Thanks to Ben Jones and Karl Floersch for clarifying discussions of the problem, and to Yan Zhang and Barnabé Monnot for helpful feedback on the presentation.

Dispute protocols are fundamentally about proving things: a proposition A

about events off chain needs to be decided on chain, so an adversarial game is set up on chain between a player affirming A

and a player denying A

whose rules have been designed so that the correct player has a winning strategy.

Example.

Alice and Bob want a system to transfer an asset back and forth arbitrarily many times without any transactions on chain except for an initial deposit and a final withdrawal. To do this, they agree that the asset will be held on chain by a contract C

, and that transferring ownership of the asset off chain will be signified by sending the other party a signature of the current time. Ownership of the asset by (respectively) Alice and Bob is signified by the propositions

\newcommand{\pro}{\textsf{P}} \newcommand{\op}{\textsf{O}} \newcommand{\attack}[2]{\mathsf{?}{#1}#2} \newcommand{\sep}{\quad\big|\quad} \newcommand{\funI}[2]{\texttt{\} \; #1 \;\, \texttt{=>} \;\, #2} \newcommand{\funE}[2]{#1 \; #2} \newcommand{\pairI}[2]{\langle #1 \, , \, #2 \rangle} \newcommand{\pairE}[4]{\texttt{match}\land \;\, #1 \;\, \texttt{with} \;\, \pairI{#2}{#3} \;\, \texttt{=>} \;\, #4} \newcommand{\eitherIL}[1]{\iota_1 \langle #1 \rangle} \newcommand{\eitherIR}[1]{\iota_2 \langle #1 \rangle} \newcommand{\eitherE}[5]{\texttt{match}\lor \;\, #1 \;\, \texttt{with} \;\, \eitherIL{#2} \;\, \texttt{=>} \;\, #3 \;\, \texttt{|} \;\, \eitherIR{#4} \;\, \texttt{=>} \;\, #5} \newcommand{\unitI}{\langle \rangle} \newcommand{\unitE}[2]{\texttt{match}\top \;\, #1 \;\, \texttt{with} \;\, \unitI \;\, \texttt{=>} \;\, #2} \newcommand{\voidE}[1]{\texttt{match}_\bot \;\, #1} \newcommand{\sub}[2]{#1[#2]} \newcommand{\subitem}[2]{#1 \mapsto #2} \newcommand{\subsep}{\;\, ; \;\,} \newcommand{\subi}[3]{\sub{#1}{\subitem{#2}{#3}}} \newcommand{\subii}[5]{\sub{#1}{\subitem{#2}{#3} \subsep \subitem{#4}{#5}}} \newcommand{\betaEq}{\;\; \sim \;\;} \newcommand{\val}[1]{#1 \;\;\; \textsf{value}} \newcommand{\step}[2]{#1 \;\; \rightsquigarrow \;\; #2} \newcommand{\hole}[1]{\texttt{?}#1} \newcommand{\holecl}[2]{\hole{#1}{#2}} \newcommand{\stuck}[2]{#1 \;\;\; \textsf{stuck} \;\;\; \hole{#2}} \begin{alignat}{1} A &= \exists t . \text{signed}(\text{Bob}, t) &&\land \lnot \left(\exists t' . t \leq t' \land \text{signed}(\text{Alice}, t') \right) \ B &= \exists t . \text{signed}(\text{Alice}, t) &&\land \lnot \left(\exists t' . t \leq t' \land \text{signed}(\text{Bob}, t') \right) \end{alignat}

and for (say) Alice to withdraw the asset on chain, she must prove the proposition A

by winning the following game, which is adjudicated by the contract C

:

- Alice is asked for a time t

, and a signature by Bob of t

. If Alice fails to provide these in reasonable time, then the game ends and Bob wins; otherwise:

- Bob is asked for a time t'

, where t \leq t'

, and a signature by Alice of t'

. If Bob fails to provide these in reasonable time, then the game ends and Alice wins; otherwise:

- The game ends and Bob wins.

Given the resemblance between the game and the proposition it decides, a natural question one might ask is: Instead of inventing and implementing such games and strategies de novo for each new dispute protocol, can we derive them automatically from the propositions and proofs they are fundamentally about?

This question was first asked and investigated by Ben Jones and Karl Floersch in their work on "predicate contracts" and later "optimistic game semantics". Inspired by dialogical logic, their work specifies a "universal adjudicator" that can interpret any proposition as a game.

This post proposes a different approach to the same question, based on type theory and ludics, which enables us to specify simultaneously a "universal adjudicator" and a corresponding "universal advocate" that can interpret any proof

of a proposition as a winning strategy

for the corresponding game.

The first two sections are mostly review of background material (with some liberties taken to keep the presentation simple). Readers already familiar with propositions-as-types can skip directly to the third and last section titled “Games and strategies”.

Dialogical logic

Dialogical logic interprets a first-order logic proposition A

as a dialogue game

between a proponent

$\pro$

and an opponent

$\op$

. The game starts with $\pro$

asserting A

, after which players take alternating turns either attacking a past assertion of their adversary or defending against a past attack by their adversary, both of which may involve making new assertions, subject to the constraint that an atomic proposition if asserted must be true. The possible attacks and defenses are listed in the following table. A player loses the game on their turn if no “productive” moves are possible.

Assertion

Attack

Defense

Comment

$\lnot A$

$\attack{\lnot}{}$

assert A

no defense possible

$A \to B$

$\attack{\to}{}$

assert A

assert B

$A_1 \land A_2$

$\attack{\land}{(i)}$

assert A_i

attacker’s choice of i

$A_1 \lor A_2$

$\attack{\lor}{}$

assert A_i

defender’s choice of i

$\forall x . A$

$\attack{\forall}{(t)}$

assert $A[x \mapsto t]$

attacker's choice of t

\exists x . A

\attack{\exists}{}

assert A[x \mapsto t]

defender's choice of t

Example.

A play of the game corresponding to \lnot (A \land \lnot A)

could go as follows (assuming the atomic proposition A

is true; otherwise \op

would lose the game on turn 4):

1. \pro

starts, asserting \lnot (A \land \lnot A)

1. \op

attacks move 1's assertion with \attack{\lnot}{}

, asserting A \land \lnot A

1. \pro

attacks move 2's assertion with \attack{\land}{(1)}

1. \op

defends against move 3's attack, asserting A

1. \pro

attacks move 2's assertion with \attack{\land}{(2)}

1. \op

defends against move 5's attack, asserting \lnot A

1. \pro

attacks move 6's assertion with \attack{\lnot}{}

, asserting A

1. \op

loses the game

These games have the necessary property that there is a winning \pro

strategy for any true proposition and a winning \op

strategy for any false proposition.

Type theory

If we want to write proofs and interpret them as winning strategies, we need a formal proof system. Arguably the best choice of formal proof system today is to be found in type theory. Most state-of-the-art software for theorem proving (e.g. Coq, Lean, Idris) is based on it, and even programmers with no background in formal logic but with some experience in functional languages will find it familiar.

The philosophy of type theory in essence is that proving a proposition means constructing a mathematical object that makes its truth evident. For example, proving that two things are isomorphic means constructing an isomorphism between them. In general, every proposition is understood as specifying a type of mathematical object to construct (which of course is sometimes possible and sometimes not).

Proposition

Evidence

Set analogy

A \to B

procedure to transform evidence for A

into evidence for B

B ^ A

A \land B

both evidence for A

and evidence for B

A \times B

A \lor B

either evidence for A

or evidence for B

A + B

\top

trivial

1

\bot

impossible

0

(\lnot A

is understood to be synonymous with A \to \bot

.)

A type theory (with the indefinite article) is a formal language for expressing these constructions. There are many different type theories, and there is no all-encompassing definition of what exactly it means to be a type theory (as with the concept of “space” in mathematics), but generally speaking a type theory looks something like the following.

There is a grammar of types

(here just enough for propositional logic):

\begin{alignat}*{1} &\text{Types} \; A,B,C \; ::= \ &\qquad \; A \to B \sep A \land B \sep \top \sep A \lor B \sep \bot \end{alignat}*

and a grammar of expressions

or terms

representing constructions (treated as abstract binding trees):

\begin{alignat}*{1} &\text{Variables} \; x, y, z \ &\text{Expressions} \; e, v \; ::= \ &\qquad \; x \ &\sep \funI{x}{e} &\sep \funE{e_0}{e} \ &\sep \pairI{e_1}{e_2} &\sep \pairE{e_0}{x}{y}{e} \ &\sep \unitI &\sep \unitE{e_0}{e} \ &\sep \eitherIL{e} \sep \eitherIR{e} &\sep \eitherE{e_0}{x}{e_1}{y}{e_2} \ & &\sep \voidE{e_0} \end{alignat}*

The typing relation

or typing judgment

"e

is of type A

", written

$e : A,$

states that the construction expressed by the term e

satisfies the specification expressed by the type A

. A construction can be parameterized by a set of variables $x_1, x_2, \ldots$

respectively assumed to be of some types $A_1, A_2, \ldots$

, so the typing judgment is generalized to "e

is of type A

in context $\Gamma = x_1 : A_1, x_2 : A_2, \ldots$

", written

$\Gamma \vdash e : A.$

Typing rules

define when a typing judgment (the conclusion

, appearing below the line) can be derived from other typing judgments (the premises

, appearing above the line).

Introduction rules

govern how to produce things of a type:

$$\frac {\Gamma, x : A \vdash e : B} {\Gamma \vdash \funI{x}{e} : A \to B} \to_\text{I}$$

$$\frac {\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B} {\Gamma \vdash \pairI{e_1}{e_2} : A \land B} \land_\text{I}$$

$$\frac {} {\Gamma \vdash \unitI : \top} \top_\text{I}$$

$$\frac {\Gamma \vdash e : A} {\Gamma \vdash \eitherIL{e} : A \lor B} \lor_\text{I1} \qquad \frac {\Gamma \vdash e : B} {\Gamma \vdash \eitherIR{e} : A \lor B} \lor_\text{I2}$$

Elimination rules

govern how to consume things of a type:

$$\frac {\Gamma \vdash e_0 : A \to B \qquad \Gamma \vdash e : A} {\Gamma \vdash \funE{e_0}{e} : B} \to_\text{E}$$

$$\frac {\Gamma \vdash e_0 : A \land B \qquad \Gamma, x : A, y : B \vdash e : C} {\Gamma \vdash \pairE{e_0}{x}{y}{e} : C} \land_\text{E}$$

$$\frac {\Gamma \vdash e_0 : \top \qquad \Gamma \vdash e : C} {\Gamma \vdash \unitE{e_0}{e} : C} \top_\text{E}$$

$$\frac {\Gamma \vdash e_0 : A \lor B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C} {\Gamma \vdash \eitherE{e_0}{x}{e_1}{y}{e_2} : C} \lor_\text{E}$$

$$\frac {\Gamma \vdash e_0 : \bot} {\Gamma \vdash \voidE{e_0} : C} \bot_\text{E}$$

Finally, structural rules

govern general features like the use of variables:

$$\frac {} {\Gamma, x : A \vdash x : A} \, \text{hyp}$$

Example.

Here is a term proving $\lnot(A \land \lnot A)$

:

$$\funI{p}{(\pairE{p}{x}{f}{\funE{f}{x}})}$$

and here is its typing derivation (suppressing unused hypotheses in contexts to save space):

\cfrac { \cfrac { \cfrac {} { p : A \land (A \to \bot) \vdash p : A \land (A \to \bot) } \, \text{hyp} \qquad \cfrac { \cfrac {} { \ldots, f : A \to \bot \vdash f : A \to \bot } \, \text{hyp} \qquad \cfrac {} { \ldots, x : A \vdash x : A } \, \text{hyp} } { \ldots, x : A, f : A \to \bot \vdash \funE{f}{x} : \bot } \to_\text{E} } { p : A \land (A \to \bot) \vdash \pairE{p}{x}{f}{\funE{f}{x}} : \bot } \land_\text{E} } { \vdash \funI{p}{(\pairE{p}{x}{f}{\funE{f}{x}})} : (A \land (A \to \bot)) \to \bot } \to_\text{I}

Computation

When different terms express the same construction is a notoriously subtle question, but at minimum there is a congruence, historically named \beta

-reduction

, generated by “cancellations” of introductions and eliminations:

\begin{alignat*}{1} \funE{(\funI{x}{e'})}{e} &\betaEq \subi{e'}{x}{e} \ \pairE{\pairI{e_1}{e_2}}{x}{y}{e'} &\betaEq \subii{e'}{x}{e_1}{y}{e_2} \ \unitE{\unitI}{e'} &\betaEq e' \ \eitherE{\eitherIL{e}}{x}{e_1'}{y}{e_2'} &\betaEq \subi{e_1'}{x}{e} \ \eitherE{\eitherIR{e}}{x}{e_1'}{y}{e_2'} &\betaEq \subi{e_2'}{x}{e} \end{alignat*}

where \subi{e'}{x}{e}

denotes the (capture-avoiding) substitution of e

for x

in e'

.

Simplifying terms with \beta

-reduction converts an implicit

representation of a construction (like “1 + 7 + 49 + 343”) into an explicit

one (like “400”), a process which can be thought of philosophically as simulating the mental act of realizing the construction.

A (closed) term that is fully reduced (ignoring subterms under binders) is a value

:

\frac {} {\val{\funI{x}{e}}}

\frac {\val{v_1} \qquad \val{v_2}} {\val{\pairI{v_1}{v_2}}}

\frac {} {\val{\unitI}}

\frac {\val{v}} {\val{\eitherIL{v}}} \qquad \frac {\val{v}} {\val{\eitherIR{v}}}

The next step of reduction is not unique in general, and is disambiguated with a reduction relation

or operational semantics

:

\frac {\step{e_0}{e_0'}} {\step{\funE{e_0}{e}}{\funE{e_0'}{e}}} \qquad \frac {\val{v_0} \qquad \step{e}{e'}} {\step{\funE{v_0}{e}}{\funE{v_0}{e'}}} \qquad \frac {\val{v}} {\step{\funE{(\funI{x}{e})}{v}}{\subi{e}{x}{v}}}

\frac {\step{e_1}{e_1'}} {\step{\pairI{e_1}{e_2}}{\pairI{e_1'}{e_2}}} \qquad \frac {\val{v_1} \qquad \step{e_2}{e_2'}} {\step{\pairI{v_1}{e_2}}{\pairI{v_1}{e_2'}}}

\frac {\step{e_0}{e_0'}} {\step{\pairE{e_0}{x}{y}{e}}{\pairE{e_0'}{x}{y}{e}}}

\frac {\val{v_1} \qquad \val{v_2}} {\step{\pairE{\pairI{v_1}{v_2}}{x}{y}{e}}{\subii{e}{x}{v_1}{y}{v_2}}}

\frac {\step{e_0}{e_0'}} {\step{\unitE{e_0}{e}}{\unitE{e_0'}{e}}}

\frac {} {\step{\unitE{\unitI}{e}}{e}}

\frac {\step{e}{e'}} {\step{\eitherIL{e}}{\eitherIL{e'}}} \qquad \frac {\step{e}{e'}} {\step{\eitherIR{e}}{\eitherIR{e'}}}

\frac {\step{e_0}{e_0'}} {\step{\eitherE{e_0}{x}{e_1}{y}{e_2}}{\eitherE{e_0'}{x}{e_1}{y}{e_2}}}

$$\frac {\val{v}} {\step{\eitherE{\eitherIL{v}}{x}{e_1}{y}{e_2}}{\subi{e_1}{x}{v}}}$$

$$\frac {\val{v}} {\step{\eitherE{\eitherIR{v}}{x}{e_1}{y}{e_2}}{\subi{e_2}{x}{v}}}$$

$$\frac {\step{e_0}{e_0'}} {\step{\voidE{e_0}}{\voidE{e_0'}}}$$

All of these definitions together ensure that well-typed terms eventually reduce to a β-equivalent value of the same type.

Theorem. If $e : A$ then $e \rightsquigarrow \ldots \rightsquigarrow e'$ where $e \sim e'$ and $e' : A$ and $e' \textsf{ value}$.

This process can be automated, effectively making a type theory a (terminating!) programming language. This is the celebrated propositions-as-types/proofs-as-programs correspondence.

Games and strategies

With a formal proof system now in hand, we could attempt to transform proofs (i.e. terms) into strategies for the dialogue games described previously. This can indeed be done with more machinery, however I want to propose instead a simpler approach inspired by ludics, which takes advantage of the intrinsic computational aspect of type theory to interpret types as games that are naturally suited to interpreting terms as strategies.

To recapitulate, the goal is to transform:

- a type A

into a game between players $\pro$ and $\op$

- a term $e_\pro : A$

into a winning strategy for $\pro$

- a term $e_\op : A \to \bot$

into a winning strategy for $\op$

A trivial noninteractive solution would be simply to require players to provide a term to win, but this runs into problems with atomic propositions about real events, for example “a hash preimage of $\texttt{0x123}$ has been revealed”. Since evidence for this event would be the preimage in question, it naturally corresponds to a primitive type S (for “secret”) whose values are hash preimages of $\texttt{0x123}$. The problem with the trivial game for this type is that while $\pro$ can win when it is true, $\op$ can never win even when it is false because there is no term of type $S \to \bot$.

From a computational point of view, we can still ask if the behavior specified by this type can be “safely” implemented by “unsafe” code (like Rust’s unsafe blocks). After all, as long as constructing a value of type S is really impossible, if there was a function $\funI{x}{e}$

of type $S \to \bot$

, then the body e

would be unreachable code anyway, so even ill-formed code could not cause a runtime error.

To allow such implementations, we introduce an “unsafe primitive” called a hole

(corresponding in ludics to the daimon

$\maltese$

):

$$\begin{alignat}{1} &\bbox[yellow]{\text{Holes} \; \hole{h}} \\ &\text{Variables} \; x, y, z \\ &\text{Expressions} \; e, v \; ::= \\ &\qquad \; x &\bbox[yellow]{\sep \hole{h}} \\ &\sep \funI{x}{e} &\sep \funE{e_0}{e} \\ &\sep \pairI{e_1}{e_2} &\sep \pairE{e_0}{x}{y}{e} \\ &\sep \unitI &\sep \unitE{e_0}{e} \\ &\sep \eitherIL{e} \sep \eitherIR{e} &\sep \eitherE{e_0}{x}{e_1}{y}{e_2} \\ & &\sep \voidE{e_0} \end{alignat}$$

A hole can occur at any type:

$$\frac {} {\Gamma \vdash \hole{h} : A} \, \text{hole}$$

and its behavior is to abort the evaluation (like a fatal exception):

$$\frac {} {\stuck{\hole{h}}{h}}$$

$$\frac {\stuck{e_0}{h}} {\stuck{\funE{e_0}{e}}{h}} \qquad \frac {\val{v_0} \qquad \stuck{e}{h}} {\stuck{\funE{v_0}{e}}{h}}$$

$$\frac {\stuck{e_1}{h}} {\stuck{\pairI{e_1}{e_2}}{h}} \qquad \frac {\val{v_1} \qquad \stuck{e_2}{h}} {\stuck{\pairI{v_1}{e_2}}{h}}$$

$$\frac {\stuck{e_0}{h}} {\stuck{\pairE{e_0}{x}{y}{e}}{h}}$$

$$\frac {\stuck{e_0}{h}} {\stuck{\unitE{e_0}{e}}{h}}$$

$$\frac {\stuck{e}{h}} {\stuck{\eitherIL{e}}{h}} \qquad \frac {\stuck{e}{h}} {\stuck{\eitherIR{e}}{h}}$$

$$\frac {\stuck{e_0}{h}} {\stuck{\eitherE{e_0}{x}{e_1}{y}{e_2}}{h}}$$

$$\frac {\stuck{e_0}{h}} {\stuck{\voidE{e_0}}{h}}$$

Now terms eventually either reduce to a value or get stuck on a hole.

Theorem.

If $e : A$

then $e \rightsquigarrow \ldots \rightsquigarrow e'$

where $e \sim e'$

and $e' : A$

and either $e' \textsf{ value}$

or $e' \textsf{ stuck } \hole{h}$

for some $\hole{h}$

contained in e

.

Let us say that a term not containing holes is total

, a term possibly containing holes is partial

, and a partial term e

is safe

when the holes it contains are unreachable by evaluation, i.e. there is no partial term e'

containing e that gets stuck on a hole contained in e. In particular, total terms are trivially safe. In the case of the type $S \to \bot$, there is no total term, but if (and only if) no value of type S has been revealed, then there is a safe partial term $\funI{x}{\hole{h}}$.

The intention is that safe partial terms should also yield winning strategies. Now the goal is to transform:

- a type A into a game between players $\pro$ and $\op$
- a safe partial term $e_\pro : A$ into a winning strategy for $\pro$
- a safe partial term $e_\op : A \to \bot$ into a winning strategy for $\op$

This is accomplished by the following game for a type A:

- $\pro$ and $\op$ respectively provide partial terms

$$\begin{alignat}{1} e_\pro &: A \\ e_\op &: A \to \bot \end{alignat}$$

- The partial term $\funE{e_\op}{e_\pro} : \bot$ is reduced until it gets stuck on a hole contained in either $e_\pro$ or $e_\op$ (which necessarily happens since there is no value of type $\bot$ to which it can reduce).
- The player on whose hole the reduction is stuck loses the game.