Poor Man's Axi: DPL-like proofs in Type Theory

Wojciech Kołowski

Layers in Poor Man's Axi

In our original proposal of Poor Man's Axi, the language was split into two layers:

- A programming layer which consists of a strongly-typed functional programming language based on the Simply Typed Lambda Calculus.
- A logical layer which consists of second-order classical logic with equality and some primitives for reasoning by cases and induction (note that it's not full second-order logic, because it only allows quantification over functions, but not over propositions, predicates or relations).

Logic without proofterms

The logic was presented with a bunch of judgements, the most important being the true proposition judgement $\Gamma \mid \Delta \vdash P$. While this presentation does a good job of explaining what the logic is like, it does not address the problem of writing proofs from the perspective of the programmer.

Logic with proofterms

To deal with this problem, we introduce proofterms (also known as proof certificates), which are expressions that will serve as the “proofs” that the user will write. The manipulation of judgements which establish their correctness (which we will describe soon) is left to the language’s proof checker.

Proofterms

Proofterms (here P are propositions, t are terms, x are variables):

e ::=

P | **true** | **exfalso** e

assume P **in** e | **modus-ponens** e_1 e_2 |

suppose-absurd P **in** e | **absurd** e_1 e_2 |

both e_1 e_2 | **left-and** e | **right-and** e |

equivalence e_1 e_2 | **left-iff** e | **right-iff** e |

left-either P e | **right-either** P e |

constructive-dilemma e_1 e_2 e_3 |

double-negation e |

$e_1; e_2$ |

pick-any x **in** e | **specialize** e **with** t |

exists t **such that** e | **pick-witness** x **for** e_1 **in** e_2 |

refl t | **rewrite** e_1 **in** e_2 |

case t **of** $(\texttt{inl}\ a \to e_1, \texttt{inr}\ b \to e_2)$ |

Proofterms – overview

A quick glance at the above grammar:

- First, propositions are included in the syntactic category of proofterms, because we will refer to an assumption P with P itself.
- Then we have some proofterms for dealing with propositional logic, divided as usual into introduction and elimination forms.
- **double-negation** is our way of making the logic classical.
- $e_1; e_2$ is a cut (which, in DPLs, corresponds to proof composition).
- Then we have some proofterms to deal with quantifiers.
- **refl** and **rewrite** are primitives for dealing with equality.
- Finally, **case** will be used for reasoning by cases on terms of type $A + B$.

Examples

Before we jump right into the rules, let's see some examples. The point of the first two is to show that our proofterms look very similar to proofs from DPLs (in particular, to proofs in $\mathcal{NDL}$ presented in chapters 4 and 6 of the DPL thesis). The third example shows how simple reasoning about programs might look like.

Example – propositional logic

Theorem: $(P \Rightarrow Q) \Rightarrow (Q \Rightarrow R) \Rightarrow P \Rightarrow R$.

Proof:

```
**assume** $P \Rightarrow Q$ **in**
 **assume** $Q \Rightarrow R$ **in**
  **assume** $P$ **in**
    **modus-ponens** $(P \Rightarrow Q)\ P$;
    **modus-ponens** $(Q \Rightarrow R)\ Q$
```

The proof looks the same as the $\mathcal{NDL}$ one (DPL thesis, chapter 4, page 71), except that we don't have **begin** and **end**.

Example – first-order logic

Theorem: $(\forall x : A.\, P\ x \wedge Q\ x) \Rightarrow (\forall x : A.\, P\ x) \wedge (\forall x : A.\, Q\ x)$

Proof:

```
assume $\forall x : A.\, P\ x \wedge Q\ x$ in
  pick-any $y$ in
    specialize $\forall x : A.\, P\ x \wedge Q\ x$ with $y$;
    left-and $P\ y \wedge Q\ y$;
  pick-any $y$ in
    specialize $\forall x : A.\, P\ x \wedge Q\ x$ with $y$;
    right-and $P\ y \wedge Q\ y$;
  both $(\forall y : A.\, P\ y)$ $(\forall y : A.\, Q\ y)$
```

Again, the proof looks the same as the $\mathcal{NDL}$ one (DPL thesis, chapter 6, page 156), except we use indentation instead of **begin** and **end**.

Example – proof about a program

Consider the following program:
$\texttt{swap} : A + B \to B + A := \lambda x.\, \texttt{case}\ x\ \texttt{of}\ (\lambda a.\, \texttt{inr}\ a, \lambda b.\, \texttt{inl}\ b)$

It's pretty clear that this function is involutive.
Theorem: $\forall x : A + B.\, \texttt{swap}\ (\texttt{swap}\ x) = x$

Proof:
pick-any x **in**
  case x **of** $(\texttt{inl}\ a \to \textbf{refl}\ (\texttt{inl}\ a), \texttt{inr}\ b \to \textbf{refl}\ (\texttt{inr}\ b))$

The proof has the same structure as the proofterm you would write in Coq, except for the syntactic differences.

New contexts

Contexts:
$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, P$

In the previous presentation of our logic, we judged the truth of a proposition in two contexts, $\Gamma \mid \Delta$, with Γ storing typing assumptions and Δ storing propositional assumptions (so it was the "assumption base"). We now change this, so that Γ will store both of these.

This change is not strictly necessary to introduce proofterms, but we go for it because it makes the rules more concise and will also bring benefits when implementation time comes.

Judgements

We will modify the language in the following way: we throw away the true proposition judgement $\Gamma \mid \Delta \vdash P$ and replace it with the correct proof judgement $\Gamma \vdash e : P$, which should be read: in context Γ, e is a proof of P.

Because we no longer have separate assumption contexts, but still need to keep track of well-formedness of assumptions, we drop the valid assumption context judgement $\Gamma \vdash \Delta\ \texttt{valid}$ and replace it with the valid context judgement $\Gamma\ \texttt{ctx}$, which should be read: Γ is a well-formed context.

The well-formed proposition judgement $\Gamma \vdash P\ \texttt{prop}$ stays unchanged.

Sanity checks

Recall that our judgements should satisfy some “sanity checks”, which can be summarized like this: a complex judgement should guarantee that its simpler components are well-formed.

In case of our logic, we will set up the judgements so that the following sanity checks hold:

- $\Gamma\ \texttt{ctx}$ entails nothing (because it is the simplest judgement)
- $\Gamma \vdash P\ \texttt{prop}$ entails $\Gamma\ \texttt{ctx}$
- $\Gamma \vdash e : P$ entails $\Gamma \vdash P\ \texttt{prop}$

Note that the programming layer would need to be slightly modified so that it satisfies the following sanity checks:

- $\Gamma \vdash e : A$ entails $\Gamma\ \texttt{ctx}$
- $\Gamma \vdash e_1 \equiv e_2 : A$ entails $\Gamma \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$

Assumptions

$$\frac{\Gamma\ \texttt{ctx} \qquad P \in \Gamma}{\Gamma \vdash P : P}\textsc{Ass}$$

The basic rule of our logic is that we can use assumptions from the context. Note that here P denotes both the proposition and its proof.

True and False

$$\frac{\Gamma\ \mathtt{ctx}}{\Gamma \vdash \textbf{true} : \top}\textsc{True-Intro}$$

$$\frac{\Gamma \vdash P\ \mathtt{prop} \qquad \Gamma \vdash e : \bot}{\Gamma \vdash \textbf{exfalso}\ e : P}\textsc{False-Elim}$$

true proves the true proposition $\top$ in any well-formed context.

exfalso e proves any well-formed proposition P whatsoever, provided that e proves the false proposition $\bot$.

Implication

$$\frac{\Gamma, P \vdash e : Q}{\Gamma \vdash \textbf{assume } P \textbf{ in } e : P \Rightarrow Q}\textsc{Impl-Intro}$$

$$\frac{\Gamma \vdash e_1 : P \Rightarrow Q \quad \Gamma \vdash e_2 : P}{\Gamma \vdash \textbf{modus-ponens } e_1\ e_2 : Q}\textsc{Impl-Elim}$$

assume P **in** e proves $P \Rightarrow Q$, provided that e proves Q in the context extended with the assumption P.

modus-ponens e_1 e_2 proves Q, provided that e_1 proves $P \Rightarrow Q$ and e_2 proves P.

Negation

$$\frac{\Gamma, P \vdash e : \bot}{\Gamma \vdash \textbf{suppose-absurd}\ P\ \textbf{in}\ e : \neg P}\textsc{Not-Intro}$$

$$\frac{\Gamma \vdash e_1 : \neg P \quad \Gamma \vdash e_2 : P}{\Gamma \vdash \textbf{absurd}\ e_1\ e_2 : \bot}\textsc{Not-Elim}$$

suppose-absurd P **in** e proves $\neg P$ provided that e proves $\bot$ in the context extended with the assumption P.

absurd e_1 e_2 proves $\bot$ provided that e_1 proves $\neg P$ and e_2 proves P.

Negation – discussion

Note that there are some differences from $\mathcal{NDL}$ (DPL thesis, chapter 4, page 65). First, **suppose-absurd** is a primitive, instead of being syntax sugar. Second, the order of arguments of **absurd** is flipped. The reason for this is that we follow a presentation of logic derived from the Curry-Howard correspondence.

However, what we do is not *the* usual Curry-Howard presentation, because in our system negation is primitive. The usual way to deal with negation would be to *define* $\neg P$ as an abbreviation for $P \Rightarrow \bot$, so that we would deal with negation by using **assume** instead of **suppose-absurd** and **modus-ponens** instead of **absurd**.

Conjunction

$$\frac{\Gamma \vdash e_1 : P \quad \Gamma \vdash e_2 : Q}{\Gamma \vdash \textbf{both}\ e_1\ e_2 : P \wedge Q}\textsc{And-Intro}$$

$$\frac{\Gamma \vdash e : P \wedge Q}{\Gamma \vdash \textbf{left-and}\ e : P}\textsc{And-Elim-L}$$

$$\frac{\Gamma \vdash e : P \wedge Q}{\Gamma \vdash \textbf{right-and}\ e : Q}\textsc{And-Elim-R}$$

both e_1 e_2 proves $P \wedge Q$ provided that e_1 proves P and e_2 proves Q. **left-and** e proves P provided that e proves $P \wedge Q$. **right-and** e proves Q provided that e proves $P \wedge Q$.

Biconditional

$$\frac{\Gamma \vdash e_1 : P \Rightarrow Q \quad \Gamma \vdash e_2 : Q \Rightarrow P}{\Gamma \vdash \textbf{equivalence}\ e_1\ e_2 : P \Leftrightarrow Q}\textsc{Iff-Intro}$$

$$\frac{\Gamma \vdash e : P \Leftrightarrow Q}{\Gamma \vdash \textbf{left-iff}\ e : P \Rightarrow Q}\textsc{Iff-Elim-L}$$

$$\frac{\Gamma \vdash e : P \Leftrightarrow Q}{\Gamma \vdash \textbf{right-iff}\ e : Q \Rightarrow P}\textsc{Iff-Elim-R}$$

equivalence e_1 e_2 proves $P \Leftrightarrow Q$ provided that e_1 proves $P \Rightarrow Q$ and e_2 proves $Q \Rightarrow P$. **left-iff** e proves $P \Rightarrow Q$ provided that e proves $P \Leftrightarrow Q$. **right-iff** e proves $Q \Rightarrow P$ provided that e proves $P \Leftrightarrow Q$.

Biconditional – discussion

Our presentation of the biconditional is exactly the same as in $\mathcal{NDL}$ (DPL thesis, chapter 4, page 65), although it differs from the usual Curry-Howard presentation: in our system $P \Leftrightarrow Q$ is a primitive, whereas in the usual Curry-Howard presentation it would be defined as $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$, so that we would use **both** instead of **equivalence**, **left-and** instead of **left-iff** and **right-and** instead of **right-iff**.

Disjunction

$$\frac{\Gamma \vdash Q\ \texttt{prop} \quad \Gamma \vdash e : P}{\Gamma \vdash \textbf{left-either}\ Q\ e : P \vee Q}\textsc{Or-Intro-L}$$

$$\frac{\Gamma \vdash P\ \texttt{prop} \quad \Gamma \vdash e : Q}{\Gamma \vdash \textbf{right-either}\ P\ e : P \vee Q}\textsc{Or-Intro-R}$$

$$\frac{\Gamma \vdash e_1 : P \vee Q \quad \Gamma \vdash e_2 : P \Rightarrow R \quad \Gamma \vdash e_3 : Q \Rightarrow R}{\Gamma \vdash \textbf{constructive-dilemma}\ e_1\ e_2\ e_3 : R}\textsc{Or-Elim}$$

left-either Q e proves $P \vee Q$ provided that e proves P.
right-either P e proves $P \vee Q$ provided that e proves Q.
constructive-dilemma e_1 e_2 e_3 proves R provided that e_1 proves $P \vee Q$, e_2 proves $P \Rightarrow R$ and e_3 proves $Q \Rightarrow R$.

Disjunction – annotations

The first argument of **left-either** Q e, i.e. Q, is a proposition, not a proofterm, and serves as an annotation that clarifies what its conclusion is. Note that it needs to be well-formed.
right-either P e is analogous.

The choice to have annotations there makes our system very close to $\mathcal{NDL}$ (DPL thesis, chapter 4, page 65), and also to the "intrinsic" presentation of the Simply Typed Lambda Calculus, but it is not the only one: we could do without the annotations, which would make the system less similar to $\mathcal{NDL}$ and more to the "extrinsic" presentation of Simply Typed Lambda Calculus.

Disjunction – choices

We also have a choice regarding the second and third arguments of **constructive-dilemma** $e_1\ e_2\ e_3$. To keep close to $\mathcal{NDL}$, they both must prove implications ($P \Rightarrow R$ for e_2 and $Q \Rightarrow R$ for e_3), but perhaps it would be more comfortable for the user if they only needed to prove the conclusion R in a context extended with the appropriate assumption, so that he would be spared from having to write **assume** twice (once in e_2 and once in e_3). The alternative rule is as follows:

$$\frac{\Gamma \vdash e_1 : P \vee Q \quad \Gamma, P \vdash e_2 : R \quad \Gamma, Q \vdash e_3 : R}{\Gamma \vdash \textbf{constructive-dilemma}\ e_1\ e_2\ e_3 : R}\textsc{Or-Elim-Alt}$$

Classical logic

Up until now, our logic is entirely constructive, closely following the Curry-Howard correspondence, although it differed slightly with regards to the treatment of negation and biconditional.

To make the logic classical, we need to extend it with one of the laws of classical logic that is strong enough to entail all the others. In our previous presentation of Axi we chose the Law of Excluded Middle, but this time, to keep close to $\mathcal{NDL}$, we will use Double Negation Elimination.

In a richer language and logic, the way to go would probably be to assert some version of the Axiom of Choice.

Double Negation Elimination

$$\frac{\Gamma \vdash e : \neg\neg P}{\Gamma \vdash \textbf{double-negation}\ e : P}\textsc{Classic}$$

double-negation e proves P provided that e proves $\neg\neg P$.

Reasoning by contradiction as an alternative

An alternative way of asserting classical logic would be to use another principle, like proof by contradiction, which might be more convenient to the user, because it automatically extends the context.

$$\frac{\Gamma, \neg P \vdash e : \bot}{\Gamma \vdash \textbf{by-contradiction}\ e : P}\textsc{By-Contradiction}$$

by-contradiction e proves P provided that e proves $\bot$ in the context extended with the assumption $\neg P$.

Backward and forward proofs

The proofterms we introduced so far only allowed for comfortable "backward" proofs, i.e. ones where we go from the conclusion to the assumptions. The other style, "forward" proofs, are also possible, but doing them is not comfortable.

To amend this problem, we introduce a new rule, called the CUT rule. In DPLs, it corresponds to proof composition (and in programming languages, to let-expressions, although there is no variable binding).

The CUT rule increases the comfort of doing forward proofs in our system, but does not add any expressive power, because $e_1; e_2$ can be desugared to **modus-ponens** (**assume** P **in** e_2) e_1.

Cut rule

$$\frac{\Gamma \vdash e_1 : P \quad \Gamma, P \vdash e_2 : Q}{\Gamma \vdash e_1; e_2 : Q}\textsc{Cut}$$

$e_1; e_2$ proves Q provided that e_1 proves P and e_2 proves Q in the context extended with the assumption P.

Quantifiers

Our syntax for quantifiers closely follows $\mathcal{NDL}$ (DPL thesis, chapter 6, page 150), except for the introduction rule of the existential quantifier, in which the argument order is flipped (and which, for this reason, uses different keywords).

Our rules, however, are subtly different in that we allow arbitrary proofs as arguments to **specialize**, **exists** and **pick-witness**, whereas $\mathcal{NDL}$ only allows formulas to appear as arguments in these places. Therefore, our approach is a slight generalization, which also closely aligns in spirit with the Curry-Howard correspondence.

Universal quantifier

$$\frac{\Gamma, y : A \vdash e : P\,[x := y]}{\Gamma \vdash \textbf{pick-any } y \textbf{ in } e : \forall x : A.\, P}\textsc{Forall-Intro}$$

$$\frac{\Gamma \vdash e : \forall x : A.\, P \qquad \Gamma \vdash t : A}{\Gamma \vdash \textbf{specialize } e \textbf{ with } t : P\,[x := t]}\textsc{Forall-Elim}$$

pick-any y **in** e proves $\forall x : A.\, P$ provided that e proves P in which y was substituted for x in the context extended with $y : A$.

specialize e **with** t proves P in which t was substituted for x, provided that e proves $\forall x : A.\, P$ and t is a term of type A.

Existential quantifier

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash e : P\,[x := t]}{\Gamma \vdash \textbf{exists } t \textbf{ such that } e : \exists x : A.\, P}\textsc{Exists-Intro}$$

$$\frac{\Gamma \vdash R \;\texttt{prop} \quad \Gamma \vdash e_1 : \exists x : A.\, P \quad \Gamma, y : A, P\,[x := y] \vdash e_2 : R}{\Gamma \vdash \textbf{pick-witness } y \textbf{ for } e_1 \textbf{ in } e_2 : R}\textsc{Exists-Elim}$$

exists t **such that** e proves $\exists x : A.\, P$ provided that t is a term of type A and e proves P in which t was substituted for x.

pick-witness y **for** e_1 **in** e_2 proves R, provided that e_1 proves $\exists x : A.\, P$ and e_2 proves R in the context extended with $y : A$ and P (in which y was substituted for x). Note that R needs to be well-formed in Γ, i.e. it can't depend on x or y.

Computational equality – refresher

The computational equality judgement $\Gamma \vdash t_1 \equiv t_2 : A$ (sometimes also called judgemental equality, or simply convertibility) is the way in which we express computation in our language. Its intuitive meaning is that t_1 and t_2 compute to the same result (although the rules we gave in the previous slides don't look like this at all).

Computational equality is a part of the programming layer of our language and does not depend on the logical layer in any way. In particular, it does not have to change to accomodate proofterms (except for a tiny change in the congruence rule for **1** to accomodate our new contexts). Computational equality is decidable and its use in proofs is handled entirely by the proof checker.

Propositional equality – refresher

Propositional equality $t_1 =_A t_2$ is not a judgement, but, as the name suggests, a proposition. It belongs to the logical layer of our language and the introduction of proofterms does affect it – we will need to change our rules to accomodate them. It is not decidable and all proofs involving propositional equality are up to the programmer.

Equality – overview

Our treatment of equality is fundamentally type-theoretic, with the conversion rule, which connects computational equality to propositional equality, playing the central role. The end product, however, is very similar to $\mathcal{NDL}$ (DPL thesis, chapter 6, page 151).

We also include a rule that asserts function extensionality, i.e. the principle that two functions are equal when they are equal for all arguments. This rule is not present in $\mathcal{NDL}$ because its language of terms doesn't have function types.

Conversion rule

$$\frac{\Gamma, x : A \vdash P\ \mathtt{prop} \quad \Gamma \vdash t_1 \equiv t_2 : A \quad \Gamma \vdash e : P\,[x := t_1]}{\Gamma \vdash e : P\,[x := t_2]}\textsc{Conv}$$

If e proves P with t_1 substituted for x and t_1 is computationally equal to t_2 at type A, then e also proves P with t_2 substituted for x.

Conversion rule – discussion

The main point of the conversion rule is to facilitate proving equalities which hold by computation. In contrast to DPLs, which would require doing some manual labor here (or calling powerful methods for help), in our system proving such equalities is free, or even more than free – the programmer doesn't need to do anything to benefit from the conversion rule, not even use the conversion rule. This is because computational equality is decidable, so using the conversion rule at the right time is entirely up to the proof checker.

Equality introduction and elimination

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \textbf{refl}\ t : t =_A t}\,\textsc{Eq-Intro}$$

$$\frac{\Gamma, x : A \vdash P\ \texttt{prop} \quad \Gamma \vdash e : t_1 =_A t_2 \quad \Gamma \vdash e' : P\,[x := t_1]}{\Gamma \vdash \textbf{rewrite}\ e\ \textbf{in}\ e' : P\,[x := t_2]}\,\textsc{Eq-Elim}$$

refl t proves that t is equal to itself, provided that it is well-typed at some type A.

rewrite e **in** e' proves P with t_2 substituted for x, provided that e proves $t_1 =_A t_2$ (for some type A) and that e' proves P with t_1 substituted for x.

refl – discussion

refl t might look like a weakling, a puny little proofterm only able to prove $t = t$, but in reality it is an incantation that summons the invincible juggernaut – the conversion rule. If t is computationally equal to t', then, by the conversion rule, to prove $t = t'$ it suffices to prove $t = t$, which is precisely what **refl** t accomplishes.

rewrite – discussion

rewrite is the workhorse for equational proofs, corresponding to $\mathcal{NDL}$'s **leibniz** (DPL thesis, chapter 6, page 151), except that, as for quantifiers, we allow its arguments to be proofs, whereas $\mathcal{NDL}$ requires them to be formulas and terms whose equality is already in the assumption base.

Compared to Athena, using **rewrite** e **in** e' is similar to performing a single step of chaining, with e being the justifier and e' being the rest of the chain.

Compared to Coq, **rewrite** e **in** e' looks most similar to Coq's tactic `rewrite e`; e', which rewrites an equation e and then goes on to e', which is the rest of the proof.

Function extensionality

$$\frac{\Gamma \vdash e : \forall x : A.\, f \;\; x =_B g \;\; x}{\Gamma \vdash \textbf{funext} \;\; e : f =_{A \to B} g}\textsc{Funext}$$

funext e proves $f = g$ provided that e proves $\forall x : A.\, f \;\; x = g \;\; x$.

Function extensionality – discussion

A separate rule for function extensionality is needed, because extensionality, despite being a very basic, useful and desirable principle, does not follow from anything else in the system: neither from the intuitionistic core, nor the classical **double-negation**, nor the rules for quantifiers, nor even the conversion rule, even though function extensionality is present at the level of computational equality. This is a well known phenomenon in type theory, with which $\mathcal{NDL}$ and other DPLs don't have to deal because they don't have function types.

Function extensionality – alternative rule

funext e is in some sense "higher-order", i.e. e is expected to be a proof of a universal statement, which means that usually it will be of the form **pick-any** x **in** e'. This suggests an alternative, "first-order" version of the rule for function extensionality, which has this variable binding baked-in:

$$\frac{\Gamma, x : A \vdash e : f\ x =_B g\ x}{\Gamma \vdash \textbf{funext}\ x\ \textbf{in}\ e : f =_{A \to B} g}\ \textsc{Funext-Alt}$$

Reasoning by cases (for sums)

$$\frac{\Gamma, x : A + B \vdash P \ \texttt{prop} \quad \Gamma \vdash t : A + B \quad \begin{array}{c}\Gamma, a : A \vdash e_1 : P\,[x := \texttt{inl}\ a] \\ \Gamma, b : B \vdash e_2 : P\,[x := \texttt{inr}\ b]\end{array}}{\Gamma \vdash \textbf{case}\ t\ \textbf{of}\ (\texttt{inl}\ a \to e_1, \texttt{inr}\ b \to e_2) : P\,[x := t]}$$

case t **of** $(\texttt{inl}\ a \to e_1, \texttt{inr}\ b \to e_2)$ proves P with t substituted for x, provided that t is of type $A + B$, that e_1 proves P with $\texttt{inl}\ a$ substituted for x in the context extended with $a : A$, and that e_2 proves P with $\texttt{inr}\ b$ substituted for x in the context extended with $b : B$.

Reasoning by cases (for sums) – discussion

From the grammar and typing rules given in Poor Man's Axi slides, we know that terms of type $A + B$ can be of only two forms: either `inl` a for some $a : A$, or `inr` b for some $b : B$.

We know this, but our logic doesn't automagically know this, so we need to tell it by adding an appropriate proofterm and a rule to handle it. This is what **case** accomplishes.

Induction

Sums are not the only type for which we need to add a principle for reasoning by cases – there are also products, the unit type, and the empty type, after all.

But for more complicated types, like the natural numbers or lists, reasoning by cases won't suffice, and so we would also need to add proofterms and rules for induction. We won't do this now for lack of space and time, but it's not much more difficult than handling reasoning by cases for sums.