line definitions for all possible lengths; that's quite tedious so we generate them using Emacs Lisp and hope the code never needs changing.)

We can now write inference rules in this more intuitive syntax; the following rule is equivalent to the initial example:

```
data X : Pred₀ Level where
  ruleX : • ℓ ≢ 1ℓ
          • ℓ ≢ 2ℓ
            ──────────
            X ℓ
```

While we are at it, why not do the same for isomorphisms, denoted by a double horizontal line, and prove rules `X` and `Y` are actually equivalent:

```
infix -50 _══════════_
_══════════_ : Type ℓ → Type ℓ′ → Type (ℓ ⊔ₗ ℓ′)
A ══════════ B = A ↔ B


_ : X ℓ
    ══════════
    Y ℓ
_ = ↝ , ↜
  where ↝ = λ where (ruleX p q) → ruleY p q
        ↜ = λ where (ruleY p q) → ruleX p q
```

Note that the syntax does not work for the dependent function space (Π types); we instead need to revert to builtin notation midway to accommodate them:

```
_ : Type
_ = ∀ {ℓ} (∃x : ∃ (_≡ ℓ)) →
    • ℓ ≢ 1ℓ
    → (∃y : ∃ (_≢ ℓ))
    → ℓ ≢ 2ℓ
    • ℓ ≢ 3ℓ
    → let x , _ = ∃x; y , _ = ∃y in
    • x ≢ y
      ──────────────────────────────────────
      ℓ ≡ 0ℓ
```

Recent support for binding variables in Agda's `syntax` notation[1] could help us implement this, but we have not yet found a solution that works in full generality:

```
──────────_ •_ : Type → Type
────────── A = A
```

[1] https://agda.readthedocs.io/en/v2.6.3/language/syntax-declarations.html

```
•    A = A

_`→`_ : Type → Type → Type
A `→` B = A → B

Π : (A : Type) → (A → Type) → Type
Π A f = ∀ (a : A) → f a

——syntax = _`→`_; ——syntax′ = Π
syntax ——syntax  A B         =     A ————— B
syntax ——syntax′ A (λ x → B) = x ⦂ A ————— B

•-syntax = _`→`_; •-syntax′ = Π
syntax •-syntax  A B         =     A • B
syntax •-syntax′ A (λ x → B) = x ⦂ A • B

_ = Type
  ∋⋮ —————
     ℕ
   ⋮ ℕ
     —————
     ℕ
   ⋮ n ⦂ ℕ
     —————
     Vec ℕ n
   ⋮ ( • ℕ
       • n ⦂ ℕ
       —————
       Vec ℕ n )
   ⋮∅
```

A.3 Deriving strategies

Throughout our work, we need to prove decidable equality of a litany of datatypes, whose methodical implementations frustrates even the most patient programmer. This is why mainstream functional languages such as Haskell have a "deriving" feature to have the compiler systematically produce these for you, usually supporting the most often used typeclasses.

We can emulate something similar using Agda's reflection machinery and its associated syntax for *unquoting* declarations.

First and foremost, a *derivation* will be a computation in the `TC` monad, which will generate the required instances of a given typeclass for a list of given types:

```
Derivation : ℕ → Set
Derivation n = List (Name × (Name ^{} n)) → TC ⊤
```

It takes as input (a list of) a type that we want the instances for, accompanied by a list of auxiliary names that might be generated, which is also recorded at the type level to make this a bit more robust. For instance, deriving `Ord` not only gives us the comparison operator but also the proofs that it is an *order* and its corresponding decision procedure.

In order to have a way to register such decision procedures, we have a `Derivable` typeclass that gathers them, which the user can then invoke by using the macro `DERIVE`
.

The Prelude currently provides deriving strategies for decidable equality (`DecEq`) and ordering (`Ord`), although they are experimental and do not work in all expected cases.

```
instance
  Derivable-DecEq : DERIVABLE DecEq 1
  Derivable-Ord   : DERIVABLE Ord   3
```

On the user side of things, we simply need to add a "deriving" clause after defining our datatypes, *e.g.,* for terms of arithmetic expressions with string variables that are indexed by their type:

```
data Ty : Type where
  `ℕ `𝔹 : Ty
data Expr : Ty → Type where
  var : String → Expr `ℕ
  ` : ℕ → Expr `ℕ
  _`+_ _`-_ : Expr `ℕ → Expr `ℕ → Expr `ℕ
  _`=_ _`<_ : Expr `𝔹 → Expr `𝔹 → Expr `𝔹
unquoteDecl DecEq-Expr = DERIVE DecEq [ quote Expr , DecEq-Expr ]
```

A.4 Decidability

The Prelude defines a class of *decidable types*, *i.e.,* ones that admit a decision procedure that either gives a proof of a value that inhabits the type or a proof of its negation:

```
record _⁇ (P : Type ℓ) : Type ℓ where
  constructor ⁇_
  field dec : Dec P

  auto : ⦃ True dec ⦄ → P
  auto ⦃ pr ⦄ = toWitness pr
```

```
contradict : ∀ {X : Type} {pr : False dec } → P → X
contradict {pr = pr} = ⊥-elim ∘ toWitnessFalse pr
```

Once we have registered the decision procedures provided by the standard library (or new ones that the Prelude contains), we can simply recall a decision procedure by instance search, rather than having to remember the exact names we have invented for each involved typed:

```
_ : ∀ {A : Type ℓ}
  → ⦃ DecEq A ⦄
  → {m : Maybe (List A)} {x₁ x₂ : A}
  → Dec $ M.Any.Any (λ xs → (xs ≡ [ x₁ ⨾ x₂ ]) ⊎ Any (const ⊤) xs) m
_ = dec
```

Alternatively, we could use special "enigma" type annotations that are specialised to decidable types:

```
¿_¿ : ∀ (X : Type ℓ) → ⦃ X ⁇ ⦄ → Dec X
¿ _ ¿ = dec
```

In other words, the type of the decidability of X is denoted as ¿ X ¿.

More importantly, we can easily prove simple (closed) formulas by embracing *proof-by-reflection*, where we crank up the gears of the associated decision procedure to get a proof out (if successful), which is precisely what calling `auto` does:

```
_ = (¬ ¬ ((true , true) ≡ (true , true)))
  × (8 ≡ 18 ∸ 10)
  ∋ auto

_ = ¬ ( ¬ ¬ ((true , true) ≡ (true , true))
      × 8 ≡ 17 ∸ 10 )
  ∋ auto
```

A.5 Accessors

Fields of a record can be accessed with the intuitive 'dot'-notation, reminiscent of *object-oriented* programming, but sometimes the field we want to access lies deeper in the record structure or is a computed/derived property computed from the primitive fields.

A cheap way to accommodate such "smarter" field accessors is to use a postfix notation of the form `_•field` that is technically a function but still resembles 'dot'-notation visually. This naive solution does not scale well though, since we would need to

continuously define aliases for our definitions, along with their precedence, leading to a lot of boilerplate code.

A more scalable solution is to define a generic typeclass of structures `A` that hold a field of some type `B` that might be dependent on a specific `a : A` although we mostly care about the non-indexed version here:

```
record HasField′ (A : Type) (B : A → Type) : Type where
  constructor mkHasField
  field _∙ : (a : A) → B a

HasField : Op₂ Type
HasField A B = HasField′ A (const B)
```

Crucially, the typeclass itself does not give a name to the field, this will be handled by renaming at the module (a record is also a module that we can open[2]).

As an example, we can define a number field for a list, computing its length:

```
instance
  _ : HasField (List A) ℕ
  _ = λ where ._∙ → length
_ = [ "single" ] ∙ ≡ 1 ∋ refl
```

But what we want is to have specific names/aliases for certain types, so let us first capture the type of things we will need to rename, namely the accessor's type, the name of the actual accessor, and a way to build an instance:

```
AccessorTy : Type → (Type → Type)
AccessorTy = flip HasField

Accessor : Type → Type₁
Accessor B = ∀ {A} → ⦃ HasField A B ⦄ → A → B

AccessorBuilder : Type → Type₁
AccessorBuilder B = ∀ {A} → (A → B) → HasField A B
```

That way we can have a more intuitive class of numeric fields and an instantiation on lists as before:

```
HasNum = AccessorTy ℕ
_∙num = Accessor ℕ ∋ _∙
∙num= = AccessorBuilder ℕ ∋ mkHasField
instance
  _ : HasNum (List A)
  _ = ∙num= length
_ = [ "single" ] ∙num ≡ 1 ∋ refl
```

[2] https://agda.readthedocs.io/en/v2.6.3/language/record-types.html#record-modules

The alias declarations are definitely boilerplate which, thanks to the meta-programming facilities of Agda[3] for *elaborator reflection* [CB16], we can automate away pretty easily:

```
genAccessor : Name → Name → Name → Name → TC ⊤
genAccessor ty f mk B = do
  declareDef (vArg ty) unknown
  defineFun ty [ ⟦⇒ quote AccessorTy ∙⟦ def B [] ⟧ ⟧ ]
  declareDef (vArg f) (quote Accessor ∙⟦ def B [] ⟧)
  defineFun f [ ⟦⇒ def (quote _∙) [] ⟧ ]
  declareDef (vArg mk) (quote AccessorBuilder ∙⟦ def B [] ⟧)
  defineFun mk [ ⟦⇒ con (quote mkHasField) [] ⟧ ]
```

This finally lets us define field accessors of different kinds (names appropriately in each case) for possibly several types, *e.g.,* define numeric fields as before, and in the same namespace define string accessors both for lists of elements we can `show` , as well as functions and permutation proofs:

```
unquoteDecl HasNum _∙num ∙num=_ = genAccessor HasNum _∙num ∙num=_ (quote ℕ)
instance
  _ : HasNum (List A)
  _ = ∙num= length
_ = [ "single" ] ∙num ≡ 1 ∋ refl

unquoteDecl HasStr _∙str ∙str=_ = genAccessor HasStr _∙str ∙str=_ (quote String)
instance
  _ : ⦃ Show A ⦄ → HasStr (List A)
  _ = ∙str= show

  _ : HasStr (A → B)
  _ = ∙str= const "«thunk»"

  _ : HasStr (xs ↭ ys)
  _ = ∙str= const "«proof»"

_ = [ 42 ]                            ∙str ≡ "{42}"    ∋ refl
_ = ([ 1 ⨾ 2 ] ↭ [ 2 ⨾ 1 ] ∋ auto) ∙str ≡ "«proof»" ∋ refl
_ = (_∸ 1)                            ∙str ≡ "«thunk»" ∋ refl
```

A.6 Collections

Frequently there is a need to collect sub-components of large structures into a list, which is implemented as a structural traversal, *e.g.,* define a function `ids` to gather all names

[3] https://agda.readthedocs.io/en/v2.6.3/language/reflection.html

used in a term. Furthermore, you typically collect the same type of elements from different types, leading to an explosion of names you have to give to your definitions, *e.g.*, `idsᵗ` for terms, `idsᵈ` for proof derivations, and so forth.

One could remedy this by introducing a typeclass of types from which we can collect a specific type of elements (*e.g.*, names), but this leads to an explosion of typeclasses because we will need to collect different kind of elements.

A more generic approach is a typeclass for collecting elements of any type from any larger type:

```
record _has_ (A : Type) (B : Type) : Type where
  field collect : A → List B
```

Apart from providing a standard way to implement such collections that does not require inventing ad-hoc names or typeclasses, it also gives us a way to remove boilerplate constructions that we would otherwise need to repeat for each type, *e.g.*, collecting from a list by iteratively collecting each element along with a proof that the collection is preserved under permutation:

```
collectFromList : (X → List Y) → (List X → List Y)
collectFromList = concatMap

collectFromList↭ : ∀ (f : X → List Y) →
  xs ↭ ys
  ─────────────────────────────────────────
  collectFromList f xs ↭ collectFromList f ys
collectFromList↭ = ↭-concatMap⁺
```

One step further, we define a heterogeneous relation between types that have the same kind of elements to collect:

```
relateOn : ∀ {ℓ} {Z A B : Type} {Z′ : Type ℓ} → ⦃ A has Z ⦄ → ⦃ B has Z ⦄
  → Rel₀ Z′
  → A
  → (∀ {X} → ⦃ X has Z ⦄ → X → Z′)
  → B → Type
relateOn _~_ a f b = f a ~ f b
syntax relateOn _~_ a f b = a ⟪ _~_ on f ⟫ b
```

We then instantiate it to the operations we will need for our use cases, letting us write `x ↭⟪ ids ⟫ y` to mean that the names used in `y` are a permutation of those in `x`.

```
module _ {Z}{A}{B}⦃ ia ⦄ ⦃ ib ⦄ where
  _→⟪_⟫_ = relateOn {Z = Z}{A}{B}{Type}⦃ ia ⦄ ⦃ ib ⦄ _`→`_
  module _ {Z′} where
    _≡⟪_⟫_ = relateOn {Z = Z}{A}{B}{Z′}⦃ ia ⦄ ⦃ ib ⦄ _≡_
    _↭⟪_⟫_ = relateOn {Z = Z}{A}{B}{List Z′}⦃ ia ⦄ ⦃ ib ⦄ _↭_
    _⊆⟪_⟫_ = relateOn {Z = Z}{A}{B}{List Z′}⦃ ia ⦄ ⦃ ib ⦄ _⊆_
```

A.7 Coercions

...we define a typeclass of coercions between any two types, along with a notation for specifying the type to resolve ambiguity.

```
record _↝_ (A B : Type) : Type where
  field to : A → B
  syntax to {B = B} = to[ B ]
_↜_ = flip _↝_
```

Crucially, we do not give *ad hoc* names for each translation (and each direction) that we might come across; instead preferring to use a standard name `to` for all coercions that we can easily keep in long-term memory, and, if it is needed for type inference or to communication to the reader, we can also specify a specific translation in a purely type-guided fashion via `to[ T ]`.

For instance, we could define coercions between booleans and some isomorphic type that is defined differently:

```
𝟚 = ⊤ ⊎ ⊤; pattern 𝕃 = inj₁ tt; pattern ℝ = inj₂ tt
instance
  _ = 𝟚 ↝ Bool ∋ λ where .to → λ where
    𝕃 → true
    ℝ → false
  _ = Bool ↝ 𝟚 ∋ λ where .to → λ where
    true → 𝕃
    false → ℝ

_ : Bool
_ = to 𝕃

_ : 𝟚
_ = to true

_ : Bool → Bool
_ = not ∘ to[ Bool ] ∘ to[ 𝟚 ]
```

When both directions are available, we get an isomorphism by instance search:

```
instance
  ↝↜ : ⦃ A ↝ B ⦄ → ⦃ A ↜ B ⦄ → A ↔ B
  ↝↜ = to , to

_ = (𝟚 ↔ Bool) ∋ it
```

A.8 No-nil list notation

Lists are one of the most fundamental data structures in functional programming, and we normally construct literal lists using the constructors of the `List` datatype: 'nil' (`[]`) and 'cons' (`∷`).

However, sometimes on paper we freely use a binary operator to mean *a list of things* without any termination operation like `[]`. If that wasn't enough, we sometimes conflate a single element with the singleton list. One example where both of the above apply is the notation for BitML branches in the original paper [BZ18].

To accommodate this in a strongly-typed proof assistant like Agda, we will need to employ some typeclass trickery in order to achieve the same kind of *ad-hoc polymorphism* as on paper.

To that end, we define a class of list-like types; either such a value holds one element or a list of elements, and this admits an elimination to list by pattern matching to see what the value actually is:

```
pattern 𝕃 = inj₁ refl; pattern ℝ = inj₂ refl

record List? (X A : Type) : Type₁ where
  field isList : (A ≡ X) ⊎ (A ≡ List X)
  syntax isList {X}{A} = A isListOf? X

  toL : A → List X
  toL with isList
  ... | 𝕃 = [_]
  ... | ℝ = id
  syntax toL {A = A} = toL[ A ]
```

We are careful to provide just the two instances of single elements and actual lists, and then restrict access to the constructors of this typeclass to the rest of the world.

```
instance
  Pick𝕃 : List? X X
  Pick𝕃 = record {isList = 𝕃}

  Pickℝ : List? X (List X)
  Pickℝ = record {isList = ℝ}
```

We can now talk about a *heterogeneous* binary operator that combines two list-like things into a list:

```
_⊕_ : ⦃ List? X A ⦄ → ⦃ List? X B ⦄ → A → B → List X
x ⊕ y = toL x ++ toL y
```

In other words, single elements are *implicitly coerced* to singleton lists, allowing us to freely interchange them as operands of a composition chain.

This finally allow us to write lists in all the following ways, akin to what we would use on paper (possibly with the operator renamed based on the application):

```
_ = List ℕ
 ∋: []
  ∶ [ 0 ]
  ∶ 0 ⊕ 1
  ∶ 0 ⊕ [ 1 ]
  ∶ [ 0 ] ⊕ 1
  ∶ [ 0 ] ⊕ [ 1 ]
  ∶ 0 ⊕ 1 ⊕ 2
  ∶ 0 ⊕ 1 ⊕ [ 2 ]
  ∶ 0 ⊕ [ 1 ] ⊕ 2
  ∶ [ 0 ] ⊕ 1 ⊕ 2
  ∶ 0 ⊕ [ 1 ] ⊕ [ 2 ]
  ∶ [ 0 ] ⊕ [ 1 ] ⊕ 2
  ∶ [ 0 ] ⊕ 1 ⊕ [ 2 ]
  ∶ [ 0 ] ⊕ [ 1 ] ⊕ [ 2 ]
  ∶∅
```

The Prelude actually also provides a more general class[4] that accommodates any *variant*-like structure that admits an implicit coercion to one of the summands, but we will not use that here.

A.9 Relation closures

In various occasions we want to take the reflexive and transitive closure of a binary relation, so as to talk about sequences of transitions. One could naively repeat the standard construction on every relation we care about, *e.g.*, for some type of transitions `_—→_`:

```
data _—↠_ : Rel₀ X where
 base : ∀ {x} → x —↠ x
 step : ∀ {x y z} → x —→ y → y —↠ z → x —↠ z
```

But we can do better, in terms of scalability and maintainability, by defining this generically for an arbitrary relations for any type X:

```
module ReflexiveTransitiveClosure {A : Type ℓ} (_—→_ : Rel A ℓ) where


pattern begin_ x = x
data _—↠_ : Rel A ℓ where
```

[4] https://omelkonian.github.io/formal-prelude/Prelude.Variants

```
  _■ : ∀ x → x —↠ x
  _—→⟨_⟩_ : ∀ x → x —→ y → y —↠ z → x —↠ z
```

We can derive some simple constructions on said relation; we need to define them just once and they will be available for any future instantiation. These include a non-reflexive version, some helper properties and notation, as well as a "snoc" version that adds the most recent transition, which can be implemented as a view that upholds the expected isomorphism:

```
data _—↠⁺_ : Rel A ℓ where
  _—→⟨_⟩_ : ∀ x → x —→ y → y —↠ z → x —↠⁺ z

—↠-trans : Transitive _—↠_
—↠-trans (x ■) xz = xz
—↠-trans (_ —→⟨ r ⟩ xy) yz = _ —→⟨ r ⟩ —↠-trans xy yz

_—↠⟨_⟩_ : ∀ x → x —↠ y → y —↠ z → x —↠ z
_—↠⟨_⟩_ _ = —↠-trans

_←—_ = flip _—→_
data _↞—_ : Rel A ℓ where
  _■ : ∀ x → x ↞— x
  _⟨_⟩←—_ : ∀ z → z ←— y → y ↞— x → z ↞— x
data _⁺↞—_ : Rel A ℓ where
  _⟨_⟩←—_ : ∀ z → z ←— y → y ↞— x → z ⁺↞— x

↞—-trans : Transitive _↞—_
↞—-trans (x ■) xz = xz
↞—-trans (_ ⟨ r ⟩←— zy) yx = _ ⟨ r ⟩←— ↞—-trans zy yx

_⟨_⟩↞—_ : ∀ z → z ↞— y → y ↞— x → z ↞— x
_⟨_⟩↞—_ _ = ↞—-trans

_`—→⟨_⟩_ : ∀ x → y ↞— x → z ←— y → z ↞— x
_ `—→⟨ st ⟩ _■ = _ ⟨ st ⟩←— _■
_ `—→⟨ st ⟩ _⟨ st′ ⟩←— p = _ ⟨ st′ ⟩←— _ `—→⟨ st ⟩ p

viewLeft : x —↠ y → y ↞— x
viewLeft (_■) = _■
viewLeft (_ —→⟨ st ⟩ p) = _ `—→⟨ st ⟩ viewLeft p

_`⟨_⟩←—_ : ∀ z → y —→ z → x —↠ y → x —↠ z
_ `⟨ st ⟩←— (_■) = _ —→⟨ st ⟩ _■
_ `⟨ st ⟩←— (_ —→⟨ st′ ⟩ p) = _ —→⟨ st′ ⟩ (_ `⟨ st ⟩←— p)

viewRight : y ↞— x → x —↠ y
viewRight (_■) = _■
```

```
viewRight (_ ⟨ st ⟩←— p) = _ `⟨ st ⟩←— viewRight p

view↔ : (x —↠ y) ↔ (y ↞— x)
view↔ = viewLeft , viewRight
```

As an example, consider an artificial transition between numbers that can either increment or decrement, *i.e.*, a relation between consecutive numbers:

```
data _—→_ : Rel₀ ℕ where
  [inc] : n —→ suc n
  [dec] : suc n —→ n
```

Simply `open`ing the module gives us the notation for chaining transitions together:

```
open ReflexiveTransitiveClosure _—→_

_ : 10 —↠ 10
_ = begin 10 ∎

_ : 10 —↠⁺ 12
_ = begin
  10 —→⟨ [inc] ⟩
  11 —→⟨ [dec] ⟩
  10 —→⟨ [inc] ⟩
  11 —→⟨ [inc] ⟩
  12 ∎

_ : 12 ⁺↞— 10
_ = begin
  12 ⟨ [inc] ⟩←—
  11 ⟨ [inc] ⟩←—
  10 ⟨ [dec] ⟩←—
  11 ⟨ [inc] ⟩←—
  10 ∎
```

Furthermore, we might need to have labelled relations, where each transition is "tagged" with a label of some other kind, from which we can always regain the non-labelled relation by internally postulating a label:

```
LRel : Type ℓ × Type → (ℓ′ : Level) → Type (ℓ ⊔ₗ lsuc ℓ′)
LRel (A , L) ℓ = A → L → A → Type ℓ


unlabel : LRel (A , L) ℓ′ → Rel A ℓ′
unlabel _—→[_]_ x y = ∃ λ α → x —→[ α ] y
```

We can easily extend the previous generic construction to also account for labels and derive the same utilities as before:

```
module LabelledReflexiveTransitiveClosure
  {A : Type ℓ} {L : Type} (_—[_]→_ : LRel (A , L) ℓ) where

data _—[_]↠_ : LRel (A , List L) ℓ where
  _∎ : ∀ x → x —[ [] ]↠ x
  _—→⟨_⟩_ : ∀ x → x —[ α ]→ y → y —[ αs ]↠ z → x —[ α ∷ αs ]↠ z
```

Back to the increment/decrement example, we can add string labels to each operation and write equational-style proofs (with some additional notation specific to labels).

```
data _—[_]→_ : LRel₀ (ℕ , String) where
  [inc] : n —[ "inc" ]→ suc n
  [dec] : suc n —[ "dec" ]→ n

open LabelledReflexiveTransitiveClosure _—[_]→_

_ : 10 —[ [] ]↠ 10
_ = begin 10 ∎

_ : 10 —↠ 10
_ = emit: begin 10 ∎

_ : 10 —↠⁺ 12
_ = emitting [ "inc" ⨾ "dec" ⨾ "inc" ⨾ "inc" ] :
    begin 10 —→⟨ [inc] ⟩
          11 —→⟨ [dec] ⟩
          10 —→⟨ [inc] ⟩
          11 —→⟨ [inc] ⟩
          12 ∎

_ : 12 ⁺↞— 10
_ = emitting [ "inc" ⨾ "dec" ⨾ "inc" ⨾ "inc" ] :
    begin 12 ⟨ [inc] ⟩←—
          11 ⟨ [inc] ⟩←—
          10 ⟨ [dec] ⟩←—
          11 ⟨ [inc] ⟩←—
          10 ∎
```

The final variation we can accommodate is for relating elements of a type equipped with a *setoid* structure; this will prove particularly handy for the semantics of the BitML calculus in Chapter 4 which come in the form of transitions between configuration that are considered equivalent *up to permutation*:

```
module UpToLabelledReflexiveTransitiveClosure
  {A : Type ℓ} {L : Type} (_—[_]→_ : LRel (A , L) ℓ) ⦃ _ : ISetoid A ⦄ where

  open LabelledReflexiveTransitiveClosure _—[_]→_ public
```

```
data _—[_]↠_ : LRel (A , List L) (ℓ ⊔ₗ relℓ) where
  _∎ : ∀ x → x —[ [] ]↠ x
  _—→⟨_⟩_⊢_ : ∀ x {x′ y y′ z}
    → x′ —[ α ]→ y′
    → x ≈ x′ × y ≈ y′
    → y —[ αs ]↠ z
      ------------------
    → x —[ α ∷ αs ]↠ z
_—↠_ = unlabel _—[_]↠_
```

Apart from the previous constructions, we also want to remove the proof obligations in case we are working on closed terms of decidable equivalence:

```
module _ ⦃ _ : DecSetoid A ⦄ where


_—→⟨_⟩_ : ∀ (x : A)
  → x′ —[ α ]→ y′
  → y —[ αs ]↠ z
  → {True $ x ≈? x′}
  → {True $ y ≈? y′}
    ------------------
  → x —[ α ∷ αs ]↠ z
(x —→⟨ x′→y′ ⟩ y↠z) {p₁} {p₂} = x —→⟨ x′→y′ ⟩ toWitness p₁ , toWitness p₂ ⊢ y↠z
```

Finally, if the type of states (*i.e.*, the type being related) has a notion of *initiality*, we can talk about *well-rooted traces* arising from a closure:

```
record HasInitial (A : Type) : Type₁ where
  field Initial : Pred₀ A


record Trace ⦃ _ : HasInitial A ⦄ : Type where
  field
    start end : A
    trace : start —↠ end
    init : Initial start
```

A.10 Views

Going a bit further than the coercions of Section A.7, we can provide standard notation for *viewing* isomorphic representation of the same objects, which is a well-known technique in functional programming [Wad87].

```
record _▷_ (A : Type) (B : Type) : Type where
 field
  view   : A → B
  unview : B → A
  unview∘view : ∀ x → unview (view x) ≡ x
  view∘unview : ∀ y → view (unview y) ≡ y
```

This lets us declare various instances of this sort and then invoke them just by referring to the types, so no need to remember *ad hoc* names:

```
view_as_ : A → (B : Type) ⦃ _ : A ▷ B ⦄ → B
view x as B = view {B = B} x
```

Let us demonstrate with the typical example of “snoc”-lists, where new elements are inserted on the right end of the tail rather than prepended at the front as usual:

```
data SnocList (A : Type) : Type where
 []  : SnocList A
 _∷_ : SnocList A → A → SnocList A

_∷ˡ_ : A → SnocList A → SnocList A
x ∷ˡ []        = [] ∷ x
x ∷ˡ (xs ∷ y) = (x ∷ˡ xs) ∷ y

snocView : List A → SnocList A
snocView = λ where
 [] → []
 (x ∷ xs) → x ∷ˡ snocView xs

snocUnview : SnocList A → List A
snocUnview = λ where
 [] → []
 (xs ∷ x) → snocUnview xs ∷ʳ x

snocUnview-∷ˡ : ∀ (x : A) xs → snocUnview (x ∷ˡ xs) ≡ x ∷ snocUnview xs
snocUnview-∷ˡ t [] = refl
snocUnview-∷ˡ x (xs ∷ _) rewrite snocUnview-∷ˡ x xs = refl

snocView-∷ʳ : ∀ (x : A) xs → snocView (xs ∷ʳ x) ≡ snocView xs ∷ x
snocView-∷ʳ _ [] = refl
snocView-∷ʳ x (_ ∷ xs) rewrite snocView-∷ʳ x xs = refl

instance
 SnocView : List A ▷ SnocList A
 SnocView .view   = snocView
 SnocView .unview = snocUnview
 SnocView .unview∘view [] = refl
```

```
  SnocView .unview∘view (x ∷ xs)
   rewrite snocUnview-∷ˡ x (view xs)
    |      unview∘view xs
    =      refl
  SnocView .view∘unview [] = refl
  SnocView .view∘unview (xs ∷ x)
   rewrite snocView-∷ʳ x (unview xs)
    |      view∘unview xs
    =      refl
```

Giving the instance entails registering the back-and-forth translation between the two isomorphic types, as well as prove they cancel out.

A more natural definition of retrieving the last element of a list can now be phrased as:

```
last : List A → Maybe A
last {A = A} xs with view xs as SnocList A
... | [] = nothing
... | _ ∷ x = just x
```

It is important to note that this does not scale to the more advanced *view from the left* [MM04], where one of the viewed types is indexed by the other and thus leads to a dependently-typed class of views:

```
record _▷_ (A : Set) (P : Pred₀ A) : Set where
  field
    view : (x : A) → P x
    view∘unview : ∀ {x} (y : P x) → view x ≡ y

  unview : ∀ {x} → P x → A
  unview {x = x} _ = x

  unview∘view : ∀ x → unview (view x) ≡ x
  unview∘view _ = refl


view_as_ : (x : A) (P : Pred₀ A) ⦃ _ : A ▷ P ⦄ → P x
view x as P = view {P = P} x
```

The new dependency also made this simpler; since we can no longer follow a bidirectional approach, it does not make sense to have `unview` anymore:

"Snoc"-lists will now record their corresponding "cons"-list in their type:

```
data SnocList {A : Set} : Pred₀ (List A) where
  [] : SnocList []
  _∷_ : ∀ {xs} → SnocList xs → (x : A) → SnocList (xs L.∷ʳ x)
```

```
snocView : (xs : List A) → SnocList xs
snocView []       = []
snocView (x ∷ xs) = x ∷ˡ snocView xs
 where
 _∷ˡ_ : (x : A) {xs : List A} → SnocList xs → SnocList (x ∷ xs)
 x ∷ˡ []       = [] ∷ x
 x ∷ˡ (xs ∷ y) = (x ∷ˡ xs) ∷ y
```

The instance we have to provide is more lightweight now, since we only need to take care of one side of the translation:

```
instance
 SnocView : List A ▷ SnocList
 SnocView .view = snocView
 SnocView .view∘unview {.[]} [] = refl
 SnocView .view∘unview {.(_ ∷ʳ x)} (_ ∷ x) =
  trans (view-∷ʳ _ _) $ cong (_∷ x) $ view∘unview _
  where
  view-∷ʳ : ∀ (x : A) xs → view (xs ∷ʳ x) ≡ view xs ∷ x
  view-∷ʳ _ [] = refl
  view-∷ʳ x (_ ∷ xs) rewrite view-∷ʳ x xs = refl
```

As before, we can use type-directed instance selection:

```
last : List A → Maybe A
last xs with view xs as SnocList
... | []     = nothing
... | _ ∷ x = just x
```

A.11 Measure-based termination

We will see many cases of well-founded recursion/induction needed in the thesis, *e.g.,* to prove the tracing properties of BitML's operational semantics in Section 6.2.3 we will need to perform induction on prefixes of the trace.

Chapter 5 introduced well-founded recursion using accessibility predicates to define the BitML compiler, but sometimes it's easier to use a *termination measure* that is computed from the structure we are recursing. Restricting to numeric measures, we can define the class of measurable types that have a cardinality method to compute their size:

```
record Measurable {ℓ} (A : Type ℓ) : Type ℓ where
 field |_| : A → ℕ
```

We can then derive the machinery for well-founded induction/recursion by appealing to the well-foundedness of natural numbers:

```
module _ ⦃ _ : Measurable A ⦄ where

  _<_ : Rel A 0ℓ
  _<_ = _<_ on ∣_∣

  <-wf : WellFounded _<_
  <-wf = On.wellFounded ∣_∣ <-wellFounded

  <-rec : Recursor (WfRec _<_)
  <-rec = All.wfRec <-wf 0ℓ
```

This approach even allows us a heterogeneous relation on measurable items, *i.e.*, where we can recurse to values of different types that have a strictly smaller measure:

```
∃Measurable : Type₁
∃Measurable = Σ[ A ∈ Type ] (Measurable A) × A
instance _ = Measurable ∃Measurable ∋ λ where .∣_∣ (_ , record {∣_∣ = f} , x) → f x

_<ᵐ_ : ∀ {A B : Type} ⦃ _ : Measurable A ⦄ ⦃ _ : Measurable B ⦄ → A → B → Type
x <ᵐ y = toMeasure x < toMeasure y
  module _ where
    toMeasure : ∀ {A : Type} ⦃ _ : Measurable A ⦄ → A → ∃Measurable
    toMeasure ⦃ mx ⦄ x = _ , mx , x
```

As an example, if we wanted to recurse to the middle of a list in a repetitive fashion, we can declare the termination measure of lists as their length and use the properties of natural numbers to prove termination:

```
instance _ = (∀ {A : Type} → Measurable (List A)) ∋ λ where .∣_∣ → length

middles : ∀ {A} → List A → List A
middles {A} = <-rec _ λ where
  [] _ → []
  (x ∷ xs) rec → x ∷ rec (take (⌊ length xs /2⌋) xs) (s≤s $ length-take/2 {xs})
```

A.12 Substitution syntax

When we want to inform the type level that two terms are propositionally equal, we typically use `subst` from the standard library. It is directed however, but the name does not suggest which way the equality should face to make it right; you always have

to return to the type signature at the definition site to remind yourself before getting on with your life.

To remedy this, the Prelude provides a more intuitive syntax that gives a strong hint as to which direction the equality should be facing:

- **forwards:** *from* the goal *to* the term Here we are rewriting terms in the goal, until we arrive at a goal type that we can precisely match with the term at hand.
- **backwards:** *from* the term *to* the goal Vice versa, we might want to start rewriting on a term we hold and successively refine it to match the final goal.

```
-- forwards
substˡ = subst; substˡ′ = substˡ
substˡ″ : {P : Pred₀ A} → x ≡ y → P x → P y
substˡ″ = substˡ _
syntax substˡ  (λ ◆ → P) eq p = p :~ eq ⟪ ◆ ∣ P ⟫
syntax substˡ′ P         eq p = p :~ eq ⟪ P ⟫
syntax substˡ″           eq p = p :~ eq
-- backwards
substʳ : (P : Pred₀ A) → y ≡ x → P x → P y
substʳ P eq p = subst P (sym eq) p
substʳ′ = substʳ
substʳ″ : {P : Pred₀ A} → y ≡ x → P x → P y
substʳ″ = substʳ _
syntax substʳ (λ ◆ → P) eq p = ⟪ ◆ ∣ P ⟫ eq ~: p
syntax substʳ′ P        eq p = ⟪ P ⟫     eq ~: p
syntax substʳ″          eq p =           eq ~: p
```

Note that we also provide a `syntax` declaration whenever there is binding involved, at the same time suggesting the *rhombus* (◆) variable name that gives strong indications as to which *hole* in the context we are focusing on.

To illustrate, here is the same proof facing both forwards and backwards:

```
module _ (n m : ℕ) (n≡m : n ≡ m) (P : Pred₀ ℕ) (p : P m × P n) where

  _ : P (n + 0) × P (0 + m)
  _ = ⟪ ◆ ∣ P ◆ × P (0 + m) ⟫ +-identityʳ _ ~:
      ⟪ ◆ ∣ P ◆ × P (0 + m) ⟫ n≡m ~:
      ⟪ ◆ ∣ P m × P ◆ ⟫ +-identityˡ _ ~:
      ⟪ ◆ ∣ P m × P ◆ ⟫ sym n≡m  ~: p

  _ : P (n + 0) × P (0 + m)
  _ = p :~ n≡m      ⟪ ◆ ∣ P m × P ◆           ⟫
        :~ sym (+-identityˡ _) ⟪ ◆ ∣ P m × P ◆ ⟫
```

```
    :~ sym n≡m ⟪ ◆ ∣ P ◆ × P (0 + m) ⟫
    :~ sym (+-identityʳ _) ⟪ ◆ ∣ P ◆ × P (0 + m) ⟫
```

Writing these contexts is quite tedious and often only serves Agda's type inference, so the Prelude also provides a term-level rewriting macro emulating `rewrite` using reflection:

https://omelkonian.github.io/formal-prelude/Prelude.Tactics.Rewrite.html

This essentially figures out the contexts for us, but also allows simultaneous substitution and/or controlling a subset of the term occurrences to replace, resembling what one has available in a tactics-based proof assistant such as Coq.[5] Alas, it is still highly unstable so we refrained from using it in the BitML work.

A.13 Meta-properties

In our BitML formalisation, we were frequently using membership proofs *computationally*, *i.e.,* thinking about them as indices to a position in a list, which is what drove the BitML compiler. When permutation came into play (by the configurations of the operational semantics), we started using *properties* that relate membership and permutation, *e.g.,* :

```
∈-resp-↭ : ∀ {x : A} → (x ∈_) Respects _↭_
```

Since we care about the computational contents of our membership proofs, we also care about the computational content of properties such as the above. This is what we refer to as *meta-properties*, since we now have to prove properties of the *ground properties*.

Let us illustrate such a meta-property, namely that the previous property commutes with the functorial action on membership proofs:

```
Any-map∘∈-resp-↭ : ∀ {A : Type} {x y : A} {xs ys : List A}
  (f : (x ≡_) ⊆¹ (y ≡_))
  (p : xs ↭ ys)
  (x∈ : x ∈ xs)
  ------------------------
  → ( L.Any.map f -- y ∈ ys
    ∘ ∈-resp-↭ p  -- x ∈ ys
    ) x∈          -- x ∈ xs
  ≡ ( ∈-resp-↭ p  -- y ∈ ys
    ∘ L.Any.map f -- y ∈ xs
    ) x∈          -- x ∈ xs
Any-map∘∈-resp-↭ _ refl _ = refl
```

[5] https://coq.inria.fr/distrib/current/refman/proofs/writing-proofs/equality.html#coq:tacn.rewrite

```
Any-map∘∈-resp-↭ f (↭-trans p p′) x∈
  rewrite sym $ Any-map∘∈-resp-↭ f p x∈
    | sym $ Any-map∘∈-resp-↭ f p′ (∈-resp-↭ p x∈)
    = refl
Any-map∘∈-resp-↭ f (prep x p) x∈ with x∈
... | here _ = refl
... | there x∈ = cong there $ Any-map∘∈-resp-↭ f p x∈
Any-map∘∈-resp-↭ f (swap x y p) x∈ with x∈
... | here _ = refl
... | there (here _) = refl
... | there (there x∈) = cong there $ cong there $ Any-map∘∈-resp-↭ f p x∈
```

Proving a meta-property involves reasoning about the previous proof of the base-property, so it suggests that we should also care about how are proofs are coded up, leading to a more software-engineering perspective on theorem proving, which we hope to have embraced throughout this thesis.

Things quickly gets out of hand, as later meta-properties will need lemmas about all other constructions on list we are mentioning. Just to illustrate, here is a humongous proof about list `concat`anation with type annotations to guide the reader, which refers to similarly scary lemmas about related operations:

```
Any-resp-↭∘Any-concat⁺ :
  (xss↭ : xss ↭ yss)
  (xs∈  : Any (Any P) xss) →
  ─────────────────────────────────────────────────
    ( Any-resp-↭ (↭-concat⁺ xss↭) -- Any P (concat yss)
    ∘ L.Any.concat⁺               -- Any P (concat xss)
    ) xs∈                         -- Any (Any P) xss
  ≡ ( L.Any.concat⁺      -- Any P (concat yss)
    ∘ Any-resp-↭ xss↭    -- Any (Any P) yss
    ) xs∈                -- Any (Any P) xss
Any-resp-↭∘Any-concat⁺ refl _ = refl
Any-resp-↭∘Any-concat⁺ (↭-prep {xs = xss}{yss} xs xss↭) xs∈
  with xs∈
... | here x∈ = Any-resp-↭∘Any-++⁺ˡˡ {zs = xs} (↭-concat⁺ xss↭) x∈
... | there xs∈
  rewrite Any-resp-↭∘Any-++⁺ˡʳ {zs = xs} (↭-concat⁺ xss↭) (L.Any.concat⁺ xs∈)
    |     Any-resp-↭∘Any-concat⁺ xss↭ xs∈
    =     refl
Any-resp-↭∘Any-concat⁺ (↭-trans p q) x∈
  rewrite Any-resp-↭∘Any-concat⁺ p x∈
  = Any-resp-↭∘Any-concat⁺ q (Any-resp-↭ p x∈)
Any-resp-↭∘Any-concat⁺ {xss = .xs ∷ .ys ∷ xss} {yss = .ys ∷ .xs ∷ yss}
                        (swap xs ys xss↭) xs∈
```

```
  rewrite ↭-trans∘reflʳ (++⁺ˡ ys $ ++⁺ˡ xs $ ↭-concat⁺ xss↭)
  with xs∈
... | here x∈ =
  begin
    ( Any-resp-↭
        (L.Perm.↭-trans (shifts xs ys) $ (++⁺ˡ ys $ ++⁺ˡ xs $ ↭-concat⁺ xss↭))
    ∘ L.Any.++⁺ˡ
    ) x∈
  ≡⟨ Any-resp-↭∘↭-trans (L.Any.++⁺ˡ x∈) (shifts xs ys)
                        (++⁺ˡ ys $ ++⁺ˡ xs $ ↭-concat⁺ xss↭) ⟩
    ( Any-resp-↭ (++⁺ˡ ys $ ++⁺ˡ xs $ ↭-concat⁺ xss↭) -- Any P (ys++xs++concat yss)
    ∘ Any-resp-↭ (shifts xs ys {concat xss})          -- Any P (ys++xs++concat xss)
    ∘ L.Any.++⁺ˡ                                      -- Any P (xs++ys++concat xss)
    ) x∈                                              -- Any P xs
  ≡⟨ Any-resp-↭∘ˡˡshifts {xs = xs} (L.Any.++⁺ˡ x∈) (↭-concat⁺ xss↭) ⟩
    ( Any-resp-↭ (shifts xs ys)                       -- Any P (ys++xs++concat yss)
    ∘ Any-resp-↭ (++⁺ˡ xs $ ++⁺ˡ ys $ ↭-concat⁺ xss↭) -- Any P (xs++ys++concat yss)
    ∘ L.Any.++⁺ˡ                                      -- Any P (xs++ys++concat xss)
    ) x∈                                              -- Any P xs
  ≡⟨ cong (Any-resp-↭ (shifts xs ys {concat yss}))
      $  Any-resp-↭∘Any-++⁺ˡˡ (++⁺ˡ ys $ ↭-concat⁺ xss↭) x∈ ⟩
    ( Any-resp-↭ (shifts xs ys {concat yss}) -- Any P (ys ++ xs ++ ws)
    ∘ L.Any.++⁺ˡ                             -- Any P (xs ++ ys ++ ws)
    ) x∈                                     -- Any P xs
  ≡⟨ Any-resp-↭∘shiftsˡ x∈ ⟩
    ( L.Any.++⁺ʳ ys -- Any P (ys ++ xs ++ ws)
    ∘ L.Any.++⁺ˡ -- Any P (xs ++ ws)
    ) x∈         -- Any P xs
  ∎
... | there (here x∈) =
  begin
    ( Any-resp-↭ ( L.Perm.↭-trans (shifts xs ys)
                 $ (++⁺ˡ ys $ ++⁺ˡ xs $ ↭-concat⁺ xss↭)
                 )
    ∘ L.Any.++⁺ʳ xs
    ∘ L.Any.++⁺ˡ
    ) x∈
  ≡⟨ Any-resp-↭∘↭-trans _ (shifts xs ys) (++⁺ˡ ys $ ++⁺ˡ xs $ ↭-concat⁺ xss↭) ⟩
    ( Any-resp-↭ (++⁺ˡ ys $ ++⁺ˡ xs $ ↭-concat⁺ xss↭) -- Any P (ys++xs++concat yss)
    ∘ Any-resp-↭ (shifts xs ys)                       -- Any P (ys++xs++concat xss)
    ∘ L.Any.++⁺ʳ xs                                   -- Any P (xs++ys++concat xss)
    ∘ L.Any.++⁺ˡ                                      -- Any P (ys++concat xss)
    ) x∈                                              -- Any P ys
  ≡⟨ Any-resp-↭∘ˡˡshifts {xs = xs}
```

```
      (L.Any.++⁺ʳ xs (L.Any.++⁺ˡ x∈)) (↭-concat⁺ xss↭) ⟩
    ( Any-resp-↭ (shifts xs ys)                       -- Any P (ys++xs++concat xss)
    ∘ Any-resp-↭ (++⁺ˡ xs $ ++⁺ˡ ys $ ↭-concat⁺ xss↭) -- Any P (xs++ys++concat yss)
    ∘ L.Any.++⁺ʳ xs                                   -- Any P (xs++ys++concat xss)
    ∘ L.Any.++⁺ˡ                                      -- Any P (ys++concat xss)
    ) x∈                                              -- Any P ys
  ≡⟨ cong (Any-resp-↭ (shifts xs ys))
      $  Any-resp-↭∘Any-++⁺ˡʳ (++⁺ˡ ys $ ↭-concat⁺ xss↭) (L.Any.++⁺ˡ x∈) ⟩
    ( Any-resp-↭ (shifts xs ys)                -- Any P (ys++ xs ++ concat yss)
    ∘ L.Any.++⁺ʳ xs                            -- Any P (xs++ ys ++ concat yss)
    ∘ Any-resp-↭ (++⁺ˡ ys $ ↭-concat⁺ xss↭) -- Any P (ys++ concat yss)
    ∘ L.Any.++⁺ˡ                               -- Any P (ys++ concat xss)
    ) x∈                                       -- Any P ys
  ≡⟨ cong (Any-resp-↭ (shifts xs ys) ∘ L.Any.++⁺ʳ xs)
      $  Any-resp-↭∘Any-++⁺ˡˡ (↭-concat⁺ xss↭) x∈ ⟩
    ( Any-resp-↭ (shifts xs ys) -- Any P (ys ++ xs ++ concat yss)
    ∘ L.Any.++⁺ʳ xs             -- Any P (xs ++ ys ++ concat yss)
    ∘ L.Any.++⁺ˡ                -- Any P (ys ++ concat xss)
    ) x∈                        -- Any P ys
  ≡⟨ Any-resp-↭∘shiftsʳˡ {ys = xs} x∈ ⟩
    L.Any.++⁺ˡ -- Any P (ys ++ xs ++ concat yss)
    x∈         -- Any P ys
  ∎
... | there (there xs∈) =
  begin
    ( Any-resp-↭ ( L.Perm.↭-trans (shifts xs ys)
                   $ (++⁺ˡ ys $ ++⁺ˡ xs $ ↭-concat⁺ xss↭)
                   )  -- Any P (ys ++ xs ++ concat yss)
    ∘ L.Any.++⁺ʳ xs -- Any P (xs ++ ys ++ concat xss)
    ∘ L.Any.++⁺ʳ ys -- Any P (ys ++ concat xss)
    ∘ L.Any.concat⁺ -- Any P (concat xss)
    ) xs∈           -- Any (Any P) xss
  ≡⟨ Any-resp-↭∘↭-trans _ (shifts xs ys) (++⁺ˡ ys $ ++⁺ˡ xs $ ↭-concat⁺ xss↭) ⟩
    ( Any-resp-↭ (++⁺ˡ ys $ ++⁺ˡ xs $ ↭-concat⁺ xss↭) -- Any P (ys++xs++concat yss)
    ∘ Any-resp-↭ (shifts xs ys)                       -- Any P (ys++xs++concat xss)
    ∘ L.Any.++⁺ʳ xs                                   -- Any P (xs++ys++concat xss)
    ∘ L.Any.++⁺ʳ ys                                   -- Any P (ys++concat xss)
    ∘ L.Any.concat⁺                                   -- Any P (concat xss)
    ) xs∈                                             -- Any (Any P) xss
  ≡⟨ Any-resp-↭∘ˡˡshifts {xs = xs}
      (L.Any.++⁺ʳ xs $ L.Any.++⁺ʳ ys $ L.Any.concat⁺ xs∈) (↭-concat⁺ xss↭) ⟩
    ( Any-resp-↭ (shifts xs ys)                       -- Any P (ys++xs++concat xss)
    ∘ Any-resp-↭ (++⁺ˡ xs $ ++⁺ˡ ys $ ↭-concat⁺ xss↭) -- Any P (xs++ys++concat yss)
    ∘ L.Any.++⁺ʳ xs                                   -- Any P (xs++ys++concat xss)
```

```
  ∘ L.Any.++⁺ʳ ys                            -- Any P (ys++concat xss)
  ∘ L.Any.concat⁺                            -- Any P (concat xss)
  ) xs∈                                      -- Any (Any P) xss
≡⟨ cong (Any-resp-↭ (shifts xs ys))
     $  Any-resp-↭∘Any-++⁺ˡʳ (++⁺ˡ ys $ ↭-concat⁺ xss↭) _ ⟩
  ( Any-resp-↭ (shifts xs ys)            -- Any P (ys ++ xs ++ concat xss)
  ∘ L.Any.++⁺ʳ xs                        -- Any P (xs ++ ys ++ concat yss)
  ∘ Any-resp-↭ (++⁺ˡ ys $ ↭-concat⁺ xss↭) -- Any P (ys ++ concat yss)
  ∘ L.Any.++⁺ʳ ys                        -- Any P (ys ++ concat xss)
  ∘ L.Any.concat⁺                        -- Any P (concat xss)
  ) xs∈                                  -- Any (Any P) xss
≡⟨ cong (Any-resp-↭ (shifts xs ys) ∘ L.Any.++⁺ʳ xs)
     $  Any-resp-↭∘Any-++⁺ˡʳ (↭-concat⁺ xss↭) _ ⟩
  ( Any-resp-↭ (shifts xs ys)    -- Any P (ys ++ xs ++ concat yss)
  ∘ L.Any.++⁺ʳ xs                -- Any P (xs ++ ys ++ concat yss)
  ∘ L.Any.++⁺ʳ ys                -- Any P (ys ++ concat yss)
  ∘ Any-resp-↭ (↭-concat⁺ xss↭) -- Any P (concat yss)
  ∘ L.Any.concat⁺                -- Any P (concat xss)
  ) xs∈                          -- Any (Any P) xss
≡⟨ Any-resp-↭∘shiftsʳʳ {xs = xs}
     (Any-resp-↭ (↭-concat⁺ xss↭) $ L.Any.concat⁺ xs∈) ⟩
  ( L.Any.++⁺ʳ ys                -- Any P (ys ++ xs ++ concat yss)
  ∘ L.Any.++⁺ʳ xs                -- Any P (xs ++ concat yss)
  ∘ Any-resp-↭ (↭-concat⁺ xss↭) -- Any P (concat yss)
  ∘ L.Any.concat⁺                -- Any P (concat xss)
  ) xs∈                          -- Any (Any P) xss
≡⟨ cong (L.Any.++⁺ʳ ys ∘ L.Any.++⁺ʳ xs) $ Any-resp-↭∘Any-concat⁺ xss↭ xs∈ ⟩
  ( L.Any.++⁺ʳ ys    -- Any P (ys ++ xs ++ concat yss)
  ∘ L.Any.++⁺ʳ xs    -- Any P (xs ++ concat yss)
  ∘ L.Any.concat⁺    -- Any P (concat yss)
  ∘ Any-resp-↭ xss↭ -- Any (Any P) yss
  ) xs∈              -- Any (Any P) xss
∎
```

This is one of the reasons why we had to resort to `postulate`s at the point where technical details became just too much to handle while at the same time not having any conceptual merits.