Glow Whitepaper

January 2020

https://bit.ly/GlowWhitepaper2020

François-René Rideau
fare@mukn.io

Mutual Knowledge Systems, Inc.

151 Dudley St #3
Cambridge, MA 02140

Abstract

We present *Glow*, a Domain-Specific Language (DSL) to develop *secure* Decentralized Applications (DApps) on the blockchain. Unlike existing languages, *Glow* covers much more than a DApp's "smart contract": the *Glow* compiler also generates crucially matching client code, and a logical model of your DApp so you can prove it correct. Formal methods are not an afterthought in *Glow*, they are built into the language and its implementation. Furthermore, *Glow*'s logic is designed to deal with the inherently adversarial aspect of DApps, that existing formal tools blatantly overlook. Underlying *Glow* is an architecture that in the future will make it possible to prove correctness of *Glow* itself, and can later grow into a complete DApp Operating System. *Mutual Knowledge Systems, Inc.* is developing *Glow* as an Open Source platform, with an ambitious Business Model to become the go-to company for all blockchain developments.

NB: This document is long and will likely be broken down into shorter documents. However, you only need to read the parts that are relevant to you, depending on your interests.

Businessmen: Start with the Motivation section (8 pages), and, if interested, continue with the Business Model section (8 pages). If you are technical, you can also read the Language Overview section (4 pages) for some high-level insight.

Programmers: The Motivation section can give some background but you can skim it and skip the Business Model section. Focus on the Language Overview and Programming Model sections. If our implementation is of interest to you, don't miss the Architecture section, and consult the Appendices for some advanced techniques we use.

Logicians: You may skim or skip the Motivation, Business Model and Language Overview sections. Be attentive to the Programming Model section, then focus on the Logical Model section. For deeper technical insight, read the Appendices. If you wonder about how we keep large proofs manageable, see the Architecture section.

Table of Contents

Motivation: The Economic Significance of *Glow*

More than a Smart Contract

Decentralized Applications promise to enable secure financial transactions in the largely trustless and recourseless world of blockchain. The potential for economic growth is tremendous. But there is a large gap between this promise and current reality: existing technology makes it too difficult to write *secure* DApps, which stifles the growth of the market. In this paper we will present a solution to this problem, by introducing a new language, *Glow*, for developers to create DApps that remain secure in an adversarial environment.

A Decentralized Application (DApp or dApp) is an interaction between two or more parties, transacting digital assets on decentralized ledgers each managed by a validation network. These ledgers, and by extension these validation networks, are typically called "blockchain" after the first and most popular data structure used for the purpose. Blockchains directly support simple DApps such as one-way payments in the blockchain's "native" token (as with Bitcoin or Ethereum) and sometimes in exchanges between multiple native or user-defined tokens (as with Algorand). More elaborate applications require custom code, called "smart contract" (or "covenant" or "script"), that runs on a special-purpose secure virtual machine (VM) on each node of the blockchain. The cost of redundantly running a "smart contract" on every node is literally millions of times greater than running code on a private cloud computer, and so "smart contracts" are typically reserved for high-value code that directly controls assets.

Today, most people involved in the DApps industry understand that a single bug in a "smart contract" may cause all the assets at stake to be lost with limited or no recourse. Indeed, at least twice, the most famous smart contract of the day, written and audited by the most reputable specialists, just a few hundred lines short, was found deficient—each time a flaw in the contract code made possible by bad design of the system. With the DAO hack, tens of millions of dollars worth of Ethereum tokens were stolen; they were only retrieved at the cost of a cataclysmic fork that divided the community. With the Parity Wallet bug, hundreds of millions of dollars worth of tokens were forever locked and lost to their owners; this time there was no fork to save the victims; and the community consensus is that there shall be no further fork to save the victims of any future similar issue. There are many other cases besides these higher profile "hacks" which all support the need for a system to develop secure DApps.

However, most people in the DApp industry fail to acknowledge how **a bug in "client" code can be as bad as a bug in "smart contract" code**.

Any DApp includes *client* code: code that runs on a participant's computers, that interacts with other participants as well as with the blockchain. The *servers* it connects to are the nodes of the blockchain and of any additional inter-client communication network used. Participants must use

this “client” code before they may even negotiate, sign, or invoke any existing or new “smart contract”. Note that in the context of a DApp, “client” does not necessarily mean something running on an end-user’s smartphone: A DApp client might well run on the redundant data-centers of a financial institution, where it would be a client to the blockchain, but also a server to further components of a larger financial system. For instance, it could bridge the blockchain to a network of semi-autonomous trading agents. Or indeed, a “client” might run on an end-user’s smartphone, where it might both talk to the blockchain and provide a user interface; even then, the “client” as such might be isolated from the user interface, that should run in separate processes, to reduce the surface for bugs and to contain attacks; furthermore, even end-user clients might rely on a distributed network of trusted servers to provide security, privacy and redundancy as a mediation layer between the user device and untrusted nodes of the blockchain.

Now, the subtlest bug in the “client” code as such, that controls the interaction with the blockchain, could allow a malicious actor to exploit the bug and steal or lock up the participant’s assets. Furthermore, it isn’t enough that all code in all “clients” and all “smart contracts” shall be bug free: if some participant uses a slightly different version of the “client” code that doesn’t exactly match the “smart contract”, then the system is not secure and they may still lose all their assets. Thus, building a safe DApp requires widening our understanding of what DApp is: contrary to widespread belief, **a DApp is much more than a “smart contract”, it also includes matching “clients”—and a reason to believe all this code is correct**.

As DApps grow larger and more complex, the opportunities for errors grow exponentially. To assess as much as possible of the correctness of a DApp, it then becomes important to use formal methods. But in most DApp development platforms, formal methods come as an afterthought at best, when there are any at all. The few serious uses of formal methods only apply to proving the correctness of a “smart contract” with no notion of either the client code or the large runtime libraries used by this client code. Thus these formal tools not only fail to cover the client code—they make it impossible to even express important security properties of a DApp, the properties that rely on there being not just one client, but multiple clients, each run by one of multiple parties.

A further problem with existing DApp development tools is that they require you to write your application many times over, in at least two languages: once as a contract in the contract language (e.g. Solidity), and again, in multiple instances, as client code for each of the participants (e.g. in JavaScript). The problem is compounded when using techniques like “generalized state channels”, where each participant has to not only fulfill their contractual obligations, but also check that the other participants are fulfilling theirs, and take appropriate measures if they did not—the very same logic being replicated multiple times in different ways. With or without state channels, writing multiple related yet distinct variants of the same code in multiple very different languages is difficult and error prone; a small subtle discrepancy, and your users may lose all their assets. Even if you get it right the first time, a small change in any part of the code, as often required to adapt it to a changing world, and you may easily introduce

such a catastrophic discrepancy. As a result, the process of developing code with existing systems is time consuming and incurs large costs and delays in addition to great risks.

To solve all these problems, *Mutual Knowledge Systems, Inc.* is developing *Glow*: a language to develop not just a "smart contract", but an entire "Decentralized Application", which also includes client code and formal proofs. From a single specification in *Glow*, our compiler will generate exactly matching code for both "clients" and "smart contracts". Our runtime also tracks which versions of which code you are using to make sure you are always using the correct, matching and trusted versions of a DApp's client and smart contract. Finally, since correctness is such a high stake concern for a DApp, our language helps you specify a logical model of your DApp, so you can formally verify that it is indeed correct.

The Importance of Being Correct

To be secure, a DApp must behave in all ways that users want (or else users might have their transactions blocked or their assets locked), and none of the ways they do not want (or else users may have their privacy breached or their assets stolen). This is what is generally called the *correctness* of the application. Correctness is an all-important property for a DApp: users of the DApp each have to trust the DApp to keep their assets safe, and must either audit the entire DApp themselves, or trust some expert authority to have properly audited it on their behalf. DApps differ greatly from centralized applications, where the application owners can rely on institutional memory and good practices to trust their own code, while users don't need to trust it as much, and can find legal recourse and sue the application owners if an issue arises. When using a DApp, there is no recourse, no other responsible party to sue, if some users lose their assets due to a bug.

Happily, there exist many formal methods, notably used in the aerospace, biotechnology or micro-electronics industries, to help automatically assess the correctness of computerized applications. However, even those industries do not have a need for correctness as high as the DApp industry: If a bug only happens in circumstances that are extremely unlikely to occur in practice, these other industries can often afford to keep this bug in their current design and fix it in the next revision, while they issue an erratum for users to avoid these rare circumstances, work around them, install counter-measures, or get insured against the small risk. But in a DApp, adversaries who would identify such a bug will actively contrive to make the critical circumstances occur on purpose; they will thus break the DApp, and there will be no defense possible after the DApp is deployed on the Internet. It will be too late for users to update to the next revision that fixes the bug or install a patch: their assets will already be gone. Only the most advanced military applications possibly face such a harsh adversarial environment as DApps—and they can afford to keep their code secret in physically protected facilities. By contrast, DApps perforce must openly publish all the code that matters on a blockchain, where the Enemy can analyze it at leisure and interact with it freely.

Currently there exist tools to formally verify programs written in existing “smart contract languages”—but these tools come short in many ways that cannot conceivably be fixed unless this approach is rejected and the “DApp language” approach is adopted instead:

1. First, these existing tools by design can only verify the “smart contract”, and not the entire DApp. Yet, most code in a DApp is client code running on the participants’ computers, outside the “smart contract” running on the blockchain. Moreover, on-chain smart contract code is never called during the regular operations of a “state channel”; it is only present to deal with disputes when one party fails to behave as contractually agreed upon. Therefore, these tools only cover a diminishing fraction of the DApp code that has to be correct, and a fraction that isn’t used in daily practice for many DApps.
2. Second, the existing tools are simple variants of previous tools used to verify programs outside the Blockchain; these tools therefore all assume a logical framework where all components of a computation cooperate towards a common result—whereas the very purpose of a “smart contract” is that this assumption cannot be made. Thus, can only express and prove *liveness* properties of a smart contract: properties supposed to hold when everyone cooperates. They cannot express, much less prove, the crucial *safety* properties of a smart contract, not to mention an entire DApp: properties that ensure good behavior of the DApp even when some participants fail to cooperate.

A DApp is an interaction between multiple parties that do not fully trust each other. Any DApp requires at least one step from each party. Since by definition no party controls the others, no one can ever be quite sure that the other parties will complete their part of the interaction. And what if one of the parties stops cooperating? Can they either steal or lock up the assets of the other parties? That's where a “smart contract” is used: it is a device to keep all the parties honest, to prevent or repair any cheating or default by any party during the interaction—it serves as referee to the interaction. If the two or more parties trusted each other, there would be no need for a smart contract, or a blockchain at all: they could transact directly and privately, without any public infrastructure, any intermediary, any spy or adversary. The *raison d’être* of a smart contract, and of a DApp beyond it, is that the participants do not blindly trust each other. They use this smart contract to create mutual accountability that wouldn’t exist without it. Instead of having to trust each other, they trust a decentralized network: any particular participant may misbehave, but the system remains honest as long as a super-majority of participants behave honestly—two thirds of them weighed by how much skin they have in the game. It has thus been said that Bitcoin, and Blockchain in general, is the technology you use to trade with your enemies—you don’t need it if trading between trusted friends.

Until you understand the purpose of a DApp, it is impossible to adequately conceive the correctness properties that matter for it, much less to express or prove them formally. Once you understand this purpose, it is not just possible but easy to think of these correctness properties, of the logic in which you want to express those properties, of the methods to prove them automatically. See the section on our Logical Model for technical details.

Taming Complexity with a Domain-Specific Language

With *Glow*, we have created a *Safe DApp Language*—a Domain-Specific Language (DSL) that will make it considerably easier to write *safe* DApps than is currently possible. Indeed, preliminary experiments show that it took one-tenth the number of lines of code (50 vs 600), one-tenth the mental effort, and one-tenth the bug-chasing frustration, to write our example DApp using *Glow* than it took us to write the same DApp using the supported combination of Solidity and JavaScript. The savings are even greater if you consider the tens of thousands of lines of framework you will have to write that are not DApp-specific: a *safe* DApp will include many general-purpose concepts and components that current platforms fail to address, and thus using other platforms will require you to build your own framework. By using our platform, you will be able to reuse the built-in framework that we developed for all our users.

To write a DApp, you need expertise in the subject matter of your DApp, be it Financial Products, Derivatives Trading, Supply Chain Tracking, Insurance Pricing, etc. But using current technology, you would additionally need expertise in seven subfields of both Economics and Computer Science to build a safe and effective DApp:

1. Cybersecurity, to ensure your system isn't amenable to hacking by dedicated criminals bent on stealing your assets or on blackmailing you with locking up your assets.
2. Economic Mechanism Design, or more specifically Crypto-economics, to ensure your DApp offers proper economic incentives for each participant at each stage of its execution.
3. Economic Modelling, necessary to know when to accept which transactions at what price. Even when transactions are under direct control of a human user, correctly choosing the price of gas requires some amount of economic modelling, especially so in an adversarial environment (see in Appendix section Semi-Automatically Estimating Proper Collateral).
4. Cryptography, to ensure the encryption protocols used properly restrict which participants are authorized to issue or read each relevant piece of information published on the network.
5. Distributed Systems, to ensure your system keeps running 24/7 under tight deadlines despite the presence of hardware or network failures—be they accidental or caused by malicious behavior.
6. Systems Programming, to avoid all the subtle mistakes you can easily make when confronting the complexity of modern operating systems, databases and network stacks.
7. Last but not least, advanced knowledge of the blockchain platform chosen to implement and run your DApps. With most development platforms your DApp will only work on the chosen Blockchain, and if conditions change and you decide to move to another Blockchain, you will have to restart all your code from scratch and deal with all the above risks again.

Glow captures, combines and encodes the relevant elements of the above skill sets in a programming language. It can thereby vastly reduce the complexity involved in developing DApps compared to existing alternatives. As direct benefits *Glow* will:

a. Provide previously unreachable safety and cut down on catastrophic failure risks.
b. Provide previously impossible portability.
c. Cut down the time from idea to market.
d. Greatly lower the barrier to entry to developing DApps in terms of skills.
e. Greatly lower the barrier to entry to developing DApps in terms of capital.
f. Save substantial costs in human resources due to having to hire experts.
g. Save substantial costs in managing these experts to effectively work together.
h. Eliminate risks from having to manage a complex project (see below).
i. Make substantial savings in quality processes, reviews, and audits.

The costs of software development increase exponentially with software complexity, rather than linearly. This exponential increase is not just reflected in required skill sets, personnel expenses, and development time, but is especially felt in risks of failure and costs to avoid them:

a. Most software projects fail, and the more complex the project, the higher the risk that something, somewhere will go wrong: from setting goals, to identifying requirements, to planning resources, to hiring suitably skilled labor, to dividing labor into teams, to aligning incentives and streaming communication inside and between teams, to choosing the correct technologies, to solving technical issues, to tracking progress along many dimensions, to achieving and retaining sufficiently high quality as software grows, to delivering on time, to keeping the software running in production, to maintaining the software as issues crop up—and being able to adapt and evolve through constant changes at each and every of these levels.
b. Most organizations cannot financially, technically and culturally afford the sophistication necessary to recruit and manage many top experts in very different specialties, who think in very different ways and speak past each other in subtly different jargons, and to get them to work together effectively, as second fiddles who support the work of the actual application developers.
c. The quality processes, internal reviews and external audits required to ensure the correctness of the software will rise sharply in cost as the software grows in size and complexity: the number of combined edge cases not to miss increases exponentially, and soon each case by itself requires more complexity than any single expert can fit in his head at the same time, or at all. Quality soon depends on fragile processes that no one fully comprehends, that will unexpectedly break following a routine change.

In the case of DApps, where the stakes are high and the risks are catastrophic, elimination of complexity is crucial for the successful implementation of your DApps. Every DApp has some *intrinsic complexity*, that stems from its goals and requirements, that cannot be avoided. This intrinsic complexity determines a floor to the costs and risks involved in building that DApp. Any *incidental complexity* added to this intrinsic complexity compounds exponential costs and risks

on top of that floor, and must be most carefully avoided. *Glow* enables you to drastically cut on that incidental complexity.

Using *Glow*, companies can focus more time and resources on quickly implementing their business solutions in a safe and secure environment. They will still want to consult an infrastructure expert for a DApp *audit*, and to have experts on call to deploy a DApp on the Internet. However, the cost for the audit and deployment will be substantially less when using *Glow* than when using other platforms, and the resulting DApps will be much more secure. When such expert services are needed, *Mutual Knowledge Systems, Inc.*, that develops *Glow*, will be available to provide them.

Commoditizing Blockchains via Portability

DApps written in *Glow* are *portable* to all blockchains: they will directly run on every blockchain that supports smart contracts—and, indirectly, even on those blockchains that don't, using suitable *bridges*: for instance, the Bitcoin-Ethereum relay network, or the Dogethereum bridge, that allow one blockchain to "read" the contents of the other and write contracts about them, albeit with some high latency to full confirmation. New blockchains can also be supported by *Glow,* with the one-time development of a new backend to our compiler and a new variant of our runtime environment. By contrast existing tools require each DApp to be rewritten for each blockchain that it will support. Using *Glow* is thus a much more economically efficient approach, as well as safer.

This *portability* also significantly decreases another important *economic risk* associated to building DApps: being locked in to a chosen blockchain for DApp development. Currently, companies planning to build DApps have to guess which blockchain will be the most profitable to support tomorrow—as opposed to being the most popular with today's users or developers, which is obvious but not quite as helpful. If they make the wrong choice, they will have invested a lot of resources in supporting the wrong blockchain. *Glow* solves this problem. Using Glow, you can develop your DApp once and deploy it simultaneously on all blockchains that make sense to support: this will include whichever will be the most profitable blockchain that you don't have to guess anymore, but also all the other ones that may be profitable, too.

By vastly reducing economic risk, *Glow* will thereby *commoditize* the existing blockchains and their tokens. Users will be able to enjoy whichever blockchain technology brings them the most value at any moment without being tied down to any specific one. They will be able to only hold their digital wealth in whichever cryptocurrency, fiat currency, digital asset, financial instrument or basket thereof they believe will best serve their financial goals. They will never have to hold more liquidity in any kind of token than necessary for their current transactions. Through the use of liquidity pools, they will be able to acquire any tokens they need just before they need them, without having to hold them. Insurance contracts and futures contracts can even help them manage the risk of these exchange operations. As the market for digital assets becomes liquid, tokens will become commodities. Volatility will decrease, and their price will be driven up or

down based on whatever objective technical advantage they do or do not possess with respect to either facilitating economic transactions as utility tokens, or holding economic value as financial assets.

Glow has the potential to change the economic landscape of decentralized digital assets by enabling modern financial practices on decentralized markets.

Unlocking the Potential of DApps

Glow will drastically lower the barrier to entry to writing, deploying, maintaining, trusting, and using Decentralized Applications (DApps). Not only will companies save orders of magnitude in costs to build and use DApps: they will be able to build and use DApps that are not affordable, not doable, sometimes not even imaginable, using current technology. No one can afford to trust current DApps: how can you audit and trust tens of millions of lines of code written by yourself, much less by other people, in multiple programming languages? Would you put any significant assets under the control of any such DApp, with no recourse whatsoever if an adversary finds any bug however subtle? The entire DApp ecosystem is currently stunted by how unsafe the current tools are. By reducing the complexity of DApps by orders of magnitude, *Glow* will make it possible to build, audit and trust DApps that would be out of reach using previous technology. We will thereby *unlock the potential of DApps*, to offer services that are just inconceivable with today's technology.

The volume of assets confided to the care of DApps is set to grow exponentially, as *Glow* makes it easier and safer for everyone to build DApps that other people can afford to trust. Come join *Mutual Knowledge Systems, Inc.* in growing and using this DApp language—and beyond the language, creating an entire ecosystem. We are actively raising funds, seeking partners, hiring developers and courting users, to build what is poised to become a very large industry.

Business Model

Multiple Revenue Streams

Our company Mutual Knowledge Systems, Inc. will be building the *Glow* language, growing its ecosystem, and servicing its community. We often abbreviate the company name as *MuKn*, which we pronounce as "Moon" (or "Myun"), since the letters stand for MU-tual KN-owledge.

As a company, *MuKn* is following a strategy to collect revenues from multiple streams, wherein developing the language *Glow* is only the first step:

1. We will make and keep our language so far ahead of competing languages, and publish it as open source software, so that anyone writing a DApp will *of course* use Glow. This

step will bring no direct revenue to *MuKn*, but will give it a lot of *traction*.

2. We will build a few “Killer DApps” in joint ventures with other companies that will operate them, to demonstrate the advantage of using our technology, and reap the low hanging fruits of what this technology enables. These DApps may include Non-Custodial Decentralized Exchanges, Cross-Cryptocurrency Payment System, Supply Chain Asset Tracking, Insurance contracts, etc.

3. We will sell services for support, maintenance, development, guarantee, audit, training, certification, etc., in a traditional Open Source model. However, if the steps above are successful and *everyone* in cryptofinance uses our language, then we stand to sell a lot of high value services to a large number of financial institutions all around the world.

4. We will sell proprietary extensions and value-added services that enable our corporate customers to interface with other proprietary systems: databases, ERPs, trading platforms, analytics platforms, data feeds, languages and platforms used in finance, etc. This revenue stream will extend the previous one as we build more complete solutions for our customers.

5. As we control the default settings for the official compiler that everyone uses, we can sell access to our users, and get a cut of fees for various services: transaction processing, scaling, hosting, insurance, etc. Then again, doing it in ways too onerous to users would lead to our losing traction to a “free software fork” that changes those default. Thus we will have a monetizable influence on the community, that we can keep only by using it moderately and not abusing it. (See for instance how Google sold the naming of the Android operating system releases to be brands of sweets and candies.)

6. Beyond providing just one default scaling solution, one default hosting solution, one default insurance solution, etc., we can provide an entire marketplace for all blockchain services, whether B2B or even B2C. As everyone uses our development platform that in turn makes it easy to buy and sell services on the *MuKn* MarketPlace, we can become the premier marketplace for everything Blockchain, and collect a small fee on everything.

Open Source Community

All experts in the domain of Blockchain understand the utmost importance for all basic infrastructure to be *Open Source*. Software being Open Source means that anyone should have the right to copy it, use it, modify it, and redistribute it, without having to get permission from any centralized entity, or pay license fees to it. If that were not the case, then the centralized entity could hold the users’ assets hostage—the blockchain software wouldn’t really be a Decentralized Application (DApp). Moreover, users of proprietary software (the opposite of Open Source) are not legally able to freely audit the software and fix bugs in it, whereas illegal adversaries would not be stopped in examining the software and creating attacks. Proprietary

software is therefore inherently insecure in the domain of Blockchain. The community of blockchain users at large is well aware of these constraints and would simply not adopt Proprietary Software as the basis for DApps.

We at *MuKn* understand and embrace these basic principles, and will make our Blockchain Infrastructure available as Open Source software. Actually, by making our basic software offering Open Source, we intend to capture a large part of the user *mindshare*, as everyone in the blockchain community in turn embraces our software without reservations, because it is technically superior as well as legally unencumbered. We will actively cultivate a community of Open Source users and cater to their needs. This community will provide us with many benefits:

1. The Open Source Community will give us a short feedback loop to improve our product-market fit. Comments and requests from community members help us identify many of the needs of the markets, and the best ways to improve our product to respond to demand. Open Source can help us keep our OODA loop tighter than our competitors who fail to cultivate such a community.

2. Open Source Community members may help identify issues with our products and provide fixes, before our software enters in production, keeping our software more secure.

3. Many Open Source users eventually become commercial users, and purchase our services, as they realize we are better equipped than they are to provide efficient and affordable services around our platform, so they can instead focus on their value added.

4. As our language and platform become popular, then ubiquitous, large institutions that need to develop blockchain DApps will find that the most used language is ours, that the easiest way to hire a DApp expert is to hire one who knows how to use our language, that our language is the natural choice to cover their DApp needs.

5. As our development is open for everyone to see, it is obvious to everyone that as the main developers of this platform, we are so much better placed than any competitor to provide service around this platform.

Killer DApps

In the early days of our company, as we struggle to build credibility, we will create “Killer DApps” in joint ventures with suitable partners—other companies that share our understanding of the market but have complementary skills, and that are willing to take a calculated risk to beat their competition. These “Killer DApps” will be as many opportunities to showcase the unique capabilities of our platform, while raising and making money with projects that have low hanging

fruits. Not all of these DApps may succeed, but only one would be enough to make our platform hugely popular.

The partners we seek will have a DApp in mind, but may lack the means to securely tackle some difficult problems about those DApps—that we can help with. These partners would develop and deploy the DApps. In particular, they would deal with any regulatory issues with deploying those DApps. Meanwhile, we would build the language primitives that make those DApps possible at all to write in a secure way. These partners would help us find funding, we would bring the missing talent, and we would share the eventual profits.

Potential Killer DApps that we are interested in helping build include Non-Custodial Decentralized Exchanges, Fast Cross-Cryptocurrency Payment Systems, Non-Custodial Decentralized Poker Servers, Decentralized Auction Houses, Decentralized Insurance, Decentralized Futures Market, Decentralized Massively-Multiplayer Online Games with fungible and non-fungible game tokens, Decentralized Advertising Platforms, Decentralized Supply-Chain Asset Tracking, etc. We have contacts with people building such applications, and will be closing deals when our platform is ready for each of them.

Service Company

As our software becomes established, we will sell services to all companies that use our software:

- We will sell subscriptions to companies that want to make sure they always use the latest version of our software with all the latest features and security updates.
- We will sell support contracts to companies that need help to write and deploy DApps in our language.
- We will sell maintenance contracts to companies that want to make sure that our software keeps working for their DApps as their own environment evolves, and that we prioritize fixing the issues that they experience if any.
- We will sell development contracts to companies that need modifications to our software so our language can compile and run their DApps in their software environment.
- We will sell guarantees to companies that need our software to pass stringent requirements before they may use it.
- We will sell audit contracts to companies that want to make sure they use our software correctly, that their DApps do not have technical issues, that they use an appropriate configuration to deploy it, etc.
- We will sell training to companies and professionals who want to follow the best practices in using our software to write correct DApps.
- We will sell certification to professionals who have demonstrated their understanding of our software, how to use it, how to extend it.
- We may sell active hosting of DApps to users and companies that do not want to host it themselves. However, we might do that with hosting partners, or as a middle-man to a hosting service. See relevant sections below.

Many companies have succeeded by publishing their software as Open Source then selling up to hundreds of millions of dollars of such services every year. Successful Open Source service companies include RedHat (now part of IBM), HashiCorp, Canonical (makers of Ubuntu), Cloudera, MuleSoft, Automattic, Elastic, MongoDB, Acquia, and many more.

As our language gains in popularity and *everyone* who writes DApps starts using it, we may sell a lot of high value services to a large number of commercial corporations and financial institutions all around the world.

System Extensions

While it is essential to our success that our basic software offering shall be completely Open Source, we may build and sell extensions that are Proprietary beyond this basic offering.

Many of our customers may want to interface our platform with various other platforms:

- A proprietary database, such as Oracle, IBM DB2, Microsoft SQL, Sybase, Kdb+, etc.
- A proprietary ERP, such as SAP, NetSuite, Syspro, etc.
- A proprietary trading platform, such as Bloomberg, Interactive Brokers, etc.
- A proprietary analytics platform, such as Tableau, Looker, etc.
- A proprietary data feed, such as Nasdaq, Knoema, Intrinio, etc.
- A proprietary language used in finance such as APL, J, K, Q, etc.
- A proprietary interface to KYC services, geolocation services, etc.
- Any proprietary platform used on some market.

We will sell proprietary extensions to bridge our software with all these other proprietary platforms. We will also sell services to help our customers integrate our software with their software on top of these platforms that they depend on. Eventually, our proprietary offering will enable us to build more complete solutions for our customers, and strengthen the value we create for them.

As financial institutions start using our software like everyone else who uses Blockchain, this revenue stream will extend the previous one and may surpass it.

Access to Users and Defaults

As our language is used everywhere, we will become a source that users generally trust. Most users will want to run the unmodified original version of our software, straight from the source—a version that was specifically audited and signed by us. They will not give as much trust to modified copies from less trustworthy third parties, that would require an additional audit. This means that most users will be using our language in the default configuration that we provide.

Most users will probably use the default settings included in our configuration. These settings will notably include:

- Which tokens those DApps will use, on which Blockchain.
- Which exchanges to use to get suitable tokens.
- Which scaling services (side-chain, etc.) those DApps will rely on for lower latency or higher throughput.
- Which hosting provider will run those DApps for redundancy against DDoS attacks.
- Which VPNs or mix networks will be used to anonymize transactions sent and avoid DDoS attacks.
- Which pre-approved insurance contracts to purchase when using known DApps.
- Which hardware wallet to buy to integrate with our DApps.

To a point, we may then sell access to our users to various service providers, so they get placed higher in the choice we offer to users, in exchange for a share in the revenues they make this way. There is a potential for a lot of revenues this way, but there is also a potential for a big loss in social capital if the interests of the public are betrayed at any moment in this process.

The situation is akin to Google placing ads with search results: sponsored ads must be well distinguished from regular search results and must not shadow them. The regular search results must be and remain whatever is most relevant to the expectations of the users. Users must not be bullied into making choices that are not in their best interest. They must be shown potentially better choices that they might not otherwise be aware of. Instead of selling access to our users wholesale with a single first choice, we may sometimes want to *randomize* which vendor is shown first, with a distribution designed to avoid tipping the balance between vendors. As an aside, Google also presumably makes money selling the names of Android operating system releases to corporations owning brands of sweets and candies, which is alright because it also doesn't interfere with the choices users make while using their Android devices. We could similarly sell unrelated branding space in our software.

We at *Mutual Knowledge Systems, Inc.*, will consider using our influence being the main source for our software, to sell access to our users to companies offering services, especially services users may choose *by default* because they are the first choice that appears. But we will be especially wary to always be steadfast in defending the interests of our users as we do so. Actually, we will have to find ways to publicly precommit very clearly to always being completely transparent as to how we make recommendations to the public, and to always making those recommendations based on the best reasonable understanding of each user's interests, while respecting every user's privacy. To increase trust in our users, we may use a variant of the slogan "don't be evil" as a canary so our users know that they will be warned when to stop trusting us—in the same way that other service companies publish a "warrant canary" to warn the public about their having received a warrant with a gag order (which we may also do if we host DApps).

Should we in any way betray the trust of our users, we would lose our precious reputation, and with it, substantial revenues. We are fully aware of it, and we will always prefer not sell some services at all than do it in a way that is less than perfectly reputable.

The Marketplace for All Blockchain Services

As the *Glow* language gives access to a growing set of services, there will come a point where there isn't just a small list of services each with a small drop-down list of providers. Instead, there will be a very large set of services, each with many more or less differentiated providers. Then, to satisfy our users, whether developers or end-users, whether companies or individuals, we will have to provide a marketplace with a search engine with a user-driven rating engine—again very much like the Android Play Store, iOS App Store, the Windows App Store, etc., or like Amazon or Alibaba, just specialized for blockchain DApps and DApp related services.

At that point in the growth of the *Glow* ecosystem, the *MuKn* Marketplace may become the world's first broker of blockchain services, both B2B and B2C. As everyone who uses Blockchain uses us, and as everyone uses Blockchain, we stand a chance to become a universal middle-man collecting a small fee on *everything*.

This last part of our business plan is quite remote. Indeed, each part of our business plan depends on the previous parts to have been well executed, and our staying ahead of our competition as we do. This is admittedly a long shot, and we will need help to reach those goals: not just capital to raise, but also key employees to recruit. But we do have a long-term vision that makes our company an admittedly high risk but potentially extremely high reward endeavor. We seek your help in making this vision a reality.

An adaptive plan

We maintain a short-term plan (next 6 months) and a longer-term plan (up to 5 years) for the development of the business. These plans obviously change often to adapt to a changing situation: business progress and available resources, variations in prices of our inputs and outputs, potential market demands and actual customer requests, happy and unhappy accidents, business opportunities that open or close, internal and external deadlines, etc. As per the military saying, "plans are worthless, but planning is everything." Thus, it doesn't make sense to include a quickly obsolete snapshot of our plan in this whitepaper. Still, we will gladly share a summary of our plan, as well as relevant details, with our actual and potential partners and investors, etc. In this last subsection, we will only explain the way we think about our plan, without giving details.

The purpose of our plan is to identify how to maximize the net present value of the company, which includes a speculative evaluation of all future discounted earnings. We focus on the low

hanging fruits to increase the value of each of our revenue streams. Each of our revenue streams will bring its own concerns, each with its own time scale:

1. Increase traction and user engagement of our language in an Open Source community. Figure out the features most needed to onboard new developers. This is our immediate concern, to bootstrap our business. It will be the most important thing over the initial few months, and remain important throughout. With a few hundreds of thousands of dollars, we can complete a minimal viable product and start building a community.

2. Determine which DApps will generate the most revenue as early as possible. We are already actively seeking partners with whom to build “Killer DApps”, analyzing their needs in terms of language features, and prioritizing our development so we can maximize our value through time as these DApps are built. This will become a major concern in a few months and for the first two years. We are looking for other young blockchain projects for us to join as technical partners. We may enter joint ventures but will be mindful to avoid conflicts of interests with future customers of our services.

3. Maximize actual customer revenue for our Service Company. The simplest DApps cost tens of thousands of dollars to build over several months, with costs easily doubled or tripled to afford an audit. Meanwhile, large software service companies charge millions of dollars for blockchain development contracts from large corporations and financial institutions. We will start with small contracts. But with *Glow*’s ability to develop safer DApps at a fraction of the cost, we will not only disrupt the existing market, we will grow it by lowering the barriers to entry. Capturing this market and staying on top of it will be our main priority as soon as our product is ready, starting in our first or second year.

4. Strategically extend our ecosystem with proprietary products and services that interface with third-party systems or ones we build, in a way that maximizes the value we add to our customers. As we grow successful after a few years, widening our offering will become an all-important question to keep growing our revenues as previous technology gets commoditized and as we seek bigger contracts with higher margins.

5. Monetize the defaults in our systems in a way respectful of our corporate customers and partners and of our open source users. Preserving the goodwill capital we have built with them is important to protect our long-term gains. Done right, we anticipate this could become our most important revenue stream within ten years, as blockchain increasingly disrupts finance and usage volumes ramp up.

6. Maintain a strong reputation and build lasting relationships with other players in the field, big and small, so we are in a good position to start our universal market for blockchain services. Over the years, we can turn our company into an institution of its own; but this is a long game, and we have to do things right at every step. This early in the game, our

concern is then to become the kind of company that succeeds this way, rather than make overly precise plans that will be meaningless before the next action item arises.

7. Be open to the many opportunities for our business to evolve, create value in new ways, capture existing markets or build new ones, etc. Our flagship technology today will not be exciting anymore in a few years, but we will keep building or adopting new exciting technologies on top of it or beside it. In the end, what makes the technologies exciting is how they can create value for people, and the proof of the pudding will always be in building a successful business around them.

Language Overview

General Design of *Glow*

This Language Overview section discusses the Design of *Glow*—an explanation of principles that guide *why* we made specific decisions in designing our Domain-Specific Language (DSL), as opposed to a detailed description of the result, which comes in the "Programming Model" and "Logical Model" sections.

The initial *goal* of *Glow* is **to enable the expression of "DApps"—financial interactions between multiple parties mediated by blockchains**. In a DApp, multiple parties do not fully trust each other, and use a blockchain "smart contract" to keep each other honest: the "smart contract" will verify that every party is following the protocol, and otherwise punish the offenders. Because the financial stakes are significant, it is very important that every party may audit and trust the DApp code, and the ultimate *constraint* on *Glow* is that it should **maximally facilitate and minimally hamper the audit and trust of DApp code by all parties**. From the above goal and constraint, we use our Programming Language (PL) expertise to derive the design of *Glow*.

The goal implies that *Glow* will primarily focus on *multi-party interactions* and on how a blockchain "smart contract" can be used to keep the parties honest. Any feature that doesn't serve this goal can be rejected or at least deferred until later. For instance, implementing a User Interface (UI) that runs on the web, on a computer's graphical console, or on a mobile phone, may be a fine and important thing to do; but for the sake of keeping *Glow* simple and its programs safe, it is better to do this UI work *outside Glow*, and "just" offer some *Foreign Function Interface* (FFI) between *Glow* and the outside language in which the UI is written.

As for the ultimate constraint on *Glow*, it is especially extreme among programming languages: **Users must be able to trust the code after they audit it** (or have experts they trust audit it). As a consequence, the syntax and semantics of the language shall be as *familiar* and as *well-defined* as possible, so the auditors may understand very precisely the meaning of a program and reason about it. Furthermore, there shall be a well-defined logical model for programs written in *Glow*; that model shall enable automated formal reasoning, and shall

precisely match the auditor's intuitions above, as well as closely represent the kind of computations that the users care about. These are very hard constraints that cannot be affordably retrofitted after the fact into a design that was conceived without them in mind.

Features of *Glow*

Glow was designed around the following features, all of them chosen under this guiding principle: ***Glow* makes it easy to write *and audit* financial interactions between multiple parties**:

Domain Specific: The *Glow* language deals purely with the interaction between the participants with each other and the blockchain. Thanks to this restriction, programs remain simple, and it is possible to prove them correct. On the flip side, *Glow* notably doesn't deal with user interface, and leaves that part to a separate program with which it communicates in a restricted way. By offering an abstraction layer between the user interface and the blockchain, isolating one from each other, it keeps the overall application secure. The front-end can be written by regular programmers without their bugs being more (or less) critical than in another application. Thus *Glow* can lower the barrier to entry to writing DApps, without sacrificing security.

Pure Typed Functional Programming Core: The core of the language was chosen so there is the most direct correspondence possible between our basic computational fragment and formal logic. This makes it as easy as possible to *reason* about program, whether by a human or a machine.

Partial Functions: Our language supports, as the only allowed local side effect, partiality of functions, whether from failure or non-termination of a computation. Partiality enables developers to express arbitrary computations, but also validation (or rejection) of computations that use parameters provided by untrusted other parties. It is thus possible to easily focus on the "honest case" where everyone participates as agreed upon, and yet know that the language will ensure the "dishonest cases" are properly handled.

Multi-Party Computation: Our language explicitly supports a notion of multiple participants, who do not trust one another. Computations can be annotated with the participant who is computing, and their data elements remain private unless and until explicitly published by the participant who knows the data.

Consensual Asset Control: As a special trusted but decentralized participant, the consensus controls assets. The "smart contracts" that run in this consensus will verify computations, and if they are valid, transfer assets.

Asynchronous communication: The multiple participants exchange messages, and also send messages to the consensus. Communications are asynchronous over a planetary network, and there is no way to guarantee "simultaneity" of messages sent by multiple participants.

Transactions: Messages sent to a participant or to the consensus can be accepted or rejected depending on whether they are valid in the context of the current interaction. Transactions can be composed or nested. A given transaction either wholly succeeds or fails. More advanced transactions are only atomic “up to economic compensation”: the transaction may fail, but honest users who follow the protocol will be made whole, modulo some economic assumption that they agreed to beforehand. For instance, in some “atomic swap” protocols, an honest participant might complete their part of the deal, then notice that the other participant didn’t complete their part; as reparations, the honest participant may thus receive damage payment from the bond that the dishonest participant forfeited.

Logical Reasoning: Our language directly supports specifying and automatically proving logical theorems about programs. Our compiler extracts from every program a precise logical model of the program’s behavior. For every logical assertion provided by the user (or automatically included by the system), the compiler also produces a logical constraint. The proofs for all the specified requirements are discharged by an automated theorem prover such as Z3.

Game-Theoretic Safety: Unlike other languages and formal verification systems, we do not assume that all participants cooperate towards the protocol (or else, no smart contract and no blockchain consensus would be needed). Instead our language is unique in allowing developers to express and prove their programs safe for each user who follows the protocol, whatever the other users may do that does or doesn’t follow the protocol. We prove rather than assume that participants’ interests are aligned. For that purpose, the logic we use includes epistemic, temporal and economic aspects.

Layered Architecture: *Glow* is layered into many *levels* of language, each with its own set of features, and each being defined in terms of the lower levels of languages. This enables developers to specify their DApp behavior at the level of abstraction that matters, without extra complexity from dealing with irrelevant lower or higher abstractions. But this also enables us to organize our own compiler infrastructure into much smaller and simpler parts than rival projects. Our compilers will be easier to audit, and, in the future, easier to formally verify with an automated proof of correctness.

Glow as a Dialect of JavaScript

To satisfy the constraint that the language syntax and semantics of *Glow* shall be familiar and well-defined, we have chosen to make it a *dialect* of the popular language JavaScript (standardized as ECMAScript). More precisely, Glow is an *extension* of a *subset* of JavaScript: first we pick a subset of Javascript that makes it easier to reason about programs, and then we extend that subset with features required to express DApps.

JavaScript is a complex language with many corner cases, advanced features, and much historical baggage, which can make it difficult and non-intuitive to reason about JavaScript

programs. To make it easier to reason about programs in Glow, we only keep the most basic features of JavaScript, and restrict programs to be in a very strict subset of it: pure functional programs that can be statically typed with a simple type system. Where Javascript itself falls short in this regard, we adopt the same amendments to its syntax and semantics as previously made by Facebook's ReasonML. Thus, our language is based on a familiar and well-defined core.

Now, to write a DApp, programmers must be able to express transfer of digital assets as well as communication of information, between the multiple parties and between them and a blockchain smart contract. For that we will extend our DSL beyond this core language, to include communication primitives, as well as annotations regarding who knows which private data and executes which part of a computation. We also add primitives to specify relevant logical properties of the program that help with the audit.

Throughout all this subsetting and extending, we stick to a *Fundamental Principle* of Programming Language Dialects: ***The syntax of some fragment of the dialect will match the syntax of the base language if and only if the semantics of the fragment also matches in the two languages.***

This Fundamental Principle is what makes a dialect familiar and helpful: programs always behave as expected from experience. Dialects that ignore this principle are confusing and misleading rather than familiar and helpful: each violation means that many programs will look like they are doing something based on familiar understanding, but will actually be doing something very different based on the actual dialect specification. This is dangerous, and may facilitate malicious actors writing code that lures people into using it while later turning out to do something different from what they thought they had agreed to. There could and probably should be legal implications to deliberately including confusing and misleading features in a programming language—at least one specially meant for financial interactions, if not in other languages.

It is acceptable to violate this Fundamental Principle in some rare occasions, but only in cases that are definitely not misleading, only with a fair amount of warnings in the documentation, and only in the name of some other Principle that somehow makes the violation worth it. For instance, the violation might make it much easier to reason about programs in the dialect, and indeed might consist in correcting a confusing and misleading feature in the base language itself. But even then, there is a price to pay: if the user gets used to the dialect doing the right thing, then when going back from the dialect to the base language, the user may incorrectly rely on his habits from using the dialect assume the base language does the right thing and get bitten badly. This will be especially dangerous when writing code in both languages, e.g. when using JavaScript to build a User Interface (UI) for a DApp written in *Glow*.

If some language design cannot stick to this Fundamental Principle most of the time, then it is better to choose for the language a syntax that is completely different and purposefully distant

from that of the familiar language, to avoid confusion. Maybe the familiar language was the wrong choice as a base from which to make a dialect, and a different language may be chosen, less familiar but more appropriate to the task.

Programming Model

Multi-Party Interactions

Glow extends a pure functional typed subset of JavaScript with primitives for communication between multiple mutually-untrusting parties, each other, and a "consensus" that algorithmically controls assets contributed by the participants.

Computation statements can be annotated to specify that they happen on a given participant's private computer. Any variable defined remains private to that participant until it is published (if it ever is published), and its definition may use other variables private to that participant, but not to other participants. Consider the following sequence of statements:

```
@Alice let nonceA = randomNonce();
@Bob let nonceB = randomNonce();
```

Two random numbers are defined, `nonceA` and `nonceB` that are respectively visible by participants Alice and Bob. Alice cannot see Bob's nonce at this point, and won't be able to see it until Bob publishes it—if he does. Similarly, Bob cannot see Alice's nonce. The compiler will report an error if an expression makes use of a private variable that hadn't been shared with the current participant or published to the consensus between the participants.

The developer can specify that a participant shall publish the contents of a variable with:

```
publish! Alice -> nonceA;
```

At that point, the protocol will mandate that participant Alice shall send a message to the consensus containing the contents of variable `nonceA`, which will be checked for validity with respect to its declared or inferred type.

A `publish!` statement may only appear in the consensus, and not within a participant private code. Indeed, the contract as well as all other participants will be able to consensually see the published variables. A single `publish!` statement may publish multiple variables (separated by commas), and may be accompanied in the same block by a `deposit!` statement whereby the participant also contributes assets to the contract. The contract itself can disburse assets back to participants or send them to third parties with the `withdraw!` statement.

Consider an interaction wherein a Buyer agrees to pay a given amount in exchange for a signature from a Seller. The signature may for instance validate a transaction on the same blockchain or on another blockchain; or it may grant a lease and activate an electronic key for some time to open some completely off-chain lock (that nevertheless has a trusted clock). The uses are many. The source code for this interaction in *Glow* would look as follows:

```
@interaction([Buyer, Seller])
let payForSignature = (digest : Digest, price : Assets) => {
   deposit! Buyer -> price;
   @Seller @verifiably let signature = sign(digest);
   publish! Seller -> signature;
   verify! signature;
   withdraw! Seller <- price; }
```

In that function, the digest of the message to sign is a parameter of the interaction, as is the convened price. The Buyer and the Seller are two participants who have agreed to the terms of this sale, and know what the signature is about—and they now want to conduct this sale without a chance of either party cheating half-way. The Buyer deposits assets onto the contract that amount to the agreed-upon price. Then the Seller signs. Crucially, the signature is consensually verified by everyone in a way that the contract enforces. Finally the money is transferred to the Seller, also in a consensual way that is managed by the contract.

Importantly, *Glow* automatically handles common failure cases. If the Buyer never pays the price, the Seller will never proceed to sign, and after waiting long enough, the Seller's software will notify him of a timeout. Similarly, if the Buyer pays, but the Seller fails to sign, then after waiting long enough, the Seller will time out, the Buyer's software will notify her and invoke the "smart contract" to claim back the assets they had deposited as the agreed upon `price`. If the Seller tries to send an invalid signature, his message is rejected because it fails the verification, and the Seller will eventually timeout on fulfilling his duty. This timeout handling is fully automated. In a DApp written using previous technology, this error handling would be hundreds of lines of manually written code, and the need to do it right would be a major opportunities for bugs and potential loss of assets. Using Glow, all this handling is automatically and safely generated.

Note that the `@verifiably` annotation remembers the formula for the `signature` such that we can `verify!` it later as part of the consensual computation. By only having to specify the formula once, we avoid errors in duplication, and even more importantly, errors in failing to propagate every modification to all the copies simultaneously as the code evolves. Also, in the case of verifying a signature, only the Seller can sign thanks to his private key, but anyone can verify the signature thanks to his public key. There again, specifying the formula for the computation and deducing the formula for its verification avoids a whole category of bugs wherein the developer would fail to match the formulas for a computation and its verification. In general, the DApp developer must still specify *when* to verify, because the verification is done in

a different context (as part of consensual computations vs as part of private computations), and can only be done after all the relevant data elements have been published, which in some interactions can be many steps after the definition takes place. The common case of verifying immediately can actually be optimized by replacing the three lines of definition, publication and verification by:

```
@Seller @publicly let signature = sign(digest);
```

There is a further complication when an expression irreversibly transforms the output of a function like sign before to publish the result; then for verification, the untransformed output must be separately published for verification.

Finally, the programming model includes two primitives `input(tag, pred)` and `output(tag, value)` that allow for arbitrary interaction between the DApp and the outside world. Input must happen privately on a participant's local computer (e.g. via graphical user interface), or use some supported "oracle" that makes data publicly available to the blockchain. Output must similarly happen privately on a participant's local computer, or output to a common channel seen by all participants. The precise meaning of input and output is specified by software outside the DApp itself, as a "foreign function interface" to code either provided by a system library or by the user, e.g. some JavaScript function and/or Solidity escape hatch. From the point of view of the logical specification of the program, the input returns an arbitrary value satisfying the predicate (assuming one exists), while the output has no observable logical effect.

Proper Collaterals

Consider a slightly more elaborate interaction, a game of rock-papers-scissors: in that common game, each player picks as a hand one of rock, paper or scissors; they show it at the same time; and a winner is determined by comparing the two hands, where rock beats scissors, paper beats rock, scissors beats paper, and two identical hands are a draw. Now, at the level of abstraction of a blockchain, messages are asynchronous: there is no such thing as showing your hands "at the same time". The game will thus be implemented using a commit-reveal protocol: the first player commits to a hand by showing a digest of the hand prefixed by a nonce; the second player shows his hand; and finally the first player reveals hers. The body of the interaction would look like that:

```
@Alice let handA = inputHand();
@Alice let saltA = randomNonce();
@Alice @verifiably let commitment = hash(saltA, handA);
publish! Alice -> commitment;
deposit! Alice -> wagerAmount;
@Bob let handB = inputHand();
publish! Bob -> handB;
deposit! Bob -> wagerAmount;
// NB: the compiler implicitly typechecks handB when published
publish! Alice -> saltA, handA;
```

```
verify! commitment; // NB: handA is implicitly typechecked
switch (computeOutcome(handA, handB)) {
  | A_WINS => withdraw! Alice <- 2*wagerAmount
  | B_WINS => withdraw! Bob <- 2*wagerAmount
  | DRAW => withdraw! Alice <- wagerAmount;
            withdraw! Bob <- wagerAmount }
```

Now, what if in this interaction, Alice finds that she is going to lose, and rationally decides that she has nothing to win and only gas costs to lose by revealing her losing hand? Then she'll stop cooperating and leave Bob hanging until she times out. At that point Bob wins by default, but then *he* will cover the gas costs, in addition to having his time wasted. The proper economic mechanism design to avoid this situation is to modify the protocol so that Alice has an incentive to keep cooperating: with her first message, Alice will also pay some extra `collateralAmount` that will interest her in the future in cooperating to the normal end of the program. If she later fails to cooperate, she will lose this additional amount. Thus she is interested in cooperating until the end, even if she picked a losing hand. Bob on the other hand doesn't have to post a collateral when he reveals his hand, because he has no duty to publish a message in the future—his first message is also his last message in the normal case.

There are many ways that *Glow* can help with correct mechanism design. As we keep improving our language, we will start with the simpler ones and end with the more advanced:

1. *Glow* can automatically insert proof obligations for any interaction protocol to require such collateral amounts for any participant who has to post a message in one of their possible futures.
2. *Glow* can even automatically insert the collaterals themselves. When a small interaction is actually part of a larger interaction, automatic insertion is simpler and cheaper to do once for the entire interaction, rather than expensively modified at each small partial interaction. The contract must keep track of who posted how much collateral.
3. *Glow* computes the minimum amount for the collaterals based on the burden that each participant imposes on the other participants if one stops cooperating. This burden includes both the gas that other participants must pay to exclude the uncooperative one (which itself requires a model of the future price of gas in ether or whichever token is used), and the interests on the capital being immobilized by the other participants while the failing participant times out (which also requires an economic model for interest rates, exchange rates, etc.).
4. For short-term DApps, this computation is better done wholly off-chain, by each participant's clients, as part of a negotiation of the terms of the contract, before any contract is even signed or invoked on the blockchain. If the participants somehow fail to agree on mutually agreeable collateral amounts, they are better off declining to further partake in the interaction.
5. For long-term DApps, the computation may have to be done as part of the contract: the participants agree to a formula or adjustment process during the negotiation phase, and

the formula or process may depend on how gas price varies in the future. This is much harder, but there again, *Glow* will provide solutions for this difficult problem, so that developers and users don't have to build and audit their own.

We discussed the need for proper collateral in a DApp in slightly more details a previous article, "Why Developing for the Blockchain is Hard Part 2: Computing Proper Collateral".

Note that a contract can hold accountable the participants who have already deposited a collateral, but not those who haven't deposited any collateral yet. Thus, in a two-person DApp, the second player can easily hold accountable the first, but the first player can easily be stood up until the second player times out. To avoid these situations, it is recommended that the first player be whichever player has the least or least enforceable reputation, while subsequent players are chosen in order of increasing reputation.

More generally, the need for proper collaterals is a special case of the need for Economic Safety as discussed in the section below on our Logical Model. And we will keep automating away all that can be about the safety of your applications—checking that the developer got it right, and automatically solving the issue whenever possible. Thus, *Glow* will automatically protect DApp developers and users from a host of issues that are extremely costly to address with existing development tools, that are impossible to even express much less verify using existing logic analysis tools, that a lot of developers are unaware of, yet that would be quite costly to users if left unaddressed.

Programmable Implementation Strategies

Glow will support many different *strategies* to implement any given program: not only can a program be implemented on different blockchains, a program can be implemented in "direct style", or using "(generalized) state channels", or a side chain, or a state channel on top of a side chain, or one of the above using zero-knowledge proofs, etc. If the program involves several assets, then each asset could be on its own possibly distinct blockchain with its own strategy, and then additional strategies must be picked to ensure the application's transactions are "atomic" across multiple blockchains. The set of combined strategies selected to implement a given program is called a *target*. The very same program can be implemented using many different targets. Thus, the signature sale or rock-paper-scissors game above could be run on Bitcoin in direct style, on Ethereum using state channels, or on Tezos using a side chain and zero-knowledge proofs, etc.

Let's consider as an *atomic fragment* a maximally long consecutive fragment of the program that involves a single participant and consensus. In the "direct style" strategy, every atomic fragment is implemented as a message sent on the blockchain, e.g. as a "contract call" on Ethereum. Due to size limitations of the target blockchain, the atomic fragment may actually be broken down into multiple consecutive messages, and this break down can already be considered an auxiliary implementation strategy. Direct style is the "naive" strategy that most programmers use

when writing a DApp using existing tools, because it has a simple mapping between fragments of the abstract interaction and code either in the smart contract or on the client side. But using it by hand is already quite tedious and error-prone, and this style has performance limitations making it a poor choice for complex interactions.

Using “state channels” is a more elaborate strategy, made popular by the Bitcoin Lightning Network, and further generalized by companies like Celer Network on Ethereum. In this strategy, a small number of participants (typically two, and realistically not more than a dozen or so) each put assets in a common “state channel” contract at the beginning. The regular way to take assets and participants out of the channel (and possibly close it), or to add further assets and participants into the channel, is for all participants to sign a common “settlement”. In between the two, the participants maintain the current “state” of the “channel”, which is made valid by all the participants signing it. The DApp makes progress by participants sending messages to each other for each atomic fragment, and all of them signing the resulting state. They can exchange as many messages as they want, in a fraction of a second, without having to go through the blockchain, to pay blockchain fees and wait for the blockchain to finalize their transactions. Where things get interesting is when one or more participants stop cooperating. In that case, the remaining participants can post to the smart contract a message that includes the latest mutually signed state; then they can challenge the uncooperative participants to either send a message corresponding to a fragment expected from them (as if in direct style, with some overhead), post a more recent state that overrides the previous one (states include a timestamp or serial number), or else timeout and forfeit any disputed assets (possibly keeping an undisputed share). A participant may also take his share out by similarly posting a state, sending messages to complete a transaction if needed, and challenging other participants to either post a state update, or let him get out with his undisputed share.

State channels are perfectly suited for interactions where the interests of all participants are aligned. They are not suited for every application: for instance, in an auction, it is not appropriate to use a state channel whereby a bidder would have to get authorization from previous bidders to outbid them. But when they are suited, they are much faster and cheaper than direct style—unless participants otherwise distrust each other and fear some would slow down the interaction by timing out. However, with previous development tools, to write a program using state channels is extremely difficult and error-prone, since you must duplicate all the code in two very different languages. For instance, today with Celer Network, you have to write the interaction code both in Solidity and in Go, and manually ensure that the two match each other exactly—or else it’s a bug wherein some participants will lose their assets. The resulting code is as difficult to audit and trust as it is to write, since you must follow what happens as asynchronous messages are exchanged between programs in two very different languages. *Glow* will make state channels usable by enabling developers to write programs that are a small fraction of the size, in a single language, with a much simpler programming model, and program verification, while automatically generating all the matching contracts and client codes, including error handling for all the corner cases.

Glow will be extensible so that new implementation strategies can be added by advanced developers. Separating the programs and their implementation strategies is very important, because it makes it much easier and cheaper not just to write DApps, but also to audit them: the business logic of a DApp needs be written and audited only once, by specialists in this business logic, and can then be reused with any implementation strategy. Meanwhile, the implementation strategy also needs to be written and audited only once, by specialists in software implementation, and can then be reused with any program. This is much less effort than with current technology, that requires a DApp to mix all these concerns in one giant program that is much more complex, at which point you have only audited the DApp on a single target.

Contracts as Logic

Our long-term vision for DApps is one that fully acknowledges the analogy between “smart contracts” and “legal contracts”—both the common parts where the analogy holds and is fruitful, and just as importantly the divergent parts where the analogy breaks down and is misleading. In both cases, we have systems that serve to prevent and resolve disputes and create alignment between the interests of participants. In both cases, the systems register titles and claims, and may resolve disputes by having parties argue their claims as an interaction before a referee. In the case of “smart contracts”, though, the participants are machines, and their arguments use formal logic, whereas in the case of “legal contracts”, the participants are humans, and their arguments use rhetoric. Logic is more rigid than rhetoric; that means that the resolution of disputes in smart contracts will be more predictable, but also less adaptable to changing circumstances. There is an entire field of Mathematics, Game Semantics, that studies how logical formulas can be translated into interactive proof “games” (and interestingly, this field indeed has its roots in formalizing rhetorical arguments). We can leverage this field to enable developers to specify their DApps in terms of logic, and automatically extract contracts that are enforced using interactive arguments.

This vision has consequences on how we will develop our language. For instance, smart contracts, like legal contracts, are meant to align the interests of participants, but the goal of the contract clauses is not to be invoked, it is to never be invoked: the purpose of the contract is to define what each party will do, and make it so obvious that not doing it will have bad consequences that no party would even think of not doing their part. Going to court is a last resort, not the intended use of the contract. Thus, the “state channel” style of contracts, where participants sign at the beginning, settle at the end, and normally don’t otherwise invoke the contract, should be the norm. The precise text of the contract doesn’t even have to be published: it can remain private, hidden under a salted hash, until one party fails and the other party needs to invoke a clause, that must then be revealed. Even then, zero-knowledge proofs can be used to keep even the logic private (until one party wants to reveal it). Furthermore, a simple multisig contract and a pre-signed transaction to a dispute-resolution address is all that is needed for all normal executions, hiding even the existence of dispute-resolution clauses; using a Schnorr signature for multisig can also hide the number and identity of participants.

There are broader consequences to thinking of contracts as a set of logical clauses with sanctions associated to violating them. This high-level view of contracts that makes it easier to write and compose them. We published an earlier article, Composing Contracts without Special Provisions — using Blockchain History, explaining how logical formulas about events recorded on the blockchain can be used to write and compose arbitrary contracts. This technique enables contracts about resources that offer no particular API to help write contracts about them. The same technique also enables contracts about resources managed by different blockchains, as long as there is a "bridge" allowing one blockchain to "read" the state of the other. Finally, this technique is itself easily bridged with traditional techniques of writing contracts as code: in one direction, programming language semantics provides a way to extract logic propositions from existing code written in an existing language; in the other direction, logical query languages provide a way to extract code from a logical proposition.

As our ecosystem grows new layers of abstractions, we will build logic layers that enable developers to write declarative DApps in a more declarative style, in terms of data structures and relationships between them, their type schemas and their logical constraints. Not only will such logic layers allow the development of DApps in a more succinct style that eases composition and minimizes development costs, it will also greatly simplify reasoning about a DApp—making its correctness "invariants" obvious to prove, and, by deriving the implementation directly from them, making the correctness of this implementation also obvious, while at the same time enabling a wide range of strategies to increase performance.
Audit of DApps and trust in DApps will be greatly facilitated, and costs reduced, by using such a declarative style.

Safe DApp Runtime

Glow makes it much easier for developers to write a DApp and compile it to contract and client code. But to run a DApp, additional code needs to be provided, that is shared between all DApps: the *runtime* system—all the infrastructure that supports running the DApp. The runtime, too, needs to be correct, and needs to be safe against known attacks. And there, too, *Glow* will be vastly superior to competing frameworks to develop DApps, that seem blissfully ignorant of common attacks, and completely unhelpful for dealing with them.

Any non-trivial DApp will have timeouts and challenge periods, whereby a participant stands to lose some or all of their assets if they don't post a message before a deadline. If somehow a malicious participant can successfully launch a *Denial of Service* (DoS) attack against another, that other participant will be unable to post their message, and will forfeit some assets. There are many obvious and less obvious ways to achieve a DoS attack.

If the attacker knows the IP address of the victim, they could try to hack into the victim's machine if it's improperly secured; even if it is, the attacker could leverage a *botnet* to launch a "Distributed Denial of Service" (DDoS) attack, whereby each zombie machine in the botnet sends messages to the victim's address, flooding its connection, so it doesn't have any

bandwidth left to talk to the blockchain anymore. For a high value transaction, the attacker could even disrupt network cables or routers connecting the victim to the Internet, and thereby prevent her from posting a message; the victim might have to go to another city, or even country, to post their message. To mitigate such DoS attacks, the participants must choose a challenge period that is long enough that sustaining the attack is economically unaffordable. Additionally, participants should have computers in multiple trusted datacenters around the globe (with armed guards and other physical security to keep them trusted), themselves synchronized with a private BFT consensus, and hiding their location by using some kind of onion routing network to send data to the blockchain. Most teams developing DApp infrastructure are unaware of these attacks, and don't have anyone competent in cybersecurity, redundant distributed systems, or anonymization protocols, who could help them write a solution, even if they were aware of it.

Another attack would be to purchase entire blocks of the blockchain, and thereby starve the victim of opportunities to post their messages. This attack has actually happened at least once, on a million-dollar contract. We described this attack and how to survive it in our earlier article "Why Developing for the Blockchain is Hard — Part 1: Posting Transactions". Importantly, this is an *economic* attack, that requires economic counter-measures. In particular, posting a message within a deadline involves winning an auction for scarce transaction space on the blockchain. A related attack applies to contracts using Merklized state, which is always the case on blockchains that use UTXOs: an attacker with a superior posting algorithm to constantly make small changes to the state and prevent other participants from posting their messages. Then you must not just win an auction, but a constantly moving target of an auction, that requires you to closely follow and react to blockchain network traffic, in addition to playing the auction game. Most teams developing DApp infrastructure are wholly unaware of these attacks. Even if they were aware of it, they don't have anyone competent in economics in their team who could help them write a solution.

Previous development platforms require their users to reinvent solutions to all these issues, when these users often don't have the expertise required to address any of them. Even when some users can handle these concerns, previous development platforms require them to do it in a way that mixes these concerns with the application code, resulting in a complex system that is difficult to audit and fragile to maintain.

We at *Glow*, have or will have experts in-house to defend against these and many more potential attacks. We will implement all the necessary counter-measures, and keep them updated, have them audited, etc., so they will be available for all DApps written in *Glow*. Users can focus solely on their value-added expertise while being confident that a competent team is fighting off all the known attack vectors for them, even those that they users haven't heard of.

Logical Model

Proving DApps Correct

Since there are such high stakes to ensuring that a DApp doesn't have a bug, *Glow* uses formal methods to help developers prove that their DApps are correct.

What does it mean, for a DApp to be correct? Informally, it means that the DApp does exactly what it is intended to do, and nothing else. "What it is intended to do" includes implementing all the rules of the interaction as understood by the users—assuming the users understand what they're trying to do. "Nothing else" includes not withholding assets from their legitimate recipients, not sending assets to attackers or to anyone but the intended recipients, not leaking secret keys or other sensitive information, etc.

Already, we see that there can be a gap between this informal understanding and what formal theorems will be proven using *Glow*—the same issue applies to whatever formal methods are used to verify programs in whatever language. We published an earlier article "What do Formal Methods actually Guarantee?" that explains what you can or cannot hope to achieve using formal methods. Formal methods are not magical; they are methods that must be used, and used correctly. But ignoring them is even less magical, and is probably irresponsible considering the stakes of getting a DApp correct. If you don't think you can get formal methods to work, you probably shouldn't be writing DApps. And if you are going to write a DApp, you should be using the best formal methods on the market to ensure it is correct—*Glow*.

Glow uses mathematics to automatically prove various important correctness properties about each DApp. Some of these properties, it automatically includes in its analysis. Some of these properties, it requires users to provide suitable parameters so it can include them. Some of these properties, the users will have to write themselves, with *Glow* providing a logic framework that makes them easy to write. In all cases, *Glow* then feeds these properties to a theorem prover (currently, Z3, from Microsoft), that will either prove them, find a counter-example scenario, or timeout. By reducing the incidental complexity of its infrastructure compared to rival solutions that are lower-level and thus include many extraneous details that the theorem prover has to deal with.

The Logic of DApps

Once you understand the purpose of a DApp as an interaction between multiple distrusting parties with digital assets at stake, it becomes easier to think of the correctness properties that matter for it, and of the logic in which you want to express those properties:

- First and foremost, there are multiple distinct parties each with access to their own private information in addition to public information. The way to formally express who knows what is by having a notion of identities, and using *Epistemic Logic*.
- Now, the point of a DApp is that who knows what changes with time with the exchange of messages. To handle the "when" aspect of this knowledge requires *Temporal Logic*, which combined with the Epistemic Logic above yields *Dynamic Epistemic Logic*.
- To express the limitation and conservation of resources is done using *Linear Logic*.
- If further you want to use zero-knowledge proofs, you must be able to reduce your computations to a finite fragment of your logic (modulo cryptographic assumptions).
- Last but not least, to prove that each user is safe whatever the other users do, you need an *Economic* model of the interaction in terms of *Game Theory*, and more precisely in terms of *Mechanism Design*.

Existing formal verification tools do not handle any of these logical aspects of a DApp, and therefore cannot even express the properties needed to assert their safety. Yet, no DApp language except *Glow*, even includes plans to use any of these variants of Logic.

By integrating the above logical formalisms, *Glow* is able to simply express and prove essential DApp correctness properties as follows.

Glow will automatically include these properties in its analysis:

- At all times, all account balances contain non-negative amounts of every asset.
- At the end of a contract, the balance of the contract is zero.
- A participant with a message in their past and another in one their future, must have deposited some collateral that will incentivize them to keep cooperating until the end.

Glow will include these properties in its analysis given proper parameters by the user:

- Assume for each user as a parameter a subjective valuation of asset positions, and setting aside the acceptance of paying some bounded gas cost. We will then show that once the participant entered the interaction, their optimal strategy is to follow the rules of the program, and will lead to higher valuation at the end. This entails *game-theoretic safety* of the DApp, whereby the interaction is *safe* for the participant if the other participants, even if they collude with each other, cannot cause the participant to lose any value in assets compared to what would happen if they cooperated. Note however that if the other participants fail to cooperate, the safe participant still has to pay the regular gas costs for participating, and loses the opportunity cost of putting his capital and time to better use while the other participants time out.

Glow will make it easy for users to specify additional properties to verify:

- If all participants cooperate and none of them times out, then the "liveness" properties hold, that describe the intended outcome of the completed interaction.
- For each participant, whichever the inputs and the behavior of other participants (that are externally determined), following the program will ensure the "safety" properties hold.

- More generally, given a set of trusted participants and a set of untrusted participants, these trusted participants by following the program can ensure that their associated "safety" properties hold, despite any effort by the untrusted participants, for all inputs.

Economic Safety

Glow leverages two distinct but complementary branches of Mathematics to prove properties of interactions: Game Theory and Game Semantics. In both cases, the word "game" is a technical term: It applies to all interactions between multiple parties that do not trust each other (technically called "players"), and doesn't at all imply that the interaction is necessarily about gambling. Thus, financial interactions as well as hiring interviews or jet-fighter duels are "games" and their participants "players", as far as Mathematics is concerned. Game Theory, also considered a branch of Economics and of Military Science, studies games from the point of view of gains and losses to each player, strategies that players may follow, and equilibria between strategies. Game Semantics studies the structural correspondence between logical formulas and interactive proofs: to each formula corresponds a verification game between a "verifier" who defends the formula and a "falsifier" who defends the opposite formula; the verifier has a winning strategy if and only if the formula is provable—a winning strategy being one that if followed, guarantees that the player will win whatever the other player does.

Previous "smart contract" formal verification platforms have blatantly overlooked both these branches of Mathematics, even though they are crucial to understanding decentralized interactions. Instead, these platforms have been direct applications of existing program verification techniques to smart contract. But these existing techniques all start with the assumption that the many parts of a program all conspire together towards a common goal—which is exactly the assumption that cannot be made in a DApp and especially not in its smart contract. Therefore, existing "smart contract" formal verification platforms are mostly worthless. They do not even begin to tackle the problem that matters. At best, they can prove some results about "liveness" properties. As far as "safety" properties, they can only handle of DApps so simple that they don't involve much interaction at all, and even then, with great effort from the developer.

In *Glow*, all assertions, as specified with the `assert!` primitive, are made from the point of view of a set of participants who are trusted—typically, either a singleton with a single participant, or the empty set. Furthermore, the predicate being asserted may use the *modal* operators `must eventually`, `must forever`, `can eventually`, `can forever`, and other operators from Computation Tree Logic, a variant of *temporal logic* that can deal with the many branching possibilities of choices made other participants and the environment. These operators allow developers to specify properties involving the necessary or possible futures of the program execution from the point where the assertion appears. Thus consider the following assertion:

```
@A assert! must eventually(
   end: end.winner == A | timeout: timeout.player == B )
```

The above predicate claims that in the future, whatever may otherwise happen, *if A follows his part of the protocol*, then the computation will either reach the regular end of the computation (as identified by label `end`) at which point the winner is A (as identified by the binding of variable `winner` at the end, and `A` as the current point of execution), or it will similarly reach a point where player B timed out. Moreover, since A is trusted, and A only, this result may depend on A making the "correct" choices based on some strategy, but is *robust* against any adversarial choice of action or inaction made by any untrusted party.

Computing the proper robust predicates for a point of execution is where Game Semantics applies: at each alternation between actions of trusted and untrusted players, the quantifiers change. In a `must eventually` predicate, choices made by the trusted players are existentially quantified whereas choices made by the untrusted players are universally quantified. That is, it is enough that there exists at least one correct choice for each move that a trusted player will make. But there must be no move that untrusted players can make that would possibly invalidate the conclusion. Thus, every temporal predicate about an interaction can be reduced to a logical predicate without temporal operators, based on the program and its control flow graph, and this game semantic alternation of quantifiers between friends and adversaries.

Logic-based Contracts

A contract can then be written in terms of each party promising to uphold some logic formula, and the legal consequences when they fail to hold—typically, the loss of a collateral previously deposited by the party found wrong as a prerequisite for participating in the DApp. Disputes will be adjudicated fairly based on a fully automated interactive proof protocol to determine which party is at fault and which party is entitled to compensation—see our Appendix for technical details. Logic provides a common language in which contracts can be composed at a much higher-level than calls to on-chain "smart contract" procedures. Using programming language semantics, we can also turn any program into a logical predicate relating its inputs to its outputs. We can then leverage existing programs, and write contracts about how the behavior of the participants shall match the results of some future agreed upon computation. In case of dispute, the logic-based DApp can later verify whether the result of the computation is as claimed, and punish the wrongful participant. Multiple programs in multiple languages can be used, as long as they can each be compiled to a virtual machine the semantics of which was formalized in terms of logic.

Thus, in *Glow*, users will be able to make claims about data structures and computations that are not actively validated by the contract itself, but by active participants outside the blockchain. These validators watch messages relevant to the DApp, counter bad claims and punish the bad claimants.

```
publish! A -> index, amount, beneficiary;
claim! A -> withdrawal_ticket[index] ==
```

```
            make_ticket(amount, beneficiary), {use_once: true};
// The following happens after the claim is successfully
defended
// against any counterclaim filed within the challenge period.
withdraw! beneficiary <- amount; // beneficiary may be A, or
not.
```

It then becomes possible to enforce the validity of side-chains with complex structural invariants, using a technique that in general we first described in February 2018, and that has since been known as “optimistic roll-up”. The resulting code is much simpler than if the verification games for those claims had to be coded manually. The correctness is also much simpler to assert, since the claimed predicate can be assumed as an invariant accepted by all parties after it has survived any adversarial counter-claim. Existing formal methods could never handle such a construction.

In the future, *Glow* will include a declarative language for describing data schema and logical queries, based on categorical databases, so that efficient code can be generated at the same time for the clients that use the data, the servers that store it, the validators that watch what the servers do, and the contracts that keep everyone honest.

Domain-Specific Logic

No existing programming language can quite express the programs that *Glow* will make possible to write, even though each of the concepts in *Glow* may be found in some predecessor. For the same reasons, no existing logic formalism can express the correctness predicates that *Glow* will make possible to verify. To minimize complexity and the associated exponential risk for catastrophic mistakes, we will develop a Domain-Specific Logic, just like we are developing a Domain-Specific Language: The logic used by *Glow* will be made of fragments of known logical systems; it will be translated into first-order logic for automatic verification using existing theorem prover technology; but the precise combination will be its own system.

Consider the fact that the strategy for each player at any moment may only depend on information that is known to that player at that moment (including public information), and not on information private to other players. This kind of restriction on strategies is natural in Game Semantics, and we can make it implicit in the logic of *Glow*. It has been studied in logics such as Dependence Logic or Independence-Friendly Logic. To express the same restriction in a general-purpose logic system, users would have to manually reduce their predicates to classical logic, for instance by Skolemization: bind the strategy to a function that must be chosen (quantified over) before the inputs are known, and that will be called only with the inputs that the function is allowed to depend on. This is easy to get wrong if done manually, but can be automated away using a Domain-Specific Logic.

In the future, we may want to express how strategies use randomness according to various distributions of possible choices, and compute expected gains and losses from using one strategy or another. There again, with a Domain-Specific Logic, we can in the future add a fragment of probabilistic logic at a later time, and preserve all the work done by our users through appropriate automatic translations. Using libraries on top of general purpose logic, users would have to translate things manually, and that would be extremely error-prone in addition to being expensive.

Inasmuch as we may have to invent our own logic, we will use the general approach of Computability Logic: putting the Game Semantics first, and the syntax afterwards. This will make it notably easier for logical predicates to be used in claims that can be verified interactively as part of a “smart contract”.

Ultimately the argument for using a Domain-Specific Logic versus a library on top of some general-purpose logic is the same as for using a Domain-Specific Language versus a library on top of some general-purpose language: language abstraction reduces complexity, removes opportunities for using the library “wrong”, and allows for linear rather than exponential cost when composing abstractions. Using libraries allows for low-level mistakes that “break the abstraction”, and combining such libraries pushes onto users an ever renewed exponential complexity that with language abstraction can be handled only once, by the language developers. Thus, with Domain-Specific Logic, we can consistently combine many logic fragments that are required to specify DApps correctly, without users having to do a lot of hard work.

The same reduction in complexity makes a Domain-Specific Logic *easier to trust*, when indeed it fits the domain. Using a general-purpose logic, each and every correctness property becomes a large formula with many low-level details. In addition to the abstract property that you care about, you then have to manually audit the correct translation of said abstract property into the resulting concrete formula. Did the manual translation correctly follow the desired “design pattern”? That’s as many opportunities for critical bugs. It is easy for some application-specific bug to hide in these details, and each formula of each application must be audited for them. By contrast, using a Domain-Specific Logic, each and every correctness property is as simple as can be. As for the translation step, it is automated, and its compiler can be audited once for all properties of all applications. Furthermore, this compiler can itself be made easier to audit thanks to its own formal or informal proof of correctness. Finally, it is much harder for the compiler to contain an application-specific bug that won’t be a glaring bug found in other applications or tests.

The *Glow* Architecture

The Power of Abstraction

The way that *Glow* enables writing safe applications is by being a *Domain Specific Language* (or DSL). Its closed programming model, tailored to the application domain, (1) empowers developers to reason precisely about what programs do or don't do for users, while at the same time (2) completely shields developers from underlying concerns that could only cause critical errors if used inconsistently with the chosen implementation strategy. This model maximizes the amount of security that can be achieved while minimizing the development costs, as long as the security properties that matter are indeed at the *level of abstraction* offered by the DSL.

Now, a DApp is made of more than what the developers write in a DApp language (or in the combination of contract and client languages, on other platforms): to deploy the DApp, you must also trust every compiler and runtime library used to build the DApp, plus the server configurations, the physical security of the machines, etc. The overall system is only as robust as its weakest link. Even if the code as written by developers is itself as secure as can be, all the other components of the system must be secure, too, for the DApp to resist attack. For *Glow* as well as for competing software platforms, it is important to assess the security of the compilers and runtime libraries used while building and deploying DApps. That is where we must generalize the principle that makes *Glow* a secure *language*: we must conceive it as part of a *platform* that is itself secure.

The security of *Glow* as a language flows from how it keeps DApps *simpler* than other approaches permit. This simplicity makes it more manageable to prove DApps correct. And *Glow* makes DApps simpler thanks to the power of *abstraction*.

A good abstraction has enough low-level details that it perfectly expresses what the program does and what it doesn't, as far as users care about. But a good abstraction is also high-level enough that it doesn't contain any lower-level details that would only confuse the concepts that matter, distract from them, and introduce opportunities for bugs. A good abstraction thus brings *clarity* as to what a program does or doesn't do—so that programmers, users, auditors or automated theorem provers can reason about it and assess that it is indeed correct.

Bad abstractions *leak*: the program's behavior critically depends on the lower-level details that the abstraction ought to have hidden away. The program is thus more complex than it appears at first, often in subtle ways that are hard to reason about. Its correctness becomes elusive and bugs become likely. When multiple levels of abstraction leak, the actual complexity of the program increases exponentially with the number of abstractions. Even experts make mistakes, while beginners cannot avoid them. Correctness becomes impossible to assess. Bugs are almost guaranteed.

An abstraction that does not leak but remains “airtight” in presence of deliberate attacks is called a *full abstraction* by programming language researchers.

Now, abstraction is not something that can be provided to users once and for all as a magic powder to sprinkle over code. Rather, it is a discipline to keeping software simple and correct that must pervade how the software is written—both by users, and by implementers of a platform. Within a unit of code that a programmer may write, “Functional Programming” and “Object-Oriented Programming” provide a variety of techniques to create and combine abstractions. To maintain abstraction in larger programs, good software platforms provide some kind of “module system”. Better software platforms also provide some more or less advanced “type system” that can greatly restrict the ways abstractions may leak. The best software platforms provide a system for “language-oriented programming” that alone can ensure full abstraction, if used properly.

Observable Language Abstractions

Glow will not only provide its users with a Domain-Specific Language at the best level of abstraction for writing DApps. *Glow* will also evolve towards providing ways for its users to build the abstractions they need as their DApps grow. But just as importantly, the internal architecture of *Glow* will be organized internally around abstractions that will keep it as simple as possible, and as easy to audit for correctness as possible. For it is not enough that your DApps shall be correct. The infrastructure that runs your DApps must also be correct. Other software that include millions of lines of code imported from insecure sources have been hacked—the larger and more complex the software and its dependencies, the harder it is to audit it and keep it secure across time.

Thus, *Glow* provides abstraction through all of modules, types and languages, to minimize the complexity of the system, and make it possible to keep it secure. In particular, *Glow* includes multiple languages, organized in layers. Each language is defined by translations to other languages in lower-level layers, until a level is reached that can be executed by existing systems. Each translation layer is kept simple enough that it can be reasoned about, and, in the future, proven correct using formal methods. But also, importantly, each layer provides a *full abstraction*, so that the correctness proofs for each layer, formal or informal, can be combined into a correctness proof for the complete system. Thus, the *Glow* platform (compiler and runtime libraries) will stand a chance to be audited for security. By contrast, a rival system organized in a *monolithic* way, in a single layer, without such abstractions, will be impossible to audit, due to its exponential complexity.

Moreover, translations between the many languages in the system must be kept “reversible”, or “observable”: operators who watch a DApp at work, whether during development or in production, need to make sense of incidents at the most suitable level of abstraction, so they may take proper action. Incidents must be translated back to as high-level as possible, so that

operators can make economically informed decisions at to which incidents to address in what priority with what resources. Yet incidents must be kept to as low-level as needed (but no more) to take effective action. This *observability* of the DApp cannot be added after the fact, but must be built into each translation layer between languages within the platform. *Glow* can achieve this observability by using taking advantage of language abstractions that each provide full abstraction and observability; thus the effort required to get it right only grows linearly with the number of layers. Our monolithic rivals will not be able to provide this observability correctly, due to the exponential complexity of their platforms.

Language Layers Within Glow

The internal architecture of *Glow* will be organized around a *Tower of Languages*. This *Tower* will enable developers to think about each issue at the most suitable level of abstraction—the one that makes it easiest to think about this issue and solve it. More generally, this Tower will be essential to our providing a uniquely robust architecture that can reduce *overall* system complexity. Indeed, it isn't just the DApp-specific code that matters when auditing and trusting a DApp; the infrastructure code also needs to be audited and trusted. There too, complexity will cause systems to crumble. Competing DApp platforms and the DApps build on top of them will fail to deliver trustworthy code unless they build their software around an architecture that can keep the platform itself simple enough to be audited.

Most *Glow* developers will write their DApps using the *Glow* DSL itself, at the top of our Tower of Languages. They won't normally write programs in any other language in our Tower. Still, advanced developers will sometimes use or modify the many languages of our tower, to extend the platform with new features, or improve its performance for their use case. Even regular developers may become accustomed to *reading* programs in these languages, if not writing them: while developing and debugging their programs, they will want to inspect how these programs are translated into the layer of abstraction where some erroneous or otherwise interesting phenomenon emerges; they will want to play with programs and reason about programs at that lower layer of abstraction; and thus they stand to detect, understand and resolve all issues—hopefully before deployment in production. Maintaining an explicit well-defined language for each of these layers of abstraction enables this logical reasoning and exploration. Logical reasoning further makes it possible to prove the programs correct, and, in the future, to prove the automatic translations themselves correct.

As detailed in Appendix B, our *Tower of Languages* will include not just many languages, but also many *translation strategies* between these languages.

At the top will be the *Glow* DSL. Below it, there will be "End-Point Languages" describing client code, contract code and logical models. Below them, will be a variety of target languages, depending on the blockchain used for contracts, and the development environment used for client or server code. In between these two layers, many intermediate languages may each address one aspect of the execution.

Multiple implementation strategies, with different tradeoffs, will cater to the variety of needs of DApps. For instance a “direct style” that translates interaction steps into messages on the blockchain is well-suited for an Auction DApp with an open set of bidders. But a “generalized state channel style” can drastically reduce cost and latency for DApps involving a small closed set of participants, at the cost of a more elaborate transformation with additional layers. Further scaling may involve using some kind of sidechain, which itself may involve interaction on the main chain as a fall back; this again requires further layers of abstractions and translation. Larger DApps may involve additional layers of to implement a virtual machine that enables them to bypass the stringent size limitations and execution costs of running code directly on the blockchain.

Interestingly, a very same DApp can be translated in different ways, depending on the amounts at stake, the degree of mutual trust of the users, their latency requirements, their sensitivity to operating costs, etc. Some parts of these translation path for a DApp is determined by which assets are used: the blockchains on which each of these assets are managed each comes with limitations that must be either worked within, or worked around using additional translation layers. Other parts of these translation paths result from economic choices made by the users of the DApps as they negotiate the terms of their interactions: relatively trusting participants who use a DApp to process a five-dollar interaction may not use the same translation as relatively distrusting participants who use the very same DApp to process a million-dollar interaction. Yet, with the *Glow* architecture, each DApp and each of the translation steps only has to be proven correct and audited once to be trusted. A different architecture would require complete rewrites and completely new audits for every useful combination—resulting in an exponentially higher number of audits, each exponentially more complex and costly.

Beyond a Language

The *Glow* language is only the first step towards something bigger. The *Glow* architecture has a potential for much more than a DApp DSL—and will *have* to become more, as it grows in popularity.

Indeed, security is and will forever remain an arms race between the builders and the breakers. As we use *language abstraction* to eliminate bugs from the application layer that communicate with blockchains, the weakest link that attackers target will move to higher layers (closer to the user interface) or to lower layers (closer to the network, to the hardware). We can apply the same method of *language abstraction* to keep DApps secure at these other layers. Eventually, the security of the DApp depends on the entirety of the environment in which the DApp runs—not just the part that directly interfaces with the blockchain, not only the user interface on top, but the entire computing system, including any other program that may run on the same machine. In other words, as attackers get more sophisticated, we eventually need an entire *Operating System* designed for security.

If our language finds the success we are working towards, we at *Mutual Knowledge Systems, Inc.* will eventually build such a secure Operating System. A system that enables all developers to safely create and compose language abstractions. A system that prevents cracks in the *semantic tower* of programming languages and models; one that ensures that the translation layers between these languages are fully abstract, through a combination of compile-time verification and runtime checks. A system that uses composable model for language abstraction that we previously invented (see our published article Climbing Up the Semantic Tower — at Runtime). A system that can provide users with a trusted computing environment while resisting the wide range of known attacks. A system that will result in overall safer applications at an affordable cost—including the cost to formally verify whichever properties we care about most. A system where the amount of effort to keep the system secure grows linearly rather than exponentially with the number of layers in the system.

Our long term vision for a *DApp Operating System* (DAOS) is that it will eventually encompass all the software that runs on a secure device with which users manage their assets. For that we can build on the efforts of existing teams building proven-correct operating systems such as seL4, *CertiKOS*, BedRock Systems, etc. It is preferable that for serious DApp usage, users should use a dedicated device that they trust, and not use untrusted code on it. However, to reduce costs and accommodate users who desire to use the same device for regular and trusted operations, the secure and insecure code can be isolated from each other by running in mutually isolated virtual machines or processing modes—for instance by using a hypervisor like Qubes, or running in a Trusted Execution Environment (TEE). With the ability to virtualize computations at the level of every programming language, and ensure full abstraction, our DAOS can potentially provide a more secure and more efficient way to integrate secure and insecure code. We have developed a reflective model that notably allows to separate the security capabilities required in objects and their meta-objects, thus improving security over other secure systems in how applications are combined, while simplifying applications themselves.

Other directions for research and development include improving the software elaboration process itself: integrating the many currently disconnected activities including editing the code, analyzing it, reasoning about it, transforming it one way and back, generating tests for it, running tests on it, interactively executing and debugging it, reviewing it, exploring variants of it, deploying it in QA or production environments, accepting feedback from users, measuring performance, comparing executions to expectations, analyzing usage data, unifying the many variants of the data as the code evolves, etc.—not just for programming experts, but also for domain experts who are not programming experts. While none of this is a short term concern, in the long run, as our development ecosystem grows and our company with is, a vision is essential to guide us along many dimensions of potential product improvement, and allow us to stay a few steps ahead of our competitors.

While this long-term vision is many years away, it establishes *Glow* as a project with a long-term plan to create sustainable value. The potential for long-term growth and diversification is

enormous. From DApps narrowly defined to users reclaiming sovereignty not just over their digital assets, but over their entire digital experience.

Bibliography

Here is a list of relevant scientific publications by past and present direct contributors to our language (in bold), many of them peer-reviewed, demonstrating our expertise in the domain at hand: the design and implementation of programming languages that enable developers to build safe decentralized applications, amenable to formal verification.

François-René Rideau, "Simple Formally Verified DApps—and not just Smart Contracts" *EthCC[3], 2020*.

Jay McCarthy and **François-René Rideau**, "Alacrity: A DSL for Simple, Formally-Verified DApps", *2019.*

Kimball Germane, **Jay McCarthy**, Michael Adams, and Matthew Might. "Demand Control-Flow Analysis". *VMCAI, 2019.*

François-René Rideau, Language Abstraction for [V]erifiable Blockchain Distributed Applications, *IOHK Summit, 2019*.

François-René Rideau, "Composing Contracts without Special Provisions—using Blockchain History". *Hackernoon*, 2019.

François-René Rideau, Binding Blockchains Together With Accountability Through Computability Logic, *LambdaConf 2018*.

François-René Rideau, "Why Developing on Blockchain is Hard? - Part 2: Computing Proper Collateral". *Hackernoon, 2018.*

François-René Rideau, "Why Developing on Blockchain is Hard? - Part 1: Posting Transactions". *Hackernoon, 2018*.

François-René Rideau, "Climbing Up the Semantic Tower — at Runtime". *Off the Beaten Track Workshop at POPL, 2018.*

Stephen Chang, **Alex Knauth,** and Ben Greenman. "Type Systems as Macros". *POPL, 2017.*

Jay McCarthy, Burke Fetscher, Max New, Daniel Felty, and Robert Bruce Findler. "A Coq Library For Internal Verification of Running-Times". *Science of Computer Programming, 2017.*

Robert Goldman, Elias Pipping, and **François-René Rideau**, "Delivering Common Lisp Applications with ASDF 3.3". *European Lisp Symposium, 2017.*

James Y. Knight, **François-René Rideau**, and Andrzej Walczak, "Building Common Lisp programs using Bazel or Correct, Fast, Deterministic Builds for Lisp". *European Lisp Symposium, 2016.*

François-René Rideau, "Why Lisp is Now an Acceptable Scripting Language". *European Lisp Symposium, 2014.*

François-René Rideau, "LIL: CLOS Reaches Higher-Order, Sheds Identity And Has A Transformative Experience". *International Lisp Conference, 2012.*

Tim Disney, Cormac Flanagan, and **Jay McCarthy,** "Temporal Higher-Order Contracts". *International Conference on Functional Programming, 2011.*

François-René Rideau and Robert Goldman, "Evolving ASDF: More Cooperation, Less Coordination". *International Lisp Conference, 2010.*

Jay McCarthy and Shriram Krishnamurthi. "Cryptographic Protocol Explication and End-Point Projection". *European Symposium on Research in Computer Security, 2008.*

Jay McCarthy, and Shriram Krishnamurthi. "Trusted Multiplexing of Cryptographic Protocols". *International Workshop on Formal Aspects in Security and Trust, 2009.*

Jay McCarthy and Shriram Krishnamurthi. "Cryptographic Protocol Explication and End-Point Projection". *European Symposium on Research in Computer Security, 2008.*

Appendix A: Logical Techniques for DApps

Game Semantics for DApps

The correspondence between verification games and logical formulas goes both ways: In the previous section, we matched an arbitrary with a logical formula that tells whether it is safe for the first player to partake in it. Conversely, we can match an arbitrary logical formula with a

“verification game” that is safe for the first player to partake in if and only if the formula is provably true. This correspondence is the subject of a branch of mathematics known as Game Semantics.

Let us see how a game is derived from a logical formula. The first player, or *verifier*, claims that the formula is true, and offers to verify it. The second player, or *falsifier*, will claim the opposite. If the outermost logical connector of the formula is a disjunction or existential quantifier, the verifier chooses a subformula or set element, and produces it as a *witness*. The verifier then recurses into the chosen subformula. When a conjunction or universal quantifier is reached, the verifier cannot afford to show a potentially astronomical number of witnesses to cover all possible cases, but can challenge the opposite claimant, the falsifier, to show his single witnesses for the opposite formula. The game thus continues by replacing the remaining formula by its opposite, and switching the roles of verifier and falsifier. When the remaining formula is simple enough, the formula can be directly evaluated with the values provided as witnesses, and the case can be adjudicated based on whether the result is true or false. Importantly, the fundamental theorem of Game Semantics is that the verifier (resp. falsifier) will have a winning strategy to convince the judge against an adversary interested in proving them wrong, if and only if the disputed statement (resp. its opposite) is provable. Thus, we can use this “interactive proof” mechanism as a general tool to ascertain the truth or falsity of arbitrary contractual claims, as long as they can be expressed as logical predicates.

Semi-Automatically Estimating Proper Collateral

Since the initial formula is finite and only contains a (usually small) finite number of alternations between disjunctions and conjunctions, the game will be completed in a finite number of steps with a (usually small) upper bound that is known in advance. The compiler can automatically determine, in advance of any execution, the maximum amount of “gas” required to argue every step of this process, using abstract interpretation. The compiler can also determine the maximum duration for the game, based on the maximum number of steps multiplied by the challenge timeout period.

Based on those determinations, each participant can estimate what collateral they require other parties to deposit before they enter the game. If the participants all agree on each other’s requirements, they will deposit their assets and collaterals, and the game will proceed. If some disagree, the game is cancelled and presumptive participants can try to find other people to interact with, or renegotiate the terms of the game. A participant may estimate the collateral they require from another participant by adding:

1. An estimate for the “smart legal” costs they would have to incur to have the “smart judge” rule in their favor if the other party fails, based on the gas costs, and an estimate of how high gas prices may increase during the duration of the DApp and its dispute resolution.
2. An estimate of the price to replace the assets that the participant stands to lose if the other party fails to cooperate, there again based on an estimate of how the exchange rates for these assets may change before the DApp and its disputes are completed.

3. An estimate of interest rates to cover the opportunity cost of having assets immobilized while completing the DApp and any disputes.

These estimations depend on the subjective evaluations of each user, their economic models, their risk profile, etc. They are computed privately by each user on their own computer using their own asset management software and models. Then the estimates are exchanged between users as part of a phase to negotiate the terms of the DApp, before the DApp contract is signed. The proof of safety of the DApp for a given participant will take as hypothesis that the terms accepted during this negotiation were indeed agreeable to this participant.

Ensuring Timely Game Dispute Resolution

The compiler can also determine the maximum duration for the game, based on the maximum number of steps, multiplied by the challenge timeout period.

Now, for participants to be able to survive DDoS attacks or attacks on network infrastructure, important contracts will probably have a challenge timeout period of at least 24h, probably more like 48h, or even as much 7 days, as in the Bitcoin Lightning Network. This means that a dispute requiring an interactive proof that has tens of steps can immobilize capital for many weeks, or even many months.

To reduce the precious time it takes to either validate or invalidate a claim, DApp developers can trade time for space using a technique called Skolemization: The current player can eliminate his next step by publishing a table or formula explaining how he'll reply to the coming challenge by the next player.

The compiler can *automate* the Skolemization of formulas used when resolving disputes. The choice of which parts of which formulas to dispute can be done manually by the developer, or it can itself be done automatically by the compiler based on various strategies and developer annotations. In any case, automating the automatic transformation allows the DApp to be specified in a way that makes audits and correctness proofs easiest, while being executed in a way that is most cost-effective, without running the risk of an error during a manual translation or during the maintenance of the complex result.

Skolemization can notably be used to compress the number of steps in a binary search by making it a 16-ary search to divide the number of steps by 4, or 256-ary search to divide the number of steps by 8, and so on. However, the space and associated cost required to publish the witness tables and formulas increases exponentially with the number and complexity of the steps being saved.

Publishing Data At Scale with a Mutual Knowledge Base

Now, consider what would be possible if there were a cheap and reliable device to publish large pieces of data, so that all parties could be guaranteed to be able to view the evidence in a

dispute. With such a device, the number of proof steps can be significantly reduced, as millionary or billionary search become possible. As long as the cost is (a) affordable, (b) ultimately covered by the failing party, and (c) not paid as long as all parties cooperate, then depositing a collateral that covers the cost of publishing a lot of evidence isn't an obstacle to making efficient DApps with fast resolution in a minimal number of steps.

We call such a device a *Mutual Knowledge Base* or MKB: indeed, what this registry creates is "data that everyone knows about", which is known in Epistemic Logic and Game Theory as "Mutual Knowledge". There are many uses to such a MKB:

- It can serve as a "smart court registry" on which to publish large amount of evidence during disputes, as above.
- It can serve to registry in advance titles and claims and transfer contracts, thus helping resolve future disputes, making such disputes less likely, and wholly preventing many of them.
- As we discussed in our February 2018 whitepaper Legicash FaCTS: Fast Cryptocurrency Transactions, Securely, such a registry can be used to publish the data for side-chains. These side-chains can enable secure scalable transactions for digital assets maintained on arbitrary blockchains, even when the main blockchain itself doesn't scale.
- It can publish large amounts of data at scale and avoid censorship of data.
- It can be used to force the managers of "Plasma" style chains to publish their data, thus preventing the "block withholding attacks" described in the Plasma whitepaper.
- This registry can also publish large input data tables and large execution traces, making affordable to verify large computations on the blockchain. For instance, prices could be computed based on complex financial models, with the formula contractually agreed upon in advance, and the computation done later when the contract reaches maturity and the data becomes available.
- The idea of a MKB is also known as a "Data Availability Engine" in the Ethereum community. Vitalik Buterin notably popularized a variant of the concept as "rollup", wherein the Ethereum blockchain itself as the MKB on which to publish the data for a side-chain. Because the blockchain doesn't scale much, a "rollup" also doesn't scale much, but (a) it still claims about thirty times more than direct use of the Ethereum blockchain, and (b) a rollup is somewhat more secure than using a different blockchain or a dedicated MKB to publish data, since it does not require to trust any validation network than the main blockchain used (in this case Ethereum). Buterin's "zk rollup" design uses zero-knowledge non-interactive proofs to ensure the integrity of the side-chain. The Plasma Group proposed an "optimistic rollup" that uses interactive proofs, following the model we proposed.

Building a MKB requires an economic validation network, akin to the networks used to validate blockchains. However, there are slight differences that make the design different and useful:

- The notion of Mutual Knowledge is in contrast with Common Knowledge, information that everyone knows that everyone knows about, etc.

- Common Knowledge is what the Consensus algorithm of a blockchain creates; it is a stronger property than Mutual Knowledge and intrinsically slower and more expensive to create.
- Common Knowledge notably requires all participants to synchronize with each other, whereas Mutual Knowledge can be done fully in parallel, without participants having to talk to each other, only with the originator of the information.
- This is why Mutual Knowledge can be achieved much faster and/or much cheaper than Common Knowledge and at larger scale (higher throughput and lower latency).
- Mutual Knowledge can thus be used to scale transactions by ensuring the data is available to all parties. Dispute resolution still involves a blockchain consensus and its Common Knowledge. But regular transactions and dispute detection only require Mutual Knowledge.
- Some functions of the MKB still require Common Knowledge: any proof-of-stake or proof-of-work system to establish who is the current MKB "committee" requires a regular blockchain. Either an existing blockchain can be used (e.g. Ethereum), or the same network that validates the MKB can also validate its own regular blockchain (with its own scalability limits), in addition to making data available in a scalable way.

DApps with More Than Two Parties

There is a notable complication when using Game Semantics in case a DApp has more than two participants.

Consider a DApp where more than two participants temporarily or permanently pool their assets in a common fund. If a thief makes a claim withdraw a large amount of money, normally some fund manager will dispute the claim. The two parties will use a verification game to convince the "smart judge" that one party is dishonest. At the end, the "smart judge" will reject the claim and condemn the thief to cover all legal fees. But what if the manager fails to properly dispute the claim? The manager could be colluding with the thief; they could be incompetent; their system may have been hacked; they could otherwise be incapacited. Then, the judge will rule in favor of the thief. The assets under control of the DApp will be depleted to the detriment of the other participants in the DApp, who didn't get to partake in the dispute.

To prevent this kind of abuse, any participant in a DApp may partake in a "smart lawsuit" started by other participants. Whichever participant makes a claim or an argument, any other participant can make counter-claims or counter-arguments, and not just a designated "fund manager". If a party to the dispute is found to have deliberately lost an argument that it could have won, it will be punished as well as the wrongful claimant on the other side. For lawsuits that have more than two steps, this means that the "smart contract" for every lawsuit will actively maintain a branching tree of on-going arguments from an arbitrary number of participants, instead of simply a sequence of arguments from two parties. Also, to prevent double-jeopardy or double-withdrawal, only the first participant who files a complete winning argument will prevail.

Claimants who made wrong claims are punished. Those who filed a winning argument too late are just ignored.

For each claim that isn't directly verifiable (last step in the proof), there is a deadline before which all direct counter-claims must be posted. If those counter-claims are not directly verifiable (last step in the proof), then a set of such counter-claims is maintained, that themselves each have a deadline for a counter-counter-claim. Thus, while all direct counterclaims are known after the challenge period, the final outcome of the interactive argument is only known after all the branches of the argument tree have either reached their conclusion or timed out, which can take as much time as the challenge period multiplied by the maximum number of steps in the interaction.

Merkleization and Posting Markets

When translating DApp interaction steps into blockchain messages, a transformation is sometimes applied to *Merkleize* part or all of the application state: the transformed application only stores a recursive digest of the original application state, and the message includes a sufficient fragment of that state to apply the current step, together with digests of the rest of the state, sufficient to establish a *Merkle proof* that the message-reminded state matches the previous state of the application.

This translation is *necessary* on some blockchains such as Bitcoin, that have only use-once UTXOs and no fixed-address stateful contract. Even on a blockchain with fixed-address stateful contracts such as Ethereum or Tezos, this translation is often *advantageous* on contracts with a closed control structure, where it is always clear which participant's turn it is to post the next message, since it then saves on storage cost at no disadvantage to any participants. However, when a contract has an *open* control structure, and anyone can join at any time, or some participants may post an arbitrary number of consecutive message, then this transformation opens the DApp to a "moving target attack": an attacker makes a lot of small changes all the time, faster than other participants can keep up with; the other participants' messages always thus fail to provide a valid Merkle proof; they are prevented from participating and can be made to time out and lose their stake and collateral, despite their good will attempts to participate.

Some mitigations are possible against this kind of attack, but they can be complex: DApps will allow each participant to sign a sub-message that only commits to the new data from their interaction step. Getting this sub-message through as part of a larger message is then delegated to a separate *posting network*, made of nodes that do not need access to the secret key of the DApp participant. This network can itself be made of machines controlled by the same organization as the DApp participant. Or it can be a separate network contracted by the participant to post the message, possibly after a reverse auction for the service price. Or this posting network can itself be a decentralized market incentivized by a fixed fee paid by the participant. Or it can be a combination of the previous. Whichever way, nodes in the posting network would closely follow the moving state of the contract, construct a message with a

correct Merkle proof to accompany the signed sub-message for the DApp interaction step, and get that complete message posted. Miners themselves, or operators with servers close to those of miners, can be expected to provide this service, as that would provide them a latency advantage in close data races. In the end, the network would provide a clearinghouse for legitimate users to compete for space in the next mined block with each other and with attackers, transforming a technical and economic issue into a simple economic issue—the same issue as for posting messages to a blockchain that supports fixed-address contracts, mentioned in the section Safe DApp Runtime.

Appendix B: A Tower of Languages for DApps

As explained in a section above, The *Glow* Architecture is structured around a *Tower of Languages*. This Appendix will describe the Tower as we will be providing it to regular developers, in terms of both the languages and the *transformations* from each language to the next.

The list of languages below is not meant to be exact or exhaustive, but only to be used as a guide. The precise architecture of *Glow* will evolve as we grow the platform to better support the needs of a wider array of users. Indeed, our prototype compiler directly does End-Point Projection from a simplified variant of our DSL to JavaScript, Solidity, and a logical model using Z3. But as we add and maintain features like Observability, as we add backends for many blockchains, and strategies for various *styles* of code generation suited for a wider variety of DApps, as we start to formally verify our tower of compilers, we will implement more of this architecture in terms of a *semantic tower* of languages. Layering *Glow* in terms of many such languages will be instrumental in making *Glow* a robust platform that is and remains auditable and trustworthy even as it grows in functionality, unlike its monolithic rivals.

The *Glow* DSL and End-Point Projection

At the top of our tower is the Domain-Specific Language (DSL) in which we offer our users to write DApps: *Glow* itself, as described in the section Programming Model. *Glow* enables the development of DApps from a single specification. From that specification, we use a technique known as *End-Point Projection* (EPP) to extract:

1. Client code for each participant, in a *client language*.
2. Contract code for each blockchain, in a *contract language*.
3. A verified logical model for the DApp, in a *logical language*.

We call the three languages involved *End-Point Languages*. The *client language* can do local computations, send and receive messages to or from other clients, and send and receive messages to or from the consensus, the latter including transfer of assets. The *contract language* does essentially the same as any other contract language does: limited computations

that anyone can see, that can *verify* the conditions that control the release of assets, though it may not be suitable to *compute* the parameters used during the verification. The *logical language* describes the logical relations between the variable elements of the previous languages, and can express the safety properties required from the DApp.

End-Point Projection in Direct Style

The simplest strategy for End-Point Projection (EPP) works as follows.

First, the program is divided in *interaction steps*: each step is a block of "elementary" actions, such as adding, subtracting or comparing numbers, copying data, as performed by a single participant and/or by the consensus. Actions are grouped together into steps that are as large as possible within these constraints (and possibly other blockchain-dependent constraints), as follows: For each participant to the protocol, in order of participation, we group together all the consecutive actions performed either by the participant or by the consensus into one interaction step, until another participant takes an action, or the end of the protocol is reached. Depending on the target blockchain, calls to simple enough functions, that are guaranteed to be completed within a single interaction step, can be either inlined or translated into function calls in the contract language. However, any more complex control structures have to first be reduced to a data structure instead, using the well-known transformation to Continuation-Passing Style (CPS), also known as Continuation-Passing Transform: the code is then divided into "atomic blocks" that contain no complex function call and no change in participant; complex control structures are implemented by explicit pushing or popping of frames onto and from an explicit control stack; that stack may itself be represented as an array or a linked list, depending on what makes sense at lower layers of abstractions.

Then, for each action to be performed by the participant of an interaction step, the EPP generates:

a. Client code for that participant's client to perform the action as specified.
b. Code in all other participants' clients to check that the action is performed correctly or else take corrective action, including invoking the contract.
a. Code in the on-chain contract to verify whether the action was performed correctly and enact appropriate consequences either way.
b. A logical specification of the DApp behavior when the participant performs the action.
c. A logical specification of the DApp behavior when the participant fails to perform it.

For every action to be performed consensually, EPP generates:

a. Code in all participants' clients to perform the action, and/or to monitor the contract as it performs the action, when the action is a transfer of assets.
b. Code in the on-chain contract to perform the action.
c. A logical specification of the DApp behavior as the contract performs the action.

Each interaction step is then directly translated into the step's active participant performing in

sequence all the actions in the step, and finally sending a single message to the contract containing all the data published together with any assets transferred. Depending on the blockchain used and the nature of the DApp, a transformation to Merklize the DApp state may be used at this point, as per above section Merkleization and Posting Markets. The contract will then verify that the message is valid, that each action applies to the current state of the contract and satisfies all requirements, and will take all the appropriate reactions as specified by the EPP above, in a single transaction. The other participants watch the blockchain to see when the message is posted and the interaction step completed; they don't have to do the consensual checks, since the contract already enforces them, but may still include assertions to detect bugs (though it is too late to fix them). Each interaction step comes with a deadline, typically based on a time limit since the previous interaction step. If the active participant fails to complete his interaction step before his deadline, the other participants can send a message to contract to declare him in default; the defaulting participant thereby forfeits his table stakes and his collateral in favor of the other participants; in case the interaction has more than two participants, a part of the collateral is specially reserved to the one who went to the trouble of declaring him in default.

The grouping into interaction steps is a correct transformation precisely because the consequences are the same regardless of which action in the group a participant stops cooperating at. Thus the "game" outcomes are the same. The grouping is a useful transformation because it minimizes the amount of messages exchanged between participants and with the blockchain, thus minimizing cost and latency.

Finally, collaterals can be automatically added to the protocol: for each message sent by new participant, if there is a future message that the same participant may have to send as part of the interaction protocol, then the participant must deposit a collateral that will ensure their continued cooperation. The precise amount of collateral for each participant is a parameter to be agreed upon between participants as part of negotiating the terms of the interaction, as per above section on Semi-Automatically Estimating Proper Collateral.

This direct style transformation of code already is the composition of many simpler transformations involving several languages or fragments of languages. It already benefits from the *Glow* architecture to minimize the complexity of the overall system by organizing it layers that can be audited separately. As we shall see immediately below, some of these layers can be reused and combined in ways that make this separate audit radically simpler than if not using layers.

End-Point Projection in State Channel Style

Instead of translating interaction steps directly into messages to the blockchain, we instead translate them into messages to a *generalized state channel*. This strategy, that we call *state channel style*, is particularly useful for applications within a small set of participants that seldom changes: In those cases, it can drastically cut both operating costs and latency by displacing

most message exchange off the slow and expensive blockchain and into fast and cheap private communication channels.

In *state channel style*, the blockchain normally only sees (1) the initial *binding* messages by which the participants agree to the contract and contribute their initial stakes, and (2) a final *settlement* message by which the assets under control of the contract are redistributed between the participants according to mutually agreed quantities. In between, intermediate settlements can be used to add or remove assets or participants to or from the DApp, according to terms mutually agreed upon by all participants old and new. All these blockchain messages are as slow and expensive as any blockchain message, though their structure can be kept minimal as far as messages for a DApp with the given number of participants go.

The real DApp happens off-chain, through message exchanged between participants: every time a participant makes an *interaction step*, they compute the next state of the DApp and sign it, and send this signature along with a message describing their step. Once every participant has signed the message, the new state is accepted as valid. If a state channel contains more than a handful of participants (which requires some modicum of trust in each and every other participant remaining active, online and honest), a Schnorr signature may be used to save on gas when validating regular state updates and settlements where everyone cooperates. Individual participant signatures are still required for adversarial message challenges and updates, and for a simpler multisig implementation used when there are very few participants (two to four or so). Until a state is signed by everyone, the previous state is valid. There may be a short period of time when both states may be valid, as the message is “in transit”: the sender and intermediate signatories do not know whether the last signatory will sign and send back the message until he does; but all signatories have already accepted both states, so either state is fine. At the very worst, the sender may have to post his message directly to the smart contract, as above, to ensure the new state is validated.

What is not fine is when some participant stops cooperating: for whatever reason, they stop sending and signing the messages they are supposed to send, or signing the messages they are supposed to receive. That is when the other participants will post some message to the blockchain smart contract to denounce them as uncooperative, and challenging them to resume cooperation or be timed out. The challenge can be met by posting to the blockchain smart contract a more recent state update where everyone has started cooperating. Or, when cooperative communication between parties has broken down, it can be met by posting an interaction step message directly to the contract—often followed by a challenge to the other party, accusing *them* back with bad faith. When communication breaks down, all parties that remain active will seek to time out those that have stopped cooperating. Failing that, each participant can get out, by completing any on-going transactions, reaching a point when they have nothing left on the table or as collateral, and they can exit the DApp with whatever assets they fully own in the DApp’s private ledger. If all parties cooperate, state channels can vastly improve the cost and latency of a DApp. When some parties are playing dumb, the state channel only add overhead. But whether other parties cooperate, fail, or play dumb, each

participant who keeps following the protocol is guaranteed to never lose the assets they own, and at worse to only waste a predictably bounded amout of time and gas at getting out of the DApp. It is still a bad idea to enter a DApp with participants you can't be reasonably trusted to cooperate, but it is not a vastly losing proposition.

Participants in a state channel must negotiate proper collateral after having agreed on which DApp to use and before to actually start the state channel. They may revise this collateral after the DApp is complete, or before a new DApp is started. This does not require a source transformation to add collaterals to the interaction, as in the direct style.

The state of a DApp in state channel style contains:

1. A nonce that identifies the current session, and changes with each instance of a DApp and each intermediate settlement within the DApp—in particular whenever a participant joins or leaves a DApp.
2. A clock or sequence number within that session, that allows the smart contract to distinguish between older and newer states, and reject older states.
3. The addresses of the participants on the blockchain, as an array; later within the DApp, each participant is identified by his index into that array.
4. A private ledger describing the assets fully owned by each participant and the assets held in escrow as a collateral for each of them, typically as an array or associative table indexed by the participants and by each kind of asset supported by the DApp.
5. For each participant, some status indicating whether this participant is active, has already been flagged as timed out, or is being challenged to post before some given deadline.
6. A "computation state" describing the state of the interaction: the current "code pointer", the value bound to each variable, every control frame in the more complex function call structure of the DApp; possibly a structure made of the states of several computations being run in parallel. This "computation state" describes how the "table stakes" that are not currently fully owned by anyone will later be distributed, depending on interaction steps taken by each participant. Often, this "computation state" will include a mapping from the roles played in the current interaction of sub-interaction to the index of the participant in the ledger.

State Channel Style starts similarly to direct style, but adds some extra transformations: Interaction steps are obtained the very same way as in direct style, by grouping consecutive actions by a single participant and reactions by the consensus, after continuation-passing transform if needed. End-Point Projection of individual actions is mostly the same, except that the transfer of assets is represented by an explicit private ledger within the DApp. A previously open state channel can be used to run any number of DApps as part of the same session, or through any number of sessions; to facilitate this versatility, when implementing a DApp in state channel style, the generated code will include an indirection from the role played in the current interaction, to the index of the participant in the DApp, to the address of the participant. The Merkleization transformation is always used, since it is needed for signing messages anyway.

Now the most complex difference in how code is generated is in sending and receiving messages.

When sending a message, a participant first tries to send it the “normal” way, by signing the new state together with a message, and sending that to everyone. If some other participant ceases to cooperate, the current participant can try to exit the interaction, if already in an acceptable state; or else the failing participant is denounced as uncooperative while messages are sent directly to the blockchain smart contract to reach a stable point from which the current participant can exit. When receiving a message, each participant monitors the private communication channels as well as the blockchain. If a message is received privately, it is validated before it is signed; the other participant is flagged as dishonest if it sends and signs invalid states. Then it is signed in turn and propagated to the next participant. If a message is received directly on the blockchain smart contract, different actions may be taken: If the message describes an obsolete state, the current participant may post a message to supersede it (or the current participant might just accept it, if that state is advantageous to them). Care is taken to detect and respond to any challenge posted on the blockchain. Each honest participant will always try to resume consensual communication, but otherwise fallback to using the blockchain smart contract and getting out of the DApp. Failing or dishonest participants are typically added to black lists by the client software of other participants, who will exit the current interaction and refuse to partake in state channels with them anymore.

There is a special case when a state channel only binds to players: the table stakes and collateral of a failing party are all completely and easily transferred to the other party that still follows the Protocol. With more than two players, the code in the DApp will have to provide an exception handler detailing what happens to the table stakes partially held by the failing participant, and to the collateral of the failing participant. Typically, the challenging and denouncing participants get part of that collateral, whereas the table stakes may be distributed between all remaining participants, either evenly, or according to some proportion to other table stakes. The cost of computing this distribution may itself be covered by another share of the same collateral, and again paid to the participant running the computation. A convention for who normally gets to denounce whom at what point, and collect which fees, can itself be enforced through staggered deadline for “the expected participant does it vs any other participant does it”. Then, it becomes the mutual interest of all participants to just sign a new settlement that acknowledges the expected outcome, rather than there being a wasteful rivalry wherein participants are incentivized to pay for the computation the hard way.

Generalized state channels can also be chained in the style of the Bitcoin Lightning Network or of the Celer Network: state channels can be chained, whereby A has a state channel with B, who shares one with C, etc. Fully owned assets can be transferred along such a route, by each pair of participants signing conditional transfers in a first phase, then in a second phase revealing the secret that makes each of them valid. Computational state can also be transferred along the way, also by signing conditional transfers. One problem though, is what happens when someone along the way stops cooperating: indeed, since messages can be in limbo when

it isn't clear whether they will be signed or not, it can take a complete timeout cycle before a channel knows whether or not the message was posted in the next channel, and what corrective action to take if any. This means that the timeouts of each state channel must be long enough to cover the timeouts of all the remaining state channels in the chain, and then some time for taking action. The one-week timeout for the Lightning Network is very large and cumbersome to deal with, and doesn't lend itself to easily extending the timeout in parts of the route. Even with shorter timeout periods and possible adaptation of timeouts to a channel's part in a route, existing state channel networks don't seem ready for the correct operation of more than trivial DApps along trivial routes.

Using a Sidechain

Some sidechains, such as those from SKALE, require the users to fully trust their validation network, at which point they otherwise behave like a regular blockchain, just more scalable. These sidechains do not require special code generation for DApps, just special configuration at runtime. On the other hand, some sidechains in the style of Plasma can survive events involving the operator failing to cooperate, and participants either reverting back to operating on the main chain (typically Ethereum), or somehow migrating en masse to another Plasma chain with a different operator. In these cases, using the Plasma chain appears at some level of abstraction like using a regular blockchain, but underneath, requires users to follow a more complex protocol that includes fallback to the main chain as a possibility, and race conditions between normal and exceptional behavior.

The details will vary with the fine interfaces of whichever sidechain or Plasma chain is used, and are out of scope for this document. Nevertheless, there will be further translation layers involved in using some sidechains. These layers can themselves be combined with both direct style and state channel style, resulting in an exponential growth in the number of useful combinations of transformations.

Enabling Machines for Larger DApps

Smart contracts are costly to register and use, and only more so as their code grows. This imposes strong limits on the size and complexity of DApps and makes them onerous to use. The following techniques can lift those limits, by introducing additional layers of languages, on top of "state channel style".

With "space compression", state channels can be used to execute a small fragment of code on an arbitrarily large data structures. Participants' computers manipulate a state made of data structures merklized into a Directed Acyclic Graph (DAG). The on-chain contract only stores the merkle root of that DAG. Whenever a participant wants to verify a step of computation, he will reveal to the contract a merkle proof for the small subset of DAG used; then the contract will run the computation, verifying that it has the correct result and only uses the revealed DAG fragment. Each step will be compiled in several ways:

1. In an optimized way, to run efficiently on each client as long as there is no dispute.
2. In a merkleized way, to run not too inefficiently on each client, yet producing merkle proofs along the way, that can be included in challenges to determine which step of the computation is actually being disputed.
3. In a tracing merkleized way, that runs inefficiently on a client but for that single step of the computation under dispute, recording along the way the smallest subset of the merkleized DAG sufficient to reveal the step.
4. In a verify-only way, for the on-chain contract to check the correctness of the step on a reconstituted minimal subset of the DAG. Due to limitations in some contract platforms, this may also involve the user publishing intermediate results (such as multiplications) that the contract can verify but not compute, in addition to merkleized data.

Space compression can notably be used to verify the validity of a large data structure: each participant will run the computation off-chain using the optimized. If an anomaly is detected or a dispute otherwise arises, the participants will run the merkleized version to challenge and/or the tracing version to partake in a dispute that will establish the validity of the computation. In some cases, the protocol may mandate a run using the merkleized version to save a round of challenges in the adversarial case at the expense of creating an additional burden in the cooperative case.

With “time compression”, state channels contract can verify computations about arbitrarily long computations. In case of dispute about a long computation, the computation is transformed into a trace of execution. Then a binary search is made through this trace, either interactively (which can involve a lot of challenge periods), or less interactively (e.g. by posting the large trace to a trusted availability engine), to determine the earliest step in that trace at which the two parties disagree about the execution. The problem has then been reduced to the space compression issue above.

By using a simple enough virtual machine, the execution of which can be verified by one simple contract, DApp code can be encoded as data on the contract side. Since the “space compression” technique above makes it possible to work with arbitrarily large data, it then becomes possible to work with arbitrarily large code. Typically, this virtual machine could be RISCV, WebAssembly, or perhaps Michelson or something DApp-specific, so that developers can use their usual toolchains, and be able to deploy arbitrary code in a DApp.

In particular, running the native validation code of other blockchains, by compiling them to RISCV then running that code in a VM as above, can crucially enable bridges between multiple blockchains. As another example, complex financial computations can be secured that involve large computations on large data: a program in an arbitrary language can be compiled to some VM, to describe what formula will be used to compute a strike price for some futures contract.

Blockchain-specific complications

The above languages still remain somewhat abstract and domain-specific. As they are translated into code that is executable for various blockchains, several intermediate languages are involved, that may vary with the target blockchain. For instance, Ethereum uses a virtual machine called the EVM for contracts, and an interface known as web3 for clients; they are very different from the family of “scripts” used by diverse forks of Bitcoin and the accompanying client interfaces, or from the equivalent languages used by Cardano, Tezos, Cosmos, Hashgraph, EOS, etc. As we add to the list of supported platforms, we will add more layers and transformations that can take advantage of each platforms’ advantages and can cope with their defects.

For instance, the limited scripting language of Bitcoin does not allow UTXOs to commit on computed outputs. Thus, “covenants” can only be written that at each step only has only a relatively small number of final outcomes.

Manu bitcoin sidechains and bitcoin forks provide a way to commit to computed output, such with the Bitcoin Cash opcode OP_CHECKDATASIG or the Blockstream Elements opcode OP_CHECKSIGFROMSTACK. This allows for arbitrary elaborate DApps to successfully run that bypass the limitations of Bitcoin proper. Still, the administrative work required to actually verify the commitment to properly computed outputs easily wastes half of the limited amount of opcodes allowed per UTXO. Interaction steps may have to be further divided into small steps to run on those blockchains. Several interaction steps can be chained within a single block of the blockchain, though there again with limitations: for instance, Bitcoin Cash will only allow a chain of 25 such frames per block mined.

When compiling simple applications for Tezos, we may find that Michelson’s very simple type system might be enough for many DApps to be directly translated to it, but that it is too limited for more advanced DApps. One solution would be to implement some virtual machine as above in a suitably restricted dialect of our contract language that allows for direct translation, then use that virtual machine for DApps that cannot fit those restrictions.

In many cases, it can be useful to target a finite state machine, such as used by Bitstream’s language Simplicity, but more importantly, such as used by various zero-knowledge proof systems. Then, the correct execution of contracts can be consensually verified by public blockchain validation networks, without anyone but the signatories of the contract being able to see what are the terms being validated. The constraints of these technologies will lead to corresponding layers of language restrictions and transformations between restricted languages.

Proving Correctness of our Transformations

When we eventually prove the correctness of our transformations between layers, we will use a sufficiently expressive logical language that is appropriate for precise formal reasoning about programs. The most likely candidates are variants of the dependently-typed lambda-calculus where all functions are pure and total: the Coq proof assistant, Lean, F*, Agda, Idris, Cur, ATS, some extensions to Haskell, and more. These languages are often far from ideal to directly write programs into, but often allow for embedding of other languages inside them that can be reasoned about. Thus, we can establish direct translations between each of the other languages we use and a formal language embedded in our logic language. Then we can prove correctness of transformations between our languages, culminating in proving the correctness of complete sequences of transformations from *Glow* to each of combinations of client and contract languages for each target blockchain.

Exactly which platform we choose will depend on how their respective ecosystems evolve, how well they can integrate with the rest of our workflow, and most importantly, what senior talent we can recruit. The endeavor of proving correctness of our platform is an expensive long haul project. We will only start that project after we have established the commercial value of the success of said endeavor, in convincing financial institutions to use our platform rather than competing ones. We may even start with competing teams using different tools on different or similar parts of our system, to see which brings the likeliest promise of completing a proof for our entire system within budget.