

An Epigram Implementation

Edwin Brady, James Chapman, Pierre-Évariste Dagand,
Adam Gundry, Conor McBride, Peter Morris, Ulf Norell, Nicolas Oury

February 16, 2011

Contents

1	Introduction	6
1.1	In the beginning	6
1.1.1	Recommended Reading	6
1.2	Language	7
2	The Evidence Language	9
2.1	Tm	9
2.1.1	TmJig	12
2.2	NameSupply	13
2.2.1	(\rightarrow) NameSupply is a NameSupplier	15
2.2.2	ReaderT NameSupply is a NameSupplier	15
2.2.3	The Check monad is a NameSupplier	15
2.3	Variable Manipulation	15
2.3.1	The <i>under</i> mangle	16
2.3.2	The deBruijnifying mangle	16
2.3.3	The substitution mangle	17
2.3.4	The <i>inc</i> mangle	17
2.4	Evaluation	17
2.4.1	Elimination	18
2.4.2	Operators	18
2.4.3	Binders	19
2.4.4	Evaluator	19
2.4.5	Alpha-conversion on the fly	20
2.5	Type-checker	21
2.5.1	Type-checking Canonicals and Eliminators	21
2.5.2	Type checking	25
2.5.3	Type inference	26
2.6	Equality and Quotation	27
2.6.1	inQuote	28
2.6.2	η -expansion	29
2.6.3	exQuote	29
2.6.4	Simplification of stuck terms	30
2.7	β -Quotation	30
2.8	Operators and primitives	31
2.9	Operator DSL	46
2.10	Observational Equality	47
2.11	Utilities	51
2.11.1	From EXTM to INTM and back again	51
2.11.2	Discharging a list of hypotheses over a term	51
2.11.3	Term construction and deconstruction	52

3	The Display Language	53
3.1	Display Terms	53
3.1.1	Structure of Display Terms	53
3.1.2	Useful Abbreviations	55
3.1.3	Sizes	57
3.2	Relative Names	58
3.2.1	Names to strings	58
3.3	Schemes	58
3.3.1	Schemes for implicit arguments	58
3.3.2	Extracting names	59
3.3.3	Turning schemes to terms	59
3.3.4	Unlifting schemes	59
3.3.5	Schemes in error messages	59
3.4	Lexer	60
3.4.1	What are tokens?	60
3.4.2	Lexer	61
3.4.3	Abstracting tokens	66
3.5	Parsing Terms	67
3.5.1	Names	67
3.5.2	Overall parser structure	67
3.5.3	Lists of sized parsers	68
3.5.4	Parser support code	71
3.5.5	Parsing schemes	72
3.6	Pretty-printing	72
4	The Proof State	79
4.1	Developments	79
4.1.1	The Dev data-structure	79
4.2	Managing Entries in a Development	81
4.2.1	Looking into an Entry	81
4.2.2	Entry equality	82
4.2.3	Changing the carrier of an Entry	82
4.3	News about updated references	82
4.3.1	News	82
4.3.2	News Bulletin	82
4.4	Proof Context	84
4.4.1	The derivative: Layer	84
4.4.2	The Zipper: ProofContext	85
4.5	The ProofState monad	86
4.5.1	Defining the Proof State monad	86
4.5.2	Error management toolkit	86
4.6	Managing Entries in a Proof Context	86
4.7	Fake references	87
4.8	Scope management	88
4.8.1	Extracting scopes as entries	88
4.8.2	Manipulating entries as scopes	88
4.9	The Get Set	89
4.9.1	Getters	89
4.9.2	Putters	91
4.9.3	Removers	93
4.9.4	Replacers	93
4.10	Navigating in the Proof Context	93
4.10.1	One-step navigation	94
4.10.2	Many-step Navigation	100

4.11	The ProofState Kit	101
4.11.1	Accessing the NameSupply	101
4.11.2	Accessing the type-checker	101
4.11.3	Being paranoiac	102
4.12	Name management	102
4.13	Making Parameters	103
4.13.1	λ -abstraction	103
4.13.2	Assumptions	104
4.13.3	Pi-abstraction	104
4.14	Making Definitions	104
4.15	Modules in Proof Context	105
4.16	Searching in the Proof Context	106
4.16.1	Searching by name	106
4.16.2	Searching for a goal	107
4.17	Solving goals	108
4.17.1	Giving a solution	108
4.17.2	Finding trivial solutions	108
4.17.3	Refining the proof state	109
4.18	Resolving and unresolving names	110
4.18.1	Resolving relative names to references	110
4.18.2	Unresolving absolute names to relative names	113
4.19	Lambda-lifting and discharging	119
4.19.1	Discharging entries in a term	119
4.19.2	Binding a term	119
4.19.3	Binding a type	119
4.19.4	Making a type out of a goal	120
4.20	Anchor resolution	120
5	The Proof Tactics	122
5.1	Presenting Information	122
5.2	Elimination with a Motive	128
5.2.1	Analyzing the eliminator	128
5.2.2	Identifying the motive	130
5.2.3	Putting things together	138
5.3	Propositional Simplification	140
5.3.1	Setting the Scene	140
5.3.2	Simplification in Action	141
5.3.3	Invoking Simplification	147
5.4	Problem Simplification	148
5.5	Record declaration	153
5.6	Relabelling	154
5.7	Programming Gadgets	157
5.7.1	The Return gadget	157
5.7.2	The Define gadget	157
5.7.3	The By gadget	157
5.7.4	The Refine gadget	158
5.7.5	The Solve gadget	158
5.8	Converting Epigram definitions to Haskell	158
5.9	Unification	161
5.9.1	Solving flex-rigid problems	161
5.10	Matching	163

6	Elaboration	167
6.1	ElabProb: syntactic representation of elaboration problems	167
6.2	The Elab monad: a DSL for elaboration	168
6.3	Using the Elab language	169
6.3.1	Tools for writing elaborators	169
6.3.2	Elaborating DInTms	171
6.3.3	Elaborating DExTms	173
6.4	Implementing the Elab monad	176
6.4.1	Running elaboration processes	176
6.4.2	Interpreting elaboration problems	178
6.4.3	Hoping, hoping, hoping	179
6.4.4	Suspending computation	184
6.5	Invoking the Elaborator	184
6.5.1	Elaborating terms	184
6.5.2	Elaborating construction commands	185
6.5.3	Elaborating programming problems	186
6.5.4	Elaborating schemes	188
6.6	The Scheduler	189
6.7	Wire Service	192
6.7.1	Updating a reference	192
6.7.2	Committing news into the ProofState	192
6.7.3	Informing a current entry about its development	194
7	Distillation	198
7.1	The distiller	198
7.1.1	Distilling INTMs	198
7.1.2	Distilling EXTMs	201
7.1.3	Distillation support	202
7.1.4	Distillation interface	203
7.2	The Scheme distiller	203
7.2.1	Distilling schemes	203
7.2.2	ProofState interface	204
7.3	The Moonshine distillery	205
7.3.1	Moonshining	205
8	Cochon	206
8.1	Loading Developments	206
8.1.1	Parsing a Development	206
8.1.2	Construction	208
8.1.3	Loading the files	209
8.2	Cochon Command Lexer	210
8.2.1	Tokens	210
8.2.2	Tokenizer combinators	211
8.2.3	Printers	212
8.3	Cochon (Command-line Interface)	212
8.4	Cochon error prettier	222
8.4.1	Catching the gremlins before they leave ProofState	222
8.4.2	Pretty-printing the stack trace	222
8.5	Main	222

9	The Source Language	225
9.1	Structure	225
9.2	Parser	226
9.3	Elaborator	226
9.4	Example	227
10	Compiler	228
10.1	OpDef	228
10.2	Compiler	229
10.2.1	Representing Epic syntax	229
10.2.2	Compiling functions	230
10.2.3	Code generation	232
10.2.4	Flattening and lambda-lifting	232
10.2.5	Invoking compilation	234
10.2.6	Operator definitions	235
A	Kit	237
A.1	BwdFwd	237
A.2	Parsley	238
A.2.1	Parsley’s Semantics	239
A.2.2	Structure	239
A.2.3	Low-level combinators	240
A.2.4	High-level combinators	240
A.2.5	Token manipulation	241
A.3	Missing Library	241
A.3.1	Renaming	241
A.3.2	Indicator Function	242
A.3.3	Newtype Unwrapping	242
A.3.4	Error Handling	243
A.3.5	Missing Applicatives	243
A.3.6	Missing Instances	244
A.3.7	HalfZip	244
A.3.8	Functor Kit	244
A.3.9	Applicative Kit	246
A.3.10	Monadic Kit	247
A.4	Trace	247
	Bibliography	248
	Index	249

Chapter 1

Introduction

"I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature." Donald Knuth, 1992.

This document is intended to realise this lofty aim for the implementation of Epigram. Although, given that it is being written by several different people at once, the plot changes constantly, and it's currently pushing 300 pages even though it has barely been started, I don't expect it'll be a best seller.

Epigram is a many faceted beast whose facets are constructed roughly one on top of the other. This document starts by describing its bowels: the core type theory which serves as Epigram's language of evidence and into which programs are elaborated. We then proceed outwards and upwards until we reach its snorting, wild eyed, and rabid face, a.k.a. the high-level language.

1.1 In the beginning

In the beginning there was Epigram 1. The Epigram 1 language was designed by Conor McBride and James McKinna in their weighty paper "The view from the left"[15]. It describes a programming language with support for inductive families and dependent pattern matching. Notably pattern matching is an added extra to the type theory. Instead, pattern matching programs in the high level language are *elaborated* into type theoretic expressions made of traditional eliminators in the underlying theory. An implementation of Epigram 1 was written by Conor and with its idiosyncratic emacs interface with 2d syntax, one could make small experiments. Unfortunately this prototype implementation proved unscalable and unmaintainable. Since then Agda 2 and Coq's Russell language have assimilated many of Epigram 1's features and many of Epigram's programs can be written in these two other languages.

Epigram 2 is not just an attempt at a better implementation of Epigram 1; it has a radically different underlying type theory which supports interesting features in the high-level language. Firstly, it supports extensional equality of functions in an intentional theory by using observational equality. Secondly, it is a closed type theory where new data definitions are new codes in a universe of datatype descriptions whose validity is internally guaranteed rather than decreed to be acceptable by an external checker.

1.1.1 Recommended Reading

The type-checker is a bidirectional one [17]. A remotely related type-checker has been described in the context of ETT [5].

Normalization is achieved by evaluation [6, 3]. The implementation has been described in James Chapman's work [4]. A graspable introduction to both normalization by evaluation and bidirectional type-checking à la Epigram can be found in Boutillier's report [2].

The story for names has been told by McBride and McKinna [14]. *Ite messa est.*

For containers, there is a lot to say. So, I will not say anything for the moment.

Concerning the `Desc` universe, Morris et al. [16] wrote a clear article, covering that topic and much more. In particular, they show why and how `Box` and `mapBox` can automatically derived from `Desc`. Conor’s work on `Ornamemts` [12] builds up from a simplified `Desc` universe: some insights can be found there too. Finally, the authoritative source shall be cited: Dybjer et al. on induction-recursion [7, 8, 9].

1.2 Language

<code>INTM ::=</code>	<code>Set</code>	Universe of sets
	<code>Prop</code>	Universe of propositions
	<code><(NOM : INTM)>+ -> INTM</code>	Π -type
	<code>INTM -> INTM</code>	nondependent function type
	<code>\NOM* -> INTM</code>	λ -abstraction
	<code>:- INTM</code>	set of proofs
	<code>NAT<+ INTM>?</code>	Elements of enumerations
	<code>con INTM</code>	Constructor for inductive definitions
	<code>sig (SIG)</code>	‘record’ signature
	<code>()</code>	unit
	<code>[]</code>	void
	<code>[INTM+ <, INTM>?]</code>	tuple
	<code>Enum INTM</code>	enumeration
	<code>'NOM INTM</code>	tag or (co)data
	<code><(NOM : INTM)>+ => INTM</code>	propositional \forall
	<code>INTM && INTM</code>	propositional And
	<code>TT FF</code>	propositional Trivial and Absurd
	<code>1 0</code>	proofs of Trivial and Absurd
	<code>EXTM == EXTM</code>	blue equality
	<code>(INTM)</code>	grouping
	<code>EXTM</code>	term with synthesizable type
<code>SIG ::=</code>	<code>ε</code>	unit signature
	<code><NOM : >?INTM SIGMORE</code>	$\Sigma x : S.T$
<code>SIGMORE ::=</code>	<code>i INTM</code>	$i \ T$ means T
	<code>i SIG</code>	$i \ S$ means S
	<code>:-INTM</code>	$:- \ P$ means $:- \ P$
	<code>:-INTM SIGMORE</code>	$:- \ P \ S$ means $\Sigma \dots PS$
<code>EXTM ::=</code>	<code>(: INTM)</code>	type-cast
	<code>NOM(^NAT)?<.NOM(^NAT)?>*</code>	relative name
	<code>EXTM INTM</code>	application
	<code>EXTM !</code>	car
	<code>EXTM -</code>	cdr
	<code>OP(INTM '*)</code>	operator
	<code>(INTM : INTM) <-> (INTM : INTM)</code>	green equality

and more to come. For developments:

TOP ::=	$\langle [GIRL^*]^? COM^{i^*}$	top-level development
GIRL ::=	$NOM \langle [LINE^*] := \rangle \langle ? INTM \rangle : INTM \langle [COM^{i^*}] \rangle^? ;$	development
	$ NOM [LINE^*] \langle [COM^{i^*}] \rangle^? ;$	module
LINE ::=	GIRL BOY	line in development
BOY ::=	$\backslash NOM : INTM \rightarrow$	λ -boy
	$ (NOM : INTM) \rightarrow$	Π -boy
	$ (NOM : INTM ;) \rightarrow$	Σ -boy
	$:- (NOM : INTM) \Rightarrow$	\forall -boy

This is all very tentative. One overriding principle is that we should stick to ASCII characters throughout. Users may use Unicode if they wish, but it should not be forced upon them.

Chapter 2

The Evidence Language

2.1 Tm

```
type Name = [(String, Int)]
```

```
data Part = Head | Body
```

```
data Status :: * where
```

```
Val :: Status
```

```
Exp :: Status
```

```
deriving SheSingleton
```

```
data Tm :: {Part, Status, Nat} → * where
```

```
L   :: Env {n, m} → String → Tm {Body, s, S m} → Tm {Body, s', n}
```

```
LK  :: Tm {Body, s, n} → Tm {Body, s', n}
```

```
(: -) :: Can → [Tm {Body, s, n}] → Tm {Body, s', n}
```

```
(: $) :: Tm {Head, s, n} → Bwd (Tm {Body, Exp, n}) → Tm {Body, s, n}
```

```
D   :: DEF → Bwd EXP → Operator {p, s} → Tm {p, s, n}
```

```
V :: Fin {n} {-dB i-} → Tm {Head, s, n}
```

```
P :: (Int, String, TY) {-dB l-} → Tm {Head, s, n}
```

```
(: /) :: Env {n, m} → Tm {p, s, m} → Tm {p', Exp, n}
```

```
data Operator :: {Part, Status} → * where
```

```
Eat :: Operator {p, s} → Operator {Body, s'}
```

```
Emit :: Tm {Body, Exp, Z} → Operator {Body, Exp}
```

```
Hole :: Operator {Head, s}
```

```
Case :: [(Can, Operator {Body, s})] → Operator {Body, s'}
```

```
StuckCase :: [(Can, Operator {Body, s})] → Operator {Head, s'}
```

```
Split :: Operator {p, s} → Operator {Body, s'}
```

```
type Env nm = (Maybe [EXP], IEnv nm)
```

```

data IEnv :: { Nat, Nat } → * where
  INix :: IEnv { n, Z }
  INil :: IEnv { n, n }
  (:<<:) :: IEnv { m, n } → Tm { Body, s, Z } → IEnv { m, S n }

```

```

exp :: Tm { p, s, n } → Tm { p', Exp, n }
exp = (: /) ENil

```

```

ev :: Tm { p, Exp, Z } → VAL
ev = (ENil / /)

```

```

(!!) :: IEnv { Z, n } → Fin { n } → Tm { Body, Exp, Z }
(ez <<< e) !! Fz = exp e
(ez <<< e) !! Fs i = ez !! i

```

```

(<<:) :: IEnv { m, n } → IEnv { n, o } → IEnv { m, o }
g <<< INix = INix
g <<< INil = g
g <<< (g' <<< e) = (g <<< g') <<< e

```

```

(< + <) :: Env { m, n } → Env { n, o } → Env { m, o }
(gl, gi) < + < (gl', gi') = (gln, gi <<< gi')
where
  gln = case (gl, gl') of
    (_, Just y) → Just y
    (Just x, _) → Just x
    _ → Nothing

```

```

pattern ENil = (Nothing, INil)

```

```

type DEF = (String, TY, Operator { Body, Exp })

```

```

type EXP = Tm { Body, Exp, Z }
type VAL = Tm { Body, Val, Z }
type TY = EXP

```

```

data Can :: * where
  Set   :: Can -- set of sets
  Pi    :: Can -- functions
  Sigma :: Can -- products
  Pair  :: Can -- pairs
  Con   :: Can -- packing
  Hd    :: Can
  Tl    :: Can
deriving (Eq, Show)

```

pattern SET = Set : - [] -- set of sets
pattern ARR $s\ t$ = Pi : - [s, (LK t)] -- simple arrow
pattern PI $s\ t$ = Pi : - [s, t] -- dependent functions
pattern TIMES $s\ t$ = Sigma : - [s, (LK t)] -- simple arrow
pattern SIGMA $s\ t$ = Sigma : - [s, t] -- dependent product
pattern PAIR $a\ b$ = Pair : - [a, b] -- pairing
pattern CON t = Con : - [t] -- Container (packing "stuff")
pattern HD = Hd : - []
pattern TL = Tl : - []

eval :: $\forall m\ n\ p\ s' \cdot pi\ (s :: Status) \cdot$
Env {Z, n} → Tm {p, s', n} → Tm {Body, s, Z}
eval {s} g (L g' x b) = L (g < + < g') x b
eval {s} g (LK e) = LK (g : / e)
eval {s} g (c : - es) = c : - (fmap (g: /) es)
eval {s} g (h : \$ es) = *aplys* {s} (eval {s} g h) (fmap (g: /) es)
eval {s} (Nothing, _) (D d ez o) = mkD {s} d ez o
eval {s} (Just es, _) (D d ez o) = mkD {s} d (fmap ((Just es, INix): /) ez) o
eval {s} (es, ez) (V i) = *eval* {s} ENil (ez !! i)
eval {s} (Nothing, _) (P lt) = P lt : \$ B0
eval {s} (Just es, _) (P (l, -, -)) = *eval* {s} ENil (es !! l)
eval {s} g (g' : / e) = *eval* {s} (g < + < g') e

(//) :: {s :: Status:} ⇒ Env {Z, n} → Tm {p, s', n} → Tm {Body, s, Z}
(//) = *eval* {s :: Status:}

apply :: $\forall s' \cdot pi\ (s :: Status) \cdot$
Tm {Body, s, Z} → Tm {Body, s', Z} → Tm {Body, s, Z}
apply {s} (L (gl, gi) _ b) a = *eval* {s} (gl, gi :<<: a) b
apply {s} (LK e) _ = *eval* {s} ENil e
apply {s} (D d es (Eat o)) a = mkD {s} d (es :< exp a) o
apply {Val} (D d es (Case os)) a =
case (ENil // a :: VAL) **of**
(c : - as) → **case** *lookup* c os **of**
(Just o) → *foldl* (\$\$) (mkD {Val} d es o) as
Nothing → *error* "You muppet"
x → D d (es :< exp x) (StuckCase os) : \$ B0
apply {Val} (D d es (Split o)) a =
mkD {Val} d es o \$\$ (exp a : \$ (B0 :< HD)) \$\$ (exp a : \$ (B0 :< TL))
apply {Exp} d@(D _ _ _) a = (ENil : / d) : \$ (B0 :< exp a)
apply {s} (PAIR a b) HD = *eval* {s} ENil a
apply {s} (PAIR a b) TL = *eval* {s} ENil b

(\$\$) :: {s :: Status:} ⇒ Tm {Body, s, Z} → Tm {Body, s', Z} → Tm {Body, s, Z}
(\$\$) = *apply* {s :: Status:}

aplys :: $pi\ (s :: Status) \cdot Tm\ \{Body, s, Z\} \rightarrow Bwd\ EXP \rightarrow Tm\ \{Body, s, Z\}$
aplys {s} v B0 = v
aplys {s} v (ez :< e) = *apply* {s} (*aplys* {s} v ez) e

(\$\$\$) :: {s :: Status:} ⇒ Tm {Body, s, Z} → Bwd EXP → Tm {Body, s, Z}
(\$\$\$) = *aplys* {s :: Status:}

```

mkD :: ∀ s' p · pi (s :: Status) ·
  DEF → Bwd EXP → Operator {p, s'} → Tm {Body, s, Z}
mkD {s} d es (Emit t)      = eval {s} (Just (trail es), INix) t
mkD {s} d es (Eat o)       = D d es (Eat o)
mkD {s} d es Hole         = D d es Hole : $ B0
mkD {s} d es (Case os)     = D d es (Case os)
mkD {s} d (es :< e) (StuckCase os) = apply {s} (D d es (Case os)) e

```

We have special pairs for types going into and coming out of stuff. We write $typ \text{ :> } thing$ to say that typ accepts the term $thing$, i.e. we can push the typ in the $thing$. Conversely, we write $thing \text{ :< } typ$ to say that $thing$ is of inferred type typ , i.e. we can pull the type typ out of the $thing$. Therefore, we can read :> as “accepts” and :< as “has inferred type”.

```

data ty :> tm = ty :> tm deriving (Show, Eq)
infix 4 :>
data tm :< ty = tm :< ty deriving (Show, Eq)
infix 4 :<

```

```

fstIn :: (a :> b) → a
fstIn (x :> _) = x

```

```

sndIn :: (a :> b) → b
sndIn (_ :> x) = x

```

```

fstEx :: (a :< b) → a
fstEx (x :< _) = x

```

```

sndEx :: (a :< b) → b
sndEx (_ :< x) = x

```

As we are discussing syntactic sugar, we define the “reduces to” symbol:

```

data t :=> v = t :=> v deriving (Show, Eq)
infix 5 :=>

```

with the associated projections:

```

valueOf :: (t :=> v) → v
valueOf (_ :=> v) = v
termOf :: (t :=> v) → t
termOf (t :=> _) = t

```

Intuitively, $t :=> v$ can be read as “the term t reduces to the value v ”.

2.1.1 TmJig

```

class Wrapper (t :: *) (n :: {Nat}) | t → n where
  wrapper :: Tm {Head, Exp, n} → Bwd (Tm {Body, Exp, n}) → t

```

```

instance Wrapper (Tm {Body, Exp, n}) n where
  wrapper h es = h :$ es

```

instance ($s \sim \text{Tm } \{\text{Body}, \text{Exp}, n\}$, $\text{Wrapper } t \ n$) \Rightarrow $\text{Wrapper } (s \rightarrow t) \ n$ **where**
wrapper f es e = wrapper f (es :< e)

instance ($\text{Wrapper } s \ n$, $\text{Wrapper } t \ n$) $=_i$ $\text{Wrapper } (s, t) \ n$ **where** *wrapper f es = (wrapper f es, wrapper f es)*

cough :: (Fin {S m} \rightarrow Tm {p, b, m}) \rightarrow Tm {p, b, m}
cough f = f Fz -- coughs up the zero you want

la :: String \rightarrow
(($\forall t \ n \cdot (\text{Wrapper } t \ n, \text{Leq } \{\text{S } m\} \ n) \Rightarrow t$)
 \rightarrow Tm {Body, Exp, S m}) \rightarrow
Tm {Body, s, m}
la s b = cough \$ \fz \rightarrow L ENil s (b (wrapper (V (finj fz)) B0))

(- >>) :: (String, Tm {Body, s, m}) \rightarrow
(($\forall t \ n \cdot (\text{Wrapper } t \ n, \text{Leq } \{\text{S } m\} \ n) \Rightarrow t$)
 \rightarrow Tm {Body, Exp, S m}) \rightarrow
Tm {Body, s, m}
(x, s) - >> t = PI s (la x t)

*(- **) :: (String, Tm {Body, s, m}) \rightarrow*
(($\forall t \ n \cdot (\text{Wrapper } t \ n, \text{Leq } \{\text{S } m\} \ n) \Rightarrow t$)
 \rightarrow Tm {Body, Exp, S m}) \rightarrow
Tm {Body, s, m}
*(x, s) - ** t = SIGMA s (la x t)*

ugly :: Vec {n} String \rightarrow Tm {p, s, n} \rightarrow String
ugly xs (L ENil x b) = "(\\ \" ++ x ++ \" -> \" ++ ugly (x :> xs) b ++ \")"
ugly xs (LK e) = "(\\ _ -> \" ++ ugly xs e ++ \")"
ugly xs (ARR s t) = "(\" ++ ugly xs s ++ \" -> \" ++ ugly xs t ++ \")"
ugly xs (PI s (L ENil x t)) = "((\" ++ x ++ \" : \" ++ ugly xs s ++ \") -> \"
++ ugly (x :> xs) t ++ \")"
*ugly xs (TIMES s t) = "(\" ++ ugly xs s ++ \" * \" ++ ugly xs t ++ \")"*
*ugly xs (SIGMA s (L ENil x t)) = "((\" ++ x ++ \" : \" ++ ugly xs s ++ \") * \"*
++ ugly (x :> xs) t ++ \")"
ugly xs SET = "Set"
ugly xs (h : \$ B0) = ugly xs h
ugly xs (h : \$ es) = "(\" ++ ugly xs h ++ foldMap (\e \rightarrow \" \" ++ ugly xs e) es ++ \")"
ugly xs (V i) = xs ! >! i
ugly xs (P (i, t)) = "! \" ++ show i
ugly xs (D (s, -, -) B0 _) = s
ugly xs (D (s, -, -) es _) = "(\" ++ s ++ foldMap (\e \rightarrow \" \" ++ ugly V0 e) es ++ \")"
ugly xs (ENil : / e) = ugly xs e
ugly _ _ = "???"

2.2 NameSupply

The NameSupply is the name generator used throughout Epigram. It is inspired by the *hierarchical names* [14] used in Epigram the First. The aim of this structure is to conveniently, provide unique free variable names.

A `NameSupply` is composed by a backward list of `(String, Int)` and an `Int`. This corresponds to a hierarchical namespace and a free name in that namespace. The structure of the namespace stack is justified as follows. The `String` component provides name advice, which may not be unique, while the `Int` uniquely identifies the namespace.

```
type NameSupply = (Bwd (String, Int), Int)
```

Therefore, creating a fresh name in a given namespace simply consists of incrementing the name counter:

```
freshName :: NameSupply → NameSupply
freshName (sis, i) = (sis, i + 1)
```

Whereas creating a fresh namespace involves stacking up a new name `s`, uniquely identified by `i`, and initializing the per-namespace counter to 0:

```
freshNSpace :: NameSupply → String → NameSupply
freshNSpace (sis, i) s = (sis :< (s, i), 0)
```

Intuitively, the function `name` computes a fresh name out of a given name generator, decorating it with the human-readable label `s`. Technically, `Name` is defined as a list of `(String, Int)`. Hence, on that structure, the effect of `trail` is to flatten the backward namespace into a (unique) `Name`.

```
mkName :: NameSupply → String → Name
mkName (sis, i) s = trail $ sis :< (s, i)
```

The `NameSupplier` type-class aims at giving the ability to use the `NameSupply` in a safe way. There is trade-off here between ease of implementation and safety. As it stands now, this version offers moderate safety but is easy to use. Ideally, we would like most of the code to use `NameSupplier` instead of manipulating the `NameSupply` explicitly.

So, what does `NameSupplier` offer?

```
class (Applicative m, Monad m) ⇒ NameSupplier m where
```

First, `freshRef` enables the safe creation of fresh names inside the structure: it is provided with an informative name, the variable type, and a `body` consuming that free variable. It returns the body with the free variable filled in, while maintaining the coherency of the namespace.

```
freshRef :: (String :<: TY) → (REF → m t) → m t
```

Similarly, `forkNSupply` is a safe wrapper around `freshName` and `freshNSpace`: `forkNSupply subname child dad` runs the `child` with the current namespace extended with `subname`, then, `dad` gets the result of `child`'s work and can go ahead with a fresh variable index.

```
forkNSupply :: String → m s → (s → m t) → m t
```

Finally, we have an `askNSupply` operation, to *read* the current `NameSupply`. This was a difficult choice: we give up the read-only access to the `NameSupply`, allowing the code to use it in potentially nasty ways. This operation has been motivated by `equal` that calls into `exQuote`. `exQuote` on a parameter uses and abuses some invariants of the name fabric, hence needs direct access to the `NameSupply` structure.

```
askNSupply :: m NameSupply
```

Because of the presence of `askNSupply`, we have here a kind of Reader monad on steroids. This might not be true forever; we can hope to replace `askNSupply` by a finer grained mechanism.

Sometimes you want a fresh value rather than a reference:

```
fresh :: NameSupplier m ⇒ (String :<: TY) → (VAL → m t) → m t
fresh xty f = freshRef xty (f · pval)
```

2.2.1 (\rightarrow) NameSupply is a NameSupplier

To illustrate the implementation of a NameSupplier, we implement the NameSupply Reader monad:

```
instance NameSupplier (( $\rightarrow$ ) NameSupply) where
  freshRef (x <: ty) f r = f (mkName r x := DECL <: ty) (freshName r)
  forkNSupply s child dad nsupply = (dad · child) (freshNSpace nsupply s) (freshName nsupply)
  askNSupply r = r
```

2.2.2 ReaderT NameSupply is a NameSupplier

Once we have a NameSupplier for the NameSupply Reader monad, we can actually get it for any ReaderT NameSupply. This is as simple as:

```
instance (Monad m, Applicative m)  $\Rightarrow$  NameSupplier (ReaderT NameSupply m) where
  freshRef st body = do
    nsupply  $\leftarrow$  ask
    lift $ freshRef st (runReaderT · body) nsupply
  forkNSupply s child dad = do
    c  $\leftarrow$  local (flip freshNSpace s) child
    d  $\leftarrow$  local freshName (dad c)
    return d
  askNSupply = ask
```

2.2.3 The Check monad is a NameSupplier

One such example is the Check monad:

```
type Check e = ReaderT NameSupply (Either (StackError e))
```

That is, a Reader of NameSupply on top of an Error of StackError. Running a type-checking process is therefore a simple *runReader* operation:

```
typeCheck :: Check e a  $\rightarrow$  NameSupply  $\rightarrow$  Either (StackError e) a
typeCheck = runReaderT
```

2.3 Variable Manipulation

Variable manipulation, in all its forms, ought be handled by the mangler. A Mangle $f x y$ is a record that describes how to deal with parameters of type x , variables and binders, producing terms with parameters of type y and wrapping the results in some applicative functor f .

It contains three fields:

mangP describes what to do with parameters. It takes a parameter value and a spine (a list of eliminators) that this parameter is applied to (handy for christening); it must produce an appropriate term.

mangV describes what to do with variables. It takes a de Bruijn index and a spine as before, and must produce a term.

mangB describes how to update the Mangle in response to a given binder name.

```
data Mangle f x y = Mang
  { mangP :: x  $\rightarrow$  f (Spine {TT} y)  $\rightarrow$  f (ExTm y)
  , mangV :: Int  $\rightarrow$  f (Spine {TT} y)  $\rightarrow$  f (ExTm y)
  , mangB :: String  $\rightarrow$  Mangle f x y
  }
```

The interpretation of a Mangle is given by the % operator. This mangles a term, producing a term with the appropriate parameter type in the relevant idiom. This is basically a traversal, but calling the appropriate fields of Mangle for each parameter, variable or binder encountered.

```
(%) :: Applicative f => Mangle f x y -> Tm {In, TT} x -> f (Tm {In, TT} y)
m % L (K t) = (| L (| K (m % t) |) |)
m % L (x : . t) = (| L (| (x : .) (mangB m x % t) |) |)
m % C c = (| C ((m%) ^ $ c) |)
m % N n = (| N (exMang m n (| [] |)) |)
```

The corresponding behaviour for ExTms is implemented by *exMang*.

```
exMang :: Applicative f => Mangle f x y ->
  Tm {Ex, TT} x -> f [Elim (Tm {In, TT} y)] -> f (Tm {Ex, TT} y)
exMang m (P x) es = mangP m x es
exMang m (V i) es = mangV m i es
exMang m (o : @ a) es = (| (| (o : @) ((m%) ^ $ a) |) $ : $ es |)
exMang m (t : $ e) es = exMang m t (| ((m%) ^ $ e) : es |)
exMang m (t : ? y) es = (| (| (m % t) : ? (m % y) |) $ : $ es |)
```

The %% and %%# operators apply mangles that use the identity functor.

```
(%%) :: Mangle Identity x y -> Tm {In, TT} x -> Tm {In, TT} y
m %% t = runIdentity $ m % t
```

```
(%%#) :: Mangle Identity x y -> Tm {Ex, TT} x -> Tm {Ex, TT} y
m %%# t = runIdentity $ exMang m t (| [] |)
```

2.3.1 The *under* mangle

The *under* *i y* mangle binds the variable with de Bruijn index *i* to the parameter *y* and leaves the term otherwise unchanged.

```
under :: Int -> x -> Mangle Identity x x
under i y = Mang
  { mangP = λx ies -> (| (P x $ : $) ies |)
  , mangV = λj ies -> (| ((if i ≡ j then P y else V j) $ : $) ies |)
  , mangB = λ_ -> under (i + 1) y
  }
}
```

The *underScope* function goes under a binding, instantiating the bound variable to the given reference.

```
underScope :: Scope {TT} x -> x -> InTm x
underScope (K t) _ = t
underScope (_ : . t) x = under 0 x %% t
```

2.3.2 The deBruijnifying mangle

This thing takes a stack of REFs and traverses a term, turning them into deBruijn indices (in the hope that somebody out there will λ-abstract them). It gets used when quoting λ-abstractions in values, and when building λ-abstractions in the proof state.

```

(-||) :: Bwd REF → INTM → INTM
es -|| t = disMangle es 0 %% t
where
  disMangle :: Bwd REF → Int → Mangle Identity REF REF
  disMangle ys i = Mang
    { mangP = λx ies → (| (h ys x i$ : $) ies |)
    , mangV = λi ies → (| (V i$ : $) ies |)
    , mangB = λ_ → disMangle ys (i + 1)
    }
  h B0 x i = P x
  h (ys :< y) x i
    | x ≡ y      = V i
    | otherwise = h ys x (i + 1)

```

2.3.3 The substitution mangle

The *substitute* function implements substitution for closed terms: given a list of typed references, a corresponding list of terms and a target term, it substitutes the terms for the references in the target.

```

substitute :: Bwd (REF :<: INTM) → Bwd INTM → INTM → INTM
substitute bs vs t = substMangle bs vs %% t
where
  substMangle :: Bwd (REF :<: INTM) → Bwd INTM → Mangle Identity REF REF
  substMangle bs vs = Mang
    { mangP = λx ies → (| (help bs vs x$ : $) ies |)
    , mangV = λi ies → (| (V i$ : $) ies |)
    , mangB = λ_ → substMangle bs vs
    }
  help :: Bwd (REF :<: INTM) → Bwd INTM → REF → EXTM
  help B0 B0 x = P x
  help (bs :< (y :<: ty)) (vs :< v) x
    | x ≡ y      = v ?? ty
    | otherwise = help bs vs x

```

2.3.4 The *inc* mangle

The *inc* mangle increments the bound variables in the term, allowing a binding to be inserted for the call term. It keeps track of how many local binders it has gone under, so as to not increment them.

```

inc :: Int → Mangle Identity x x
inc n = Mang
  { mangP = λx ies → (| (P x$ : $) ies |)
  , mangV = λj ies → (| (V (if j ≥ n then j + 1 else j)$ : $) ies |)
  , mangB = \_ → inc (n + 1)
  }

```

2.4 Evaluation

In this section, we implement an interpreter for Epigram. As one would expect, it will become handy during type-checking. We assume that evaluated terms have been type-checked beforehand, that is: the interpreter always terminates.

The interpreter is decomposed in four sections. First, the application of eliminators, implemented by `$$`. Second, the execution of operators, implemented by `@@`. Third, reduction under binder, implemented by `body`. Finally, full term evaluation, implemented by `eval`. At the end, this is all wrapped inside `evTm`, which evaluate a given term in an empty environment.

2.4.1 Elimination

The `$$` function applies an elimination principle to a value. Note that this is open to further extension as we add new constructs and elimination principles to the language. Extensions are added through the `ElimComputation` aspect.

Formally, the computation rules of the featureless language are the following:

$$(\lambda_.v)u \mapsto v \quad (2.1)$$

$$(\lambda x.t)v \mapsto \text{eval } t[x \mapsto v] \quad (2.2)$$

$$\text{unpack}(Con\ t) \mapsto t \quad (2.3)$$

$$(Nn)\$e \mapsto N(n\ \$e) \quad (2.4)$$

The rules 2.1 and 2.2 are standard lambda calculus stories. Rule 2.3 is the expected “unpacking the packer” rule. Rule 2.4 is justified as follow: if no application rule applies, this means that we are stuck. This can happen if and only if the application is itself stuck. The stuckness therefore propagates to the whole elimination.

This translates into the following code:

```

($$) :: VAL → Elim VAL → VAL
L (K v)  $$ A _ = v          -- By 2.1
L (H (vs, rho) x t) $$ A v
  = eval t (vs :< v, naming x v rho) -- By 2.2
L (x : . t) $$ A v
  = eval t (B0 :< v, naming x v []) -- By 2.2
C (Con t) $$ Out = t        -- By 2.3
-- import j- ElimComputation - Extensions
-- [Feature = Labelled]
LRET t $$ Call l = t
-- [/Feature = Labelled]
-- [Feature = Sigma]
PAIR x y $$ Fst = x
PAIR x y $$ Snd = y
-- [/Feature = Sigma]
N n      $$ e      = N (n : $ e)      -- By 2.4
f        $$ e      = error $ "Can't eliminate\n" ++ show f ++
                          "\nwith eliminator\n" ++ show e

```

The `naming` operation amends the current naming scheme, taking account the instantiation of `x`: see below.

The left fold of `$$` applies a value to a bunch of eliminators:

```

($$$) :: (Foldable f) ⇒ VAL → f (Elim VAL) → VAL
($$$) = Data.Foldable.foldl ($)

```

2.4.2 Operators

Running an operator is quite simple, as operators come with the mechanics to run them. However, we are not ensured to get a value out of an applied operator: the operator might get stuck

by a neutral argument. In such case, the operator will blame the argument by returning it on the Left. Otherwise, it returns the computed value on the Right.

Hence, the implementation of $@@$ is as follow. First, run the operator. On the left, the operator is stuck, so return the neutral term consisting of the operation applied to its arguments. On the right, we have gone down to a value, so we simply return it.

$$\begin{aligned} (@@) &:: \text{Op} \rightarrow [\text{VAL}] \rightarrow \text{VAL} \\ \text{op } @@ \text{ vs} &= \text{either } (_ \rightarrow \mathbf{N} (\text{op} : @ \text{ vs})) \text{ id } (\text{opRun op vs}) \end{aligned}$$

Note that we respect the invariant on \mathbf{N} values: we have an $: @$ that, for sure, is applying at least one stuck term to an operator that uses it.

2.4.3 Binders

Evaluation under binders needs to distinguish two cases:

- the binder ignores its argument, or
- a variable x is defined and bound in a term t

In the first case, we can trivially go under the binder and innocently evaluate. In the second case, we turn the binding – a term – into a closure – a value. The body grabs the current environment, extends it with the awaited argument, and evaluate the whole term down to a value.

This naturally leads to the following code:

$$\begin{aligned} \text{body} &:: \text{Scope } \{\text{TT}\} \text{ REF} \rightarrow \text{ENV} \rightarrow \text{Scope } \{\text{VV}\} \text{ REF} \\ \text{body } (\mathbf{K} \ v) \ g &= \mathbf{K} \ (\text{eval } v \ g) \\ \text{body } (x : . \ t) \ (\text{B0}, \ \text{rho}) &= \text{txtSub rho } x : . \ t \quad \text{-- closed lambdas stay syntax} \\ \text{body } (x : . \ t) \ g@(_, \ \text{rho}) &= \mathbf{H} \ g \ (\text{txtSub rho } x) \ t \end{aligned}$$

Now, as well as making closures, the current renaming scheme is applied to the bound variable name, for cosmetic purposes.

2.4.4 Evaluator

Putting the above pieces together, plus some machinery, we are finally able to implement an evaluator. On a technical note, we are working in the $\text{Applicative} \rightarrow \text{ENV}$ and use She's notation for writing applicatives.

The evaluator is typed as follow: provided with a term and a variable binding environment, it reduces the term to a value. The implementation is simply a matter of pattern-matching and doing the right thing. Hence, we evaluate under lambdas by calling *body* (a). We reduce canonical term by evaluating under the constructor (b). We drop off bidirectional annotations from Ex to In , just reducing the inner term n (c). Similarly for type ascriptions, we ignore the type and just evaluate the term (d).

If we reach a parameter, either it is already defined or it is still not binded. In both case, *pval* is the right tool: if the parameter is intrinsically associated to a value, we grab that value. Otherwise, we get the neutral term consisting of the stuck parameter (e).

A bound variable simply requires to extract the corresponding value from the environment (f). Elimination is handled by $$$$ defined above (g). And similarly for operators with $@@$ (h).

```

eval :: Tm {d, TT} REF → ENV → VAL
eval (L b)      = (| L (body b) |)      -- By (a)
eval (C c)      = (| C (eval ^$ c) |)    -- By (b)
eval (N n)      = eval n                -- By (c)
eval (t :? _)   = eval t                -- By (d)
eval (P x)      = (| (pval x) |)        -- By (e)
eval (V i)      = evar i                -- By (f)
eval (t :$ e)   = (| eval t $$ (eval ^$ e) |) -- By (g)
eval (op :@ vs) = (| (op@@) (eval ^$ vs) |) -- By (h)
eval (Yuk v)    = (| v |)

```

```

evar :: Int → ENV → VAL
evar i (vs, ts) = fromMaybe (error "eval: bad index") (vs !. i)

```

Finally, the evaluation of a closed term simply consists in calling the interpreter defined above with the empty environment.

```

evTm :: Tm {d, TT} REF → VAL
evTm t = eval t (B0, [])

```

2.4.5 Alpha-conversion on the fly

Here's a bit of a dirty trick which sometimes results in better name choices. We firstly need the notion of a textual substitution from `Tm.lhs`.

```

type TXTSUB = [(Char, String)] -- renaming plan

```

That's a plan for mapping characters to strings. We apply them to strings like this, with no change to characters which aren't mapped.

```

txtSub :: TXTSUB → String → String
txtSub ts = foldMap blat where blat c = fromMaybe [c] $ lookup c ts

```

The `ENV` type packs up a renaming scheme, which we apply to every bound variable name advice string that we encounter as we go: the deed is done in `body`, above.

The renaming scheme is amended every time we instantiate a bound variable with a free variable. Starting from the right, each character of the bound name is mapped to the corresponding character of the free name. The first character of the bound name is mapped to the whole remaining prefix. So instantiating `"xys"` with `"monks"` maps `'y'` to `"k"` and `'x'` to `"mon"`. The idea is that matching the target of an eliminator in this way will give good names to the variables bound in its methods, if we're lucky and well prepared.

```

naming :: String → VAL → TXTSUB → TXTSUB
naming x (N (P y)) rho
  = mkMap (reverse x) (reverse (refNameAdvice y)) rho where
    mkMap "" _ rho = rho
    mkMap _ "" rho = rho
    mkMap [c] s rho | s ≠ [c] = (c, s) : rho
    mkMap (c : cs) (c' : s) rho | c ≠ c' = mkMap cs s ((c, [c']) : rho)
    mkMap (_ : cs) (_ : s) rho = mkMap cs s rho
naming _ _ rho = rho

```

2.5 Type-checker

2.5.1 Type-checking Canonicals and Eliminators

Canonical objects

Historically, canonical terms were type-checked by the following function:

```
canTy :: (t → VAL) → (Can VAL :>: Can t) → Maybe (Can (TY :>: t))
canTy ev (Set :>: Set) = Just Set
canTy ev (Set :>: Pi s t) = Just (Pi (SET :>: s) ((ARR (ev s) SET) :>: t))
canTy _ _ = Nothing
```

If we temporarily forget Features, we have here a type-checker that takes an evaluation function ev , a type, and a term to be checked against this type. When successful, the result of typing is a canonical term that contains both the initial term and its normalized form, as we get it for free during type-checking.

However, to avoid re-implementing the typing rules in various places, we had to generalize this function. The generalization consists in parameterizing $canTy$ with a type-directed function $TY :>: t \rightarrow s$, which is equivalent to $TY \rightarrow t \rightarrow s$. Because we still need an evaluation function, both functions are fused into a single one, of type: $TY :>: t \rightarrow (s, VAL)$. To support failures, we extend this type to $TY :>: t \rightarrow m (s, VAL)$ where m is a `MonadError`.

Hence, by defining an appropriate function $chev$, we can recover the previous definition of $canTy$. We can also do much more: intuitively, we can write any type-directed function in term of $canTy$. That is, any function traversing the types derivation tree can be expressed using $canTy$.

```

canTy :: (Alternative m, MonadError (StackError t) m) =>
  (TY :>: t → m (s :=>: VAL)) →
  (Can VAL :>: Can t) →
  m (Can (s :=>: VAL))
canTy chev (Set :>: Set) = return Set
canTy chev (Set :>: Pi s t) = do
  ssv@(s :=>: sv) ← chev (SET :>: s)
  ttv@(t :=>: tv) ← chev (ARR sv SET :>: t)
  return $ Pi ssv ttv
-- import j- CanTyRules
-- [Feature = Anchor]
canTy chev (Set :>: Anchors) = return Anchors
canTy chev (Anchors :>: Anchor u t ts) = do
  uv ← chev (UID :>: u)
  ttv@(t :=>: tv) ← chev (SET :>: t)
  tsv ← chev (ALLOWEDBY tv :>: ts)
  return $ Anchor uv ttv tsv
canTy chev (Set :>: AllowedBy t) = do
  ttv ← chev (SET :>: t)
  return $ AllowedBy ttv
canTy chev (AllowedBy t :>: AllowedEpsilon) = do
  return $ AllowedEpsilon
canTy chev (AllowedBy ty :>: AllowedCons _S _T q s ts) = do
  _SSv@(_S :=>: _Sv) ← chev (SET :>: _S)
  _TTv@(_T :=>: _Tv) ← chev (ARR _Sv SET :>: _T)
  qqv ← chev (PRF (EQBLUE (SET :>: ty) (SET :>: PI _Sv _Tv)) :>: q)
  ssv@(s :=>: sv) ← chev (_Sv :>: s)
  tsv ← chev (ALLOWEDBY (_Tv $$ (A sv)) :>: ts)
  return $ AllowedCons _SSv _TTv qqv ssv tsv
-- [/Feature = Anchor]
-- [Feature = Enum]
canTy chev (Set :>: EnumT e) = do
  eev@(e :=>: ev) ← chev (enumU :>: e)
  return $ EnumT eev
canTy _ (EnumT (CON e) :>: Ze) | CONSN ← e $$ Fst = return Ze
canTy chev (EnumT (CON e) :>: Su n) | CONSN ← e $$ Fst = do
  nnv@(n :=>: nv) ← chev (ENUMT (e $$ Snd $$ Snd $$ Fst) :>: n)
  return $ Su nnv
-- [/Feature = Enum]
-- [Feature = Equality]
canTy chev (Prop :>: EqBlue (y0 :>: t0) (y1 :>: t1)) = do
  y0y0v@(y0 :=>: y0v) ← chev (SET :>: y0)
  t0t0v@(t0 :=>: t0v) ← chev (y0v :>: t0)
  y1y1v@(y1 :=>: y1v) ← chev (SET :>: y1)
  t1t1v@(t1 :=>: t1v) ← chev (y1v :>: t1)
  return $ EqBlue (y0y0v :>: t0t0v) (y1y1v :>: t1t1v)
canTy chev (Prf (EQBLUE (y0 :>: t0) (y1 :>: t1)) :>: Con p) = do
  ppv@(p :=>: pv) ← chev (PRF (eqGreen @@ [y0, t0, y1, t1]) :>: p)
  return $ Con ppv
-- [/Feature = Equality]
-- [Feature = IDesc]
canTy chev (Set :>: IMu (ml :? = (Id ii : & Id x)) i) = do
  iiiv@(ii :=>: iiv) ← chev (SET :>: ii)
  mlv ← traverse (chev · (ARR iiv ANCHORS:>:)) ml
  xxv@(x :=>: xv) ← chev (ARR iiv (idesc $$ A iiv) :>: x)
  iiv ← chev (iiv :>: i)
  return $ IMu (mlv :? = (Id iiiv : & Id xxv)) iiv
canTy chev (IMu tt@(_ :? = (Id ii : & Id x)) i :>: Con y) = do
  yyv ← chev (idescOp @@ [ii
    , x $$ A i
    , L $ "i" : . [i .

```

```

canTy chev (Set :=> Sch) = return Sch
canTy chev (Sch :=> SchTy s) = do
  ssv ← chev (SET :=> s)
  return $ SchTy ssv
canTy chev (Sch :=> SchExpPi s t) = do
  ssv@(_ :=> sv) ← chev (SCH :=> s)
  ttv ← chev (ARR (schTypeOp @@ [sv]) SCH :=> t)
  return $ SchExpPi ssv ttv
canTy chev (Sch :=> SchImpPi s t) = do
  ssv@(_ :=> sv) ← chev (SET :=> s)
  ttv ← chev (ARR sv SCH :=> t)
  return $ SchImpPi ssv ttv
  -- [/Feature = Problem]
  -- [Feature = Prop]
canTy _ (Set :=> Prop) = return Prop
canTy chev (Set :=> Prf p) = (| Prf (chev (PROP :=> p)) |)
canTy chev (Prop :=> All s p) = do
  ssv@(_ :=> sv) ← chev (SET :=> s)
  ppv ← chev (ARR sv PROP :=> p)
  return $ All ssv ppv
canTy chev (Prop :=> And p q) =
  (| And (chev (PROP :=> p)) (chev (PROP :=> q)) |)
canTy _ (Prop :=> Trivial) = return Trivial
canTy _ (Prop :=> Absurd) = return Absurd
canTy chev (Prf p :=> Box (Irr x)) = (| (Box · Irr) (chev (PRF p :=> x)) |)
canTy chev (Prf (AND p q) :=> Pair x y) = do
  (| Pair (chev (PRF p :=> x)) (chev (PRF q :=> y)) |)
canTy _ (Prf TRIVIAL :=> Void) = return Void
canTy chev (Prop :=> Inh ty) = (| Inh (chev (SET :=> ty)) |)
canTy chev (Prf (INH ty) :=> Wit t) = (| Wit (chev (ty :=> t)) |)
  -- [/Feature = Prop]
  -- [Feature = Sigma]
canTy _ (Set :=> Unit) = return Unit
canTy chev (Set :=> Sigma s t) = do
  ssv@(s :=> sv) ← chev (SET :=> s)
  ttv@(t :=> tv) ← chev (ARR sv SET :=> t)
  return $ Sigma ssv ttv
canTy _ (Unit :=> Void) = return Void
canTy chev (Sigma s t :=> Pair x y) = do
  xxv@(x :=> xv) ← chev (s :=> x)
  yyv@(y :=> yv) ← chev ((t $$ A xv) :=> y)
  return $ Pair xxv yyv
  -- [/Feature = Sigma]
  -- [Feature = UId]
canTy _ (Set :=> UId) = return UId
canTy _ (UId :=> Tag s) = return (Tag s)
  -- [/Feature = UId]
canTy chev (ty :=> x) = throwError' $ err "canTy: the proposed value "
  ‡ errCan x
  ‡ err " is not of type "
  ‡ errTyVal ((C ty) :=> SET)

```

Eliminators

Type-checking eliminators mirrors *canTy*. *elimTy* is provided with a checker-evaluator, a value *f* of inferred typed *t*, ie. a *f* $\text{<: } t$ of VAL <: Can VAL , and an eliminator of $\text{Elim } t$. If the operation is type-safe, we are given back the eliminator enclosing the result of *chev* and the type of the eliminated value.

it computes the type of the argument, ie. the eliminator, in $\text{Elim } (s \text{ :=> } \text{VAL})$ and the type of the result in TY .

```
elimTy :: MonadError (StackError t) m =>
  (TY :=> t → m (s :=> VAL)) →
  (VAL <: Can VAL) → Elim t →
  m (Elim (s :=> VAL), TY)
elimTy chev (f <: Pi s t) (A e) = do
  eev@(e :=> ev) ← chev (s :=> e)
  return $ (A eev, t $$ A ev)
-- import {- ElimTyRules
-- [Feature = Equality]
elimTy chev (_ <: Prf (EQBLUE (t0 :=> x0) (t1 :=> x1))) Out =
  return (Out, PRF (eqGreen @@ [t0, x0, t1, x1]))
-- [/Feature = Equality]
-- [Feature = IDesc]
elimTy chev (_ <: (IMu tt@(_ :? =: (ld ii : & ld x)) i)) Out =
  return (Out,
    idescOp @@ [ii, x $$ A i
      , L $ "i" : . [· i · C (IMu (fmap (-$[]) tt) (NV i))]])
-- [/Feature = IDesc]
-- [Feature = Labelled]
elimTy chev (_ <: Label _ t) (Call l) = do
  llv@(l :=> lv) ← chev (t :=> l)
  return (Call llv, t)
-- [/Feature = Labelled]
-- [Feature = Prop]
elimTy chev (f <: Prf (ALL p q)) (A e) = do
  eev@(e :=> ev) ← chev (p :=> e)
  return $ (A eev, PRF (q $$ A ev))
elimTy chev (_ <: Prf (AND p q)) Fst = return (Fst, PRF p)
elimTy chev (_ <: Prf (AND p q)) Snd = return (Snd, PRF q)
-- [/Feature = Prop]
-- [Feature = Sigma]
elimTy chev (_ <: Sigma s t) Fst = return (Fst, s)
elimTy chev (p <: Sigma s t) Snd = return (Snd, t $$ A (p $$ Fst))
-- [/Feature = Sigma]
elimTy _ (v <: t) e = throwError' $ err "elimTy: failed to eliminate"
  + errTyVal (v <: (C t))
  + err "with"
  + errElim e
```

question: Why not asking *m* to be *Alternative* too?

Operators

The *opTy* function explains how to interpret the telescope *opTyTel*: it labels the operator's arguments with the types they must have and delivers the type of the whole application. To do

that, one must be able to evaluate arguments. It is vital to type-check the sub-terms (left to right) before trusting the type at the end. This corresponds to the following type:

$$\begin{aligned} opTy &:: \forall t \cdot (t \rightarrow \text{VAL}) \rightarrow [t] \rightarrow \text{Maybe} ([TY \text{ :> } t], \text{TY}) \\ opTy \text{ ev args} &= (\dots) \end{aligned}$$

Where we are provided an evaluator *ev* and the list of arguments, which length should be the arity of the operator. If the type of the arguments is correct, we return them labelled with their type and the type of the result.

However, we had to generalize it. Following the evolution of *canTy* in Section 2.5.1, we have adopted the following scheme:

$$\begin{aligned} opTy &:: (\text{Alternative } m, \text{MonadError } (\text{StackError } t) m) \Rightarrow \\ &\text{Op} \rightarrow (\text{TY} \text{ :> } t \rightarrow m (s \text{ :=> } \text{VAL})) \rightarrow [t] \rightarrow \\ &m ([s \text{ :=> } \text{VAL}], \text{TY}) \end{aligned}$$

First, the *MonadError* constraint allows seamless integration in the world of things that might fail. Second, we have extended the evaluation function to perform type-checking at the same time. We also liberalise the return type to *s*, to give more freedom in the choice of the checker-evaluator. This change impacts on *exQuote*, *infer*, and *useOp*. If this definition is not clear now, it should become clear after the definition of *canTy* in Section 2.5.1.

$$\begin{aligned} opTy \text{ op chev ss} & \\ &| \text{length } ss \equiv opArity \text{ op} = telCheck \text{ chev } (opTyTel \text{ op} \text{ :> } ss) \\ opTy \text{ op } _ _ &= throwError' \$ (err "operator arity error: ") \\ &\quad \# (err \$ opName \text{ op}) \end{aligned}$$

2.5.2 Type checking

Here starts the bidirectional type-checking story. In this section, we address the Checking side. In the next section, we implement the Inference side. Give Conor a white-board, three pens of different colors, 30 minutes, and you will know what is happening below in the Edinburgh style. If you can bear with some black-and-white boring sequents, keep reading.

The checker works as follow. In a valid typing environment Γ , it checks that the term *t* is indeed of type *T*, ie. *t* can be pushed into *T*: $T \text{ :> } t$:

$$\Gamma \vdash \text{TY} \ni \text{Tm} \{ \text{In}, \} p$$

Technically, we also need a name supply and handle failure with a convenient monad. Therefore, we jump in the Check monad defined in Section 2.2.3.

$$check :: (\text{TY} \text{ :> } \text{INTM}) \rightarrow \text{Check INTM} (\text{INTM} \text{ :=> } \text{VAL})$$

Type-checking a canonical term is rather easy, as we just have to delegate the hard work to *canTy*. The checker-evaluator simply needs to evaluate the well-typed terms.

$$\begin{aligned} check (C \text{ cty} \text{ :> } C \text{ ctm}) &= \mathbf{do} \\ cc' &\leftarrow canTy \text{ check } (\text{cty} \text{ :> } \text{ctm}) \\ \text{return } \$ C \text{ ctm} \text{ :=> } &(C \$ fmap \text{valueOf } cc') \end{aligned}$$

As for lambda, it is simple too. We wish the code was simple too. But, hey, it isn't. The formal typing rule is the following:

$$\frac{x : S \vdash Tx \ni t}{\Pi S T \ni \lambda x.t}$$

As for the implementation, we apply the by-now standard trick of making a fresh variable $x \in S$ and computing the type $T x$. Then, we simply have to check that $T x \ni t$.

```

check (PI s t :>: L sc) = do
  freshRef ("__check" :<: s)
    (λref → check (t $$ A (pval ref)) :>:
      underScope sc ref))
  return $ L sc :=>: (evTm $ L sc)

```

Formally, we can bring the Ex terms into the In world with the rule:

$$\frac{n \in Y \quad \star \ni W \equiv Y}{W \ni n}$$

This translates naturally into the following code:

```

check (w :>: N n) = do
  r ← askNSupply
  yv :<: yt ← infer n
  case (equal (SET :>: (w, yt)) r) of
    True → return $ N n :=>: yv
    False → throwError' $ err "check: inferred type"
      † errTyVal (yt :<: SET)
      † err "of"
      † errTyVal (yv :<: yt)
      † err "is not"
      † errTyVal (w :<: SET)

```

Finally, we can extend the checker with the Check aspect. If no rule has matched, then we have to give up.

```

-- import {- Check
-- [Feature = Prop]
check (PRF (ALL p q) :>: L sc) = do
  freshRef (" " :<: p)
    (λref → check (PRF (q $$ A (pval ref)) :>:
      underScope sc ref))
  return $ L sc :=>: (evTm $ L sc)
-- [/Feature = Prop]

check (ty :>: tm) = throwError' $ err "check: type mismatch: type"
  † errTyVal (ty :<: SET)
  † err "does not admit"
  † errTm tm

```

2.5.3 Type inference

On the inference side, we also have a valid typing environment Γ that is used to pull types TY out of Ex terms:

$$\Gamma \vdash \text{Tm } \{ \text{Ex}, \cdot \} p \in \text{TY}$$

This translates into the following signature:

```
infer :: EXTM → Check INTM (VAL :<: TY)
```

We all know the rule to infer the type of a free variable from the context:

$$\overline{\Gamma, x : A, \Delta \vdash x \in A}$$

In Epigram, parameters carry their types, so it is even easier:

```
infer (P x) = return $ pval x <: pty x
```

The rule for eliminators is a generalization of the rule for function application. Let us give a look at its formal rule:

$$\frac{f \in \Pi S T \quad S \ni x}{fx \in (Bx)^\downarrow}$$

The story is the same in the general case: we infer the eliminated term t and we type-check the eliminator, using `elimTy`. Because `elimTy` computes the result type, we have inferred the result type.

```
infer (t : $ s) = do
  val <: ty ← infer t
  case ty of
  C cty → do
    (s', ty') ← elimTy check (val <: cty) s
    return $ (val $$ (fmap valueOf s')) <: ty'
  _ → throwError' $ err "infer: inferred type"
    ++ errTyVal (ty <: SET)
    ++ err "of"
    ++ errTyVal (val <: ty)
    ++ err "is not canonical."
```

Following exactly the same principle, we can infer the result of an operator application:

```
infer (op : @ ts) = do
  (vs, t) ← opTy op check ts
  return $ (op @@ (fmap valueOf vs)) <: t
```

Type ascription is formalized by the following rule:

$$\frac{\star \ni \text{ty} \quad \text{ty}^\downarrow \ni t}{(t : \in T) \in \text{ty}^\downarrow}$$

Which translates directly into the following code:

```
infer (t :? ty) = do
  _ :=> vty ← check (SET :=> ty)
  _ :=> v ← check (vty :=> t)
  return $ v <: vty
```

Obviously, if none of the rule above applies, then there is something fishy.

```
infer _ = throwError' $ err "infer: unable to infer type"
```

2.6 Equality and Quotation

Testing for equality is a direct application of normalization by evaluation[6, 4, 3]: to compare two values, we first bring them to their normal form. Then, it is a simple matter of syntactic equality, as defined in Section ??, to compare the normal forms.

```
equal :: (TY :=> (VAL, VAL)) → NameSupply → Bool
equal (ty :=> (v1, v2)) r = quote (ty :=> v1) r ≡ quote (ty :=> v2) r
```

quote is a type-directed operation that returns a normal form INTM by recursively evaluating the value VAL of type TY.

```
quote :: (TY :> VAL) → NameSupply → INTM
```

The normal form corresponds to a β -normal η -long form: there are no β -redexes present and all possible η -expansions have been performed.

This is achieved by two mutually recursive functions, *inQuote* and *exQuote*:

```
inQuote :: (TY :> VAL) → NameSupply → INTM
exQuote :: NEU → NameSupply → (EXTM :<: TY)
```

Where *inQuote* quotes values and *exQuote* quotes neutral terms. As we are initially provided with a value, we quote it with *inQuote*, in a fresh namespace.

```
quote vty r = inQuote vty (freshNSpace r "quote")
```

2.6.1 inQuote

Quoting a value consists in, if possible, η -expanding it. So it goes:

```
inQuote :: (TY :> VAL) → NameSupply → INTM
inQuote (C ty :> v) r | Just t ← etaExpand (ty :> v) r = t
```

Needless to say, we can always η -expand a closure. Therefore, if η -expansion has failed, there are two possible cases: either we are quoting a neutral term, or a canonical term. In the case of neutral term, we switch to *exQuote*, which is designed to quote neutral terms.

However, we are allowed to *simplify* the neutral term before handling it to *exQuote*. Simplification is discussed in greater length below. To give an intuition, it consists in transforming stuck terms into equivalent, yet simpler, stuck terms. Typically, we use this opportunity to turn some laws (monad law, functor law, etc.) to hold *definitionally*. This is known as Boutillier's trick [2].

```
inQuote (_ :> N n) r = N t
  where (t :<: _) = exQuote (simplify n r) r
```

In the case of a canonical term, we use *canTy* to check that *cv* is of type *cty* and, more importantly, to evaluate *cty*. Then, it is simply a matter of quoting this *typ :> val* inside the canonical constructor, and returning the fully quoted term. The reason for the presence of *Just* is that *canTy* asks for a *MonadError*. Obviously, we cannot fail in this code, but we have to be artificially cautious.

```
inQuote (C cty :> C cv) r = either
  (error $
    "inQuote: impossible! Type " ++ show (fmap (\_ → ()) cty) ++
    " doesn't admit " ++ show cv)
  id $ do
    ct ← canTy chev (cty :> cv)
    return $ C $ fmap termOf ct
      where chev (t :> v) = do
        let tv = inQuote (t :> v) r
            return $ tv :=> v
inQuote (x :> t) r = error $
  "inQuote: type " ++ show x ++ " doesn't admit " ++ show t
```

2.6.2 η -expansion

As mentioned above, λeta -expansion is the first sensible thing to do when quoting. Sometimes it works, especially for closures and features for which a `CanEtaExpand` aspect is defined. Quoting a closure is a bit tricky: you cannot compute under a binder as you would like to. So, we first have to generate a fresh variable v . Then, we apply v to the function f , getting a value of type $t v$. At this point, we can safely quote the term. The result is a binding of v in the quoted term.

```

etaExpand :: (Can VAL :> VAL) → NameSupply → Maybe INTM
etaExpand (Pi s t :> f) r = Just $
  L ("__etaExpandA" : .
    fresh ("__etaExpandB" :<: s)
      (\v → inQuote (t $$ A v :> (f $$ A v))) r)
  -- import j- CanEtaExpand
  -- [Feature = Prop]
etaExpand (Prf p :> x) r = Just (BOX (Irr (inQuote (PRF p :> x) r)))
  -- [/Feature = Prop]
  -- [Feature = Sigma]
etaExpand (Unit :> v) r = Just VOID
etaExpand (Sigma s t :> p) r = let x = p $$ Fst in
  Just (PAIR (inQuote (s :> x) r) (inQuote (t $$ (A x) :> (p $$ Snd)) r))
  -- [/Feature = Sigma]

etaExpand _ _ = Nothing

```

2.6.3 exQuote

Now, let us examine the quotation of neutral terms. Remember that a neutral term is either a parameter, a stuck elimination, or a stuck operation. Hence, we consider each cases in turn.


```
exQuote :: NEU → NameSupply → (EXTM :<: TY)
```

To quote a free variable, ie. a parameter, the idea is the following. If we are asked to quote a free variable P , there are two possible cases. First case, we have introduced it during an η -expansion (see `etaExpand`, above). Hence, we know that it is bound by a lambda: it needs to be turned into the bound variable V , with the right De Bruijn index. Second case, we have not introduced it: we can simply return it as such.

```

exQuote (P x) r = quop x r :<: pty x
  where quop :: REF → NameSupply → EXTM
        quop ref@(ns := _) r = help (bwdList ns) r
          where
            help (ns :<: (-, i)) (r, j) = if ns ≡ r then V (j - i - 1) else P ref

```

. The code above relies on the very structure of the names, as provided by the `NameSupply`. We know that a free variable has been created by `quote` if and only if the current name supply and the namespace `ns` of the variable are the same. Hence, the test `ns ≡ r`. Then, we compute the De Bruijn index of the bound variable by counting the number of lambdas traversed up to now – by looking at $j - 1$ in our current name supply (r, j) – minus the number of lambdas traversed at the time of the parameter creation, ie. i . Do some math, pray, and you get the right De Bruijn index. □

If an elimination is stuck, it is because the function is stuck while the arguments are ready to go. So, we have to recursively *exQuote* the neutral application, while *inQuote*-ing the arguments.

```

exQuote (n : $ v) r = (n' : $ e') :<: ty'
  where (n' :<: ty) = exQuote n r
        e' = fmap termOf e
        Right (e, ty') = elimTy chev (N n :<: unC ty) v
        chev (t :>: x) = do
          let tx = inQuote (t :>: x) r
              return $ tx :=>: x
        unC :: VAL → Can VAL
        unC (C ty) = ty

```

Similarly, if an operation is stuck, this means that one of the value passed as an argument needs to be *inQuote*-ed. So it goes. Note that the operation itself cannot be stuck: it is a simple fully-applied constructor which can always compute.

```

exQuote (op : @ vs) r = (op : @ vals) :<: v
  where (vs', v) = either (error "exQuote: impossible happened.")
                        id $ opTy op chev vs
        vals = map termOf vs'
        chev (t :>: x) = do
          let tx = inQuote (t :>: x) r
              return $ tx :=>: x

```

2.6.4 Simplification of stuck terms

question: Got to write something about that. Me tired. Another time.

```

simplify :: NEU → NameSupply → NEU
simplify n r = exSimp n r


inSimp :: VAL → NameSupply → VAL
inSimp (N n) = (| N (exSimp n) |)
inSimp v = (| v |)

exSimp :: NEU → NameSupply → NEU
exSimp (P x) = (| (P x) |)
exSimp (n : $ el) = (| exSimp n : $ (inSimp ^ $ el) |)
exSimp (op : @ vs) = opS op ⊗ (inSimp ^ $ vs)
  where
    opS op r vs = case opSimp op vs r of
      Nothing → op : @ vs
      Just n → n

```

2.7 β -Quotation

As we are in the quotation business, let us define β -quotation, ie. *bquote*. Unlike *quote*, *bquote* does not perform η -expansion, it just brings the term in β -normal form. Therefore, the code is much more simpler than *quote*, although the idea is the same.

 It is important to note that we are in a NameSupplier and we don't require a specially crafted NameSupply (unlike *quote* and *quop*). Because of that, we have to maintain the variables we have generated and to which De Bruijn index they correspond. This is the role of the backward list

of references. Note also that we let the user provide an initial environment of references: this is useful to discharge a bunch of assumptions inside a term. □

Apart from that, this is a standard β -quotation:

$$bquote :: \text{NameSupplier } m \Rightarrow \text{Bwd REF} \rightarrow \text{Tm } \{d, \text{VV}\} \text{ REF} \rightarrow m (\text{Tm } \{d, \text{TT}\} \text{ REF})$$

If binded by one of our lambda, we bind the free variables to the right lambda. We don't do anything otherwise.

$$\begin{aligned} bquote \text{ refs } (P \ x) = & \\ \text{case } x \text{ 'elemIndex' refs of} & \\ \text{Just } i \rightarrow \text{pure } \$ \ V \ i & \\ \text{Nothing} \rightarrow \text{pure } \$ \ P \ x & \end{aligned}$$

Constant lambdas are painlessly structural.

$$bquote \text{ refs } (L \ (K \ t)) = (| \text{LK } (bquote \text{ refs } t) |)$$

When we see a syntactic lambda value, we are very happy, because quotation is just renaming.

Pierre: This is part of Conor's experiment on Term representations. The following line could be de-commented and that would be ok. However, this does not compute as much as the next case, hence the current Cochon user has to see things he doesn't want to see. This could be solved by some Distillation work but we prefer to avoid over-engineering Cochon's Distillation for the time being.

$$\text{-- } bquote \text{ refs } (L \ (x \ . : t)) = ((\text{refs } - | \ L \ (x \ . : t)))$$

For all other lambdas, it's the usual story: we create a fresh variable, evaluate the applied term, quote the result, and bring everyone under a binder.

$$\begin{aligned} bquote \text{ refs } f @ (L \ _) = & \\ (| \ (L \ \cdot (x \ . : _)) & \\ \text{freshRef } (x \ . : \text{error "bquote: type undefined"}) & \\ (\lambda x \rightarrow bquote \text{ refs } _ < x) & \\ (f \ \$ \$ \ A \ (pval \ x))) & |) \\ \text{where } x = \text{fortran } f & \end{aligned}$$

For the other constructors, we simply go through and do as much damage as we can. Simple, easy.

$$\begin{aligned} bquote \text{ refs } (C \ c) &= (| \ C \ (\text{traverse } (bquote \text{ refs}) \ c) |) \\ bquote \text{ refs } (N \ n) &= (| \ N \ (bquote \text{ refs } n) |) \\ bquote \text{ refs } (n \ : \ \$ \ v) &= (| \ (bquote \text{ refs } n) : \$ \ (\text{traverse } (bquote \text{ refs}) \ v) |) \\ bquote \text{ refs } (op \ : \ @ \ vs) &= (| \ (op \ : \ @) \ (\text{traverse } (bquote \text{ refs}) \ vs) |) \end{aligned}$$

2.8 Operators and primitives

In this section, we weave some She aspects. In particular, we bring inside `Rules.lhs` the *operators* defined by feature files, along with any auxiliary code.

```

operators :: [Op]
operators = (
  -- import j- Operators
  -- [Feature = Enum]
  branchesOp :
  switchOp :
  enumInductionOp :
    -- [/Feature = Enum]
    -- [Feature = Equality]
  eqGreen :
  coe :
  coh :
    -- [/Feature = Equality]
    -- [Feature = IDesc]
  idescOp :
  iboxOp :
  imapBoxOp :
  iinductionOp :
    -- [/Feature = IDesc]
    -- [Feature = Problem]
  argsOp :
  schTypeOp :
    -- [/Feature = Problem]
    -- [Feature = Prop]
  nEOp :
  inhEOp :
    -- [/Feature = Prop]
    -- [Feature = Sigma]
  splitOp :
    -- [/Feature = Sigma]
  [])

```

The list of *primitives* includes axioms and fundamental definitions provided by the Primitives aspect, plus a reference corresponding to each operator.

```

primitives :: [(String, REF)]
primitives = map ( $\lambda op \rightarrow (opName\ op, mkRef\ op)$ ) operators ++ (
  -- import j- Primitives
  -- [Feature = Enum]
  ("EnumU", enumREF) :
  ("EnumD", enumDREF) :
  ("EnumConstructors", enumConstructorsREF) :
  ("EnumBranches", enumBranchesREF) :
  -- [/Feature = Enum]
  -- [Feature = Equality]
  ("cohAx", cohAx) :
  ("refl", refl) :
  ("substEq", substEq) :
  ("symEq", symEq) :
  -- [/Feature = Equality]
  -- [Feature = IDesc]
  ("IDesc", idescREF) :
  ("IDescD", idescDREF) :
  ("IDescConstructors", idescConstREF) :
  ("IDescBranches", idescBranchesREF) :
  -- [/Feature = IDesc]
  [])
where
mkRef :: Op  $\rightarrow$  REF
mkRef op = [("Operators", 0), (opName op, 0)] := (DEFN opEta <: opTy)
  where
    opTy = pity (opTyTel op) (((BO <: ("Operators", 0) <:
      (opName op, 0) <:
      ("opTy", 0)), 0) :: NameSupply)

    ari = opAriety op
    args = map NV [(ari - 1), (ari - 2)..0]
    names = take (ari - 1) (map ( $\lambda x \rightarrow [x]$ ) ['b' ..])
    opEta = L $ "a" : . Prelude.foldr ( $\lambda s\ x \rightarrow L\ (s : . x)$ ) (N $ op : @ args) names

```

We can look up the primitive reference corresponding to an operator using *lookupOpRef*. This ensures we maintain sharing of these references.

```

lookupOpRef :: Op  $\rightarrow$  REF
lookupOpRef op = case lookup (opName op) primitives of
  Just ref  $\rightarrow$  ref
  Nothing  $\rightarrow$  error $ "lookupOpRef: missing operator primitive " ++ show op

```

```

pity :: NameSupplier m  $\Rightarrow$  TEL TY  $\rightarrow$  m TY
pity (Target t) = return t
pity (x <: s : - : t) = do
  freshRef (x <: error "pity': type undefined")
  ( $\lambda xref \rightarrow$  do
    t  $\leftarrow$  pity $ t (pval xref)
    t  $\leftarrow$  bquote (BO <: xref) t
    return $ PI s (L $ x : . t))

```

```

-- import j- OpCode
-- [Feature = Enum]
type EnumDispatchTable = (VAL, VAL → VAL → VAL)
mkLazyEnumDef :: VAL → EnumDispatchTable → Either NEU VAL
mkLazyEnumDef arg (nilECase, consECase) = let args = arg $$ Snd in
  case arg $$ Fst of
    NILN   → Right $ nilECase
    CONSN  → Right $ consECase (args $$ Fst) (args $$ Snd $$ Fst)
    N t    → Left t
    _     → error "mkLazyEnumDef: invalid constructor!"

branchesOp = Op
  { opName = "branches"
  , opArity = 2
  , opTyTel = bOpTy
  , opRun = {-bOpRun -} runOpTree $
    oData [oTup $ OLam $ \_P → ORet UNIT
          , oTup $ λ() _E → OLam $ \_P → ORet $
            TIMES (_P $$ A ZE)
            (branchesOp @@ [_E, L $ "x" : . [x · _P - $ [SU (NV x)]])]
          ]
  , opSimp = \_ → empty
  } where
  bOpTy = "e" :<: enumU : - : λe →
    "p" :<: ARR (ENUMT e) SET : - : λp →
    Target SET

switchOp = Op
  { opName = "switch"
  , opArity = 4
  , opTyTel = sOpTy
  , opRun = runOpTree $ {-sOpRun - makeOpRun "switch" switchTest -}
    OLam $ \_ →
    OCase (map projector [0..])
  , opSimp = \_ → empty
  } where
  projector i = OLam $ \_ → OLam $ λbs → ORet (proj i bs)
  proj 0 bs = bs $$ Fst
  proj i bs = proj (i - 1) (bs $$ Snd)
  sOpTy =
    "e" :<: enumU : - : λe →
    "x" :<: ENUMT e : - : λx →
    "p" :<: ARR (ENUMT e) SET : - : λp →
    "b" :<: branchesOp @@ [e, p] : - : λb →
    Target (p $$ A x)

```

```

enumInductionOp = Op
  { opName = "enumInduction"
  , opArity = 5
  , opTyTel = eOpTy
  , opRun = runOpTree $
    oData [ oTup $ OSet $ λc → OBarf
          , oTup $ λt e →
            OSet $ λc →
              oLams $ λp mz ms →
                case c of
                  Ze      → ORet (mz $$ A t $$ A e)
                  (Su x) → ORet (ms $$ A t $$ A e $$ A x $$ A
                                (enumInductionOp @@ [e, x, p, mz, ms]))
                ]
  , opSimp = λ_ _ → empty
  } where
  eOpTy =
    "e" :<: enumU : - : λe →
    "x" :<: ENUMT e : - : λx →
    "p" :<: ARR (SIGMA enumU (L $ "e" : . [.e · ENUMT (NV e)])) SET : - : λp →
    "mz" :<: (PI UID $ L $ "t" : . [.t ·
      PI enumU $ L $ "e" : . [.e ·
        p -$ [PAIR (CONSE (NV t) (NV e)) ZE]
      ]]) : - : λmz →
    "ms" :<: (PI UID $ L $ "t" : . [.t ·
      PI enumU $ L $ "e" : . [.e ·
        PI (ENUMT (NV e)) $ L $ "x" : . [.x ·
          ARR (p -$ [PAIR (NV e) (NV x)])
            (p -$ [PAIR (CONSE (NV t) (NV e)) (SU (NV x))])
          ]]) : - : λms →
      Target (p $$ A (PAIR e x))
  -- [/Feature = Enum]

  -- [Feature = Equality]
eqGreen = Op { opName = "eqGreen"
  , opArity = 4
  , opTyTel = "S" :<: SET : - : λsS → "s" :<: sS : - : λs →
    "T" :<: SET : - : λtT → "t" :<: tT : - : λt →
      Target PROP
  , opRun = opRunEqGreen
  , opSimp = \_ _ → empty
  } where
  opty chev [y0, t0, y1, t1] = do
    (y0 :=>: y0v) ← chev (SET :=>: y0)
    (t0 :=>: t0v) ← chev (y0v :=>: t0)
    (y1 :=>: y1v) ← chev (SET :=>: y1)
    (t1 :=>: t1v) ← chev (y1v :=>: t1)
    return ([y0 :=>: y0v
      , t0 :=>: t0v
      , y1 :=>: y1v
      , t1 :=>: t1v]
      , PROP)
  opty _ _ = throwError' "eqGreen: invalid arguments."

```

```

coe = Op { opName = "coe"
, opArity = 4
, opTyTel = "S" :<: SET : - : λsS → "T" :<: SET : - : λtT →
          "Q" :<: PRF (EQBLUE (SET :>: sS) (SET :>: tT)) : - : λ_ →
          "s" :<: sS : - : λ_ → Target tT
, opRun = oprun
, opSimp = λ[sS, tT, _, s] r → case s of
  N s | equal (SET :>: (sS, tT)) r → pure s
  _ → ()
} where
oprun :: [VAL] → Either NEU VAL
oprun [_S, _T, q, v] | partialEq _S _T q = Right v
oprun [C x, C y, q, s] = case halfZip x y of
  Nothing → Right $ nEOp @@ [q $$ Out, C y]
  Just fxy → coerce fxy (q $$ Out) s
oprun [N x, y, q, s] = Left x
oprun [x, N y, q, s] = Left y
oprun vs = error ("coe: undefined for arguments"
  + unlines (map show vs))

```

```

coh = Op { opName = "coh"
, opArity = 4
, opTyTel =
  "S" :<: SET : - : λ_S → "T" :<: SET : - : λ_T →
  "Q" :<: PRF (EQBLUE (SET :>: _S) (SET :>: _T)) : - : λ_Q →
  "s" :<: _S : - : λs → Target $ PRF $
  EQBLUE (_S :>: s) (_T :>: (coe @@ [_S, _T, _Q, s]))
, opRun = oprun
, opSimp = λ[_S, _T, _, s] r →
  if equal (SET :>: (_S, _T)) r
  then pure $ P refl : $ A _S : $ A s
  else ()
} where
oprun :: [VAL] → Either NEU VAL
oprun [_S, _T, q, s] | partialEq _S _T q =
  Right (pval refl $$ A _S $$ A s)
oprun [N x, y, q, s] = Left x
oprun [x, N y, q, s] = Left y
oprun [_S, _T, _Q, s] = Right $
  pval cohAx $$ A _S $$ A _T $$ A _Q $$ A s
oprun vs = error ("coh: undefined for arguments"
  + unlines (map show vs))
-- [/Feature = Equality]
-- [Feature = IDesc]
unTag :: VAL → String
unTag (TAG u) = u
unTag _ = "x"

```

```

idescOp :: Op
idescOp = Op
  { opName = "idesc"
  , opArity = 3
  , opTyTel = idOpTy
  , opRun = runOpTree $ OLam $ \I → oData {-idOpRun -}
  [ {-VAR -} oTup $ λi → OLam $ \P → ORet $ _P $$ A i
  , {-CONST -} oTup $ \A → OLam $ \P → ORet _A
  , {-PI -} oTup $ \S _T → OLam $ \P → ORet $
    PI _S $ L $ "s" : . [·s · N $
      idescOp : @ [I -$ [], _T -$ [NV s], _P -$ []]]
  , {-FPI -} oTup $ \E _Df → OLam $ \P → ORet $
    branchesOp @@
      [ _E
      , (L $ "e" : . [·e · N $
        idescOp : @ [I -$ []
          , _Df -$ [NV e]
          , _P -$ []
        ])]
      ]
  , {-SIGMA -} oTup $ \S _T → OLam $ \P → ORet $
    SIGMA _S $ L $ (fortran _T) : . [·s · N $
      idescOp : @ [I -$ [], _T -$ [NV s], _P -$ []]]
  , {-FSIGMA -} oTup $ \E _Ds → OLam $ \P → ORet $
    SIGMA (ENUMT _E) (L $ (fortran _Ds) : . [·s · N $
      idescOp : @ [I -$ []
        , N $ switchOp : @
          [ _E -$ []
          , NV s
          , LK (idesc -$ [I -$ []])
          , _Ds -$ []
          , _P -$ []])
      ]
  , {-PROD -} oTup $ λu _D _D' → OLam $ \P → ORet $
    SIGMA (idescOp @@ [I, _D, _P]) $ L $ (unTag u) : .
      (N (idescOp : @ [I -$ [], _D' -$ [], _P -$ []]))
  ]
, opSimp = \_ → empty
} where
idOpTy =
  "I" :<: SET : - : λiI →
  "d" :<: (idesc $$ A iI) : - : λd →
  "x" :<: ARR iI SET : - : λx →
  Target SET

```

```

iboxOp :: Op
iboxOp = Op
  { opName = "ibox"
  , opArity = 4
  , opTyTel = iboxOpTy
  , opRun = runOpTree $ OLam $ \I → oData {-iboxOpRun -}
    [ {-VAR -} oTup $ λi → oLams $ λ() v → ORet $
      IVAR (PAIR i v)
    , {-CONST -} oTup $ λ() → oLams $ λ() () → ORet $
      ICONST (PRF TRIVIAL)
    , {-PI -} oTup $ \S _T → oLams $ \P f → ORet $
      IPI _S (L $ "s" : . [·s · N $
        iboxOp : @ [ _I -$ [], _T -$ [NV s]
          , _P -$ [], f -$ [NV s]])
    , {-FPI -} oTup $ \E _Df → oLams $ \P v → ORet $
      IFPI _E (L $ "e" : . [·e · N $
        iboxOp : @ [ _I -$ [], _Df -$ [NV e], _P -$ []
          , N $ switchOp : @
            [ _E -$ [], NV e
              , L $ "f" : . [·f · N $
                idescOp : @ [ _I -$ []
                  , _Df -$ [NV f]
                  , _P -$ []]])
          , v -$ []]])
    , {-SIGMA -} oTup $ λ() _T → OLam $ \P → OPr $ oLams $ λs d → ORet $
      iboxOp @@ [ _I, _T $$ A s, _P, d]
    , {-FSIGMA -} oTup $ \E _Ds → OLam $ \P → OPr $ oLams $ λe d → ORet $
      iboxOp @@ [ _I
        , switchOp @@ [ _E
          , e
          , LK (idesc $$ A _I)
          , _Ds]
        , _P
        , d]
    , {-PROD -} oTup $ λu _D _D' → OLam $ \P → OPr $ oLams $ λd d' → ORet $
      IPROD (TAG (unTag u ++ "h")) (iboxOp @@ [ _I, _D, _P, d])
        (iboxOp @@ [ _I, _D', _P, d'])
    ]
  , opSimp = \_ → empty
} where
iboxOpTy =
  "I" <: SET          : - : λ_I →
  "D" <: (idesc $$ A _I) : - : λ_D →
  "P" <: ARR _I SET   : - : λ_P →
  "v" <: idescOp @@ [ _I, _D, _P ] : - : λ_v →
  Target $ idesc $$ A (SIGMA _I (L $ "i" : . [·i · _P -$ [NV i]]))

```

```

imapBoxOp :: Op
imapBoxOp = Op
  { opName = "imapBox"
  , opArity = 6
  , opTyTel = imapBoxOpTy
  , opRun = runOpTree $ OLam $ \l → oData {-imapBoxOpRun -}
    [ {-VAR -} oTup $ λi → oLams $ λ() () p v → ORet $ p $$ A (PAIR i v)
    , {-CONST -} oTup $ λ() → oLams $ λ() () () → ORet $ VOID
    , {-PI -} oTup $ λ() _T → oLams $ \X _P p f → ORet $
      L $ "s" : . [·s · N $
        imapBoxOp : @ [ _I -$ [], _T -$ [NV s]
          , _X -$ [], _P -$ [], p -$ [], f -$ [NV s]]]
    , {-FPI -} oTup $ λ() _Df → oLams $ \X _P p v → ORet $
      L $ "s" : . [·s · N $
        imapBoxOp : @ [ _I -$ [], _Df -$ [NV s]
          , _X -$ [], _P -$ [], p -$ [], v -$ [NV s]]]
    , {-SIGMA -} oTup $ λ() _T → oLams $ \X _P p → OPr $ oLams $ λs d → ORet $
      imapBoxOp @@ [ _I, _T $$ A s, _X, _P, p, d]
    , {-FSIGMA -} oTup $ \_E _Ds → oLams $ \X _P p → OPr $ oLams $ λe d → ORet $
      imapBoxOp @@ [ _I
        ,
          switchOp @@ [ _E, e
            ,
              LK (idesc $$ A _I)
            ,
              _Ds
            ]
          ,
            _X, _P, p, d]
    , {-PROD -} oTup $ λ() _D _D' → oLams $ \X _P p → OPr $ oLams $ λd d' → ORet $
      PAIR (imapBoxOp @@ [ _I, _D, _X, _P, p, d])
        (imapBoxOp @@ [ _I, _D', _X, _P, p, d'])
    ]
  , opSimp = \_ → empty
  } where
imapBoxOpTy =
  "I" :<: SET : - : \l →
  "D" :<: (idesc $$ A _I) : - : λ_D →
  "X" :<: ARR _I SET : - : λ_X →
  let _IX = SIGMA _I (L $ "i" : . [·i · _X -$ [NV i]]) in
  "P" :<: ARR _IX SET : - : λ_P →
  "p" :<: (PI _IX $ L $ "ix" : . [·ix · _P -$ [NV ix]]) : - : λ_ →
  "v" :<: (idescOp @@ [ _I, _D, _X]) : - : λv →
  Target (idescOp @@ [ _IX, iboxOp @@ [ _I, _D, _X, v], _P])

iinductionOpMethodType _I _D _P =
PI _I $ L $ "i" : . [·i ·
  let _It = _I -$ []
    mud = L $ "j" : . [·j · IMU Nothing _It (_D -$ []) (NV j)]
  in PI (N (idescOp : @ [ _It, _D -$ [NV i], mud])) $ L $ "x" : . [·x ·
  ARR (N (idescOp : @ [SIGMA _It mud
    , N (iboxOp : @ [ _It, _D -$ [NV i], mud, NV x])
    , _P -$ [ ]))
    (_P -$ [PAIR (NV i) (CON (NV x))]))]

```

```

iinductionOp :: Op
iinductionOp = Op
  { opName = "iinduction"
  , opArity = 6
  , opTyTel = iinductionOpTy
  , opRun = runOpTree $ oLams $ \_I _D i → OCon $ oLams $ λv _P p → ORet $
    p $$ A i $$ A v
      $$ A (imapBoxOp @@ [_I, _D] $$ A i
        , (L $ "i" : . [.i .
          IMU Nothing (_I -$ []) (_D -$ []) (NV i)]
        , _P
        , L $ "ix" : . [.ix . N $
          iinductionOp : @
            [_I -$ [], _D -$ []
            , N (V ix : $ Fst), N (V ix : $ Snd)
            , _P -$ [], p -$ []])
        , v])
    , opSimp = \_ → empty
  } where
  iinductionOpTy =
    "I" :<: SET : - : \_I →
    "D" :<: ARR _I (idesc $$ A _I) : - : \_D →
    "i" :<: _I : - : λi →
    "v" :<: IMU Nothing _I _D i : - : λv →
    "P" :<: (ARR (SIGMA _I (L $ "i" : . [.i .
      IMU Nothing (_I -$ []) (_D -$ []) (NV i)])) SET) : - : \_P →
    "p" :<: (iinductionOpMethodType _I _D _P) : - : λp →
    Target (_P $$ A (PAIR i v))
  -- [/Feature = IDesc]
  -- [Feature = Problem]
argsOp = Op
  { opName = "args"
  , opArity = 1
  , opTyTel = "s" :<: SCH : - : λ_ → Target SET
  , opRun = λ[s] → argsOpRun s
  , opSimp = λ_ → empty
  }

schTypeOp = Op
  { opName = "schType"
  , opArity = 1
  , opTyTel = "s" :<: SCH : - : λ_ → Target SET
  , opRun = λ[s] → schTypeOpRun s
  , opSimp = λ_ → empty
  }

```

```

argsOpRun :: VAL → Either NEU VAL
argsOpRun (SCHTY _) = Right UNIT
argsOpRun (SCHEXPPI s t) =
  Right $ SIGMA (schTypeOp @@ [s])
    (L ("x" : . [·x · N $ argsOp : @ [t -$ [NV x]]]))
argsOpRun (SCHIMPPI s t) =
  Right $ SIGMA s
    (L ("x" : . [·x · N $ argsOp : @ [t -$ [NV x]]]))
argsOpRun (N v) = Left v

```

```

schTypeOpRun :: VAL → Either NEU VAL
schTypeOpRun (SCHTY s)      = Right s
schTypeOpRun (SCHEXPPI s t) =
  Right $ PI (schTypeOp @@ [s])
    (L ("x" : . [·x · N $ schTypeOp : @ [t -$ [NV x]]]))
schTypeOpRun (SCHIMPPI s t) =
  Right $ PI s
    (L ("x" : . [·x · N $ schTypeOp : @ [t -$ [NV x]]]))
schTypeOpRun (N v)          = Left v
-- [/Feature = Problem]
-- [Feature = Prop]
nEOp = Op { opName = "naughtE"
, opArity = 2
, opTyTel = "z" :<: PRF ABSURD : - : λ_ →
           "x" :<: SET : - : λxX → Target xX
, opRun = runOpTree $ OCon $ OBarf
, opSimp = \_ → empty
}
inhEOp = Op { opName = "inh"
, opArity = 4
, opTyTel = "s" :<: SET : - : λty →
           "p" :<: PRF (INH ty) : - : λp →
           "P" :<: IMP (PRF (INH ty)) PROP : - : λpred →
           "m" :<: PI ty (L $ "s" : . [·t ·
           pred -$ [WIT (NV t)]]) : - : λ_ →
           Target (PRF (pred $$ A p))
, opRun = λ[_ , p , -, m] → case p of
  WIT t → Right $ m $$ A t
  N n → Left n
, opSimp = \_ → empty
}
-- [/Feature = Prop]
-- [Feature = Sigma]
splitOp = Op
{ opName = "split", opArity = 5
, opTyTel = "A" :<: SET : - : λaA →
           "B" :<: ARR aA SET : - : λbB →
           "ab" :<: SIGMA aA bB : - : λab →
           "P" :<: ARR (SIGMA aA bB) SET : - : λpP →
           "p" :<: (
             PI aA $ L $ "a" : . [·a ·
             PI (bB -$ [NV a]) $ L $ "b" : . [·b ·
             pP -$ [PAIR (NV a) (NV b)]]]) : - : λp →
           Target $ pP $$ A ab
, opRun = runOpTree $
  oLams $ λ() () ab () p → ORet $ p $$ A (ab $$ Fst) $$ A (ab $$ Snd)
, opSimp = \_ → empty
}
-- [/Feature = Sigma]

-- import j- RulesCode
-- [Feature = Enum]
enumConstructors :: Tm {In, p} x
enumConstructors = CONSE (TAG "nil") (CONSE (TAG "cons") NILE)

```

```

enumBranches :: Tm {ln, p} x
enumBranches =
  PAIR (ICONST UNIT)
    (PAIR (ISIGMA UID (L $ "t" :. (IPROD (TAG "E") (IVAR VOID) (ICONST UNIT))))
      VOID)

enumD :: Tm {ln, p} x
enumD = LK $
  IFSIGMA enumConstructors enumBranches

enumU :: Tm {ln, p} x
enumU = IMU (Just (LK $ ANCHOR (TAG "EnumU") SET ALLOWEDEPSILON))
  UNIT enumD VOID

enumREF :: REF
enumREF = [("Primitive", 0), ("EnumU", 0)] := DEFN enumU <: SET

enumDREF :: REF
enumDREF = [("Primitive", 0), ("EnumD", 0)] := DEFN enumD <:
  ARR UNIT (idesc $$ A UNIT)

enumConstructorsREF :: REF
enumConstructorsREF = [("Primitive", 0), ("EnumConstructors", 0)] :=
  DEFN enumConstructors <: enumU

enumBranchesREF :: REF
enumBranchesREF = [("Primitive", 0), ("EnumBranches", 0)] :=
  DEFN enumBranches <:
    branchesOp @@ [enumConstructors, LK (idesc $$ A UNIT)]
  -- [/Feature = Enum]
  -- [Feature = Equality]
cohAx = [("Axiom", 0), ("coh", 0)] := (DECL <: cohType) where
  cohType = PRF $
    ALL SET $ L $ "s" :. [.S ·
    ALL SET $ L $ "T" :. [.T ·
    ALL (PRF (EQBLUE (SET >: NV _S) (SET >: NV _T)))
      $ L $ "Q" :. [.Q ·
    ALL (NV _S) $ L $ "s" :. [.s ·
    EQBLUE (NV _S >: NV s)
      (NV _T >: N (coe : @ [NV _S, NV _T, NV _Q, NV s]))
    ]]]
refl = [("Axiom", 0), ("refl", 0)] := (DECL <: reflType) where
  reflType = PRF $ ALL SET $ L $ "s" :. [.S ·
    ALL (NV _S) $ L $ "s" :. [.s ·
    EQBLUE (NV _S >: NV s) (NV _S >: NV s)]

```

```

substEq = [("Primitive",0),("substEq",0)] := DEFN seDef :<: seTy where
  seTy = PI SET $ L $ "x" : . [._X .
    PI (NV _X) $ L $ "x" : . [.x .
    PI (NV _X) $ L $ "y" : . [.y .
    PI (PRF (EQBLUE (NV _X :>: NV x) (NV _X :>: NV y))) $ L $ "q" : . [.q .
    PI (ARR (NV _X) SET) $ L $ "p" : . [._P .
    ARR (N (V _P : $ A (NV x))) (N (V _P : $ A (NV y)))
  ]]]]
  seDef = L $ "x" : . [._X .
    L $ "x" : . [.x .
    L $ "y" : . [.y .
    L $ "q" : . [.q .
    L $ "p" : . [._P .
    L $ "px" : . [.px .
    N (coe : @ [N (V _P : $ A (NV x)), N (V _P : $ A (NV y)),
      CON (N (P refl : $ A (ARR (NV _X) SET) : $ A (NV _P) : $ Out
        : $ A (NV x) : $ A (NV y) : $ A (NV q))),
      NV px])
  ]]]]

symEq = [("Primitive",0),("symEq",0)] := DEFN def :<: ty where
  ty = PRF $ ALL SET $ L $ "x" : . [._X .
    ALL (NV _X) $ L $ "x" : . [.x .
    ALL (NV _X) $ L $ "y" : . [.y .
    IMP (EQBLUE (NV _X :>: NV x) (NV _X :>: NV y))
      (EQBLUE (NV _X :>: NV y) (NV _X :>: NV x))
  ]]]
  def = L $ "x" : . [._X .
    L $ "x" : . [.x .
    L $ "y" : . [.y .
    L $ "q" : . [.q .
    N (P refl : $ A (ARR (NV _X) SET)
      : $ A (L $ "z" : . [.z .
        PRF (EQBLUE (NV _X :>: NV z) (NV _X :>: NV x))))
      : $ Out
      : $ A (NV x)
      : $ A (NV y)
      : $ A (NV q)
      : $ Fst
      : $ A (N (P refl : $ A (NV _X) : $ A (NV x))))
  ]]]]
  -- [/Feature = Equality]
  -- [Feature = IDesc]
inIDesc :: VAL
inIDesc = L $ "I" : . [._I . LK $ IFSIGMA constructors (cases (NV _I))]

constructors = (CONSE (TAG "varD")
  (CONSE (TAG "constD")
    (CONSE (TAG "piD")
      (CONSE (TAG "fpiD")
        (CONSE (TAG "sigmaD")
          (CONSE (TAG "fsigmaD")
            (CONSE (TAG "prodD")
              NILE)))))))))

```

```

cases :: INTM → INTM
cases _I =
  {-varD: -} (PAIR (ISIGMA _I (LK $ ICONST UNIT)))
  {-constD: -} (PAIR (ISIGMA SET (LK $ ICONST UNIT)))
  {-piD: -} (PAIR (ISIGMA SET (L $ "S" : . [..S .
    (IPROD (TAG "T") (IPI (NV _S) (LK $ IVAR VOID))
    (ICONST UNIT))))))
  {-fpiD: -} (PAIR (ISIGMA (enumU -$ []) (L $ "E" : . [..E .
    (IPROD (TAG "T") (IPI (ENUMT (NV _E)) (LK $ IVAR VOID))
    (ICONST UNIT))))))
  {-sigmaD: -} (PAIR (ISIGMA SET (L $ "S" : . [..S .
    (IPROD (TAG "T") (IPI (NV _S) (LK $ IVAR VOID))
    (ICONST UNIT))))))
  {-fsigmaD: -} (PAIR (ISIGMA (enumU -$ []) (L $ "E" : . [..E .
    (IPROD (TAG "T") (IFPI (NV _E) (LK $ IVAR VOID))
    (ICONST UNIT))))))
  {-prodD: -} (PAIR (ISIGMA UID (L $ "u" : . (IPROD (TAG "C") (IVAR VOID) (IPROD (TAG "D") (IVAR VOID))))))

```

```

idescFakeREF :: REF
idescFakeREF = [("Primitive", 0), ("IDesc", 0)]
  := (FAKE <: ARR SET (ARR UNIT SET))
idesc :: VAL
idesc = L $ "I" : . [..I .
  IMU (Just (L $ "i" : . [..i . ANCHOR (TAG "IDesc")
    (ARR SET SET)
    (ALLOWEDCONS SET
      (LK SET)
      (N (P refl : $ A SET : $ A (ARR SET SET)))
      (NV _I)
      ALLOWEDEPSILON))))
    UNIT (inIDesc -$ [NV _I]) VOID]

```

```

idescREF :: REF
idescREF = [("Primitive", 0), ("IDesc", 0)]
  := (DEFN idesc <: ARR SET SET)
idescDREF :: REF
idescDREF = [("Primitive", 0), ("IDescD", 0)]
  := (DEFN inIDesc
    <: ARR SET (ARR UNIT (idesc $$ A UNIT)))

```

```

idescConstREF :: REF
idescConstREF = [("Primitive", 0), ("IDescConstructors", 0)]
  := (DEFN constructors <: enumU)

```

```

idescBranchesREF :: REF
idescBranchesREF = [("Primitive", 0), ("IDescBranches", 0)]
  := (DEFN (L $ "I" : . [..I . cases (NV _I)])) <:
  PI SET (L $ "I" : . [..I .
    N $ branchesOp : @ [ constructors,
      LK $ N (P idescREF : $ A UNIT)]])

```

```

sumilike :: VAL → VAL → Maybe (VAL, VAL → VAL)
sumilike _I (IFSIGMA e b) =
  Just (e, λt → switchOp @@ [e, t, LK (idesc $$ A _I), b])
sumilike _ _ = Nothing

-- [/Feature = IDesc]

```

2.9 Operator DSL

Pierre: Crying for documentation (and more user-friendly syntax).

Based on, but not quite the same as Edwin's experimental operator DSEL, try this.

```

data OpTree
  = OLam (VAL → OpTree)
  | OPr OpTree
  | OCase [OpTree]
  | OCon OpTree
  | OSet (Can VAL → OpTree)
  | ORet VAL
  | OBarf

oData :: [OpTree] → OpTree
oData = OCon · OPr · OCase

class OLams t where
  oLams :: t → OpTree
instance OLams OpTree where
  oLams = id
instance OLams t ⇒ OLams (() → t) where
  oLams f = OLam $ λ_ → oLams (f ())
instance OLams t ⇒ OLams (VAL → t) where
  oLams = OLam · (oLams·)

class OTup t where
  oTup :: t → OpTree
instance OTup OpTree where
  oTup = OLam · const
instance OTup t ⇒ OTup (() → t) where
  oTup f = OPr · OLam $ λ_ → oTup (f ())
instance OTup t ⇒ OTup (VAL → t) where
  oTup = OPr · OLam · (oTup·)

runOpTree :: OpTree → [VAL] → Either NEU VAL
runOpTree (OLam f) (x : xs) = runOpTree (f x) xs
runOpTree (OPr f) (v : xs) = runOpTree f (v $$$ Fst : v $$$ Snd : xs)
runOpTree (OCase bs) (i : xs) = (| (bs!!) (num i) |) ≫ λb → runOpTree b xs where
  num :: VAL → Either NEU Int
  num ZE = (| 0 |)
  num (SU n) = (| (1+) (num n) |)
  num (N e) = Left e
runOpTree (OCon f) (CON t : xs) = runOpTree f (t : xs)
runOpTree (OSet f) (C c : xs) = runOpTree (f c) xs
runOpTree (ORet v) xs = Right (v $$$ map A xs)
runOpTree _ (N e : xs) = Left e

```

Grot! Why does the Monad (Either e) instance demand (Error e)? I shut it up.

```
instance Error NEU where
  strMsg = error
```

2.10 Observational Equality

Let's have some observational equality, now! [1]

question: Can we move this to the appropriate feature file? Does nested use of import aspects work?

The *eqGreen* operator, defined in section ??, computes the proposition that two values are equal if their containing sets are equal. We write \leftrightarrow for application of this operator.

$$(\leftrightarrow) :: (\text{TY} \text{:>: VAL}) \rightarrow (\text{TY} \text{:>: VAL}) \rightarrow \text{VAL}$$

$$(y0 \text{:>: } t0) \leftrightarrow (y1 \text{:>: } t1) = \text{eqGreen } @@ [y0, t0, y1, t1]$$

$$(<: - \text{:>}) :: (\text{INTM} \text{:>: INTM}) \rightarrow (\text{INTM} \text{:>: INTM}) \rightarrow \text{INTM}$$

$$(y0 \text{:>: } t0) <: - \text{:> } (y1 \text{:>: } t1) = \text{N } \$ \text{eqGreen} : @ [y0, t0, y1, t1]$$

We define the computational behaviour of the *eqGreen* operator as follows,

```
opRunEqGreen :: [VAL] → Either NEU VAL
```

```
-- import j- OpRunEqGreen
-- [Feature = Prop]
opRunEqGreen [PROP, t1, PROP, t2] = Right $ AND (IMP t1 t2) (IMP t2 t1)
opRunEqGreen [PRF -, -, PRF -, -] = Right TRIVIAL
-- [/Feature = Prop]
-- [Feature = Sigma]
opRunEqGreen [UNIT, -, UNIT, -] = Right TRIVIAL
opRunEqGreen [SIGMA s1 t1, p1, SIGMA s2 t2, p2] = Right $
  AND (eqGreen @@ [s1, p1 $$ Fst, s2, p2 $$ Fst])
    (eqGreen @@ [t1 $$ A (p1 $$ Fst), p1 $$ Snd, t2 $$ A (p2 $$ Fst), p2 $$ Snd])
-- [/Feature = Sigma]
-- [Feature = UId]
opRunEqGreen [UID, TAG s1, UID, TAG s2] | s1 ≡ s2 = Right $ TRIVIAL
opRunEqGreen [UID, TAG -, UID, TAG -] = Right $ ABSURD
-- [/Feature = UId]

opRunEqGreen [C (Pi sS1 tT1), f1, C (Pi sS2 tT2), f2] = Right $
  ALL sS1 $ L $ "s1" : . [·s1 ·
    ALL (sS2 -$ []) $ L $ "s2" : . [·s2 ·
      IMP (EQBLUE ((sS1 -$ []) :>: NV s1) ((sS2 -$ []) :>: NV s2)) $
        (tT1 -$ [NV s1] :>: f1 -$ [NV s1])
          <: - :> (tT2 -$ [NV s2] :>: f2 -$ [NV s2])]]]

opRunEqGreen [SET, PI sS1 tT1, SET, PI sS2 tT2] = Right $
  AND ((SET :>: sS2) ↔ (SET :>: sS1)) $
    ALL sS2 $ L $ "s2" : . [·s2 ·
      ALL (sS1 -$ []) $ L $ "s1" : . [·s1 ·
        IMP (EQBLUE ((sS2 -$ []) :>: NV s2) ((sS1 -$ []) :>: NV s1)) $
          (SET :>: (tT1 -$ [NV s1]) <: - :> (SET :>: (tT2 -$ [NV s2])))]]]
```

Unless overridden by a feature or preceding case, we determine equality of canonical values in canonical sets by labelling subterms of the values with their types, half-zipping them together (ensuring that the head constructors are identical) and requiring that the subterms are equal.

```

opRunEqGreen [C ty0, C t0, C ty1, C t1] =
  case halfZip (fmap termOf t0') (fmap termOf t1') of
    Nothing → Right ABSURD
    Just x  → Right $ mkEqConj (trail x)
  where
    Right t0' = canTy (λtx@(t :>: x) → Right (tx :=>: x)) (ty0 :>: t0)
    Right t1' = canTy (λtx@(t :>: x) → Right (tx :=>: x)) (ty1 :>: t1)

```

If we are trying to equate a function and a canonical value, we don't have much hope.

```

opRunEqGreen [_, L _, _, C _] = Right ABSURD
opRunEqGreen [_, C _, _, L _] = Right ABSURD

```

If one of the arguments is neutral, we blame it for being unable to compute.

```

opRunEqGreen [C _, N t0, C _, _] = Left t0
opRunEqGreen [C _, _, C _, N t1] = Left t1
opRunEqGreen [N y0, _, _, _] = Left y0
opRunEqGreen [_, _, N y1, _] = Left y1

```

Otherwise, something has gone horribly wrong.

```

opRunEqGreen as = error $ "opRunEqGreen: unmatched " ++ show as

```

The *mkEqConj* function builds a conjunction of *eqGreen* propositions by folding over a list. It is uniformly structural for canonical terms, ignoring contravariance. Therefore, this requires a special case for Pi in *opRunEqGreen*.

```

mkEqConj :: [(TY :>: VAL, TY :>: VAL)] → VAL
mkEqConj ((tt0, tt1) : []) = tt0 ↔ tt1
mkEqConj ((tt0, tt1) : xs) = AND (tt0 ↔ tt1) (mkEqConj xs)
mkEqConj [] = TRIVIAL

```

The *coeh* function takes two types, a proof that they are equal and a value in the first type; it produces a value in the second type and a proof that it is equal to the original value. If the sets are definitinoally equal then this is easy, otherwise it applies *coe* to the value and uses the coherence axiom *coh* to produce the proof.

```

coeh :: TY → TY → VAL → VAL → (VAL, VAL)
coeh s t q v | partialEq s t q = (v, pval refl $$ A s $$ A v)
coeh s t q v = (coe @@ [s, t, q, v]
  ,
  coh @@ [s, t, q, v])

```

```

coehin :: TY → TY → VAL → INTM → (INTM, INTM)
coehin s t q v | partialEq s t q = (v, pval refl -$ [s -$ [], v])
coehin s t q v = (N $ coe : @ [s -$ [], t -$ [], q -$ [], v]
  ,
  N $ coh : @ [s -$ [], t -$ [], q -$ [], v])

```

The *coerce* function transports values between equal canonical sets. Given two sets built from the same canonical constructor (represented as Can (VAL, VAL), a proof of their equality and an element of the first set, it will try to return Right *v* where *v* is an element of the second set. If computation is blocked by a neutral value *n*, it will return Left *n*.

Features must extend this definition using the `Coerce` aspect for every canonical set-former they introduce. They must handle coercions between all canonical inhabitants of such sets, but need not deal with neutral inhabitants. To ensure we can add arbitrary consistent axioms to the system, they should not inspect the proof, but may eliminate it with `naughtE` if asked to coerce between incompatible sets.

```

coerce :: (Can (VAL, VAL)) → VAL → VAL → Either NEU VAL
coerce Set q x = Right x
coerce (Pi (sS1, sS2) (tT1, tT2)) q f1 =
  Right · L $ (fortran tT2) : . [·s2 · N $
    let (s1, sq) = coehin sS2 sS1 (CON $ q $$$ Fst) (NV s2)
        t1 = f1 -$ [s1]
    in coe : @ [tT1 -$ [s1], tT2 -$ [NV s2]
      , CON $ (q $$$ Snd) -$ [NV s2, s1, sq], t1]]
-- import j- Coerce
-- [Feature = Enum]
coerce (EnumT (CONSE _ _ , CONSE _ _)) _ ZE = Right ZE
coerce (EnumT (CONSE _ e1 , CONSE _ e2)) q (SU x) = Right · SU $
  coe @@ [ENUMT e1, ENUMT e2, CON $ q $$$ Snd $$$ Snd $$$ Fst, x] -- CONSE tails
coerce (EnumT (NILE, NILE)) q x = Right x
coerce (EnumT (NILE, t@(CONSE _ _))) q x = Right $
  nEOp @@ [q, ENUMT t]
coerce (EnumT (CONSE _ _ , NILE)) q x = Right $
  nEOp @@ [q, ENUMT NILE]
-- [/Feature = Enum]
-- [Feature = IDesc]
coerce (IMu (Just (l0, l1) :?= :
  (ld (iI0, iI1) : & ld (d0, d1))) (i0, i1)) q (CON x) =
let ql = CON $ q $$$ Fst
    qiI = CON $ q $$$ Snd $$$ Fst
    qi = CON $ q $$$ Snd $$$ Snd $$$ Snd
    qd = CON $ q $$$ Snd $$$ Snd $$$ Fst
    typ =
      PI SET $ L $ "iI" : . [·iI ·
        ARR (ARR (NV iI) (idesc -$ [NV iI])) $
        ARR (NV iI) $
        ARR (ARR (NV iI) ANCHORS) SET]
    vap =
      L $ "iI" : . [·iI ·
        L $ "d" : . [·d ·
          L $ "i" : . [·i ·
            L $ "l" : . [·l · N $
              idescOp : @ [NV iI, N (V d : $ A (NV i))
                , L $ "j" : . [·j ·
                  IMU (| (NV l) |) (NV iI) (NV d) (NV j)]
                ]]]]
in Right · CON $
  coe @@ [idescOp @@ [ iI0, d0 $$$ A i0
    , L $ "i" : . [·i ·
      IMU (| (l0 -$ []) |) (iI0 -$ []) (d0 -$ []) (NV i)
    ]
  ]
  , idescOp @@ [iI1, d1 $$$ A i1
    , L $ "i" : . [·i ·
      IMU (| (l1 -$ []) |) (iI1 -$ []) (d1 -$ []) (NV i)
    ]
  ]
  , CON $ pval refl $$$ A typ $$$ A vap $$$ Out
    $$$ A iI0 $$$ A iI1 $$$ A qiI
    $$$ A d0 $$$ A d1 $$$ A qd
    $$$ A i0 $$$ A i1 $$$ A qi
    $$$ A l0 $$$ A l1 $$$ A ql
  , x]
coerce (IMu (Nothing :?= : (ld (iI0, iI1) : & ld (d0, d1))) (i0, i1)) q (CON x) =
let qiI = CON $ q $$$ Fst
    qi = CON $ q $$$ Snd $$$ Snd
    qd = CON $ q $$$ Snd $$$ Fst

```

The *partialEq* function takes two sets together with a proof that they are equal; it returns True if they are known to be definitionally equal. This is sound but not complete for the definitional equality, so if it returns False they might still be equal. It is safe to call during bquote, and hence during evaluation, because it avoids forcing the types of references.

```
partialEq :: VAL → VAL → VAL → Bool
partialEq _ _ (N (P r : $ _ : $ _)) | r ≡ refl = True
partialEq (C (IMu t1 _)) (C (IMu t2 _)) _ | eqLabelIncomplete t1 t2 = True
partialEq _ _ _ = False
```

Sadly we cannot do the following, because it is not safe to invent a name supply.

```
partialEq s t _ = bquote B0 s ns ≡ bquote B0 t ns
  where ns = (B0 :< ("__partialEq", 0), 0) :: NameSupply
```

2.11 Utilities

2.11.1 From EXTM to INTM and back again

Various commands yield an EXTM $:=>$: VAL, and we sometimes need to convert this to an INTM $:=>$: VAL.

```
neutralise :: Monad m ⇒ (EXTM :=> VAL) → m (INTM :=> VAL)
neutralise (n :=> v) = return $ N n :=> v
```

Conversely, sometimes we have an INTM and the value representation of its type, but need an EXTM. We avoid *bquote* if possible.

```
annotate :: NameSupplier m ⇒ INTM → TY → m EXTM
annotate (N n) _ = return n
annotate t      ty = bquote B0 ty ≫ return · (t?)
```

2.11.2 Discharging a list of hypotheses over a term

The *dischargeLam* function discharges and λ -binds a list of references over a term.

```
dischargeLam :: Bwd REF → INTM → INTM
dischargeLam bs v = wrapLambdas bs (bs -|| v)
  where
    wrapLambdas :: Bwd REF → INTM → INTM
    wrapLambdas B0 tm = tm
    wrapLambdas (bs :< (n := _)) tm = wrapLambdas bs (L (fst (last n) : . tm))
```

The *dischargeF* function discharges and binds a list of typed references over a term, using the given *binder* function at each step. The *binder* takes a Bool indicating whether the corresponding reference occurred in the original term, the name advice for the binder, the type of the reference and the term to be bound.

```

dischargeF :: (Bool → String → INTM → INTM → INTM) →
             Bwd (REF :<: INTM) → INTM → INTM
dischargeF binder bs v =
  wrapFs bs (fmap (v' contains') bs') (bs' -|| v)
where
  bs' = fmap fstEx bs
  contains :: INTM → REF → Bool
  contains = flip elem
  wrapFs :: Bwd (REF :<: INTM) → Bwd Bool → INTM → INTM
  wrapFs B0 B0 tm = tm
  wrapFs (bs :< ((n := _) :<: ty)) (cs :< c) tm =
    wrapFs bs cs (binder c (fst (last n)) ty tm)

```

Using the above, we can easily discharge and \forall -bind or discharge and Π -bind. Note that when the bound variable is not used, a K binder is used. For *dischargeAll*, the initial term must be in the form PRF *q* for some proposition *q*.

```

dischargeAll :: Bwd (REF :<: INTM) → INTM → INTM
dischargeAll = dischargeF f
where
  f :: Bool → String → INTM → INTM → INTM
  f False x (PRF p) (PRF q) = PRF (IMP p q)
  f _ x s (PRF q) = PRF (ALLV x s q)

dischargePi :: Bwd (REF :<: INTM) → INTM → INTM
dischargePi = dischargeF f
where
  f :: Bool → String → INTM → INTM → INTM
  f _ x p q = PIV x p q

```

The *dischargeAllREF* function calls *dischargeAll* on the type of a reference, producing a reference with the same name but whose type is \forall -abstracted over the list of references. This should be used with caution, as it could lead to having two references with the same name but different types.

```

dischargeAllREF :: Bwd (REF :<: INTM) → REF :<: INTM → REF :<: INTM
dischargeAllREF bs ((n := DECL :<: _) :<: ty) =
  (n := DECL :<: evTm ty') :<: ty'
where ty' = dischargeAll bs ty

```

The *mkFun* function turns a Haskell function into a term by applying it to a fresh reference and discharging over that reference.

```

mkFun :: NameSupplier m ⇒ (REF → INTM) → m INTM
mkFun f = freshRef ("fy" :<: error "mkFun: reference type undefined") $
  λref → return $ dischargeLam (B0 :<: ref) (f ref)

```

2.11.3 Term construction and deconstruction

The *splitSpine* function takes a neutral value and tries to split it into a reference and a spine of arguments to which it is applied. **Adam**: where should this live?

```

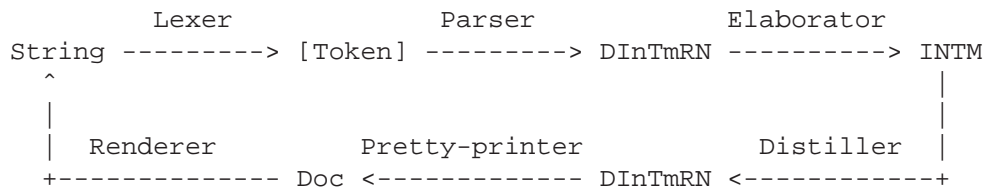
splitSpine :: NEU → Maybe (REF, [VAL])
splitSpine (P r) = return (r, [])
splitSpine (n : $ A a) = do
  (r, as) ← splitSpine n
  return (r, as ++ [a])
splitSpine _ = Nothing

```

Chapter 3

The Display Language

The life cycle of a term in the system looks like this, where vertices are labelled with the type of a representation, and edges are labelled with the transformation between representations.



In the beginning was the String. This gets lexed (section 3.4) to produce a list of Tokens, which are parsed (section 3.5) to give an DInTm RelName (a term in the display syntax containing relative names). The display term is then elaborated (section 6.5) in the ProofState monad to produce an INTM (a term in the evidence language).

Reversing the process, the distiller (section 7.1) converts an evidence term back to a display term, and the pretty-printer (section 3.6) renders this as a String.

3.1 Display Terms

3.1.1 Structure of Display Terms

Display terms mirror and extend the $Tm \{d, TT\}$ terms of the Evidence language. While the Evidence language is the language of the type-checker, the Display language is the one of humans in an interactive development. Hence, in addition to the terms from the Evidence language, we have the following:

- Question marks (holes), which are turned into subgoals during elaboration (Chapter 6) ;
- Underscores (jokers), which are inferred during elaboration ;
- Embedding of evidence terms into display terms ;
- Type annotations ; and
- Feature-specific extensions, which are imported from an aspect.

However, we have removed the following:

- Type ascriptions, replaced by type annotations ; and
- Operators, replaced by a parameter containing the corresponding reference in *primitives* (Section 2.8)



. Because of a limitation of GHC **deriving** Traversable, we define two mutually recursive data types instead of taking a `Dir` parameter. Thanks to this hack, we can use **deriving** Traversable. □

```
data DInTm :: * -> * -> * where
DL :: DScope p x      -> DInTm p x  -- λ
DC :: Can (DInTm p x) -> DInTm p x  -- canonical
DN :: DExTm p x       -> DInTm p x  -- neutral
DQ :: String          -> DInTm p x  -- hole
DU ::                 -> DInTm p x  -- underscore
DT :: InTmWrap p x    -> DInTm p x  -- embedding
-- import {-#- DInTmConstructors
-- [Feature = Anchor]
DAnchor :: String -> DInTm p x -> DInTm p x
-- [/Feature = Anchor]
-- [Feature = Equality]
```

In the display syntax, a blue equality can be between arbitrary DExTms, rather than ascriptions. To allow this, we add a suitable constructor DEqBlue to DInTm, along with appropriate elaboration and distillation rules.

```
DEqBlue :: DExTm p x -> DExTm p x -> DInTm p x
-- [/Feature = Equality]
-- [Feature = IDesc]
DIMu :: Labelled (Id : * : Id) (DInTm p x) -> DInTm p x -> DInTm p x
-- [/Feature = IDesc]
-- [Feature = UId]
DTag :: String -> [DInTm p x] -> DInTm p x
-- [/Feature = UId]
deriving (Functor, Foldable, Traversable, Show)
data DExTm p x = DHead p x $DSpine p x
deriving (Functor, Foldable, Traversable, Show)
data DHead :: * -> * -> * where
DP   :: x          -> DHead p x  -- parameter
DType :: DInTm p x -> DHead p x  -- type annotation
DTEx :: ExTmWrap p x -> DHead p x -- embedding
deriving (Functor, Foldable, Traversable, Show)
```

Note that, again, we are polymorphic in the representation of free variables. The variables in Display terms are denoted here by x . The variables of embedded Evidence terms are denoted by p . Hence, there is two potentially distinct set of free variables.

While we reuse the `Can` and `Elim` functors from `Tm`, we redefine the notion of scope. We store DExTms so as to give easy access to the head and spine for elaboration and pretty-printing.

Scopes, canonical objects and eliminators

The `DScope` functor is a simpler version of the `Scope` functor: we only ever consider *terms* here, while `Scope` had to deal with *values*. Hence, we give this purely syntactic, first-order representation of scopes:

```
data DScope :: * -> * -> * where
(:•) :: String -> DInTm p x -> DScope p x  -- binding
DK   :: DInTm p x      -> DScope p x  -- constant
deriving (Functor, Foldable, Traversable, Show)
```

We provide handy projection functions to get the name and body of a scope:

```
dScopeName :: DScope p x → String
dScopeName (x :•_) = x
dScopeName (DK _) = "_"
```

```
dScopeTm :: DScope p x → DInTm p x
dScopeTm (_ :•tm) = tm
dScopeTm (DK tm) = tm
```

Spines of eliminators are just like in the evidence language:

```
type DSpine p x = [Elim (DInTm p x)]
```

```
($ :: $) :: DExTm p x → Elim (DInTm p x) → DExTm p x
(h $s) $ :: $ a = h $(s ++ [a])
```

Embedding evidence terms

The DT and DTE_x constructors allow evidence terms to be treated as In and Ex display terms, respectively. This is useful for elaboration, because it allows the elaborator to combine finished terms with display syntax and continue elaborating. Such terms cannot be pretty-printed, however, so they should not be used in the distiller.



To make **deriving** Traversable work properly, we have to **newtype**-wrap them and manually give trivial Traversable instances for the wrappers. The instantiation code is hidden in the literate document.

□

```
newtype InTmWrap p x = InTmWrap (InTm p) deriving Show
newtype ExTmWrap p x = ExTmWrap (ExTm p) deriving Show
```

```
pattern DTIN x = DT (InTmWrap x)
pattern DTEX x = DTEx (ExTmWrap x)
```

Type annotations

Because type ascriptions are horrible things to parse¹, in the display language we use type annotations instead. The type annotation DType *ty* gets elaborated to the identity function for type *ty*, thereby pushing the type into its argument. The distiller removes type ascriptions and replaces them with appropriate type annotations if necessary.

3.1.2 Useful Abbreviations

The convention for display term pattern synonyms is that they should match their evidence term counterparts, but with the addition of Ds in appropriate places.

¹Left nesting is not really a friend of our damn parser

```

pattern DSET      = DC Set
pattern DARR s t = DPI s (DL (DK t))
pattern DPI s t   = DC (Pi s t)
pattern DCON t   = DC (Con t)
pattern DNP n    = DN (DP n :$[])
pattern DLAV x t = DL (x :•t)
pattern DPIV x s t = DPI s (DLAV x t)
pattern DLK t    = DL (DK t)
pattern DTY ty tm = DType ty :$[A tm]
  -- import j- CanDisplayPats
  -- [Feature = Anchor]
pattern DANCHOR s args = DAnchor s args
  -- [/Feature = Anchor]
  -- [Feature = Enum]
pattern DENUMT e = DC (EnumT e)
pattern DNILE     = DCON (DPAIR {-(DTAG "nil") -} DZE DVOID)
pattern DCONSE t e = DCON (DPAIR {-(DTAG "cons") -} (DSU DZE) (DPAIR t (DPAIR e DVOID)))
pattern DZE      = DC Ze
pattern DSU n   = DC (Su n)
  -- [/Feature = Enum]
  -- [Feature = IDesc]
pattern DIVARN    = DZE
pattern DICONSTN = DSU DZE
pattern DIPIN     = DSU (DSU DZE)
pattern DIFPIN    = DSU (DSU (DSU DZE))
pattern DISIGMAN  = DSU (DSU (DSU (DSU DZE)))
pattern DIFSIGMAN = DSU (DSU (DSU (DSU (DSU DZE))))
pattern DIPRODN  = DSU (DSU (DSU (DSU (DSU (DSU DZE))))))

```

```

pattern DIMU  $l\ ii\ x\ i$  = DIMu ( $l\ :? =: (Id\ ii\ : \&\ Id\ x)$ )  $i$ 
pattern DIVAR  $i$  = DCON (DPAIR DIVARN (DPAIR  $i$  DVOID))
pattern DIPI  $s\ t$  = DCON (DPAIR DIPIN (DPAIR  $s$  (DPAIR  $t$  DVOID)))
pattern DIFPI  $s\ t$  = DCON (DPAIR DIFPIN (DPAIR  $s$  (DPAIR  $t$  DVOID)))
pattern DISIGMA  $s\ t$  = DCON (DPAIR DISIGMAN (DPAIR  $s$  (DPAIR  $t$  DVOID)))
pattern DIFSIGMA  $s\ t$  = DCON (DPAIR DIFSIGMAN (DPAIR  $s$  (DPAIR  $t$  DVOID)))
pattern DICONST  $p$  = DCON (DPAIR DICONSTN (DPAIR  $p$  DVOID))
pattern DIPROD  $u\ x\ y$  = DCON (DPAIR DIPRODN (DPAIR  $u$  (DPAIR  $x$  (DPAIR  $y$  DVOID))))
  -- [/Feature = IDesc]
  -- [Feature = Labelled]
pattern DLABEL  $l\ t$  = DC (Label  $l\ t$ )
pattern DLRET  $t$  = DC (LRet  $t$ )
  -- [/Feature = Labelled]
  -- [Feature = Prop]
pattern DPRP = DC Prop
pattern DPRF  $p$  = DC (Prf  $p$ )
pattern DALL  $p\ q$  = DC (All  $p\ q$ )
pattern DIMP  $p\ q$  = DALL (DPRF  $p$ ) (DL (DK  $q$ ))
pattern DALLV  $x\ s\ p$  = DALL  $s$  (DLAV  $x\ p$ )
pattern DAND  $p\ q$  = DC (And  $p\ q$ )
pattern DTRIVIAL = DC Trivial
pattern DABSURD = DC Absurd
pattern DBOX  $p$  = DC (Box  $p$ )
pattern DINH  $ty$  = DC (Inh  $ty$ )
pattern DWIT  $t$  = DC (Wit  $t$ )
  -- [/Feature = Prop]
  -- [Feature = Sigma]
pattern DSIGMA  $p\ q$  = DC (Sigma  $p\ q$ )
pattern DPAIR  $p\ q$  = DC (Pair  $p\ q$ )
pattern DUNIT = DC Unit
pattern DVOID = DC Void
pattern DTimes  $x\ y$  = Sigma  $x$  (DL (DK  $y$ ))
pattern DTIMES  $x\ y$  = DC (DTimes  $x\ y$ )
  -- [/Feature = Sigma]
  -- [Feature = UId]
pattern DUID = DC UId
pattern DTAG  $s$  = DTag  $s$  []
  -- [/Feature = UId]

```

3.1.3 Sizes

We keep track of the Size of terms when parsing, to avoid nasty left recursion, and when pretty-printing, to minimise the number of brackets we output. In increasing order, the sizes are:

```

data Size = ArgSize | AppSize | EqSize | AndSize | ArrSize | PiSize
deriving (Show, Eq, Enum, Bounded, Ord)

```

When a higher-size term has to be put in a lower-size position, it must be wrapped in parentheses. For example, an application has AppSize but its arguments have ArgSize, so $g\ (f\ x)$ cannot be written $g\ f\ x$, whereas EqSize is bigger than AppSize so $f\ x \equiv g\ x$ means the same thing as $(f\ x) \equiv (g\ x)$.

3.2 Relative Names

For display and storage purposes, we have a system of local longnames for referring to entries. Any component of a local name may have a \hat{n} or $_n$ suffix, where n is an integer, representing a relative or absolute offset. A relative offset \hat{n} refers to the n^{th} occurrence of the name encountered when searching upwards, so $x^{\hat{0}}$ refers to the same reference as x , but $x^{\hat{1}}$ skips past it and refers to the next thing named x . An absolute offset $_n$, by contrast, refers to the exact numerical component of the name.

```
data Offs = Rel Int | Abs Int deriving (Show, Eq)
type RelName = [(String, Offs)]
```

As a consequence, there is whole new family of objects: terms which variables are relative names. So it goes:

```
type InTmRN = InTm RelName
type ExTmRN = ExTm RelName
type DInTmRN = DInTm REF RelName
type DExTmRN = DExTm REF RelName
type DSPINE = DSpine REF RelName
type DHEAD = DHead REF RelName
type DSCOPE = DScope REF RelName
```

3.2.1 Names to strings

The *showRelName* function converts a relative name to a string by inserting the appropriate punctuation.

```
showRelName :: RelName → String
showRelName = intercalate " ." · map showOffName
where showOffName (x, Rel 0) = x
       showOffName (x, Rel i) = x ++ "^" ++ show i
       showOffName (x, Abs i) = x ++ "_" ++ show i
```

The *showName* function converts an absolute name to a string absolutely.

```
showName :: Name → String
showName = showRelName · map (λ(x, i) → (x, Abs i))
```

3.3 Schemes

3.3.1 Schemes for implicit arguments

A definition may have a Scheme, which allows us to handle implicit syntax. A Scheme is defined by:

```
SCHEME ::= Ty                a real, tangible type
         | (x : SCHEME) → SCHEME  an explicit Π
         | {x : Ty} → SCHEME     an implicit Π
```

Crucially, an implicit Π hides a real type, not another scheme: we forbid “higher-schemes” for mental sanity reasons. For the sake of generality, we will parameterise over the exact representation of types:

```
data Scheme x = SchType x
             | SchExplicitPi (String :<: Scheme x) (Scheme x)
             | SchImplicitPi (String :<: x) (Scheme x)
deriving Show
```

3.3.2 Extracting names

Given a scheme, we can extract the names of its IIs:

```
schemeNames :: Scheme x → [String]
schemeNames (SchType _) = []
schemeNames (SchExplicitPi (x :<: _) sch) = x : schemeNames sch
schemeNames (SchImplicitPi (x :<: _) sch) = x : schemeNames sch
```

3.3.3 Turning schemes to terms

We can also convert a Scheme x to an x , if we are given a way to interpret II-bindings:

```
schemeToType :: (String → x → x → x) → Scheme x → x
schemeToType _ (SchType ty) = ty
schemeToType piv (SchExplicitPi (x :<: s) t) =
  piv x (schemeToType piv s) (schemeToType piv t)
schemeToType piv (SchImplicitPi (x :<: s) t) =
  piv x s (schemeToType piv t)
```

With two direct special cases:

```
schemeToInTm :: Scheme (InTm x) → InTm x
schemeToInTm = schemeToType PIV
schemeToDInTm :: Scheme (DInTm p x) → DInTm p x
schemeToDInTm = schemeToType DPIV
```

3.3.4 Unlifting schemes

Schemes are stored fully II-lifted with SchExplicitPis, so we may need to apply them to a spine of shared parameters:

```
applyScheme :: Scheme INTM → [REF] → Scheme INTM
applyScheme sch [] = sch
applyScheme (SchExplicitPi (x :<: SchType s) schT) (r : rs) =
  applyScheme (underScheme 0 r schT) rs
where
  underScheme :: Int → REF → Scheme INTM → Scheme INTM
  underScheme n r (SchType ty) = SchType (under n r %% ty)
  underScheme n r (SchExplicitPi (x :<: schS) schT) =
    SchExplicitPi (x :<: underScheme n r schS) (underScheme (n + 1) r schT)
  underScheme n r (SchImplicitPi (x :<: s) schT) =
    SchImplicitPi (x :<: under n r %% s) (underScheme (n + 1) r schT)
```

3.3.5 Schemes in error messages

We can cheaply embed schemes in error messages by converting them to types and evaluating. Really, we ought to add schemes as a kind of ErrorTok.

```
errScheme :: Scheme INTM → ErrorItem t
errScheme sch = errTyVal (evTm (schemeToInTm sch) :<: SET)
```

3.4 Lexer

The lexical structure is extremely simple. The reason is that Cochon being an interactive theorem prover, its inputs will be straightforward, 1-dimension terms. Being interactive, our user is also more interested in knowing where she did a mistake, rather than having the ability to write terms in 3D.

We want to recognize “valid” identifiers, and keywords, with all of this structures by brackets. Interestingly, we only consider correctly paired brackets: we never use left-over brackets, and it is much simpler to work with well-parenthesized expressions when parsing terms. Brackets are round, square, or curly, and you can make fancy brackets by wedging an identifier between open-and-bar, or bar-and-close without whitespace. Sequences of non-whitespace are identifiers unless they’re keywords.

3.4.1 What are tokens?

We lex into tokens, classified as follows.

```
data Token
= Identifier String      -- identifiers
| Keyword Keyword      -- keywords
| Brackets Bracket [Token] -- bracketted tokens
deriving (Eq, Show)
```

Brackets are the structuring tokens. We have:

```
data Bracket
= Round | RoundB String -- ( or (foo |
| Square | SquareB String -- [ or [foo |
| Curly | CurlyB String -- { or {foo |
deriving (Eq, Show)
```

As we are very likely to look at our tokens all too often, let us implement a function to crush tokens down to strings.

```
crushToken :: Token → String
crushToken (Identifier s) = s
crushToken (Keyword s) = key s
crushToken (Brackets bra toks) = showOpenB bra ++
  (intercalate " " (map crushToken toks)) ++ showCloseB bra
where
  showOpenB Round      = "( "
  showOpenB Square     = "[ "
  showOpenB Curly      = "{ "
  showOpenB (RoundB s) = "( " ++ s ++ " | "
  showOpenB (SquareB s) = "[ " ++ s ++ " | "
  showOpenB (CurlyB s) = "{ " ++ s ++ " | "
  showCloseB Round     = ") "
  showCloseB Square    = "]" "
  showCloseB Curly     = "}" "
  showCloseB (RoundB s) = "| " ++ s ++ " )" "
  showCloseB (SquareB s) = "| " ++ s ++ " ]" "
  showCloseB (CurlyB s) = "| " ++ s ++ " }" "
```

3.4.2 Lexer

We implement the tokenizer as a Parsley on Chars. That's a cheap solution for what we have to do. The previous implementation was running over the string of characters, wrapped in a StateT monad transformer.

question: What was the benefit of a StateT lexer, as we have a parser combinator library at hand?

A token is either a bracketted expression, a keyword, or, failing all that, an identifier. Hence, we can recognize a token with the following parser:

```
parseToken :: Parsley Char Token
parseToken = (| id parseBrackets
              | id parseKeyword
              | id parseIdent
              |)
```

Tokenizing an input string then simply consists in matching a bunch of token separated by spaces. For readability, we are also glutton in spaces the user may have put before and after the tokens.

```
tokenize :: Parsley Char [Token]
tokenize = spaces * > pSep spaces parseToken < * spaces
```

In the following, we implement these combinators in turn: *spaces*, *parseIdent*, *parseKeyword*, and *parseBrackets*.

Lexing spaces

A space is one of the following character:

```
space :: String
space = " \n\r\t"
```

So, we look for *many* of them:

```
spaces :: Parsley Char ()
spaces = (many $ tokenFilter (flip elem space)) * > pure ()
```

Parsing words

As an intermediary step before keyword, identifier, and brackets, let us introduce a parser for words. A word is any non-empty string of characters that doesn't include a space, a bracketting symbol, or one of the protected symbols. A protected symbol is, simply, a one-character symbol which can be prefix or suffix a word, but will not be merged into the parsed word. For example, "foo," lexes into first Identifier *foo* then Keyword,. In Parsley, this translates to:

```
parseWord :: Parsley Char String
parseWord = (| id (some $ tokenFilter (\t → ¬ $ elem t $ space ++ bracketChars ++ protected))
              | (:[]) (tokenFilter (flip elem protected)) |)
  where protected = " , ' ; "
```

As we are at it, we can test for word equality, that is build a parser matching a given word:

```
wordEq :: String → Parsley Char ()
wordEq " " = pure ()
wordEq word = pFilter filter parseWord
  where filter s | s ≡ word = Just ()
        | otherwise = Nothing
```

Equipped with *parseWord* and *wordEq*, the following lexers win a level of abstraction, working on words instead of characters.

Lexing keywords

Keywords are slightly more involved. A keyword is one of the following things...

data Keyword **where**

```
-- import {- KeywordConstructors
-- [Feature = Enum]
KwEnum :: Keyword
KwPlus  :: Keyword
  -- [/Feature = Enum]
  -- [Feature = Equality]
KwEqBlue :: Keyword
  -- [/Feature = Equality]
  -- [Feature = IDesc]
KwIMu :: Keyword
  -- [/Feature = IDesc]
  -- [Feature = Labelled]
KwCall   :: Keyword
KwLabel  :: Keyword
KwLabelEnd :: Keyword
KwRet    :: Keyword
  -- [/Feature = Labelled]
  -- [Feature = Problem]
KwProb   :: Keyword
KwProbLabel :: Keyword
KwPatPi  :: Keyword
KwSch    :: Keyword
KwSchTy  :: Keyword
KwExpPi  :: Keyword
KwImpPi  :: Keyword
  -- [/Feature = Problem]
  -- [Feature = Prop]
KwProp  :: Keyword
KwAbsurd :: Keyword
KwTrivial :: Keyword
KwPrf   :: Keyword
KwAnd   :: Keyword
KwArr   :: Keyword
KwImp   :: Keyword
KwAll   :: Keyword
KwInh   :: Keyword
KwWit   :: Keyword
  -- [/Feature = Prop]
  -- [Feature = Sigma]
KwFst  :: Keyword
KwSnd  :: Keyword
KwSig  :: Keyword
  -- [/Feature = Sigma]
  -- [Feature = UId]
KwUId  :: Keyword
KwTag  :: Keyword
  -- [/Feature = UId]
```

KwAsc :: Keyword
KwComma :: Keyword
KwSemi :: Keyword
KwDefn :: Keyword
KwUnderscore :: Keyword
KwEq :: Keyword
KwBy :: Keyword

KwSet :: Keyword
KwPi :: Keyword
KwLambda :: Keyword

KwCon :: Keyword
KwOut :: Keyword

deriving (Bounded, Enum, Eq, Show)

...and they look like this:

key :: Keyword → String

```

-- import j- KeywordTable
-- [Feature = Enum]
key KwEnum    = "Enum"
key KwPlus    = "+"
-- [/Feature = Enum]
-- [Feature = Equality]
key KwEqBlue = "=="
-- [/Feature = Equality]
-- [Feature = IDesc]
key KwIMu    = "IMu"
-- [/Feature = IDesc]
-- [/Feature = Labelled]
key KwCall   = "call"
key KwLabel  = "<"
key KwLabelEnd = ">"
key KwRet    = "return" -- rename me
-- [/Feature = Labelled]
-- [Feature = Problem]
key KwProb   = "Prob"
key KwProbLabel = "ProbLabel"
key KwPatPi  = "PatPi"
key KwSch    = "Sch"
key KwSchTy  = "SchTy"
key KwExpPi  = "ExpPi"
key KwImpPi  = "ImpPi"
-- [/Feature = Problem]
-- [Feature = Prop]
key KwProp   = "Prop"
key KwAbsurd = "FF"
key KwTrivial = "TT"
key KwPrf    = ":-"
key KwAnd    = "&&"
key KwArr    = "->"
key KwImp    = "=>"
key KwAll    = "All"
key KwInh    = "Inh"
key KwWit    = "wit"
-- [/Feature = Prop]
-- [Feature = Sigma]
key KwFst    = "!"
key KwSnd    = "-"
key KwSig    = "Sig"
-- [/Feature = Sigma]
-- [Feature = UId]
key KwUId    = "UId"
key KwTag    = "'"
-- [/Feature = UId]

```

```

key KwAsc      = ":"
key KwComma    = ","
key KwSemi     = ";"
key KwDefn     = ":@"
key KwUnderscore = "_"
key KwEq       = "="
key KwBy       = "<="

```

```

key KwSet      = "Set"
key KwPi       = "π"
key KwLambda   = "\\\"

```

```

key KwCon      = "con"
key KwOut      = "⌘"

```

```

key k = error ("key: missing keyword " ++ show k)

```

It is straightforward to make a translation table, *keywords*:

```

keywords :: [(String, Keyword)]
keywords = map (\k → (key k, k)) (enumFromTo minBound maxBound)

```

To implement *parseKeyword*, we can simply filter by words that can be found in the *keywords* list.

```

parseKeyword :: Parsley Char Token
parseKeyword = pFilter (\t → fmap (Keyword · snd) $ find ((t ≡) · fst) keywords) parseWord

```

Lexing identifiers

Hence, parsing an identifier simply consists in successfully parsing a word – which is not a keyword – and saying “oh! it’s an Identifier”.

```

parseIdent = (| id (%parseKeyword%) ()
              | Identifier parseWord |)

```

Lexing brackets

Brackets, open and closed, are one of the following.

```

openBracket, closeBracket, bracketChars :: String
openBracket = " [ { "
closeBracket = " } ] "
bracketChars = "| " ++ openBracket ++ closeBracket

```

Parsing brackets, as you would expect, requires a monad: we’re not context-free my friend. This is slight variation around the *pLoop* combinator.

First, we use *parseOpenBracket* to match an opening bracket, and get it’s code. Thanks to this code, we can already say that we hold a Brackets. We are left with tokenizing the content of the bracket, up to parsing the corresponding closing bracket.

Parsing the closing bracket is made slightly more complex by the presence of fancy brackets: we have to match the fancy name of the opening bracket with the one of the closing bracket.

```

parseBrackets :: Parsley Char Token
parseBrackets = do
  bra ← parseOpenBracket
  (| (Brackets bra)
    (| id tokenize (%parseCloseBracket bra%) |) |)
  where parseOpenBracket :: Parsley Char Bracket
    parseOpenBracket = (| id (%tokenEq ' (' %%)
      (| RoundB possibleWord (%tokenEq ' | ' %%)
        | Round (%spaces%) |)
      | id (%tokenEq ' [ ' %%)
        (| SquareB possibleWord (%tokenEq ' | ' %%)
          | Square (%spaces%) |)
      | id (%tokenEq ' { ' %%)
        (| CurlyB possibleWord (%tokenEq ' | ' %%)
          | Curly (%spaces%) |)
      |)
    parseCloseBracket :: Bracket → Parsley Char ()
    parseCloseBracket Round = tokenEq ' ) '
    parseCloseBracket Square = tokenEq ' ] '
    parseCloseBracket Curly = tokenEq ' } '
    parseCloseBracket (RoundB s) = matchBracketB s ' ) '
    parseCloseBracket (SquareB s) = matchBracketB s ' ] '
    parseCloseBracket (CurlyB s) = matchBracketB s ' } '
    parseBracket x = tokenFilter (flip elem x)
    matchBracketB s bra = (| id~() (%tokenEq ' | ' %%)
      (%wordEq s%)
      (%tokenEq bra%) |)
    possibleWord = parseWord ⊕ pure " "

```

3.4.3 Abstracting tokens

As we are very likely to use these tokens in a parser, let us readily define parser combinators for them. Hence, looking for a given keyword is not more difficult than that:

```

keyword :: Keyword → Parsley Token ()
keyword s = tokenEq (Keyword s)

```

And we can match any keyword (though we rarely want to) using:

```

anyKeyword :: Parsley Token Keyword
anyKeyword = pFilter filterKeyword nextToken
  where filterKeyword (Keyword k) = Just k
        filterKeyword _ = Nothing

```

Parsing an identifier or a number is as simple as:

```

ident :: Parsley Token String
ident = pFilter filterIdent nextToken
  where filterIdent (Identifier s) | ¬ (isDigit $ head s) = Just s
        filterIdent _ = Nothing
digits :: Parsley Token String
digits = pFilter filterInt nextToken
  where filterInt (Identifier s) | all isDigit s = Just s
        filterInt _ = Nothing

```

Occasionally we may want to match a specific identifier:

```
identEq :: String → Parsley Token ()
identEq s = ident >>= pGuard · (≡ s)
```

Finally, we can match a bracketted expression and use a specific parser for the bracketted tokens:

```
bracket :: Bracket → Parsley Token x → Parsley Token x
bracket bra p = pFilter filterBra nextToken
  where filterBra (Brackets bra' toks) | bra ≡ bra' =
    either (\_ → Nothing) Just $ parse p toks
    filterBra _ = Nothing
```

3.5 Parsing Terms

The term parser eats structured Tokens as defined in `Lexer.lhs`. It uses the monadic parser combinators to translate the grammar of terms defined in Section 1.2.

3.5.1 Names

A relative name is a list of idents separated by dots, and possibly with `↑` or `_` symbols (for relative or absolute offsets).

```
nameParse :: Parsley Token RelName
nameParse = do
  s ← ident
  case parse pName s of
    Right rn → return rn
    Left e   → fail "nameParse failed"
```

```
pName :: Parsley Char RelName
pName = (| pNamePart : (many (tokenEq ' . ' * > pNamePart)) |)
```

```
pNamePart :: Parsley Char (String, Offs)
pNamePart = (| (,) pNameWord (%tokenEq '^' %) (| Rel (| read pNameOffset |) |)
  | (,) pNameWord (%tokenEq '-' %) (| Abs (| read pNameOffset |) |)
  | (,) pNameWord~(Rel 0)
  |)
```

```
pNameWord :: Parsley Char String
pNameWord = some (tokenFilter (λc → ¬ (c ∈ "_^.")))
```

```
pNameOffset :: Parsley Char String
pNameOffset = some (tokenFilter isDigit)
```

3.5.2 Overall parser structure

The `pDEXTm` and `pDINTm` functions start parsing at the maximum size.

```
pDEXTm :: Parsley Token DEXTmRN
pDEXTm = sizedDEXTm maxBound
```

```

pDInTm :: Parsley Token DInTmRN
pDInTm = sizedDInTm maxBound

```

We do not allow ascriptions in the term syntax, but they are useful in commands, so we provide *pAscription* to parse an ascription into separate components, and *pAscriptionTC* to parse an ascription as an appropriate type annotation.

```

pAscription :: Parsley Token (DInTmRN <: DInTmRN)
pAscription = (| pDInTm (%keyword KwAsc%) <: pDInTm |)

```

```

pAscriptionTC :: Parsley Token DExTmRN
pAscriptionTC = (| typeAnnot pDInTm (%keyword KwAsc%) pDInTm |)
  where typeAnnot tm ty = DType ty :$[A tm]

```

Each *sized* parser tries the appropriate *special* parser for the size, then falls back to parsing at the previous size followed by a *more* parser. At the smallest size, brackets must be used to start parsing from the largest size again. Concrete syntax is matched using the lists of parsers defined in the following subsection.

```

sizedDExTm :: Size → Parsley Token DExTmRN
sizedDExTm z = (| (:$[]) (specialHead z) |) ⊕
  (if z > minBound then pLoop (sizedDExTm (pred z)) (moreDExTm z)
   else bracket Round pDExTm)

```

```

sizedDInTm :: Size → Parsley Token DInTmRN
sizedDInTm z = specialDInTm z ⊕ (| (DN · (:$[])) (specialHead z) |) ⊕
  (if z > minBound then pLoop (sizedDInTm (pred z)) (moreInEx z)
   else bracket Round pDInTm)

```

```

specialHead :: Size → Parsley Token DHEAD
specialHead = sizeListParse headParsers

```

```

specialDInTm :: Size → Parsley Token DInTmRN
specialDInTm = sizeListParse inDTmParsersSpecial

```

```

moreInEx :: Size → DInTmRN → Parsley Token DInTmRN
moreInEx z (DN e) = (| DN (moreDExTm z e) |) ⊕ moreDInTm z (DN e)
moreInEx z t      = moreDInTm z t

```

```

moreDExTm :: Size → DExTmRN → Parsley Token DExTmRN
moreDExTm s e = (| (e$ :: $) (sizeListParse elimParsers s) |)

```

```

moreDInTm :: Size → DInTmRN → Parsley Token DInTmRN
moreDInTm = paramListParse inDTmParsersMore

```

3.5.3 Lists of sized parsers

A *SizedParserList* is a list of parsers associated to every size; a *ParamParserList* allows the parser to depend on a parameter.

```

type SizedParserList a    = [(Size, [Parsley Token a])]
type ParamParserList a b = [(Size, [a → Parsley Token b])]

```

We can construct such a list from a list of size-parser pairs thus:

```

arrange :: [(Size, b)] → [(Size, [b])]
arrange xs = map (λs → (s, pick s xs)) (enumFromTo minBound maxBound)
where
  pick :: Eq a ⇒ a → [(a, b)] → [b]
  pick x = concatMap (λ(a, b) → if a ≡ x then [b] else [])

```

To parse using a list at a particular size, we try all the parsers at that size in order:

```

sizeListParse :: SizedParserList a → Size → Parsley Token a
sizeListParse sps s = pTryList · unJust $ lookup s sps

```

```

paramListParse :: ParamParserList a b → Size → a → Parsley Token b
paramListParse sfs s a = pTryList · map ($a) · unJust $ lookup s sfs

```

```

unJust :: Maybe a → a
unJust (Just x) = x

```

```

pTryList :: [Parsley Token a] → Parsley Token a
pTryList (p : ps) = p ⊕ pTryList ps
pTryList []         = ()

```

Now we define the lists of parsers that actually match bits of the concrete syntax. Note that each list has a corresponding aspect so features can extend the parser.

```

headParsers :: SizedParserList DHEAD
headParsers = arrange $
  (ArgSize, (| DType (bracket Round (keyword KwAsc * > pDInTm)) |)) :
  (ArgSize, (| DP nameParse |)) :
  []

```

```

elimParsers :: SizedParserList (Elim DInTmRN)
elimParsers = arrange $
  -- import j- ElimParsers
  -- [Feature = Labelled]
  (AppSize, (| Call (%keyword KwCall%)~DU |)) :
  -- [/Feature = Labelled]
  -- [Feature = Sigma]
  (AppSize, (| Fst (%keyword KwFst%) |)) :
  (AppSize, (| Snd (%keyword KwSnd%) |)) :
  -- [/Feature = Sigma]
  (AppSize, (| Out (%keyword KwOut%) |)) :
  (AppSize, (| A (sizedDInTm ArgSize) |)) :
  []

```

```

inDTmParsersSpecial :: SizedParserList DInTmRN
inDTmParsersSpecial = arrange $
  -- import j- DInTmParsersSpecial
  -- [Feature = Enum]
  (ArgSize, (| mkNum (| read digits |) (optional $ (keyword KwPlus) * > sizedDInTm ArgSize) |)) :
  (AndSize, (| DENUMT (%keyword KwEnum%) (sizedDInTm ArgSize) |)) :
  -- [/Feature = Enum]
  -- [Feature = IDesc]
  (AndSize, (| (DIMU Nothing) (%keyword KwIMu%) (sizedDInTm ArgSize) (sizedDInTm ArgSize) (sizedDInTm ArgSize) |)) :
  -- [/Feature = IDesc]
  -- [Feature = Labelled]
  (ArgSize, (| DLABEL (%keyword KwLabel%) (sizedDInTm AppSize) (%keyword KwAsc%) (sizedDInTm ArgSize) |)) :
  (ArgSize, (| DLRET (%keyword KwRet%) (sizedDInTm ArgSize) |)) :
  -- [/Feature = Labelled]
  -- [Feature = Prop]
  (ArgSize, (| DPROP (%keyword KwProp%) |)) :
  (ArgSize, (| DABSURD (%keyword KwAbsurd%) |)) :
  (ArgSize, (| DTRIVIAL (%keyword KwTrivial%) |)) :
  (AndSize, (| DPRF (%keyword KwPrf%) (sizedDInTm AndSize) |)) :
  (AndSize, (| DINH (%keyword KwInh%) (sizedDInTm ArgSize) |)) :
  (AndSize, (| DWIT (%keyword KwWit%) (sizedDInTm ArgSize) |)) :
  (AndSize, (| DALL (%keyword KwAll%) (sizedDInTm ArgSize) (sizedDInTm ArgSize) |)) :
  -- [/Feature = Prop]
  -- [Feature = Sigma]
  (ArgSize, (| id (bracket Square tuple) |)) :
  (ArgSize, (| id (%keyword KwSig%) (bracket Round sigma) |)) :
  (ArgSize, (| DSIGMA (%keyword KwSig%) (sizedDInTm ArgSize) (sizedDInTm ArgSize) |)) :
  -- [/Feature = Sigma]
  -- [Feature = UID]
  (ArgSize, (| DUID (%keyword KwUId%) |)) :
  (ArgSize, (| DTAG (%keyword KwTag%) ident |)) :
  (AppSize, (| DTag (%keyword KwTag%) ident (many (sizedDInTm ArgSize)) |)) :
  -- [/Feature = UID]

  (ArgSize, (| DSET (%keyword KwSet%) |)) :
  (ArgSize, (| DQ (pFilter questionFilter ident) |)) :
  (ArgSize, (| DU (%keyword KwUnderscore%) |)) :
  (ArgSize, (| DCON (%keyword KwCon%) (sizedDInTm ArgSize) |)) :
  (ArgSize, (| (iter mkDLAV) (%keyword KwLambda%) (some (ident ⊕ underscore)) (%keyword KwArr%) pDInTm |)) :
  (AndSize, (| DPI (%keyword KwPi%) (sizedDInTm ArgSize) (sizedDInTm ArgSize) |)) :
  (PiSize, (| (flip iter)
    (some (bracket Round
      (| (ident ⊕ underscore), (%keyword KwAsc%) pDInTm |)))
    (| (uncurry mkDPiV) (%keyword KwArr%)
      | (uncurry mkDALLV) (%keyword KwImp%) |)
    pDInTm |)) :
  []

```

```

inDTmParsersMore :: ParamParserList DInTmRN DInTmRN
inDTmParsersMore = arrange $
  -- import j- DInTmParsersMore
  -- [Feature = Equality]
  (EqSize, λt → (| DEqBlue (pFilter isEx (pure t)) (%keyword KwEqBlue%)
                      (pFilter isEx (sizedDInTm (pred EqSize))) |)) :
  -- [/Feature = Equality]
  -- [Feature = Prop]
  (AndSize, λs → (| (DAND s) (%keyword KwAnd%) (sizedDInTm AndSize) |)) :
  (ArrSize, λs → (| (DIMP s) (%keyword KwImp%) (sizedDInTm PiSize) |)) :
  -- [/Feature = Prop]

  (ArrSize, λs → (| (DARR s) (%keyword KwArr%) (sizedDInTm PiSize) |)) :
  []

```

3.5.4 Parser support code

```

-- import j- ParserCode
-- [Feature = Enum]
mkNum :: Int → Maybe DInTmRN → DInTmRN
mkNum 0 Nothing = DZE
mkNum 0 (Just t) = t
mkNum n t = DSU (mkNum (n - 1) t)
-- [/Feature = Enum]

-- [Feature = Equality]
isEx :: DInTmRN → Maybe DExTmRN
isEx (DN tm) = Just tm
isEx _       = Nothing
-- [/Feature = Equality]

-- [Feature = Sigma]
tuple :: Parsley Token DInTmRN
tuple =
  (| DPAIR (sizedDInTm ArgSize) (| id (%keyword KwComma%) pDInTm
  | id tuple |)
  | DVOID (%pEndOfStream%)
  |)

sigma :: Parsley Token DInTmRN
sigma = (| mkSigma (optional (ident < * keyword KwAsc)) pDInTm sigmaMore
  | DUNIT (%pEndOfStream%)
  |)

sigmaMore :: Parsley Token DInTmRN
sigmaMore = (| id (%keyword KwSemi%) (sigma ⊕ pDInTm)
  | (λp s → mkSigma Nothing (DPRF p) s) (%keyword KwPrf%) pDInTm sigmaMore
  | (λx → DPRF x) (%keyword KwPrf%) pDInTm
  |)

```

```

mkSigma :: Maybe String → DInTmRN → DInTmRN → DInTmRN
mkSigma Nothing s t = DSIGMA s (DL (DK t))
mkSigma (Just x) s t = DSIGMA s (DL (x :•t))
-- [/Feature = Sigma]

```

```

questionFilter :: String → Maybe String
questionFilter ('?' : s) = Just s
questionFilter _       = Nothing

```

```

underscore :: Parsley Token String
underscore = keyword KwUnderscore >> pure "_"

```

```

mkDLAV :: String → DInTmRN → DInTmRN
mkDLAV "_" t = DL (DK t)
mkDLAV x t = DLAV x t

```

```

mkDPIV :: String → DInTmRN → DInTmRN → DInTmRN
mkDPIV "_" s t = DPI s (DL (DK t))
mkDPIV x s t = DPIV x s t

```

```

mkDALLV :: String → DInTmRN → DInTmRN → DInTmRN
mkDALLV "_" s p = DALL s (DL (DK p))
mkDALLV x s p = DALLV x s p

```

3.5.5 Parsing schemes

```

pScheme :: Parsley Token (Scheme DInTmRN)
pScheme = (| mkScheme (many pSchemeBit) (%keyword KwAsc%) pDInTm |)
  where
    pSchemeBit :: Parsley Token (String, Either (Scheme DInTmRN) DInTmRN)
    pSchemeBit = bracket Round (| ident, (%keyword KwAsc%) (| (Left · SchType) pDInTm |) |)
      ⊕ bracket Curly (| ident, (%keyword KwAsc%) (| Right pDInTm |) |)
    mkScheme :: [(String, Either (Scheme DInTmRN) DInTmRN)] → DInTmRN → Scheme DInTmRN
    mkScheme [] ty = SchType ty
    mkScheme ((x, Left s) : bits) ty = SchExplicitPi (x <: s) (mkScheme bits ty)
    mkScheme ((x, Right s) : bits) ty = SchImplicitPi (x <: s) (mkScheme bits ty)

```

3.6 Pretty-printing

We use the HughesPJ pretty-printing combinators. This section defines how to pretty-print everything defined in the Core chapter, and provides the aspects to allow features to add their own pretty-printing support.

The *keyword* function gives the document representing a Keyword.

```

keyword :: Keyword → Doc
keyword = text · key

```

The *Pretty* class describes things that can be pretty-printed. The *pretty* function takes a value *x* and the *Size* at which it should be printed, and should return a document representation of *x*.

```
class Show  $x \Rightarrow$  Pretty  $x$  where  
  pretty ::  $x \rightarrow$  Size  $\rightarrow$  Doc
```

The *wrapDoc* operator takes a document, its size and the size it should be printed at. If the document's size is larger than the current size, it is wrapped in parentheses.

```
wrapDoc :: Doc  $\rightarrow$  Size  $\rightarrow$  Size  $\rightarrow$  Doc  
wrapDoc d dSize curSize  
  | dSize > curSize = parens d  
  | otherwise       = d
```

When defining instances of Pretty, we will typically pattern-match on the first argument and construct a function that takes the current size by partially applying *wrapDoc* to a document and its size.

The Can functor is fairly easy to pretty-print, the only complexity being with Π -types.

```

instance Pretty (Can DInTmRN) where
  pretty Set      = const (kword KwSet)
  pretty (Pi s t) = prettyPi empty (DPI s t)
  pretty (Con x)  = wrapDoc (kword KwCon < + > pretty x ArgSize) AppSize
    -- import j- CanPretty
    -- [Feature = Anchor]
  pretty (Anchor (DTAG u) t ts) = wrapDoc (text u < + > pretty ts ArgSize) ArgSize
  pretty AllowedEpsilon = const empty
  pretty (AllowedCons _ _ _ s ts) = wrapDoc (pretty s ArgSize < + > pretty ts ArgSize) ArgSize
    {-Not yet implemented -}
    -- [/Feature = Anchor]
    -- [Feature = Enum]
  pretty (EnumT t) = wrapDoc (kword KwEnum < + > pretty t ArgSize) AppSize
  pretty Ze        = const (int 0)
  pretty (Su t)    = prettyEnumIndex 1 t
    -- [/Feature = Enum]
    -- [Feature = Equality]
  pretty (EqBlue pp qq) = pretty (DEqBlue (foo pp) (foo qq))
    where
      foo :: (DInTmRN => DInTmRN) -> DExTmRN
      foo (_ => DN x) = x
      foo (xty => x _) = DType xty :$[A x]
    -- [/Feature = Equality]
    -- [Feature = IDesc]
  pretty (IMu (Just l :? =: _) i) = wrapDoc
    (pretty l AppSize < + > pretty i ArgSize)
    AppSize
  pretty (IMu (Nothing :? =: (ld ii : & ld d)) i) = wrapDoc
    (kword KwIMu < + > pretty ii ArgSize < + > pretty d ArgSize < + > pretty i ArgSize)
    AppSize
    -- [/Feature = IDesc]
    -- [Feature = Labelled]
  pretty (Label l t) = const (kword KwLabel < + >
    pretty l maxBound < + > kword KwAsc < + > pretty t maxBound
    < + > kword KwLabelEnd)
  pretty (LRet x) = wrapDoc (kword KwRet < + > pretty x ArgSize) ArgSize
    -- [/Feature = Labelled]
    -- [Feature = Prop]
  pretty Prop      = const (kword KwProp)
  pretty (Prf p)   = wrapDoc (kword KwPrf < + > pretty p AndSize) AppSize
  pretty (All p q) = prettyAll empty (DALL p q)
  pretty (And p q) = wrapDoc
    (pretty p (pred AndSize) < + > kword KwAnd < + > pretty q AndSize)
    AndSize
  pretty Trivial   = const (kword KwTrivial)
  pretty Absurd    = const (kword KwAbsurd)
  pretty (Box (Irr p)) = pretty p
  pretty (Inh ty)  = wrapDoc (kword KwInh < + > pretty ty ArgSize) AppSize
  pretty (Wit t)   = wrapDoc (kword KwWit < + > pretty t ArgSize) AppSize
    -- [/Feature = Prop]
    -- [Feature = Sigma]
  pretty Unit      = wrapDoc (kword KwSig < + > parens empty) AppSize
  pretty Void      = prettyPair DVOID
  pretty (Sigma s t) = prettySigma empty (DSIGMA s t)
  pretty (Pair a b) = prettyPair (DPAIR a b)
    -- [/Feature = Sigma]
    -- [Feature = UId]
  pretty UId       = const (kword KwUId)
  pretty (Tag s)   = const (kword KwTag <> text s)
    -- [/Feature = UId]

```

The *prettyPi* function takes a document representing the domains so far, a term and the current size. It accumulates domains until a non(dependent) Π -type is found, then calls *prettyPiMore* to produce the final document.

```
prettyPi :: Doc → DInTmRN → Size → Doc
prettyPi bs (DPI s (DL (DK t))) = prettyPiMore bs
  (pretty s (pred PiSize) < + > kword KwArr < + > pretty t PiSize)
prettyPi bs (DPI s (DL (x :•t))) =
  prettyPi (bs <> parens (text x < + > kword KwAsc < + > pretty s maxBound)) t
prettyPi bs (DPI s t) = prettyPiMore bs
  (kword KwPi < + > pretty s minBound < + > pretty t minBound)
prettyPi bs tm = prettyPiMore bs (pretty tm PiSize)
```

The *prettyPiMore* function takes a bunch of domains (which may be empty) and a codomain, and represents them appropriately for the current size.

```
prettyPiMore :: Doc → Doc → Size → Doc
prettyPiMore bs d
  | isEmpty bs = wrapDoc d PiSize
  | otherwise = wrapDoc (bs < + > kword KwArr < + > d) PiSize
```

The *Elim* functor is straightforward.

```
instance Pretty (Elim DInTmRN) where
  pretty (A t) = pretty t
  pretty Out = const (kword KwOut)
  -- import j- ElimPretty
  -- [Feature = Labelled]
  pretty (Call _) = const (kword KwCall)
  -- [/Feature = Labelled]
  -- [Feature = Sigma]
  pretty Fst = const (kword KwFst)
  pretty Snd = const (kword KwSnd)
  -- [/Feature = Sigma]
  pretty elim = const (quotes · text · show $ elim)
```

To pretty-print a scope, we accumulate arguments until something other than a λ -term is reached.

```
instance Pretty DSCOPE where
  pretty s = prettyLambda (B0 :< dScopeName s) (dScopeTm s)

prettyLambda :: Bwd String → DInTmRN → Size → Doc
prettyLambda vs (DL s) = prettyLambda (vs :< dScopeName s) (dScopeTm s)
prettyLambda vs tm = wrapDoc
  (kword KwLambda < + > text (intercalate " " (trail vs)) < + > kword KwArr
  < + > pretty tm ArrSize)
ArrSize
```

```

instance Pretty DInTmRN where
  pretty (DL s)           = pretty s
  pretty (DC c)           = pretty c
  pretty (DN n)           = pretty n
  pretty (DQ x)           = const (char '?' <> text x)
  pretty DU               = const (keyword KwUnderscore)
  -- import j- DInTmPretty
  -- [Feature = Anchor]
  pretty (DANCHOR s args) = wrapDoc (text s < + > pretty args ArgSize) ArgSize
  -- [/Feature = Anchor]
  -- [Feature = Equality]
  pretty (DEqBlue t u) = wrapDoc
    (pretty t ArgSize < + > keyword KwEqBlue < + > pretty u ArgSize)
    ArgSize
  -- [/Feature = Equality]
  -- [Feature = IDesc]
  pretty (DIMu (Just s  :? =: _) _) = pretty s
  pretty (DIMu (Nothing :? =: (Id ii : & Id d)) i) = wrapDoc
    (keyword KwIMu < + > pretty ii ArgSize < + > pretty d ArgSize < + > pretty i ArgSize)
    AppSize
  -- [/Feature = IDesc]
  -- [Feature = UId]
  pretty (DTAG s) = const (keyword KwTag <> text s)
  pretty (DTag s xs) = wrapDoc (keyword KwTag <> text s
    < + > hsep (map (flip pretty ArgSize) xs)) AppSize
  -- [/Feature = UId]
  pretty indtm = const (quotes · text · show $ indtm)

```

```

instance Pretty DExTmRN where
  pretty (n :$[]) = pretty n
  pretty (n :$els) = wrapDoc
    (pretty n AppSize < + > hsep (map (flip pretty ArgSize) els))
    AppSize

```

```

instance Pretty DHEAD where
  pretty (DP x) = const (text (showRelName x))
  pretty (DType ty) = const (parens (keyword KwAsc < + > pretty ty maxBound))
  pretty (DTE ex) = const (quotes · text · show $ ex)

```

```

instance Pretty (Scheme DInTmRN) where
  pretty (SchType ty) = wrapDoc (keyword KwAsc < + > pretty ty maxBound) ArrSize
  pretty (SchExplicitPi (x <: schS) schT) = wrapDoc (
    parens (text x < + > pretty schS maxBound)
    < + > pretty schT maxBound
  ) ArrSize
  pretty (SchImplicitPi (x <: s) schT) = wrapDoc (
    braces (text x < + > keyword KwAsc < + > pretty s maxBound)
    < + > pretty schT maxBound
  ) ArrSize

```

```

-- import j- Pretty
-- [Feature = Enum]
prettyEnumIndex :: Int → DInTmRN → Size → Doc
prettyEnumIndex n DZE      = const (int n)
prettyEnumIndex n (DSU t) = prettyEnumIndex (succ n) t
prettyEnumIndex n tm       = wrapDoc
  (int n < + > kword KwPlus < + > pretty tm ArgSize)
  AppSize
-- [/Feature = Enum]

-- [Feature = Prop]
prettyAll :: Doc → DInTmRN → Size → Doc
prettyAll bs (DALL (DPRF p) (DL (DK q))) = prettyAllMore bs
  (pretty p (pred PiSize) < + > kword KwImp < + > pretty q PiSize)
prettyAll bs (DALL s (DL (x :•t))) =
  prettyAll (bs <> parens (text x < + > kword KwAsc < + > pretty s maxBound)) t
prettyAll bs (DALL s (DL (DK t))) = prettyAll bs (DALL s (DL ("_" :•t)))
prettyAll bs (DALL s t) = prettyAllMore bs
  (kword KwAll < + > pretty s minBound < + > pretty t minBound)
prettyAll bs tm = prettyAllMore bs (pretty tm PiSize)
prettyAllMore :: Doc → Doc → Size → Doc
prettyAllMore bs d
  | isEmpty bs = wrapDoc d PiSize
  | otherwise  = wrapDoc (bs < + > kword KwImp < + > d) PiSize
-- [/Feature = Prop]

-- [Feature = Sigma]
prettyPair :: DInTmRN → Size → Doc
prettyPair p = const (brackets (prettyPairMore empty p))

prettyPairMore :: Doc → DInTmRN → Doc
prettyPairMore d DVOID      = d
prettyPairMore d (DPAIR a b) = prettyPairMore (d < + > pretty a minBound) b
prettyPairMore d t          = d < + > kword KwComma < + > pretty t maxBound

prettySigma :: Doc → DInTmRN → Size → Doc
prettySigma d DUNIT          = prettySigmaDone d empty
prettySigma d (DSIGMA s (DL (x :•t))) = prettySigma
  (d < + > text x < + > kword KwAsc < + > pretty s maxBound < + > kword KwSemi) t
prettySigma d (DSIGMA s (DL (DK t))) = prettySigma
  (d < + > pretty s maxBound < + > kword KwSemi) t
prettySigma d (DSIGMA s t) = prettySigmaDone d
  (kword KwSig < + > pretty s minBound < + > pretty t minBound)
prettySigma d t = prettySigmaDone d (pretty t maxBound)

prettySigmaDone :: Doc → Doc → Size → Doc
prettySigmaDone s t
  | isEmpty s = wrapDoc t AppSize
  | otherwise = wrapDoc (kword KwSig < + > parens (s < + > t)) AppSize
-- [/Feature = Sigma]

```

The *prettyBKind* function pretty-prints a ParamKind if supplied with a document representing its name and type.

```
prettyBKind :: ParamKind → Doc → Doc
prettyBKind ParamLam d = keyword KwLambda < + > d < + > keyword KwArr
prettyBKind ParamAll  d = keyword KwLambda < + > d < + > keyword KwImp
prettyBKind ParamPi   d = parens d < + > keyword KwArr
```

The *renderHouseStyle* hook allows us to customise the document rendering if necessary.

```
renderHouseStyle :: Doc → String
renderHouseStyle = render
```

Chapter 4

The Proof State

4.1 Developments

4.1.1 The Dev data-structure

A Development is a structure containing entries, some of which may have their own developments, creating a nested tree-like structure. Developments can be of different nature: this is indicated by the Tip. A development also keeps a NameSupply at hand, for namespace handling purposes. Initially we had the following definition:

```
type Dev = (Bwd Entry, Tip, NameSupply)
```

but generalised this to allow other Traversable functors f in place of Bwd, and to store a SuspendState, giving:

```
data Dev  $f$  = Dev { devEntries      ::  $f$  (Entry  $f$ )  
                  , devTip          :: Tip  
                  , devNSupply     :: NameSupply  
                  , devSuspendState :: SuspendState  
                  }
```

Tip

There are two kinds of Developments available: modules and definitions. A Module is a development that cannot have a type or value, but simply packs up some other developments. A development holding a definition can be in one of three states: an Unknown of the given type, a Suspended elaboration problem for producing a value of the type (see section 6.2), or a Defined term of the type. Note that the type is presented as both a term and a value for performance purposes.

```
data Tip  
  = Module  
  | Unknown (INTM :=> TY)  
  | Suspended (INTM :=> TY) EProb  
  | Defined INTM (INTM :=> TY)  
deriving Show
```

Entry

As mentioned above, a Dev is a kind of tree. The branches are introduced by the container f (Entry f) where f is Traversable, typically a backward list.

An Entry leaves a choice of shape for the branches. Indeed, it can either be:

- an Entity with a REF, the last component of its Name (playing the role of a cache, for performance reasons), and the term representation of its type, or
- a module, ie. a Name associated with a Dev that has no type or value

```

data Traversable  $f \Rightarrow$  Entry  $f$ 
  = EEntity { ref      :: REF
             , lastName :: (String, Int)
             , entity   :: Entity  $f$ 
             , term    :: INTM
             , anchor   :: Maybe String }
  | EModule { name    :: Name
             , dev    :: (Dev  $f$ ) }

```

In the Module case, we have already tied the knot, by defining M with a sub-development. In the Entity case, we give yet another choice of shape, thanks to the Entity f constructor. This constructor is defined in the next section.

Typically, we work with developments that use backwards lists, hence f is Bwd:

```

type Entries = Bwd (Entry Bwd)

```



Name caching. We have mentioned above that an Entity E caches the last component of its Name in the (String, Int) field. Indeed, grabbing that information asks for traversing the whole Name up to the last element:

```

mkLastName :: REF  $\rightarrow$  (String, Int)
mkLastName ( $n := \_$ ) = last  $n$ 

```

As we will need it quite frequently for display purposes, we extract it once and for all with *lastName* and later rely on the cached version. □

Entity

An Entity is either a Parameter or a Definition. A Definition can have children, that is sub-developments, whereas a Parameter cannot.

```

data Traversable  $f \Rightarrow$  Entity  $f$ 
  = Parameter ParamKind
  | Definition DefKind (Dev  $f$ )

```

For readability, let us collapse the Entity into the Entry with these useful patterns:

```

pattern EPARAM ref name paramKind term anchor =
  EEntity ref name (Parameter paramKind) term anchor
pattern EDEF ref name defKind dev term anchor =
  EEntity ref name (Definition defKind dev) term anchor

```

Kinds of Definitions: A *definition* eventually constructs a term, by a (possibly empty) development of sub-objects. The Tip of this sub-development will be Unknown, Suspended or Defined.

A programming problem is a special kind of definition: it follows a type Scheme (Section 3.3), the high-level type of the function we are implementing.

```

data DefKind = LETG | PROG (Scheme INTM)

```

Kinds of Parameters: A *parameter* is either a λ , \forall or Π abstraction. It scopes over all following entries and the definitions (if any) in the enclosing development.

```
data ParamKind = ParamLam | ParamAll | ParamPi
deriving (Show, Eq)
```

The link between a type and the kind of parameter allowed is defined by *lambdable*:

```
lambdable :: TY  $\rightarrow$  Maybe (ParamKind, TY, VAL  $\rightarrow$  TY)
lambdable (PI s t) = Just (ParamLam, s, (t$$$)  $\cdot$  A)
lambdable (PRF (ALL s p)) = Just (ParamAll, s,  $\lambda v \rightarrow$  PRF (p $$$ A v))
lambdable _ = Nothing
```

Suspension states

Definitions may have suspended elaboration processes attached, indicated by a Suspended tip. These may be stable or unstable. For efficiency in the scheduler, each development stores the state of its least stable child.

```
data SuspendState = SuspendUnstable | SuspendStable | SuspendNone
deriving (Eq, Show, Enum, Ord)
```

4.2 Managing Entries in a Development

4.2.1 Looking into an Entry

In the following, we define a bunch of helper functions to access specific fields of an Entry. The field we look for might not exist for all possible Entry, therefore we work in a Maybe world. The naming rule of thumb of these function is: prefix *entry* followed by the name of one field of the E or M cases, starting with a capital letter.

Hence, we have:

```
entryRef :: Traversable f  $\Rightarrow$  Entry f  $\rightarrow$  Maybe REF
entryRef (EEntity r _ _ _ _) = Just r
entryRef (EModule _ _) = Nothing

entryName :: Traversable f  $\Rightarrow$  Entry f  $\rightarrow$  Name
entryName (EEntity (n := _) _ _ _ _) = n
entryName (EModule n _) = n

entryLastName :: Traversable f  $\Rightarrow$  Entry f  $\rightarrow$  (String, Int)
entryLastName (EEntity _ xn _ _ _) = xn
entryLastName (EModule n _) = last n

entryScheme :: Traversable f  $\Rightarrow$  Entry f  $\rightarrow$  Maybe (Scheme INTM)
entryScheme (EDEF _ _ (PROG sch) _ _ _) = Just sch
entryScheme _ = Nothing

entryDev :: Traversable f  $\Rightarrow$  Entry f  $\rightarrow$  Maybe (Dev f)
entryDev (EDEF _ _ _ d _ _) = Just d
entryDev (EModule _ d) = Just d
entryDev (EPARAM _ _ _ _ _) = Nothing

entrySuspendState :: Traversable f  $\Rightarrow$  Entry f  $\rightarrow$  SuspendState
entrySuspendState e = case entryDev e of
  Just dev  $\rightarrow$  devSuspendState dev
  Nothing  $\rightarrow$  SuspendNone

entryAnchor :: Traversable f  $\Rightarrow$  Entry f  $\rightarrow$  Maybe String
entryAnchor (EEntity _ _ _ _ anchor) = anchor
entryAnchor (EModule _ _) = Nothing
```

4.2.2 Entry equality

Two entries are equal if and only if they have the same name:

```
instance Traversable  $f \Rightarrow$  Eq (Entry  $f$ ) where  
   $e1 \equiv e2 = \text{entryName } e1 \equiv \text{entryName } e2$ 
```

4.2.3 Changing the carrier of an Entry

The `entryCoerce` function is quite a thing. When defining `Dev`, we have been picky in letting any `Traversable f` be the carrier of the `f (Entry f)`. As shown in Section 4.4, we sometimes need to jump from one `Traversable f` to another `Traversable g` . In this example, we jump from a `NewsyFwd` – a `Fwd` list – to some `Entries` – a `Bwd` list.

Changing the type of the carrier is possible for parameters, in which case we return a `Right $entry$` . It is not possible for definitions and modules, in which case we return an unchanged `Left dev` .

```
 $entryCoerce ::$  (Traversable  $f$ , Traversable  $g$ )  $\Rightarrow$   
  Entry  $f \rightarrow$  Either (Dev  $f$ ) (Entry  $g$ )  
 $entryCoerce$  (EPARAM  $ref\ xn\ k\ ty\ anchor$ ) = Right $ EPARAM  $ref\ xn\ k\ ty\ anchor$   
 $entryCoerce$  (EDEF _ _ _  $dev$  _ _) = Left  $dev$   
 $entryCoerce$  (EModule _  $dev$ ) = Left  $dev$ 
```

4.3 News about updated references

The news system represents stored updates to references. For performance reasons, we do not wish to traverse the entire proof state every time modifications are made to one part of the tree. Instead, we store news entries below the cursor, and update following entries when the cursor moves down. This section describes the data that is stored in the proof state, and section 6.7 describes how news is propagated.

4.3.1 News

News represents possible changes to references. At the moment, it may be `GoodNews` (the reference has become more defined) or `NoNews` (even better from our perspective, as the reference has not changed). Note that `News` is ordered by increasing “niceness”.

When we come to implement functionality to remove definitions from the proof state, we will also need `BadNews` (the reference has changed but is not more informative) and `DeletedNews` (the reference has gone completely).

```
data News = DeletedNews | BadNews | GoodNews | NoNews deriving (Eq, Ord, Show)
```

Handily, `News` is a monoid where the neutral element is `NoNews` and composing two `News` takes the less nice.

```
instance Monoid News where  
   $mempty =$  NoNews  
   $mappend =$  min
```

4.3.2 News Bulletin

A `NewsBulletin` is a list of pairs of updated references and the news about them.

```
type NewsBulletin = [(REF, News)]
```

Adding news

The *addNews* function adds the given news to the bulletin, if it is newsworthy.

```
addNews :: (REF, News) → NewsBulletin → NewsBulletin
addNews (_, NoNews) old = old
addNews (r, n) old = (r, n `mappend` n') : old' where
  -- Find previous versions n' (if any),
  -- Remove duplicate in old':
  (n', old') = seek old
  seek [] = (NoNews, [])
  seek ((r', n') : old) | r ≡ r' = (n', old)
  seek (rn : old) = (n', rn : old') where (n', old') = seek old
```

Using *seek*, we enforce the invariant that any reference appears at most once in a *NewsBulletin*.

Getting the latest news

The *lookupNews* function returns the news about a reference contained in the bulletin, which may be *NoNews* if the reference is not present.

```
lookupNews :: NewsBulletin → REF → News
lookupNews nb ref = fromMaybe NoNews (lookup ref nb)
```

The *getNews* function looks for a reference in the news bulletin, and returns it with its news if it is found, or returns *Nothing* if not.

```
getNews :: NewsBulletin → REF → Maybe (REF, News)
getNews nb ref = find ((≡ ref) . fst) nb
```

The *getLatest* function returns the most up-to-date copy of the given reference, either the one from the bulletin if it is present, or the one passed in otherwise. If given a *FAKE* reference, it will always return one, regardless of the status of the reference in the bulletin. This ensures that fake references in labels have their types updated without turning into real definitions unexpectedly.

```
getLatest :: NewsBulletin → REF → REF
getLatest news ref@(nom := FAKE <: _) = nom := FAKE <: ty
  where _ := _ <: ty = realGetLatest news ref
getLatest news ref = realGetLatest news ref
```

This is implemented via *realGetLatest*, which ignores fakery. The slightly odd recursive case arises because equality for references just compares their names.

```
realGetLatest :: NewsBulletin → REF → REF
realGetLatest [] ref = ref
realGetLatest ((ref', _) : news) ref
  | ref ≡ ref' = ref'
  | otherwise = getLatest news ref
```

Merging news

The *mergeNews* function takes older and newer bulletins, and composes them to produce a single bulletin with the worst news about every reference mentioned in either.

```
mergeNews :: NewsBulletin → NewsBulletin → NewsBulletin
mergeNews new [] = new
mergeNews new old = foldr addNews old new
```

Read all about it

The *tellNews* function applies a bulletin to a term. It returns the updated term and the news about it (i.e. the least nice news of any reference used in the term). Using the Writer monad allows the term to be updated and the news about it calculated in a single traversal. Note that we ensure FAKE references remain as they are, as in *getLatest*.

```
tellNews :: NewsBulletin → Tm { d, TT } REF → (Tm { d, TT } REF, News)
tellNews [] tm = (tm, NoNews)
tellNews news tm = runWriter $ traverse teller tm
  where
    teller :: REF → Writer News REF
    teller r = case getNews news r of
      Nothing → return r
      Just (r', n) → tell n >> return (fixFake r r')
    fixFake :: REF → REF → REF
    fixFake (_ := FAKE <: _) (n := _ <: ty) = n := FAKE <: ty
    fixFake _ r = r
```

The *tellNewsEval* function takes a bulletin, term and its present value. It updates the term with the bulletin and re-evaluates it if necessary.

```
tellNewsEval :: NewsBulletin → INTM :=> VAL → (INTM :=> VAL, News)
tellNewsEval news (tm :=> tv) = case tellNews news tm of
  (_, NoNews) → (tm :=> tv, NoNews)
  (tm', GoodNews) → (tm' :=> evTm tm', GoodNews)
```

4.4 Proof Context

Recall from Section 4.1 the definition of a development:

```
type Dev = (f (Entry f), Tip, NameSupply)
```

We “unzip” (cf. Huet’s Zipper [11]) this type to produce a type representing its one-hole context. This allows us to keep track of the location of a working development and perform local navigation easily.

4.4.1 The derivative: Layer

Hence, we define Layer by unzipping Dev. Each Layer of the zipper is a record with the following fields:

aboveEntries appearing *above* the working development

currentEntry data about the working development

belowEntries appearing *below* the working development

layTip the Tip of the development that contains the current entry

layNSupply the NameSupply of the development that contains the current entry

laySuspendState the state of the development that contains the current entry

```

data Layer = Layer
  { aboveEntries    :: Entries
  , currentEntry   :: CurrentEntry
  , belowEntries   :: NewsyEntries
  , layTip         :: Tip
  , layNSupply     :: NameSupply
  , laySuspendState :: SuspendState
  }
deriving Show

```

The derivative makes sense only for definitions and modules, which have sub-developments. Parameters being childless, they ‘derive to 0’. Hence, the data about the working development is the derivative of the Definition and Module data-types defined in Section 4.1.1.

```

data CurrentEntry = CDefinition DefKind REF (String, Int) INTM (Maybe String)
                  | CModule Name
deriving Show

```

One would expect the *belowEntries* to be an Entries, just as the *aboveEntries*. However, the *belowEntries* needs to be a richer structure to support the news infrastructure (Section 4.3). Indeed, we propagate reference updates lazily, by pushing news bulletin below the current cursor.

Hence, the *belowEntries* are not only normal entries but also contain news. We define a **newtype** for the composition of the Fwd and Either NewsBulletin functors, and use this functor for defining the type of *belowEntries*.

```

newtype NewsyFwd x = NF { unNF :: Fwd (Either NewsBulletin x) }
type NewsyEntries = NewsyFwd (Entry NewsyFwd)

```

Note that *aboveEntries* are Entries, that is Bwd list. *belowEntries* are NewsyEntries, hence Fwd list. This justifies some piece of kit to deal with this global context.

4.4.2 The Zipper: ProofContext

Once we have the derivative, the zipper is almost here. The proof context is represented by a stack of layers (*pcLayers*), ending with the working development (*pcDev*) above the cursor and the entries below the cursor (*pcBelowCursor*).

```

data ProofContext = PC
  { pcLayers       :: Bwd Layer
  , pcAboveCursor :: Dev Bwd
  , pcBelowCursor :: Fwd (Entry Bwd)
  }
deriving Show

```

The *emptyContext* corresponds to the following (purposely verbose) definition:

```

emptyContext :: ProofContext
emptyContext = PC { pcLayers = B0
                  , pcAboveCursor = Dev { devEntries    = B0
                                           , devTip       = Module
                                           , devNSupply   = (B0, 0)
                                           , devSuspendState = SuspendNone }
                  , pcBelowCursor = F0 }

```

4.5 The ProofState monad

4.5.1 Defining the Proof State monad

The proof state monad provides access to the ProofContext as in a State monad, but with the possibility of command failure represented by Either (StackError *e*).

```
type ProofStateT e = StateT ProofContext (Either (StackError e))
```

Most of the time, we will work in a ProofStateT carrying errors composed with Strings and terms in display syntax. Hence the following type synonym:

```
type ProofState = ProofStateT DInTmRN
```

4.5.2 Error management toolkit

Some functions, such as *distill*, are defined in the ProofStateT INTM monad. However, Cochon lives in a ProofStateT DInTmRN monad. Therefore, in order to use it, we will need to lift from the former to the latter.

```
mapStackError :: (ErrorTok a → ErrorTok b) → StackError a → StackError b
mapStackError = fmap · fmap
```

```
liftError :: (a → b) → Either (StackError a) c → Either (StackError b) c
liftError f = either (Left · mapStackError (fmap f)) Right
```

```
liftError' :: (ErrorTok a → ErrorTok b) → Either (StackError a) c
           → Either (StackError b) c
liftError' f = either (Left · mapStackError f) Right
```

```
liftErrorState :: (a → b) → ProofStateT a c → ProofStateT b c
liftErrorState f = mapStateT (liftError f)
```

4.6 Managing Entries in a Proof Context

Manipulating the CurrentEntry

As with entries in Section 4.2, we need some kit operating on any kind of CurrentEntry. So far, this is restricted to getting its name:

```
currentEntryName :: CurrentEntry → Name
currentEntryName (CDefinition _ (n := _) _ _ _) = n
currentEntryName (CModule n)                  = n
```

There is an obvious (forgetful) map from entry (Definition or Module) to a current entry:

```
mkCurrentEntry :: Traversable f ⇒ Entry f → CurrentEntry
mkCurrentEntry (EDEF ref xn dkind _ ty a) = CDefinition dkind ref xn ty a
mkCurrentEntry (EModule n _)             = CModule n
```

From Above to Below, and back

The *aboveEntries* and *belowEntries* give a certain twist to the visit of a Layer: on one hand, *aboveEntries* go Bwd; on the other hand, *belowEntries* go Fwd with news. Therefore, when moving the cursor, we sometimes need to change the structure that contains entries.

We define such ‘rearranging’ function by mutual induction on Entry *f* and Dev *f*:

```

rearrangeEntry :: (Traversable f, Traversable g) =>
  (∀ a · f a → g a) → Entry f → Entry g
rearrangeEntry h (EPARAM ref xn k ty a) = EPARAM ref xn k ty a
rearrangeEntry h (EDEF ref xn k dev ty a) =
  EDEF ref xn k (rearrangeDev h dev) ty a
rearrangeEntry h (EModule n d) = EModule n (rearrangeDev h d)
rearrangeDev :: (Traversable f, Traversable g) =>
  (∀ a · f a → g a) → Dev f → Dev g
rearrangeDev h d@(Dev { devEntries = xs }) = d { devEntries = rearrangeEntries h xs }
  where rearrangeEntries :: (Traversable f, Traversable g) =>
    (∀ a · f a → g a) →
      f (Entry f) → g (Entry g)
    rearrangeEntries h xs = h (fmap (rearrangeEntry h) xs)

```

Hence, we can change the carrier of Entry from Bwd to Fwd or a variation thereof:

```

reverseEntry :: Entry Bwd → Entry NewsyFwd
reverseEntry = rearrangeEntry (NF · (fmap Right) · (<>>F0))
reverseEntries :: Fwd (Entry Bwd) → NewsyEntries
reverseEntries es = NF $ fmap (Right · reverseEntry) es

```

Or we can change the carrier of a whole Dev from Bwd to Fwd:

```

reverseDev :: Dev Bwd → Dev Fwd
reverseDev = rearrangeDev (<>>F0)

```

4.7 Fake references

The *getFakeCurrentEntry* command returns a neutral application of a fake reference that represents the current entry of the current location. Note that its type is λ -lifted over its parameters in global scope, but it is then applied to them (as shared parameters).

Pierre: Soon enough, this should disappear. We hope to introduce “high-level names” cleanly into the ProofState. They will cover the role currently played by FakeRefs to name label and let us find them into the ProofState.

```

getFakeRef :: ProofState REF
getFakeRef = do
  CDefinition _ (cEntryName := HOLE _ :<: ty) _ _ _ ← getCurrentEntry
  return $ cEntryName := FAKE :<: ty

```

```

getFakeCurrentEntry :: ProofState (EXTM :=>: VAL)
getFakeCurrentEntry = do
  r ← getFakeRef
  inScope ← getInScope
  let tm = P r $ $ (paramSpine inScope)
  return $ tm :=>: evTm tm

```

4.8 Scope management

4.8.1 Extracting scopes as entries

The *globalScope* function returns the parameters and definitions available in the current development, not including the ones involved in its construction.

```
globalScope :: ProofContext → Entries
globalScope pc = foldMap aboveEntries (pcLayers pc)
```

The *inScope* function returns all parameters and definitions above the cursor. These are all entries rightfully usable at the cursor's location.

```
inScope :: ProofContext → Entries
inScope pc@PC {pcAboveCursor = Dev {devEntries = es}} = globalScope pc < + > es
```

The *definitionsToImpl* function lists the entries above the cursor that have been issued during elaboration of a programming problem (Section 6.5.3). **Pierre**: That's all I know about it, sorry about that.

Pierre: This really is a hack. I hope it will disappear any time soon.

```
magicImplName = "impl"
definitionsToImpl :: ProofContext → [REF :<: INTM]
definitionsToImpl pc@PC {pcAboveCursor = Dev {devEntries = es}} =
  help (pcLayers pc) (params es)
  where
    help :: Bwd Layer → [REF :<: INTM] → [REF :<: INTM]
    help B0 xs = xs
    help (ls :< Layer {currentEntry = CDefinition _ _ (n, _) _}) xs
      | n ≡ magicImplName = xs
    help (ls :< l) xs = help ls (params (aboveEntries l) ++ xs)
    params = foldMap param
    param (EPARAM r _ _ t _) = [r :<: t]
    param _ = []
```

4.8.2 Manipulating entries as scopes

We often need to turn the sequence of parameters under which we work into the argument spine of a λ -lifted definition. Therefore, let us extract such spine from a list of entries:

```
paramREFs :: Entries → [REF]
paramREFs = foldMap param where
  param :: Entry Bwd → [REF]
  param (EPARAM r _ _ _ _) = [r]
  param _ = []
```

```
paramSpine :: Entries → Spine {p} REF
paramSpine = fmap (A · N · P) · paramREFs
```

Similarly, *applySpine* applies a reference to a given spine of parameters, provided as a spine. These are the shared parameters of a λ -lifted definition.

```
applySpine :: REF → Entries → EXTMTM :=>: VAL
applySpine ref aus = tm :=>: evTm tm
  where tm = P ref $ : $ paramSpine aus
```

4.9 The Get Set

We provide various functions to get information from the proof state and store updated information, providing a friendlier interface than *get* and *put*. The rule of thumb for naming these functions is to prefix the name of a field by the action (*get*, *put*, *remove*, or *replace*).

question: That would be great to have an illustration of the behavior of each of these functions on a development.

Pierre: Some of these functions pattern-match aggressively, at the risk of failing. Others carefully wrap their results in a *Maybe*. It would be good to decide a uniform approach there.

4.9.1 Getters

Getting in ProofContext

```
getLayers :: ProofStateT e (Bwd Layer)
getLayers = gets pcLayers

getAboveCursor :: ProofStateT e (Dev Bwd)
getAboveCursor = gets pcAboveCursor

getBelowCursor :: ProofStateT e (Fwd (Entry Bwd))
getBelowCursor = gets pcBelowCursor
```

And some specialized versions:

```
getLayer :: ProofStateT e Layer
getLayer = do
  _ :< l ← getLayers
  return l
```

Getting in AboveCursor

```
getEntriesAbove :: ProofStateT e Entries
getEntriesAbove = do
  dev ← getAboveCursor
  return $ devEntries dev

getDevNSupply :: ProofStateT e NameSupply
getDevNSupply = do
  dev ← getAboveCursor
  return $ devNSupply dev

getDevTip :: ProofStateT e Tip
getDevTip = do
  dev ← getAboveCursor
  return $ devTip dev
```

And some specialized versions:

```

getEntryAbove :: ProofStateT e (Entry Bwd)
getEntryAbove = do
  _:< e ← getEntriesAbove
  return e

getGoal :: String → ProofStateT e (INTM :=>: TY)
getGoal s = do
  tip ← getDevTip
  case tip of
    Unknown (ty :=>: tyTy) → return (ty :=>: tyTy)
    Defined _ (ty :=>: tyTy) → return (ty :=>: tyTy)
    _ → throwError' $ err "getGoal: fail to match a goal in "
        † err s

withGoal :: (VAL → ProofState ()) → ProofState ()
withGoal f = do
  (_ :=>: goal) ← getGoal "withGoal"
  f goal

```

Getting in the Layers

```

getCurrentEntry :: ProofStateT e CurrentEntry
getCurrentEntry = do
  ls ← getLayers
  case ls of
    _:< l → return (currentEntry l)
    B0 → return (CModule [])

```

Getting in the CurrentEntry

```

getCurrentName :: ProofStateT e Name
getCurrentName = do
  cEntry ← getCurrentEntry
  case cEntry of
    CModule [] → return []
    _ → return $ currentEntryName cEntry

getCurrentDefinition :: ProofStateT e (EXTM :=>: VAL)
getCurrentDefinition = do
  CDefinition _ ref _ _ _ ← getCurrentEntry
  scope ← getGlobalScope
  return (applySpine ref scope)

```

Getting in the HOLE

```

getHoleGoal :: ProofStateT e (INTM :=>: TY)
getHoleGoal = do
  CDefinition _ (_ := HOLE _:<: _) _ _ _ ← getCurrentEntry
  getGoal "getHoleGoal"

getHoleKind :: ProofStateT e HKind
getHoleKind = do
  CDefinition _ (_ := HOLE hk :<: _) _ _ _ ← getCurrentEntry
  return hk

```

Getting the Scopes

```
getInScope :: ProofStateT e Entries
getInScope = gets inScope

getDefinitionsToImpl :: ProofStateT e [REF :<: INTM]
getDefinitionsToImpl = gets definitionsToImpl

getGlobalScope :: ProofStateT e Entries
getGlobalScope = gets globalScope

getParamsInScope :: ProofStateT e [REF]
getParamsInScope = do
  inScope ← getInScope
  return $ paramREFs inScope
```

4.9.2 Putters

Putting in ProofContext

```
putLayers :: Bwd Layer → ProofStateT e ()
putLayers ls = do
  pc ← get
  put pc { pcLayers = ls }

putAboveCursor :: Dev Bwd → ProofStateT e ()
putAboveCursor dev = do
  replaceAboveCursor dev
  return ()

putBelowCursor :: Fwd (Entry Bwd) → ProofStateT e (Fwd (Entry Bwd))
putBelowCursor below = do
  pc ← get
  put pc { pcBelowCursor = below }
  return (pcBelowCursor pc)
```

And some specialized versions:

```
putLayer :: Layer → ProofStateT e ()
putLayer l = do
  pc@PC { pcLayers = ls } ← get
  put pc { pcLayers = ls :< l }

putEntryBelowCursor :: Entry Bwd → ProofStateT e ()
putEntryBelowCursor e = do
  below ← getBelowCursor
  putBelowCursor (e :> below)
  return ()
```

Putting in AboveCursor

```
putEntriesAbove :: Entries → ProofStateT e ()
putEntriesAbove es = do
  replaceEntriesAbove es
  return ()

putDevNSupply :: NameSupply → ProofStateT e ()
putDevNSupply ns = do
  dev ← getAboveCursor
  putAboveCursor dev { devNSupply = ns }

putDevSuspendState :: SuspendState → ProofStateT e ()
putDevSuspendState ss = do
  dev ← getAboveCursor
  putAboveCursor dev { devSuspendState = ss }

putDevTip :: Tip → ProofStateT e ()
putDevTip tip = do
  dev ← getAboveCursor
  putAboveCursor dev { devTip = tip }
```

And some specialized versions:

```
putEntryAbove :: Entry Bwd → ProofStateT e ()
putEntryAbove e = do
  dev ← getAboveCursor
  putAboveCursor dev { devEntries = devEntries dev :< e }
```

Putting in the Layers

```
putCurrentEntry :: CurrentEntry → ProofStateT e ()
putCurrentEntry m = do
  l ← getLayer
  _ ← replaceLayer l { currentEntry = m }
  return ()

putNewsBelow :: NewsBulletin → ProofStateT e ()
putNewsBelow news = do
  l ← getLayer
  replaceLayer l { belowEntries = NF (Left news :> unNF (belowEntries l)) }
  return ()
```

Putting in the CurrentEntry

Putting in the PROG

```
putCurrentScheme :: Scheme INTM → ProofState ()
putCurrentScheme sch = do
  CDefinition _ ref xn ty a ← getCurrentEntry
  putCurrentEntry $ CDefinition (PROG sch) ref xn ty a
```

Putting in the HOLE

```
putHoleKind :: HKind → ProofStateT e ()
putHoleKind hk = do
  CDefinition kind (name := HOLE _ :<: ty) xn tm a ← getCurrentEntry
  putCurrentEntry $ CDefinition kind (name := HOLE hk :<: ty) xn tm a
```

4.9.3 Removers

Remove in ProofContext

```
removeLayer :: ProofStateT e Layer
removeLayer = do
  pc@PC {pcLayers = ls :< l} ← get
  put pc {pcLayers = ls}
  return l
```

Removing in AboveEntries

```
removeEntryAbove :: ProofStateT e (Maybe (Entry Bwd))
removeEntryAbove = do
  es ← getEntriesAbove
  case es of
    B0 → return Nothing
    (es' :< e) → do
      putEntriesAbove es'
      return $ Just e
```

4.9.4 Replacers

Replacing into ProofContext

```
replaceAboveCursor :: Dev Bwd → ProofStateT e (Dev Bwd)
replaceAboveCursor dev = do
  pc ← get
  put pc {pcAboveCursor = dev}
  return (pcAboveCursor pc)
```

And some specialized version:

```
replaceLayer :: Layer → ProofStateT e Layer
replaceLayer l = do
  (ls :< l') ← getLayers
  putLayers (ls :< l)
  return l'
```

Replacing in AboveCursor

```
replaceEntriesAbove :: Entries → ProofStateT e Entries
replaceEntriesAbove es = do
  dev ← getAboveCursor
  putAboveCursor dev {devEntries = es}
  return (devEntries dev)
```

4.10 Navigating in the Proof Context

In Section 4.1, we have developed the notion of Development, a tree reifying the proof construction process. In order to navigate this tree, we have computed its zipper in Section 4.4, the ProofContext. At this stage, we have a notion of *movement* in the proof context.

However, we had to postpone the development of navigation commands to this stage, where we have the ability to *edit* the ProofState (Section 4.5). Indeed, when moving down, we might hit a news bulletin. A news bulletin is a lazy edition process. In order to move, we have to propagate the news, hence effectively editing the proof state.

4.10.1 One-step navigation

We shall now develop this navigation kit, comfortably installed in the ProofState monad. First, some vocabulary. The *focus* is the current development; it contains a *cursor* which is the point at which changes take place. Consider the following development presented in Figure 4.1: we have that the development B is in focus, with *y* above the cursor and *z* below it.

The navigation commands are the following:

- Current entry navigation:
 - *putEnterCurrent*
 - *leaveEnterCurrent*
- Cursor navigation:
 - *cursorUp* moves the cursor up by one entry (under E in the example);
 - *cursorDown* moves the cursor down by one entry (under *z* in the example);
- Focus navigation:
 - *goIn* moves the focus in the first definition above the cursor, and brings the cursor at the bottom of the newly focused development (inside E and below *b* in the example);
 - *goOut* moves the focus out to the development that contains it, with the cursor at the bottom of the development (under G in the example);
 - *goOutBelow* moves the focus out to the development that contains it, with the cursor under the previously focused entry (under B in the example);
 - *goUp* moves the focus up to the closest definition and leaves the cursor at the bottom (inside A in the example); and
 - *goDown* moves the focus down to the closest definition and leaves the cursor at the bottom (inside C in the example).

These commands fail with an error if they are impossible because the proof context is not in the required form. Things are slightly more complicated than the above description because of the possibility of news bulletins below the cursor; as these are propagated lazily, they must be pushed down when the cursor or focus move.

From Entry to Current entry, and back

With *getCurrentEntry* and *putCurrentEntry*, we know how to access the current entry, and overwrite it. However, when navigating through the proof context, we will change focus, therefore *leaving* the current entry, or *entering* into another.

When leaving the current entry, the current entry is turned back into a conventional entry, so that it can be inserted with its peers in the development (above or below). In a word, this operation *zip* back the current entry.

```

[
  A [
    I : S
    \ e : S
    \ f : S
  ] : S
  \ u : S
  \ v : S
  B [
    D := ? : S
    \ x : S
    E [
      \ a : S
      F := ? : S
      \ b : S
    ] : S
    \ y : S
    -= Cursor here ==
    \ z : S
  ] : S
  \ w : S
  C [
    \ g : S
    H := ? : S
    \ h : S
  ] : S
  G := ? : S
]
(a) Example

```

```

[
  A [
    I : S
    \ e : S
    \ f : S
  ] : S
  \ u : S
  \ v : S
  B [
    D := ? : S
    \ x : S
    E [
      \ a : S
      F := ? : S
      \ b : S
    ] : S
    -= Cursor here ==
    \ y : S
    \ z : S
  ] : S
  \ w : S
  C [
    \ g : S
    H := ? : S
    \ h : S
  ] : S
  G := ? : S
]
(b) After cursorUp

```

```

[
  A [
    I : S
    \ e : S
    \ f : S
  ] : S
  \ u : S
  \ v : S
  B [
    D := ? : S
    \ x : S
    E [
      \ a : S
      F := ? : S
      \ b : S
    ] : S
    \ y : S
    -= Cursor here ==
    \ z : S
  ] : S
  \ w : S
  C [
    \ g : S
    H := ? : S
    \ h : S
  ] : S
  G := ? : S
]
(c) After cursorDown

```

```

[
  A [
    I : S
    \ e : S
    \ f : S
  ] : S
  \ u : S
  \ v : S
  B [
    D := ? : S
    \ x : S
    E [
      \ a : S
      F := ? : S
      \ b : S
    ] : S
    -= Cursor here ==
    \ y : S
    \ z : S
  ] : S
  \ w : S
  C [
    \ g : S
    H := ? : S
    \ h : S
  ] : S
  G := ? : S
]
(d) After goIn

```

```

[
  A [
    I : S
    \ e : S
    \ f : S
  ] : S
  \ u : S
  \ v : S
  B [
    D := ? : S
    \ x : S
    E [
      \ a : S
      F := ? : S
      \ b : S
    ] : S
    \ y : S
    \ z : S
  ] : S
  \ w : S
  C [
    \ g : S
    H := ? : S
    \ h : S
  ] : S
  G := ? : S
  -= Cursor here ==
]
(e) After goOut

```

```

[
  A [
    I : S
    \ e : S
    \ f : S
  ] : S
  \ u : S
  \ v : S
  B [
    D := ? : S
    \ x : S
    E [
      \ a : S
      F := ? : S
      \ b : S
    ] : S
    \ y : S
    \ z : S
  ] : S
  \ w : S
  -= Cursor here ==
  C [
    \ g : S
    H := ? : S
    \ h : S
  ] : S
  G := ? : S
]
(f) After goOutBelow

```

```

[
  A [
    I : S
    \ e : S
    \ f : S
  ] : S
  -= Cursor here ==
  ] : S
  \ u : S
  \ v : S
  B [
    D := ? : S
    \ x : S
    E [
      \ a : S
      F := ? : S
      \ b : S
    ] : S
    \ y : S
    \ z : S
  ] : S
  \ w : S
  C [
    \ g : S
    H := ? : S
    \ h : S
  ] : S
  G := ? : S
]
(g) After goUp

```

```

[
  A [
    I : S
    \ e : S
    \ f : S
  ] : S
  \ u : S
  \ v : S
  B [
    D := ? : S
    \ x : S
    E [
      \ a : S
      F := ? : S
      \ b : S
    ] : S
    \ y : S
    \ z : S
  ] : S
  \ w : S
  C [
    \ g : S
    H := ? : S
    \ h : S
  ] : S
  -= Cursor here ==
  G := ? : S
]
(h) After goDown

```

Figure 4.1: Navigation in a development

```

getLeaveCurrent :: ProofState (Entry Bwd)
getLeaveCurrent = do
  currentEntry ← getCurrentEntry
  Dev above tip root state ← getAboveCursor
  below ← getBelowCursor
  let dev = Dev (above <>< below) tip root state
  case currentEntry of
    CDefinition dkind ref xn ty a → return $ EDEF ref xn dkind dev ty a
    CModule n → return $ EModule n dev

```

Conversely, when entering a new development, the former entry needs to be *unzipped* to form the current development.

```

putEnterCurrent :: Entry Bwd → ProofState ()
putEnterCurrent (EDEF ref xn dkind dev ty a) = do
  l ← getLayer
  replaceLayer $ l { currentEntry = CDefinition dkind ref xn ty a }
  putAboveCursor dev
putEnterCurrent (EModule [] dev) = putAboveCursor dev
putEnterCurrent (EModule n dev) = do
  l ← getLayer
  replaceLayer $ l { currentEntry = CModule n }
  putAboveCursor dev

```

Cursor navigation

Cursor movement is straightforward, as there is no news to worry about. We simply move an entry above the cursor to one below, or vice versa.

```

cursorUp :: ProofState ()
cursorUp = do
  -- Look above
  above ← getEntriesAbove
  case above of
    aboveE :< e → do
      below ← getBelowCursor
      -- Move e from above to below
      putEntriesAbove aboveE
      putBelowCursor $ e :> below
      return ()
    B0 → do
      -- There is no above..
      throwError' $ err "cursorUp: cannot move cursor up."
cursorDown :: ProofState ()
cursorDown = do
  -- Look below
  above ← getEntriesAbove
  below ← getBelowCursor
  case below of
    e :> belowE → do
      -- Move e from below to above
      putEntriesAbove (above :< e)
      putBelowCursor belowE
      return ()
    F0 → do
      -- There is no below..
      throwError' $ err "cursorDown: cannot move cursor down."

```

Focus navigation

Pierre: I'm somewhat uneasy with this kind of definition. On one hand, it is thriving to avoid failure (looking up for an actual definition, then moving into it). On the other hand, it is doing two things, non-compositionally. However, writing two distinct functions — one doing the lookup, the other moving in a definition — exposes some invariants ("called on a definition" for the second one) that I don't know how to enforce with types instead of dynamic checks.

The *goIn* command moves the cursor upward, until it reaches a definition. If one can be found, it enters it and goes at the bottom.

```

goIn :: ProofState ()
goIn = do
  above ← getEntriesAbove
  case above of
    B0 → do
      -- Nothing above: we cannot go above
      throwError' $ err "goIn: you can't go that way."
    aboveE :< e → case entryDev e of
      Nothing → do
        -- This entry is not a Definition: look further up
        cursorUp >> goIn
      Just dev → do
        -- We are right into a Definition
        oldFocus ← getAboveCursor
        below ← getBelowCursor
        -- Set the focus to this Definition
        putLayer $ Layer { aboveEntries = aboveE
                          , currentEntry = mkCurrentEntry e
                          , belowEntries = reverseEntries below
                          , layTip = devTip oldFocus
                          , layNSupply = devNSupply oldFocus
                          , laySuspendState = devSuspendState oldFocus }
        -- Set cursor at the bottom
        putAboveCursor dev
        putBelowCursor F0
        return ()

```

The *goOut* command moves the focus to the outer layer, with the cursor at the bottom of it. Therefore, we zip back the current development, with the additional burden of dealing with news.

```

goOut :: ProofState ()
goOut = do
  -- Leave the current entry
  e ← getLeaveCurrent
  -- Move one layer out
  mLayer ← optional removeLayer
  case mLayer of
    Just l → do
      -- Update the current development
      putAboveCursor $ Dev { devEntries      = aboveEntries l :< e
                            , devTip          = layTip l
                            , devNSupply     = layNSupply l
                            , devSuspendState = laySuspendState l }
      putBelowCursor F0
      propagateNews NormalPropagate [] (belowEntries l)
      -- Here, the cursor is at the bottom of the current development
      return ()
    Nothing → do
      -- Already at outermost position
      throwError' $ err "goOut: you can't go that way."

```

The *goOutBelow* variant has a similar effect than *goOut*, excepted that it brings the cursor right under the previous point of focus.

```

goOutBelow :: ProofState ()
goOutBelow = do
  -- Retrieve the number of entries below the previous point of focus
  ls ← getLayers
  case ls of
  _ :< Layer { belowEntries = below } → do
    -- Go out: end up at the bottom of the development
    goOut
    -- Move cursor up by as many entries there was
    Data.Traversable.mapM (const cursorUp) below
    return ()
  B0 → throwError' $ err "goOutBelow: you can't go that way."

```

The *goUp* command moves the focus upward, looking for a definition. If one can be found, the cursor is moved at the bottom of the new development.

```

goUp :: ProofState ()
goUp = goUpAcc (NF F0)
  where
    goUpAcc :: NewsyEntries → ProofState ()
    goUpAcc (NF visitedBelow) = do
      -- Get the directly enclosing layer
      l ← getLayer
      case l of
      (Layer (aboveE :< e) m (NF below) tip nsupply state) →
        -- It has at least one entry
        case entryDev e of
        Just dev → do
          -- The entry is a Definition
          -- Leave our current position
          currentE ← getLeaveCurrent
          -- Put the cursor at the bottom of the development
          putAboveCursor dev
          putBelowCursor F0
          -- Set focus on this definition
          let belowE = NF $ visitedBelow
              < + > (Right (reverseEntry currentE) :> below)
          replaceLayer $ l { aboveEntries = aboveE
                          , currentEntry = mkCurrentEntry e
                          , belowEntries = belowE }
          return ()
        Nothing → do
          -- The entry is a Parameter
          -- Move up, accumulating the the current entry
          replaceLayer $ l { aboveEntries = aboveE }
          goUpAcc $ NF (Right (reverseEntry e) :> visitedBelow)
      _ → do
        -- There is no up
        throwError' $ err "goUp: you can't go that way."

```

Similarly to *goUp*, the *goDown* command moves the focus downward, looking for a definition. If one can be found, the cursor is placed at the bottom of the new development. As often, moving down implies dealing with news: we accumulate them as we go, updating the parameters on our way.

```

goDown :: ProofState ()
goDown = goDownAcc B0 []
  where
    goDownAcc :: Entries → NewsBulletin → ProofState ()
    goDownAcc visitedAbove visitedNews = do
      -- Get the directly enclosing layer
      l ← getLayer
    case l of
      (Layer { aboveEntries = above, belowEntries = NF (ne :> belowNE) }) →
        -- What is the entry below?
        case ne of
          Left newsBulletin → do
            -- A news bulletin:
            -- Keep going down, accumulating the news
            replaceLayer $ l { belowEntries = NF belowNE }
            goDownAcc visitedAbove $ mergeNews visitedNews newsBulletin
          Right e →
            -- A real entry:
            -- Definition or Parameter?
            case entryCoerce e of
              Left (Dev es' tip' nsupply' ss') → do
                -- Definition:
                -- Leave our current position
                currentE ← getLeaveCurrent
                -- Set focus on this definition
                let aboveE = (above :< currentE) < + > visitedAbove
                    in replaceLayer $ l { aboveEntries = aboveE
                                          , currentEntry = mkCurrentEntry e
                                          , belowEntries = NF belowNE }
                -- Put the cursor at the bottom of the development
                -- The suspend state is cleared because there are no
                -- entries in the Dev; the state will be updated
                -- during news propagation.
                putAboveCursor (Dev B0 tip' nsupply' SuspendNone)
                putBelowCursor F0
                -- Push the collected news from above into the entries
                propagateNews NormalPropagate visitedNews es'
                return ()
              Right param → do
                -- Parameter:
                -- Push the news into it
                (news, param') ← tellEntry visitedNews param
                -- Keep going down
                replaceLayer $ l { belowEntries = NF belowNE }
                goDownAcc (visitedAbove :< param') news
            _ → do
              -- There is no down
              throwError' $ err "goDown: you can't go that way."

```

4.10.2 Many-step Navigation

The following functions are trivial iterations of the ones developed above.

```

cursorTop :: ProofState ()
cursorTop = much cursorUp
cursorBottom :: ProofState ()
cursorBottom = much cursorDown

goTop :: ProofState ()
goTop = much goUp
goBottom :: ProofState ()
goBottom = much goDown
goRoot :: ProofState ()
goRoot = much goOut

```

4.11 The ProofState Kit

The proof state lives on a rich substrate of operations, inherited from the ProofContext as well as the ProofState monad. In this module, we export these operations as part of the Interface.

4.11.1 Accessing the NameSupply

By definition of the Development in Section 4.1, we have that every entry is associated a namespace by the mean of a local name supply. As a result, the ProofState can almost be made a NameSupplier. The exception being that it cannot fork the name supply, because it cannot generates new namespaces.

```

instance NameSupplier (ProofStateT e) where
  freshRef (s <: ty) f = do
    nsupply ← getDevNSupply
    freshRef (s <: ty) (λref nsupply' → do
      putDevNSupply nsupply'
      f ref
    ) nsupply
  forkNSupply = error "ProofState does not provide forkNSupply"
  askNSupply = getDevNSupply


```

We also provide an operator to lift functions from a name supply to proof state commands.

```

withNSupply :: (NameSupply → x) → ProofStateT e x
withNSupply f = getDevNSupply >>= return . f

```

 *Read-only name supply.* Note that this function has no way to return an updated name supply to the proof context, so it must not leave any references around when it has finished. □

4.11.2 Accessing the type-checker

First off, we can access the β -normalizer: the *bquoteHere* command β -quotes a term using the local name supply.

```

bquoteHere :: Tm {d, VV} REF → ProofStateT e (Tm {d, TT} REF)
bquoteHere tm = withNSupply $ bquote B0 tm

```

Secondly, any type-checking problem (defined in the Check monad) can be executed in the ProofState.

```

runCheckHere :: (ErrorTok e → ErrorTok DInTmRN) → Check e a → ProofState a
runCheckHere f c = do
  me ← withNSupply $ liftError' f · typeCheck c
  lift me

```

As a consequence, we have *checkHere* to *check* terms against types:

```

checkHere :: (TY :>: INTM) → ProofState (INTM :=>: VAL)
checkHere tt = runCheckHere (fmap DTIN) $ check tt

```

and *inferHere* to *infer* types from terms:

```

inferHere :: EXTm → ProofState (VAL <: TY)
inferHere tm = runCheckHere (fmap DTIN) $ infer tm

```

4.11.3 Being paranoid

The *validateHere* command performs some sanity checks on the current location, which may be useful for paranoia purposes.

```

validateHere :: ProofState ()
validateHere = do
  m ← getCurrentEntry
  case m of
    CDefinition _ (_ := DEFN tm <: ty) _ _ _ → do
      ty' ← bquoteHere ty
      checkHere (SET >: ty')
      'pushError' (err "validateHere: definition type failed to type-check: SET does not admit"
        ++ errTyVal (ty <: SET))
      tm' ← bquoteHere tm
      checkHere (ty >: tm')
      'pushError' (err "validateHere: definition failed to type-check:"
        ++ errTyVal (ty <: SET)
        ++ err "does not admit"
        ++ errTyVal (tm <: ty))
      return ()
    CDefinition _ (_ := HOLE _ <: ty) _ _ _ → do
      ty' ← bquoteHere ty
      checkHere (SET >: ty')
      'pushError' (err "validateHere: hole type failed to type-check: SET does not admit"
        ++ errTyVal (ty <: SET))
      return ()
  _ → return ()

```

4.12 Name management

The *lookupName* function looks up a name in the context (including axioms and primitives); if found, it returns the reference applied to the spine of shared parameters.

```

lookupName :: Name → ProofStateT e (Maybe (EXTM :=> VAL))
lookupName name = do
  inScope ← getInScope
  case find ((name ≡) · entryName) inScope of
    Just (EEntity ref _ _ _) → return $ Just $ applySpine ref inScope
    Nothing →
      case find ((name ≡) · refName · snd) primitives of
        Just (_, ref) → return $ Just $ applySpine ref inScope
        Nothing → return Nothing

```

The `pickName` command takes a prefix suggestion and a name suggestion (either of which may be empty), and returns a more-likely-to-be-unique name if the name suggestion is empty.

```

pickName :: String → String → ProofState String
pickName "" s = pickName "x" s
pickName prefix "" = do
  m ← getCurrentName
  r ← getDevNSupply
  return $ prefix ++ show (foldMap snd m + snd r)
pickName _ s = return s

```

4.13 Making Parameters

4.13.1 λ -abstraction

When working at solving a goal, we might be able to introduce an hypothesis. For instance, if the goal type is `Nat → Nat → Nat`, we can introduce two hypotheses x and y . Further, the type of the goal governs the kind of the parameter (a lambda, or a forall) and its type. This automation is implemented by `lambdaParam` that lets you introduce a parameter above the cursor while working on a goal.

```

lambdaParam :: String → ProofState REF
lambdaParam x = do
  tip ← getDevTip
  case tip of
    Unknown (pi :=> ty) →
      -- Working at solving a goal
      case lambdaable ty of
        Just (paramKind, s, t) →
          -- Where can rightfully introduce a lambda
          freshRef (x <: s) $ λref → do
            sTm ← bquoteHere s
            -- Insert the parameter above the cursor
            putEntryAbove $ EPARAM ref (mkLastName ref) paramKind sTm Nothing
            -- Update the Tip
            let tipTy = t $ pval ref
                tipTyTm ← bquoteHere tipTy
            putDevTip (Unknown (tipTyTm :=> tipTy))
            -- Return the reference to the parameter
            return ref
          _ → throwError' $ err "lambdaParam: goal is not a pi-type or all-proof."
          _ → throwError' $ err "lambdaParam: only possible for incomplete goals."

```

4.13.2 Assumptions

With *lambdaParam*, we can introduce parameters under a proof goal. However, when working under a module, we would like to be able to introduce hypothesis of some type. This corresponds to some kind of “Assume” mechanism, where we assume the existence of an object of the provided type under the given module.

```
assumeParam :: (String <: (INTM ==> TY)) → ProofState REF
assumeParam (x <: (tyTm ==> ty)) = do
  tip ← getDevTip
  case tip of
    Module →
      -- Working under a module
      freshRef (x <: ty) $ λref → do
        -- Simply make the reference
        putEntryAbove $ EPARAM ref (mkLastName ref) ParamLam tyTm Nothing
        return ref
    _ → throwError' $ err "assumeParam: only possible for modules."
```

4.13.3 Pi-abstraction

When working at defining a type (an object in Set), we can freely introduce Pi-abstractions. This is precisely what *piParam* let us do.

```
piParam :: (String <: INTM) → ProofState REF
piParam (s <: ty) = do
  ttv ← checkHere $ SET >: ty
  piParamUnsafe $ s <: ttv
```

The variant *piParamUnsafe* will not check that the proposed type is indeed a type, so it requires further attention.

```
piParamUnsafe :: (String <: (INTM ==> TY)) → ProofState REF
piParamUnsafe (s <: (tyTm ==> ty)) = do
  tip ← getDevTip
  case tip of
    Unknown (_ ==> SET) →
      -- Working on a goal of type Set
      freshRef (s <: ty) $ λref → do
        -- Simply introduce the parameter
        putEntryAbove $ EPARAM ref (mkLastName ref) ParamPi tyTm Nothing
        return ref
    Unknown _ → throwError' $ err "piParam: goal is not of type SET."
    _ → throwError' $ err "piParam: only possible for incomplete goals."
```

4.14 Making Definitions

The *make* command adds a named goal of the given type above the cursor. The meat is actually in *makeKinded*, below.

```
make :: (String <: INTM) → ProofState (EXTM ==> VAL)
make = makeKinded Nothing Waiting
```

When making a new definition, the reference to this definition bears a *hole kind* (Section ??). User-generated goals are of kind *Waiting*: waiting for the user to solve it (or, if lucky, an automation

tool could nail it down). For making these kind of definition, we will use the *make* command above. However, during Elaboration for instance (Section 6.5), the proof system will insert goals itself, with a somewhat changing mood such as Hoping or Crying.

```

makeKinded :: Maybe String → HKind → (String <: INTM) →
            ProofState (EXTM :=>: VAL)
makeKinded manchor holeKind (name <: ty) = do
  -- Check that the type is indeed a type
  _ :=>: tyv ← checkHere (SET >: ty)
  ‘pushError‘
  (err "make: " ++ errTm (DTIN ty) ++ err " is not a set.")
  -- Make a name for the goal, from name
  nsupply ← getDevNSupply
  goalName ← pickName "Goal" name
  let n = mkName nsupply goalName
  -- Make a reference for the goal, with a lambda-lifted type
  inScope ← getInScope
  let liftedTy = liftType inScope ty
      ref      = n := HOLE holeKind <: evTm liftedTy
  -- Make an entry for the goal, with an empty development
  let dev = Dev { devEntries = B0
                , devTip      = Unknown (ty :=>: tyv)
                , devNSupply  = freshNSpace nsupply goalName
                , devSuspendState = SuspendNone }
  -- Put the entry in the proof context
  putDevNSupply $ freshName nsupply
  putEntryAbove $ EDEF ref (last n) LETG dev liftedTy manchor
  -- Return a reference to the goal
  return $ applySpine ref inScope

```

4.15 Modules in Proof Context

A module is a pretty dull thing: it is just a name and a development. As far as I know, there are three usages of modules. The first one is the one we are used to: introducing namespaces and avoiding name clashes. This is mostly used at the programming level. For making modules, we use *makeModule*.

```

makeModule :: String → ProofState Name
makeModule s = do
  nsupply ← getDevNSupply
  let n = mkName nsupply s
  -- Insert a new entry above, the empty module s
  let dev = Dev { devEntries = B0
                , devTip      = Module
                , devNSupply  = freshNSpace nsupply s
                , devSuspendState = SuspendNone }
  putEntryAbove $ EModule { name = n
                           , dev  = dev }
  putDevNSupply $ freshName nsupply
  return n

```

The second usage is more technical, and occurs exclusively in the implementation (such as in tactics, for instance). It consists in making a module, going in it, doing some constructions and analyses, and at some stage wanting to say that this module is actually an open definition of a

certain type (a goal). Turning a module into a goal is implemented by *moduleToGoal*. An instance of this pattern appears in Section 5.2.1.

```

moduleToGoal :: INTM → ProofState (EXTM :=>: VAL)
moduleToGoal ty = do
  (_ :=>: tyv) ← checkHere (SET :=>: ty)
  CModule n ← getCurrentEntry
  inScope ← getInScope
  let ty' = liftType inScope ty
      ref = n := HOLE Waiting :<: evTm ty'
  putCurrentEntry $ CDefinition LETG ref (last n) ty' Nothing
  putDevTip $ Unknown (ty :=>: tyv)
  return $ applySpine ref inScope

```

The last usage of modules is to mess around: introducing things in the proof context to later, in one go, remove it all. One need to be extremely careful with the removed objects: the risk of introducing dangling references is high.

```

draftModule :: String → ProofState t → ProofState t
draftModule name draftyStuff = do
  makeModule name
  goIn
  t ← draftyStuff
  goOutBelow
  mm ← removeEntryAbove
  case mm of
    Just (EModule _ _) → return t
    _ → throwError' · err $ "draftModule: drafty " ++ name
      ++ " did not end up in the right place!"

```

4.16 Searching in the Proof Context

In Section 4.10, we have developed several commands to navigate in the proof context. Here, we extend these to be able *search* in the proof context.

4.16.1 Searching by name

The *goTo* command moves the focus to the entry with the given name. Recall that a long name is an itinerary in the proof context, listing short names from the root down to the entry. Hence, we simply have to follow this itinerary to reach our destination.

```

goTo :: Name → ProofState ()
goTo name = do
  -- Start from the root
  goRoot
  -- Eat the name as you move in the context
  goTo' name
where
  goTo' :: Name → ProofState ()
  goTo' [] = do
    -- Reached the end of the journey
    return ()
  goTo' x@(xn : xns) = goIn >> seek xn >> goTo' xns
    'pushError'
    (err "goTo: could not find " ++ err (showName x))
  -- seek find the local short name on our itinerary
  seek :: (String, Int) → ProofState ()
  seek xn = do
    goUp 'whileA' (guard · (≡ xn) · last ≪≪ getCurrentName)

```

4.16.2 Searching for a goal

First off, a *goal* is a development which Tip is of the Unknown type. Hence, to spot if the focus is set on a goal, we just have to check the Tip.

```

isGoal :: ProofState ()
isGoal = do
  Unknown _ ← getDevTip
  return ()

```

Let us start with some gymnastic. We implement *prevStep* and *nextStep* that respectively look for the previous and the next definition in the proof context. By *previous*, we mean contained in an entry directly above, or, if there is none, to the enclosing development. In other words, it has been defined “just previously”. The definition transposes to the case of *nextStep*.

```

prevStep :: ProofState ()
prevStep = (goUp >> much goIn) ⊕ goOut
  'pushError'
  (err "prevStep: no previous steps.")
nextStep :: ProofState ()
nextStep = (goIn >> goTop) ⊕ goDown ⊕ (goOut 'untilA' goDown)
  'pushError'
  (err "nextStep: no more steps.")

```

It is then straightforward to navigate relatively to goals: we move from steps to steps, looking for a step that would be a goal.

```

prevGoal :: ProofState ()
prevGoal = prevStep 'untilA' isGoal
  'pushError'
  (err "prevGoal: no previous goals.")
nextGoal :: ProofState ()
nextGoal = nextStep 'untilA' isGoal
  'pushError'
  (err "nextGoal: no more goals.")

```

In the very spirit of a theorem prover, we sometimes want to stay at the current location if it is a goal, and go to the next goal otherwise.

```
seekGoal :: ProofState ()
seekGoal = isGoal ⊕ nextGoal
```

4.17 Solving goals

4.17.1 Giving a solution

The *give* command takes a term and solves the current goal with it, if it type-checks. At the end of the operation, the cursor has not moved: we are still under the goal, which has now been Defined. Note that entries below the cursor are (lazily) notified of the good news.

```
give :: INTM → ProofState (EXTM :=>: VAL)
give tm = do
  tip ← getDevTip
  case tip of
    Unknown (tipTyTm :=>: tipTy) → do
      -- Working on a goal
      -- The tm is of the expected type
      checkHere (tipTy :=>: tm)
      'pushError'
      (err "give: typechecking failed:" ++ errTm (DTIN tm)
        ++ err "is not of type" ++ errTyVal (tipTy :=>: SET))
      -- Lambda lift the given solution
      globalScope ← getGlobalScope
      above ← getEntriesAbove
      let tmv = evTm $ parBind globalScope above tm
          -- Update the entry as Defined, together with its definition
          CDefinition kind (name := _ :=>: ty) xn tyTm a ← getCurrentEntry
          let ref = name := DEFN tmv :=>: ty
              putDevTip $ Defined tm $ tipTyTm :=>: tipTy
              putCurrentEntry $ CDefinition kind ref xn tyTm a
          -- Propagate the good news
              updateRef ref
          -- Return the reference
              return $ applySpine ref globalScope
      _ → throwError' $ err "give: only possible for incomplete goals."
```

For convenience, we combine giving a solution and moving. Indeed, after *give*, the cursor stands in a rather boring position: under a Defined entry, with no work to do. So, a first variant is *giveOutBelow* that gives a solution and moves just below the now-defined entry. A second variant is *giveNext* that gives as well and moves to the next goal, if one is available.

```
giveOutBelow :: INTM → ProofState (EXTM :=>: VAL)
giveOutBelow tm = give tm < * goOutBelow
giveNext :: INTM → ProofState (EXTM :=>: VAL)
giveNext tm = give tm < * (nextGoal ⊕ goOut)
```

4.17.2 Finding trivial solutions

Often, to solve a goal, we make a definition that contains further developments and, eventually, leads to a solution. To solve the goal, we are therefore left to *give* this definition. This is the role of the *done* command that tries to *give* the entry above the cursor.

```

done :: ProofState (EXTM :=>: VAL)
done = do
  devEntry ← getEntryAbove
  case devEntry of
    EDEF ref _ _ _ _ _ → do
      -- The entry above is indeed a definition
      giveOutBelow $ NP ref
    _ → do
      -- The entry was not a definition
      throwError' $ err "done: entry above cursor must be a definition."

```

Slightly more sophisticated is the well-known *apply* tactic in Coq: we are trying to solve a goal of type T while we have a definition of type Pi S T. We can therefore solve the goal T provided we can solve the goal S. We have this tactic too and, guess what, it is *apply*.

```

apply :: ProofState (EXTM :=>: VAL)
apply = do
  devEntry ← getEntryAbove
  case devEntry of
    EDEF f@(_ := _ :<: (PI _S _T)) _ _ _ _ _ → do
      -- The entry above is a proof of Pi S T
      -- Ask for a proof of S
      _STm ← bquoteHere _S
      sTm :=>: s ← make $ "s" :<: _STm
      -- Make a proof of T
      _TTm ← bquoteHere $ _T $$ A s
      make $ "t" :<: _TTm
      goIn
      giveOutBelow $ N $ P f : $ A (N sTm)
    _ → throwError' $ err $ "apply: last entry in the development"
      ++ " must be a definition with a pi-type."

```

The *ungawa* command looks for a truly obvious thing to do, and does it.

```

ungawa :: ProofState ()
ungawa = ignore done ⊕ ignore apply ⊕ ignore (lambdaParam "ug")
  'pushError' (err "ungawa: no can do.")

```

4.17.3 Refining the proof state

To refine the proof state, we must supply a new goal type and a realiser, which takes values in the new type to values in the original type. The *refineProofState* command creates a new subgoal at the top of the working development, so the entries in that development are not in scope.

```

refineProofState :: INTM → (EXTM → INTM) → ProofState ()
refineProofState ty realiser = do
  cursorTop
  t :=>: _ ← make ("refine" :<: ty)
  cursorBottom
  give (realiser t)
  cursorTop
  cursorDown
  goIn

```

4.18 Resolving and unresolving names

Typographical note: in this section, we write f for a relative name (component) and ε_0 for an absolute name (component).

A `BScopeContext` contains information from the `ProofContext` required for name resolution: a list of the above entries and last component of the current entry's name for each layer, along with the entries in the current development.

```
type BScopeContext = (Bwd (Entries, (String, Int)), Entries)
```

We can extract such a thing from a `ProofContext` using `inBScope`:

```
inBScope :: ProofContext → BScopeContext
inBScope (PC layers dev _) =
  (fmap ( $\lambda l \rightarrow$  (aboveEntries  $l$ , last · currentEntryName · currentEntry $  $l$ )) layers
  , devEntries dev)
```

An `FScopeContext` is the forwards variant:

```
type FScopeContext = (Fwd (Entry Bwd)
  , Fwd ((String, Int), Fwd (Entry Bwd)))
```

We can reverse the former to produce the latter:

```
bToF :: BScopeContext → FScopeContext
bToF (uess :< (es, u), es') =
  let (fs, vfss) = bToF (uess, es) in
  (fs, (u, es' <>> F0) :> vfss)
bToF (B0, es) = (es <>> F0, F0)
```

The `flat` function, up to currying, takes a `BScopeContext` and flattens it to produce a list of entries.

```
flat :: Bwd (Entries, (String, Int)) → Entries → Entries
flat B0 es = es
flat (esus :< (es', _) es = flat esus (es' < + > es)
```

The `flatNom` function produces a name by prepending its second argument with the name components from the backwards list.

```
flatNom :: Bwd (Entries, (String, Int)) → Name → Name
flatNom B0 nom = nom
flatNom (esus :< (_, u) nom = flatNom esus (u : nom)
```

4.18.1 Resolving relative names to references

We need to resolve local longnames as references. We resolve $f.x.y.z$ by searching outwards for f , then inwards for a child x , x 's child y , y 's child z . References are fully λ -lifted, but as f 's parameters are held in common with the point of reference, we automatically supply them.

When in the process of resolving a relative name, we keep track of a `ResolveState`. **question:** **What do the components mean?**

```
type ResolveState = (Either FScopeContext Entries
  , [REF]
  , Maybe REF
  , Maybe (Scheme INTM)
  )
```

The outcome of the process is a `ResolveResult`: a reference, a list of shared parameters to which it should be applied, and a scheme for it (if there is one).

```
type ResolveResult = (REF, [REF], Maybe (Scheme INTM))
```

The `resolveHere` command resolves a relative name to a reference, a spine of shared parameters to which it should be applied, and possibly a scheme. If the name ends with `"/"`, the scheme will be discarded, so all parameters can be provided explicitly. **question: What should the syntax be for this, and where should it be handled?**

```
resolveHere :: RelName → ProofState ResolveResult
resolveHere x = do
  let (x', b) = shouldDiscardScheme x
  uess ← gets inBScope
  (r, s, ms) ← resolve x' uess
  'catchEither' (err $ "resolveHere: cannot resolve name: "
    ++ showRelName x')
  return (r, s, if b then Nothing else ms)
where
  shouldDiscardScheme :: RelName → (RelName, Bool)
  shouldDiscardScheme x = if fst (last x) ≡ "/" then (init x, True)
    else (x, False)
```

The `resolveDiscard` command resolves a relative name to a reference, discarding any shared parameters it should be applied to.

```
resolveDiscard :: RelName → ProofState REF
resolveDiscard x = resolveHere x >>= (λ(r, -, _) → return r)
```

There are four stages relating to whether we are looking up or down (`^` or `_`) and whether or nor we are navigating the part of the proof state which is on the way back to (or from) the root of the tree to the cursor position.

We start off in `resolve`, which calls `lookupUp` (for `^`) or `lookupDown` (for `_`) to find the first name element. Then `lookFor` and `lookFor'` recursively call each other and `lookupDown` until we find the target name, in which case we stop, or we reach the local part of the context, in which case `lookLocal` is called. Finally, `lookLocal` calls `huntLocal` with an appropriate list of entries, so it looks up or down until it finds the target name.

The `resolve` function starts the name resolution process: if the name is a primitive we are done, otherwise it invokes `lookupUp` or `lookupDown` as appropriate then continues with `lookFor`.

```
resolve :: RelName → BScopeContext → Either (StackError t) ResolveResult
resolve [(y, Rel 0)] _
  | Just ref ← lookup y primitives = Right (ref, [], Nothing)
resolve ((x, Rel i) : us) bsc = lookupUp (x, i) bsc (bToF bsc) >>= lookFor us
resolve ((x, Abs i) : us) bsc = lookupDown (x, i) (bToF bsc) [] >>= lookFor us
resolve [] bsc = Left [err "Oh no, the empty name"]
```

```
lookFor :: RelName → ResolveState → Either (StackError t) ResolveResult
lookFor [] (_, sp, Just r, ms) = Right (r, sp, ms)
lookFor [] (Left fsc, sp, Nothing, _) = Left [err "Direct ancestors are not in scope!"]
lookFor us (Left fsc, sp, -, _) = do
  (x, -, z) ← lookFor' us fsc
  return (x, sp, z)
lookFor us (Right es, sp, mr, ms) = lookLocal us es sp mr ms
```

```

lookFor' :: RelName → FScopeContext → Either (StackError t) ResolveResult
lookFor' ((x, Abs i) : us) fsc = lookDown (x, i) fsc [] ≫≧ lookFor us
lookFor' ((x, Rel 0) : us) fsc = lookDown (x, 0) fsc [] ≫≧ lookFor us
lookFor' ((x, Rel i) : us) fsc = Left [err "Yeah, good luck with that"]
lookFor' [] fsc = Left [err "Oh no, the empty name"]

```

```

lookUp :: (String, Int) → BScopeContext → FScopeContext →
  Either (StackError t) ResolveState
lookUp (x, i) (B0, B0) fs = Left [err $ "Not in scope " ++ x]
lookUp (x, i) ((esús :< (es, (y, j))), B0) (fs, vfss) | x ≡ y =
  if i ≡ 0 then Right (Left (fs, vfss), paramREFs (flat esús es), Nothing, Nothing)
  else lookUp (x, i - 1) (esús, es) (F0, ((y, j), fs) :> vfss)
lookUp (x, i) ((esús :< (es, (y, j))), B0) (fs, vfss) =
  lookUp (x, i) (esús, es) (F0, ((y, j), fs) :> vfss)
lookUp (x, i) (esús, es :< e@(EModule n (Dev {devEntries = es'}))) (fs, vfss) | lastNom n ≡ x =
  if i ≡ 0 then Right (Right es', paramREFs (flat esús es), Nothing, Nothing)
  else lookUp (x, i - 1) (esús, es) (e :> fs, vfss)
lookUp (x, i) (esús, es :< e@(EDEF r (y, j) dkind (Dev {devEntries = es'} - _)) (fs, vfss) | x ≡ y =
  if i ≡ 0 then Right (Right es', paramREFs (flat esús es), Just r, entryScheme e)
  else lookUp (x, i - 1) (esús, es) (e :> fs, vfss)
lookUp (x, i) (esús, es :< e@(EPARAM r (y, j) - - -)) (fs, vfss) | x ≡ y =
  if i ≡ 0 then Right (Right B0, [], Just r, Nothing)
  else lookUp (x, i - 1) (esús, es) (e :> fs, vfss)
lookUp u (esús, es :< e) (fs, vfss) = lookUp u (esús, es) (e :> fs, vfss)

```

```

lookDown :: (String, Int) → FScopeContext → [REF] →
  Either (StackError t) ResolveState

```

```

lookDown (x, i) (e :> es, uess) sp =
  if x ≡ (fst $ entryLastName e)
  then if i ≡ 0
    then case (| devEntries (entryDev e) |) of
      Just zs → Right (Right zs, sp, entryRef e, entryScheme e)
      Nothing → Right (Right B0, [], entryRef e, entryScheme e)
    else lookDown (x, i - 1) (es, uess) (pushSpine e sp)
  else lookDown (x, i) (es, uess) (pushSpine e sp)
where
  pushSpine :: Entry Bwd → [REF] → [REF]
  pushSpine (EPARAM r - - -) sp = r : sp
  pushSpine _ sp = sp

```

```

lookDown (x, i) (F0, (((y, j), es) :> uess)) sp =
  if x ≡ y
  then if i ≡ 0
    then Right (Left (es, uess), sp, Nothing, Nothing)
    else lookDown (x, i - 1) (es, uess) sp
  else lookDown (x, i) (es, uess) sp

```

```

lookDown (x, i) (F0, F0) fs = Left [err $ "Not in scope " ++ x]

```

```

lookLocal :: RelName → Entries → [REF] → Maybe REF → Maybe (Scheme INTM) →
  Either (StackError t) ResolveResult
lookLocal ((x, Rel i) : ys) es sp _ = huntLocal (x, i) ys (reverse $ trail es) sp
lookLocal ((x, Abs i) : ys) es sp _ = huntLocal (x, i) ys (trail es) sp
lookLocal [] _ sp (Just r) ms      = Right (r, sp, ms)
lookLocal [] _ _ Nothing _        = Left [err "Modules have no term representation"]

```

```

huntLocal :: (String, Int) → RelName → [Entry Bwd] → [REF] →
  Either (StackError t) ResolveResult
huntLocal (x, i) ys (e : es) as =
  if x ≡ (fst $ entryLastName e)
  then if i ≡ 0
    then case (| devEntries (entryDev e) |) of
      Just zs → lookLocal ys zs as (entryRef e) (entryScheme e)
      Nothing → Left [err "Params in other Devs are not in scope"]
    else huntLocal (x, i - 1) ys es as
  else huntLocal (x, i) ys es as
huntLocal (x, i) ys [] as = Left [err $ "Had to give up looking for " ++ x]

```

4.18.2 Unresolving absolute names to relative names

Just as resolution automatically supplies parameters to references which are actually lifted, so its inverse, *christening*, must hide parameters to lifted references which can be seen locally. For example, here

```

f [
  x : S
  g => t : T
] => g : T

```

g is actually represented as $f.g \ f.x$, but should be displayed as, er, g .

In more detail

Our job is to take a machine name and print as little of it a possible, while at the same time, turning the IANAN representation into a more human friendly, (hah!) relative offset form. Consider:

```

X [ \ a : A
  f [ \ b : B ->
    g [ ] => ? : T
    -= We are here -=
  ] => ? : S
]

```

How should we print the computer name $X_0.f_0.g_0$? A first approximation would be g since this is the bit that differs from the name of the development we are in ($X_0.f_0$). And, indeed we will always have to print this bit of the name. But there's more to it, here we are assuming that we are applying g to the same spine of parameters as the parameters we are currently working under, which isn't always true. We need to be able to refer to, for instance, $f.g$, which would have type $(b : B) \rightarrow T$. So we must really resolve names with their spines compared to the current name and parameters spine. So:

- $X_0.f_0.g_0 \ a \ b$ resolves to g
- $X_0.f_0.g_0 \ a$ resolves to $f.g$

- $X_0.f_0.g_0 \ a \ c$ resolves to $f.g \ c$
- $X_0.f_0.g_0$ resolves to $X.f.g$

The job of naming boils down to unwinding the current name and spine until both are a prefix of the name we want to print, and its spine. We then print the suffix of the name applied to the suffix of the spine. So, far, so simple, but there are complications:

1) The current development is, kind of, in scope

```
X [ \ a : A
  f [ ] => ? : U
  f [ \ b : B ->
    g [ ] => ? : T
    -= We are here ==
  ] => ? : S
]
```

We never want the current development to be in scope, but with this naming scheme, we need to be very careful since $f.g$ is a valid name. Thus we must call $X_0.f_0$ by the name $f\hat{1}$ even though $X_0.f_1$ is not in scope.

2) Counting back to the top When we start looking for the first part of the name we need to print, we can't possibly know what it is, so we can't count how many times it is shadowed (without writing a circular program) This requires us to make two passes through the proof state. If we name $X_0.f_0 \ a$ in the 2nd example above, we must 1st work out the first part of the name is f and then go back out work out how many f 's we jump over to get there.

3) Counting down from the top Consider naming $X_0.f_1.g_0$ with no spine (again in the 2nd example dev) how do we render f_1 . It's my contention that or reference point cannot be where the cursor is, since we've escaped that context, instead we should name it absolutely, counting down from X , so we should print $X.f.1.g$. Note that $f.1$ as a relative name component has a different meaning from f_1 as an absolute name component, and in:

```
X [ \ a : A
  f [ ] => ? : U
  h [ ] => ? : V
  f [ \ b : B ->
    g [ ] => ? : T
    -= We are here ==
  ] => ? : S
]
```

$X_0.f_2.g_0$ also resolves to $X.f.1.g$.

We can split the name into 3 parts:

- the section when the name differs from our current position;
- the section where the name is the same but the spine is different; and
- the section where both are the same.

We must only print the last part of the 1st, and we must print the 2nd absolutely. As far as I remember the naming of these three parts is dealt with by (respectively) *nomTop*, *nomAbs* and *nomRel*.

4) Don't snap your spine Final problem! Consider this dev:

```
x [
  f [ \ a : A ->
      \ b : B ->
      g [ ] => ? : T
      -= We are here -=
    ] => ? : S
]
```

How should we render `x_0.f_0.g_0 a?`. Clearly there is some sharing of the spine with the current position, but we must still print `f.g a` since `f.g` should have type $(a : A) (b : B) \rightarrow T$. Thus we must only compare spines when we unwind a section from the name of the current development.

Here goes...

To *unresolve* an absolute name, we need its reference kind, the spine of arguments to which it is applied, the context in which we are viewing it and a list of entries in local scope. We obtain a relative name, the number of shared parameters to drop, and the scheme of the name (if there is one).

```
unresolve :: Name -> RKind -> Spine {TT} REF -> BScopeContext
           -> Entries -> (RelName, Int, Maybe (Scheme INTM))
unresolve tar rk tas msc@(mesus, mes) les =
```

We first check if the name refers to an element of the *primitives* list:

```
case find ((tar ≡) · refName · snd) primitives of
```

If so, we return its short name with no shared parameters and no scheme.

```
Just (s, _) -> [(s, Rel 0)], 0, Nothing)
```

Otherwise, we actually have to do some work. We work in the Maybe monad and *failNom* will be called if unresolution fails.

```
Nothing -> maybe (failNom tar, 0, Nothing) id $
  case (partNoms tar msc [] B0, rk) of
```

If the reference is a DECL, then it had better be a parameter above, and we do not need to worry about shared parameters. We simply call *nomTop* to find it.

```
(_, DECL) -> do
  (x, ms) ← nomTop tar (mesus, mes < + > les)
  return ([x], 0, ms)
```

```
(Just (xs, Just (top, nom, sp, es)), _) -> do
  let (top', nom', i, fsc) = matchUp (xs :<
      (top, nom, sp, (F0, F0))) tas
      mnom = take (length nom' - length nom) nom'
      (tn, tms) ← nomTop top' (mesus, mes < + > les)
      (an, ams) ← nomAbs mnom fsc
      (rn, rms) ← nomRel nom (es < + > les) Nothing
  in let ms = case (null nom, null mnom) of
      (True, True) -> tms
      (True, False) -> ams
      (False, _) -> rms
  return ((tn : an) ++ rn, i, ms)
```

```

(Just (xs, Nothing), FAKE) → do
  let (top', nom', i, fsc) = matchUp xs tas
      (tn, tms) ← nomTop top' (mesus, mes < + > les)
      (an, ams) ← nomAbs nom' fsc
  return ((tn : an), i, if null nom' then tms else ams)

```

If nothing else matches, we had better give up and go home.

```
_ → Nothing
```

Parting the noms question: Does anyone know what this does?

```

partNoms :: Name → BScopeContext → Name
  → Bwd (Name, Name, Spine { TT } REF, FScopeContext)
  → Maybe (Bwd (Name, Name, Spine { TT } REF, FScopeContext)
    , Maybe (Name, Name, Spine { TT } REF, Entries))
partNoms [] bsc _ xs = Just (xs, Nothing)
partNoms nom@(top : rest) bsc n xs = case partNom n top bsc (F0, F0) of
  Just (sp, Left es) → Just (xs, Just (n ++ [top], rest, sp, es))
  Just (sp, Right fsc) →
    partNoms rest bsc (n ++ [top]) (xs :< (n ++ [top], rest, sp, fsc))
  Nothing → Nothing

```

```

partNom :: Name → (String, Int) → BScopeContext → FScopeContext
  → Maybe (Spine { TT } REF, Either Entries FScopeContext)
partNom hd top ((esús :< (es, top')), B0) fsc | hd ++ [top] ≡ (flatNom esús []) ++ [top'] =
  Just (paramSpine (flat esús es), Right fsc)
partNom hd top ((esús :< (es, ¬)), B0) (js, vfss) =
  partNom hd top (esús, es) (F0, (¬, js) :> vfss)
partNom hd top (esús, es :< EModule n (Dev { devEntries = es' })) fsc | (hd ++ [top]) ≡ n =
  Just (paramSpine (flat esús es), Left es')
partNom hd top (esús, es :< EDEF _ top' _ (Dev { devEntries = es' }) _ _) fsc | hd ++ [top] ≡ (flatNom esús [])
  Just (paramSpine (flat esús es), Left es')
partNom hd top (esús, es :< e) (fs, vfss) = partNom hd top (esús, es) (e :> fs, vfss)
partNom _ _ _ _ = Nothing

```

Matching up If we have a backward list of gibberish and a spine, it is not hard to go back until the spine from the gibberish is a prefix of the given spine, then return the gibberish.

```

matchUp :: Bwd (Name, Name, Spine { TT } REF, FScopeContext)
  → Spine { TT } REF → (Name, Name, Int, FScopeContext)
matchUp (xs :< (x, nom, sp, fsc)) tas
  | sp 'isPrefixOf' tas = (x, nom, length sp, fsc)
matchUp (xs :< _) tas = matchUp xs tas

```

Different name First, `nomTop` handles the section where the name differs from our current position. We call it by its `lastNom` but need to look up the offset and scheme.

```

nomTop :: Name → BScopeContext → Maybe ((String, Offs), Maybe (Scheme INTM))
nomTop n bsc = do
  (i, ms) ← countB 0 n bsc
  return ((lastNom n, Rel i), ms)

```

To determine the relative offset, *nomTop* uses *countB*, which looks backwards through the context, counting the number of things in scope with the same last name component. This also returns the scheme attached, if there is one.

```

countB :: Int → Name → BScopeContext → Maybe (Int, Maybe (Scheme INTM))
countB i n (esus :< (es', u'), B0)
  | last n ≡ u' ∧ flatNom esus [] ≡ init n = (| (i, Nothing) |)
countB i n (esus :< (es', u'), B0)
  | lastNom n ≡ fst u'                                = countB (i + 1) n (esus, es')
countB i n (esus :< (es', u'), B0)                    = countB i n (esus, es')
countB i n (esus, es :< EModule n' (Dev { devEntries = es' }))
  | n ≡ n'                                              = (| (i, Nothing) |)
countB i n (esus, es :< EModule n' _)
  | lastNom n ≡ lastNom n'                            = countB (i + 1) n (esus, es)
countB i n (esus, es :< e@(EEntity r u' _ _ _))
  | last n ≡ u' ∧ refName r ≡ n                        = (| (i, entryScheme e) |)
countB i n (esus, es :< EEntity _ u' _ _ _)
  | lastNom n ≡ fst u'                                = countB (i + 1) n (esus, es)
countB i n (esus, es :< _)                            = countB i n (esus, es)
countB _ n _                                          = Nothing

```

Same name, different spine Next, *nomAbs* handles the section where the name is the same as the current location but the spine is different.

```

nomAbs :: Name → FScopeContext → Maybe (RelName, Maybe (Scheme INTM))
nomAbs [u] (es, (_, es') :> uess) = do
  (v, ms) ← findF 0 u es
  (| ([v], ms) |)
nomAbs ((x, _) : nom) (es, (_, es') :> uess) = do
  (nom', ms) ← nomAbs nom (es', uess)
  case countF x es of
    0 → (| ((x, Rel 0) : nom', ms) |)
    j → (| ((x, Abs j) : nom', ms) |)
nomAbs [] _ = Just ([], Nothing)
nomAbs _ _ = Nothing

```

```

countF :: String → Fwd (Entry Bwd) → Int
countF x F0 = 0
countF x (EModule n _ :> es) | (fst · last $ n) ≡ x = 1 + countF x es
countF x (EEntity _ (y, _) _ _ :> es) | y ≡ x = 1 + countF x es
countF x (_ :> es) = countF x es

```

```

findF :: Int → (String, Int) → Fwd (Entry Bwd)
      → Maybe ((String, Offs), Maybe (Scheme INTM))
findF i u (EModule n _ :> es) | (last $ n) ≡ u =
  Just ((fst u, if i ≡ 0 then Rel 0 else Abs i), Nothing)
findF i u@(x, _) (EModule n _ :> es) | (fst · last $ n) ≡ x = findF (i + 1) u es
findF i u (e@(EDEF _ v dkind _ _ _ :> es) | v ≡ u =
  Just ((fst u, if i ≡ 0 then Rel 0 else Abs i), entryScheme e)
findF i u (EEntity _ v _ _ _ :> es) | v ≡ u =
  Just ((fst u, if i ≡ 0 then Rel 0 else Abs i), Nothing)
findF i u@(x, _) (EEntity _ (y, _) _ _ _ :> es) | y ≡ x = findF (i + 1) u es
findF i u (_ :> es) = findF i u es
findF _ _ _ = Nothing

```

Same name and spine Finally, *nomRel* handles the section where the name and spine both match the current location.

```

nomRel :: Name → Entries
  → Maybe (Scheme INTM) → Maybe (RelName, Maybe (Scheme INTM))
nomRel [] _ ms = (| ([], ms) |)
nomRel (x : nom) es _ = do
  (i, es', ms) ← nomRel' 0 x es
  (nom', ms') ← nomRel nom es' ms
  return ((fst x, Rel i) : nom', ms')

nomRel' :: Int → (String, Int) → Entries
  → Maybe (Int, Entries, Maybe (Scheme INTM))
nomRel' o (x, i) (es :< EModule n (Dev { devEntries = es' })) | (fst · last $ n) ≡ x =
  if i ≡ (snd · last $ n) then (| (o, es', Nothing) |)
  else nomRel' (o + 1) (x, i) es
nomRel' o (x, i) (es :< e@(EDEF _ (y, j) dkind (Dev { devEntries = es' }) _)) | y ≡ x =
  if i ≡ j then (| (o, es', entryScheme e) |) else nomRel' (o + 1) (x, i) es
nomRel' o (x, i) (es :< EEntity _ (y, j) _ _ _) | y ≡ x =
  if i ≡ j then (| (o, B0, Nothing) |) else nomRel' (o + 1) (x, i) es
nomRel' o (x, i) (es :< e) = nomRel' o (x, i) es
nomRel' _ _ _ = Nothing

```

Useful oddments for unresolution

The common *lastNom* function extracts the String component of the last part of a name.

```

lastNom :: Name → String
lastNom = fst · last

```

The *failNom* function is used to give up and convert an absolute name that cannot be unresoloved into a relative name. This can happen when distilling erroneous terms, which may not be well-scoped.

```

failNom :: Name → RelName
failNom nom = ("!!!", Rel 0) : (map (λ(a, b) → (a, Abs b)) nom)

```

Invoking unresolution

The *christenName* and *christenREF* functions call *unresolve* for names, and the name part of references, respectively.

```

christenName :: BScopeContext → Name → RKind → RelName
christenName bsc target rk = s
  where (s, _, _) = unresolve target rk (paramSpine · (uncurry flat) $ bsc) bsc B0
christenREF :: BScopeContext → REF → RelName
christenREF bsc (target := rk :<: _) = christenName bsc target rk

```

The *showEntries* function folds over a bunch of entries, christening them with the given entries in scope and current name, and intercalating to produce a comma-separated list.

```

showEntries :: (Traversable f, Traversable g) ⇒ BScopeContext → f (Entry g) → String
showEntries bsc = intercalate ", " · foldMap f
  where
    f e | Just r ← entryRef e = [showRelName (christenREF bsc r)]
        | otherwise           = []

```

The `showEntriesAbs` function works similarly, but uses absolute names instead of christening them.

```
showEntriesAbs :: Traversable f => f (Entry f) -> String
showEntriesAbs = intercalate " , " · foldMap f
  where
    f e = [showName (entryName e)]
```

4.19 Lambda-lifting and discharging

Pierre: I think that, after some clean-up, the following could well be moved in `ProofState.Edition.Scope`

In the following, we define 4 useful functions manipulating terms in a context of entries. These functions provide the basic toolkit for operations like lambda-lifting, or the manipulation of the proof state. Therefore, this section has to be read with the tired eye of the implementer.

4.19.1 Discharging entries in a term

The “discharge into” operator `(-|)` takes a list of entries and a term, and changes the term so that parameters in the list of entries are represented by de Bruijn indices. It makes key use of the `(-||)` mangler.

```
(-|) :: Entries -> INTM -> INTM
es -| tm = (bwdList $ paramREFs es) -|| tm
```

4.19.2 Binding a term

The `parBind` function λ -binds a term over a list Δ of entries and λ - and Π -binds over a list ∇ of entries.

```
parBind :: {-Δ :: -} Bwd (Entry Bwd) {-Γ -} ->
          {-∇ :: -} Bwd (Entry Bwd) {-Γ, Δ -} ->
          INTM {-Γ, Δ, ∇ -} ->
          INTM {-Γ -}
parBind delta nabla t = help delnab nabla (delnab -| t) where
  delnab = delta < + > nabla
  help B0                                B0          t = t
  help (delta <: EPARAM _ (x, _) _ _ _) B0          t =
    help delta B0 (L (x : . t))
  help (delta <: _)                        B0          t =
    help delta B0 t
  help (delnab <: EPARAM _ (x, _) ParamLam _ _) (nabla <: _) t =
    help delnab nabla (L (x : . t))
  help (delnab <: EPARAM _ (x, _) ParamAll _ _) (nabla <: _) t =
    help delnab nabla (L (x : . t))
  help (delnab <: EPARAM _ (x, _) ParamPi s _) (nabla <: _) t =
    help delnab nabla (PI (delnab -| s) (L (x : . t)))
  help (delnab <: _)                        (nabla <: _) t =
    help delnab nabla t
```

4.19.3 Binding a type

The `liftType` function Π -binds a type over a list of entries.

```

liftType :: Entries → INTM → INTM
liftType es = liftType' (bwdList $ foldMap param es)
  where param (EPARAM r _ _ t _) = [r :<: t]
        param _ = []

```

```

liftType' :: Bwd (REF :<: INTM) → INTM → INTM
liftType' rtys t = pis rs tys (rs -|| t)
  where
    (rs, tys) = unzipBwd rtys
    unzipBwd B0 = (B0, B0)
    unzipBwd (rtys :< (r :<: ty)) = (rs :< r, tys :< ty)
    where (rs, tys) = unzipBwd rtys

```

```

pis B0 B0 t = t
pis (rs :< r) (tys :< ty) t = pis rs tys (PI (rs -|| ty) (L ((fst · last · refName $ r) : . t)))

```

4.19.4 Making a type out of a goal

The *inferGoalType* function Π -binds the type when it encounters a λ in the list of entries, and produces SET when it encounters a Π .

```

inferGoalType :: Bwd (Entry Bwd) → INTM → INTM
inferGoalType B0 t = t
inferGoalType (es :< EPARAM _ (x, _) ParamLam s _) t =
  inferGoalType es (PI (es -| s) (L (x : . t)))
inferGoalType (es :< EPARAM _ (x, _) ParamAll s _) (PRF t) =
  inferGoalType es (PRF (ALL (es -| s) (L (x : . t))))
inferGoalType (es :< EPARAM _ (x, _) ParamPi s _) SET =
  inferGoalType es SET
inferGoalType (es :< _) t =
  inferGoalType es t

```

4.20 Anchor resolution

```

isAnchor :: Traversable f ⇒ Entry f → Bool
isAnchor (EEntity _ _ _ (Just _)) = True
isAnchor _ = False

```

```

anchorsInScope :: ProofState Entries
anchorsInScope = do
  scope ← getInScope
  return $ foldMap anchors scope
  where anchors t | isAnchor t = B0 :<: t
        | otherwise = B0

```

To cope with shadowing, we will need some form of RelativeAnchor:

```

type RelativeAnchor = (Anchor, Int)

```

With shadowing punished by De Bruijn. Meanwhile, let's keep it simple.

```

resolveAnchor :: String → ProofStateT e (Maybe REF)
resolveAnchor anchor = do
  scope ← getInScope
  case seekAnchor scope of
    B0 → return $ Nothing
    _ :< ref → return $ Just ref
  where seekAnchor :: Entries → Bwd REF
    seekAnchor B0 = ()
    seekAnchor (scope :< EPARAM ref _ _ _ (Just anchor'))
      | anchor' ≡ anchor = B0 :< ref
    seekAnchor (scope :< EPARAM ref _ _ _ Nothing) = seekAnchor scope
    seekAnchor (scope :< EDEF ref _ _ dev _ (Just anchor'))
      | anchor' ≡ anchor = B0 :< ref
    seekAnchor (scope :< EDEF ref _ _ dev _ Nothing) =
      seekAnchor (devEntries dev)
      < + > seekAnchor scope
    seekAnchor (scope :< EModule _ dev) =
      seekAnchor (devEntries dev)
      < + > seekAnchor scope

```

Find the entry corresponding to the given anchor:

```

findAnchor :: String → ProofState ()
findAnchor = ⊥

```

Redefine the entry corresponding from the given anchor, so that's name is the second anchor:

```

renameAnchor :: String → String → ProofState ()
renameAnchor = ⊥

```

Chapter 5

The Proof Tactics

5.1 Presenting Information

```
infoInScope :: ProofState String
infoInScope = do
  pc ← get
  inScope ← getInScope
  return (showEntries (inBScope pc) inScope)
```

```
infoDump :: ProofState String
infoDump = gets show
```

The *infoElaborate* command calls *elabInfer* on the given neutral display term, evaluates the resulting term, bquotes it and returns a pretty-printed string representation. Note that it works in its own module which it discards at the end, so it will not leave any subgoals lying around in the proof state.

```
infoElaborate :: DEXtmRN → ProofState String
infoElaborate tm = draftModule "__infoElaborate" (do
  (tm' :=>: tmv :<: ty) ← elabInfer' tm
  tm'' ← bquoteHere tmv
  s ← prettyHere (ty :>: tm'')
  return (renderHouseStyle s)
)
```

The *infoInfer* command is similar to *infoElaborate*, but it returns a string representation of the resulting type.

```
infoInfer :: DEXtmRN → ProofState String
infoInfer tm = draftModule "__infoInfer" (do
  (_ :<: ty) ← elabInfer' tm
  ty' ← bquoteHere ty
  s ← prettyHere (SET :>: ty')
  return (renderHouseStyle s)
)
```

The *infoContextual* command displays a distilled list of things in the context, parameters if the argument is False or definitions if the argument is True.

```
infoHypotheses = infoContextual False
infoContext    = infoContextual True
```

```

infoContextual :: Bool → ProofState String
infoContextual gals = do
  inScope ← getInScope
  bsc ← gets inBScope
  d ← help bsc inScope
  return (renderHouseStyle d)
where
  help :: BScopeContext → Entries → ProofState Doc
  help bsc B0 = return empty
  help bsc (es :< EPARAM ref _ k _ _) | ¬ gals = do
    ty ← bquoteHere (pty ref)
    docTy ← prettyHereAt (pred ArrSize) (SET :>: ty)
    d ← help bsc es
    return $ d $$ prettyBKind k (text (showRelName (christenREF bsc ref))
      < + > kword KwAsc < + > docTy)
  help bsc (es :< EDEF ref _ _ _ _ _) | gals = do
    ty ← bquoteHere $ removeShared (paramSpine es) (pty ref)
    docTy ← prettyHere (SET :>: ty)
    d ← help bsc es
    return $ d $$ (text (showRelName (christenREF bsc ref))
      < + > kword KwAsc < + > docTy)
  help bsc (es :< _) = help bsc es

removeShared :: Spine {TT} REF → TY → TY
removeShared [] ty = ty
removeShared (A (NP r) : as) (PI s t) = t Evidences.Eval.$$ A (NP r)

```

This old implementation is written using a horrible imperative hack that saves the state, throws away bits of the context to produce an answer, then restores the saved state. We can get rid of it once we are confident that the new version (above) produces suitable output.

```

infoContextual' :: Bool → ProofState String
infoContextual' gals = do
  save ← get
  let bsc = inBScope save
      me ← getCurrentName
      ds ← many (hypsHere bsc me < * optional killBelow < * goOut < * removeEntryAbove)
      d ← hypsHere bsc me
      put save
  return (renderHouseStyle (vcat (d : reverse ds)))
where
  hypsHere :: BScopeContext → Name → ProofState Doc
  hypsHere bsc me = do
    es ← getEntriesAbove
    d ← hyps bsc me
    putEntriesAbove es
    return d

  killBelow = do
    l ← getLayer
    replaceLayer (l { belowEntries = NF F0 })

  hyps :: BScopeContext → Name → ProofState Doc
  hyps bsc me = do
    es ← getEntriesAbove
    case (gals, es) of
      (_, B0) → return empty
      (False, es' :< EPARAM ref _ k _ _) → do
        putEntriesAbove es'
        ty' ← bquoteHere (pty ref)
        docTy ← prettyHere (SET :>: ty')
        d ← hyps bsc me
        return (d $$ prettyBKind k (text (showRelName (christenREF bsc ref)) < + > kword KwAsc < + >))
      (True, es' :< EDEF ref _ _ _ _) → do
        goIn
        es ← getEntriesAbove
        (ty :=>: _) ← getGoal "hyps"
        ty' ← bquoteHere (evTm (inferGoalType es ty))
        docTy ← prettyHere (SET :>: ty')
        goOut
        putEntriesAbove es'
        d ← hyps bsc me
        return (d $$ (text (showRelName (christenREF bsc ref)) < + > kword KwAsc < + > docTy))
      (_, es' :< _) → putEntriesAbove es' >> hyps bsc me

infoScheme :: RelName → ProofState String
infoScheme x = do
  (_, as, ms) ← resolveHere x
  case ms of
    Just sch → do
      d ← prettySchemeHere (applyScheme sch as)
      return (renderHouseStyle d)
    Nothing → return (showRelName x ++ " does not have a scheme.")

```

The *infoWhatIs* command displays a term in various representations.

```

infoWhatIs :: DExTmRN → ProofState String
infoWhatIs tmd = draftModule "__infoWhatIs" (do
  (tm :=>: tmv <: tyv) ← elabInfer' tmd
  tmq ← bquoteHere tmv
  tms :=>: _ ← distillHere (tyv :=>: tmq)
  ty ← bquoteHere tyv
  tys :=>: _ ← distillHere (SET :=>: ty)
  return (unlines
    [ "Parsed term:", show tmd
      , "Elaborated term:", show tm
      , "Quoted term:", show tmq
      , "Distilled term:", show tms
      , "Pretty-printed term:", renderHouseStyle (pretty tms maxBound)
      , "Inferred type:", show tyv
      , "Quoted type:", show ty
      , "Distilled type:", show tys
      , "Pretty-printed type:", renderHouseStyle (pretty tys maxBound)
    ])
  )

```

The *prettyProofState* command generates a pretty-printed representation of the proof state at the current location.

```

prettyProofState :: ProofState String
prettyProofState = do
  inScope ← getInScope
  me ← getCurrentName
  d ← prettyPS inScope me
  return (renderHouseStyle d)

prettyPS :: Entries → Name → ProofState Doc
prettyPS aus me = do
  es ← replaceEntriesAbove B0
  cs ← putBelowCursor F0
  case (es, cs) of
    (B0, F0) → prettyEmptyTip
  - → do
    d ← prettyEs empty (es <>> F0)
    d' ← case cs of
      F0 → return d
      - → do
        d'' ← prettyEs empty cs
        return (d $$ text "----" $$ d'')
    tip ← prettyTip
    putEntriesAbove es
    putBelowCursor cs
    return (lbrack < + > d' $$ rbrack < + > tip)

where
prettyEs :: Doc → Fwd (Entry Bwd) → ProofState Doc
prettyEs d F0 = return d
prettyEs d (e := es) = do
  putEntryAbove e
  ed ← prettyE e
  prettyEs (d $$ ed) es

prettyE (EPARAM (_ := DECL <: ty) (x, _) k _ anchor) = do
  ty' ← bquoteHere ty
  tyd ← prettyHereAt (pred ArrSize) (SET :=: ty')
  return (prettyBKind k
    (text x < + > (maybe empty (brackets · brackets · text) anchor)
      < + > kword KwAsc
      < + > tyd))

prettyE e = do
  goIn
  d ← prettyPS aus me
  goOut
  return (sep [ text (fst (entryLastName e))
    < + > (maybe empty (brackets · brackets · text) $ entryAnchor e)
    , nest 2 d < + > kword KwSemi
  ])

prettyEmptyTip :: ProofState Doc
prettyEmptyTip = do
  tip ← getDevTip
  case tip of
    Module → return (brackets empty)
  - → do
    tip ← prettyTip
    return (kword KwDefn < + > tip)

```

```

prettyTip :: ProofState Doc
prettyTip = do
  tip ← getDevTip
  case tip of
    Module → return empty
    Unknown (ty :=>: _) → do
      hk ← getHoleKind
      tyd ← prettyHere (SET :=>: ty)
      return (prettyHKind hk < + > kword KwAsc < + > tyd)
    Suspended (ty :=>: _) prob → do
      hk ← getHoleKind
      tyd ← prettyHere (SET :=>: ty)
      return (text (" (SUSPENDED: " ++ show prob ++ " ")
        < + > prettyHKind hk < + > kword KwAsc < + > tyd)
    Defined tm (ty :=>: tyv) → do
      tyd ← prettyHere (SET :=>: ty)
      tmd ← prettyHereAt (pred ArrSize) (tyv :=>: tm)
      return (tmd < + > kword KwAsc < + > tyd)

```

```

prettyHKind :: HKind → Doc
prettyHKind Waiting = text "?"
prettyHKind Hoping = text "HOPE?"
prettyHKind (Crying s) = text ("CRY <<" ++ s ++ ">>")

```

The *elm* Cochon tactic elaborates a term, then starts the scheduler to stabilise the proof state, and returns a pretty-printed representation of the final type-term pair (using a quick hack).

```

elmCT :: DExTmRN → ProofState String
elmCT tm = do
  suspend ("elab" <: sigSetTM :=>: sigSetVAL) (ElabInferProb tm)
  startScheduler
  infoElaborate (DP [("elab", Rel 0)] :$[])

```

import → CochonTactics **where**

: *unaryExCT* "elm" *elmCT* "elm <term> - elaborate <term>, stabilise and print type-

: *unaryExCT* "elaborate" *infoElaborate*

"elaborate <term> - elaborates, evaluates, quotes, distills and pretty-prints <

: *unaryExCT* "infer" *infoInfer*

"infer <term> - elaborates <term> and infers its type."

: *unaryInCT* "parse" (*return* · *show*)

"parse <term> - parses <term> and displays the internal display-sytnax represen

: *unaryNameCT* "scheme" *infoScheme*

"scheme <name> - looks up the scheme on the definition <name>."

```

: unaryStringCT "show" ( $\lambda s \rightarrow$  case  $s$  of
  "inscope"  $\rightarrow$  infoInScope
  "context"  $\rightarrow$  infoContext
  "dump"      $\rightarrow$  infoDump
  "hyyps"     $\rightarrow$  infoHypotheses
  "state"     $\rightarrow$  prettyProofState
  _           $\rightarrow$  return "show: please specify exactly what to show."
)
"show <inscope/context/dump/hyyps/state> - displays useless information."

: unaryExCT "whatis" infoWhatIs
  "whatis <term> - prints the various representations of <term>."

```

5.2 Elimination with a Motive

Elimination with a motive works on a goal prepared *by the user* in the form

$$\Gamma, \Delta \vdash ? : T$$

where Γ is the list of external hypotheses, Δ is the list of internal hypotheses and T is the goal to solve. The difference between external and internal hypotheses is the following. External hypotheses scope over the whole problem, that is the current goal and later sub-goals. They are the “same” in all subgoals. On the other hand, internal hypotheses are consumed by the eliminator, hence are “different” in all subgoals.

We need a way to identify where to divide the context into Γ and Δ . One way to do that is to ask the user to pass in the reference to the first term of Δ in the context. If the user provides no reference, we will assume that Γ is empty, so the hypotheses are all internal.

Obviously, we also need to be provided the eliminator we want to use. We expect the user to provide the eliminator for us in the form

$$\Gamma, \Delta \vdash e : (P : \Xi \rightarrow \text{SET}) \rightarrow \vec{m} \rightarrow P\vec{t}$$

where we call P the *motive* of the elimination, Ξ the *indices*, \vec{m} the *methods*, \vec{t} the *targets* and $P\vec{t}$ the *return type*.

We will define *elim* this way:

```

elim :: Maybe REF  $\rightarrow$  (TY  $\rightarrow$  INTM)  $\rightarrow$  ProofState ()
elim comma eliminator = (...)

```

5.2.1 Analyzing the eliminator

Presented as a development, *elim* is called in the context

$$[\begin{array}{l} (\Gamma) \rightarrow \\ (\Delta) \rightarrow \\] := ? : T$$

where Γ and Δ are introduced and T is waiting to be solved.

We have to analyze the eliminator we are given for two reasons. First, we need to check that it *is* an eliminator, that is:

- it has a motive,
- it has a bunch of methods, and
- the target consists of the motive applied to some arguments.

Second, we have to extract some vital pieces of information from the eliminator, namely:

- the type $\Xi \rightarrow \text{SET}$ of the motive, and
- the targets \vec{t} applied to the motive.

To analyze the eliminator, we play a nice game. One option would be to jump in a *draftModule*, introduce *lambdaParams*, then retrieve and check the different parts as we go along. However, this would mean that the terms would be built from references that would become invalid when the draft module was dropped. Therefore, we would suffer the burden (and danger) of manually renaming those references.

The way we actually proceed is the following. The trick consists of rebuilding the eliminator in the current development:

```
[ (Γ) →
  (Δ) →
  makeE [ P := ? : Ξ → Set
          m1 := ? : M1 P
          (...)
          mn := ? : Mn P
        ] := e P m̄ : P t̄
] := ? : T
```

We will build the motive *in-place*, instead of analyzing the eliminator *and then* making the motive. Moreover, we are able to safely check the shape of the eliminator and extract the interesting bits.

The *introElim* command takes the eliminator with its type, which should be a function whose domain is the motive type and whose range contains the methods and return type. It creates a new definition for the rebuilt eliminator, with subgoals for the motive and methods. It returns the name of the rebuilt eliminator, the motive type and the list of targets; the cursor is left on the motive subgoal.

```
introElim :: (TY :> EXTM) → ProofState (Name, TY, Bwd INTM)
introElim (P1 motiveType telType :> elim) = do
```

Make a module, which we will convert to a goal of type $P\vec{t}$ later:

```
elimName ← makeModule "makeE"
goIn
```

Make a goal for the motive:

```
motiveTypeTm ← bquoteHere motiveType
-- telTypeTm j- bquoteHere telType
motive :=> motiveVal ← make $ "motive" :<: motiveTypeTm
```

Make goals for the methods and find the return type:

```
(methods, returnType) ← makeMethods B0 (telType $$ A motiveVal)
```

Check motive and grab the targets (the terms that are applied to it):

```
targets ← checkTargets returnType motiveType (motiveVal :<: motiveType)
```

Convert the module to a goal, solve it (using the subgoals created above) and go to the next subproblem, i.e. making the motive P :

```

moduleToGoal returnType
give $ N $ elim : $ A (N motive) $## methods
goIn
goTop
return (elimName, motiveType, targets)

```

If the eliminator is not a function from motive and methods to return type, there is nothing we can do:

```

introElim _ = throwError' $ err "elimination: eliminator not a pi-type!"

```

Making the methods

Above, we have used `makeMethods` to introduce the methods and retrieve the return type of the eliminator. Remember that the eliminator is a telescope of Π -types. To get the type of the motive, we have matched the first component of the telescope. To get the methods, we simply iterate that process, up to the point where all the Π s have been consumed.

```

makeMethods :: Bwd INTM → TY → ProofState (Bwd INTM, INTM)
makeMethods ms (PI s t) = do
  sTm ← bquoteHere s
  m :=>: mv ← make $ "method" :<: sTm
  makeMethods (ms :< N m) (t $$ A mv)
makeMethods ms target = do
  targetTm ← bquoteHere target
  return (ms, targetTm)

```

Checking the motive and targets

The `checkTargets` command verifies that the motive is of type $\Xi \rightarrow Set$ and that the return type is the motive applied to some arguments (the targets), which are returned.

```

checkTargets :: INTM → VAL → (VAL :<: VAL) → ProofState (Bwd INTM)
checkTargets returnType SET (motive :<: motiveType) = do
  isEqual ← withNSupply $ equal (motiveType :=>: (evTm returnType, motive))
  if isEqual
  then return B0
  else throwError' $ err "elimination: return type does not use motive!"
checkTargets (N (f : $ A x)) (PI s t) (motive :<: motiveType) =
  freshRef ("s" :<: s) $ λs → do
    xs ← checkTargets (N f) (t $$ (A $ pval s)) (motive :<: motiveType)
    return (xs :< x)
checkTargets _ _ _ = throwError' $ err "elimination: your motive is suspicious!"

```

Pierre: There are also some conditions on the variables that can be used in these terms! I have to look that up too. This is a matter of traversing the terms to collect the REFS in them and check that they are of the right domains. **question:** Do these conditions actually matter?

5.2.2 Identifying the motive

The `introElim` command has generated a subgoal for the motive and left us inside it. That's a good thing because this is what we are going to do.

Overview

Now the question is: what term are we supposed to build? To answer that question, the best is still to read the Sanctified Papers [13, 10]. If you can bear with me, here is my strawman explanation.

First try: life's too short, the paper's too long. Remember that the eliminator will evaluate to the following term:

$$P\vec{t}$$

So, we could define P as:

$$P \equiv \lambda(\Xi).T$$

Which trivially solves the goal. However, we will have a hard time making the methods! Indeed, we are asking to solve exactly the same problem, with the same knowledge.

Second try: learning from the targets. We need to hand back some knowledge to the user in the methods. Where could any knowledge come from? Well, we are sure of one thing: (Ξ) will be instantiated to \vec{t} . So, we could define our motive as:

$$P \equiv \lambda(\Xi).(\Xi) == \vec{t} \Rightarrow T$$

Hence, the methods will have some additional knowledge, presented as constraints on the value that their arguments can take.

Third try: the problem with inductive eliminators. This definition would work for the "non-inductive" eliminators, that is eliminators for which the method types are all of the following form:

$$MP \equiv \Pi\Delta_S \rightarrow P\vec{s}$$

These eliminators are simply doing a case analysis, without referring to an induction principle.

However, for "inductive" eliminators, this motive is too weak. An eliminator will be inductive-ish if a method is using the motive anywhere else than as a target. If this is the case, we would like to be able to appeal to some induction principle: we need the freedom to apply the motive in another context than the fixed context Δ . We can get this by abstracting Δ in P , without forgetting to bring \vec{t} and T under this new context:

$$P \equiv \lambda(\Xi).\Pi(\Delta').(\Xi) == \vec{t}[\Delta'/\Delta] \Rightarrow T[\Delta'/\Delta]$$

Fourth try: being less parametric. However, there is a slight problem here. The motive and the methods are very likely to be parameterized over Δ . If we use the motive above, we are asking for trouble in the definition of the methods. Remember that a method typically looks like:

$$MP \equiv \Pi\Delta_S \rightarrow P\vec{s}$$

This will therefore compute to:

$$\Pi\Delta_S\Pi\Delta \Rightarrow \vec{s} == \vec{t}[\Delta'/\Delta] \Rightarrow T[\Delta'/\Delta]$$

The effect is therefore that this type becomes more parametric than it was before, for no good reason. Indeed, there is no way a method can take advantage of this parametricity: it will be used as a boring parameter (because it *is* just a parameter).

So, we need to chunk Δ in two contexts:

- Δ_0 , containing terms involved in the type of the methods and motive, and their respective dependencies; and
- Δ_1 , containing the free terms

Instead of brutally abstracting over Δ , we subtly abstract over Δ_1 :

$$P \equiv \lambda(\Xi).\Pi(\Delta'_1).(\Xi) == \bar{t}[\Delta'_1/\Delta_1] \Rightarrow T[\Delta'_1/\Delta_1]$$

Fifth try: simplifying equations. Although the previous version would be morally correct, the user will appreciate if we simplify some trivial equations by directly instantiating them. Our motto being “the user is king” (don’t quote me on that), let’s simplify.

Hence, if we have $\xi_i : \Xi_i == t_i : T_i$ with $\Xi_i == T_i$ and t_i a term defined in the context Δ'_1 , then we can replace t_i by ξ_i everywhere it appears, in the telescope and in the goal. Having replaced t_i , we can remove the corresponding abstraction in the telescope $\Pi(\Delta'_1)$.

As one would expect, this process must be carried from the left to the right, as equations might simplify further down the line. However, there is a subtlety in that, when discovering a simplification, one must also rename the previously discovered constraints, as they might mention the simplified term.

Pierre: We are simplifying references here but we should do more. We are making a (syntactic) distinction which is finer than what the theory (semantically) describes. There is something in the pipe but I don’t understand it in the general case. Finishing that work will consist in extending *matchRef* (defined later) with the relevant cases.

This concludes our overview. Now, we have to implement the last proposal. That’s not for the faint of heart.

Finding parametric hypotheses

We have the internal context Δ lying around. We also have access to the type of the eliminator, that is the type of the motive and the methods. Therefore, we can already split Δ into its parametric and non-parametric parts, Δ_0 and Δ_1 . As we are not interested in Δ_0 , we fearlessly throw it away.

We aim to implement the *findNonParametricHyps* command, which takes the context Δ as a list of references with their types in term form and the type of the eliminator, and returns the filtered context Δ_1 . The initial dependencies are those of the motive and methods.

```

findNonParametricHyps :: Bwd (REF :<: INTM) → TY
  → ProofState (Bwd (REF :<: INTM), Bwd (REF :<: INTM))
findNonParametricHyps delta elimTy = do
  argTypes ← unfoldTelescope elimTy
  let deps = foldMap collectRefs argTypes
      removeDependencies delta deps

```



Note that we have been careful in asking for *elimTy* here, the type of the eliminator. One might have thought of getting the type of the motive and methods during *introElim*, and using those. That would not work: the motive is defined under the scope of Δ , so its lambda-lifted form includes Δ . Hence, the type of the methods are defined in terms of the motive. Hence, all Δ is innocently included into these types, making them useless. \square

The real solution is to go back to the type of the eliminator. We unfold it with fresh references. Doing so, we are ensured that there is no pollution, and we get what we asked for.

```

unfoldTelescope :: TY → ProofState [INTM]
unfoldTelescope (PI _S _T) = do
  _Stm ← bquoteHere _S
  freshRef ("unfoldTelescope" :<: _S) $ λs → do
    t ← unfoldTelescope (_T $$ (A $ pval s))
    return $ _Stm : t
unfoldTelescope _ = return []

```

The dependencies can be extracted from terms in INTM form using the following helper function:

```

collectRefs :: INTM → [REF]
collectRefs = foldMap (λx → [x])

```

Now, we are left with implementing *removeDependencies*. In the case where $r \in \Delta$ belongs to the dependency set, we exclude it from Δ_1 . We add the references in the type of r to the dependency set, then continue. If r is not in the dependency set, we continue and add r to Δ_1 .

```

removeDependencies :: Bwd (REF :<: INTM) → [REF]
→ ProofState (Bwd (REF :<: INTM), Bwd (REF :<: INTM))
removeDependencies (rs :<: (r :<: rTy)) deps
| r ∈ deps = do
  (delta0, delta1) ← removeDependencies rs (deps ‘union‘ collectRefs rTy)
  return (delta0 :<: (r :<: rTy), delta1)
| otherwise = do
  (delta0, delta1) ← removeDependencies rs deps
  return (delta0, delta1 :<: (r :<: rTy))
removeDependencies B0 deps = return (B0, B0)

```

Finding removable hypotheses

We need to do something like this to find removable hypotheses (those about which we gain no information and hence might as well not abstract over). However, the problem is more complex than just dependency analysis, because of labelled types. The *shouldKeep* function doesn't work properly and should be replaced with a proper type-directed traversal for this to make sense.

```

findNonRemovableHyps :: Bwd (REF :<: INTM) → INTM → Bwd INTM → Bwd (REF :<: INTM)
findNonRemovableHyps delta goal targets = help delta []

```

```

where
  deps :: [REF]
  deps = collectRefs goal ++ foldMap collectRefs targets

```

```

help :: Bwd (REF :<: INTM) → [REF :<: INTM] → Bwd (REF :<: INTM)
help B0 xs = bwdList xs
help (delta :<: (r :<: ty)) xs = help delta
  (if (r ∈ deps) ∨ shouldKeep ty
   then (r :<: ty) : xs else xs)

```

```

shouldKeep :: Tm { d, TT } REF → Bool
shouldKeep (LABEL _ _) = True
shouldKeep (C c) = Data.Foldable.any shouldKeep c
shouldKeep (L (_: . t)) = shouldKeep t
shouldKeep (L (H _ _ t)) = shouldKeep t
shouldKeep (L (K t)) = shouldKeep t
shouldKeep (N t) = shouldKeep t
shouldKeep (t :? _) = shouldKeep t
shouldKeep (t :$ A u) = shouldKeep t ∨ shouldKeep u
shouldKeep (_ :@ ts) = Data.Foldable.any shouldKeep ts
shouldKeep _ = False

```

Representing the context as Binders

As we have seen, simplifying the motive will involve considering the non-parametric context Δ_1 and, as we go along, remove some of its components. We will work with Δ_1 in the following form. A Binder is a reference with the INTM representation of its type, and a corresponding argument term that will be used when applying the motive.

```

type Binder = (REF :<: INTM, INTM)

```

Note that binders are DECL references which are copied from the original context and modified. This is not really a problem: remember that they are an imitative fiction. Once we have found which binders we keep, we will discharge them over the goal type to produce the motive.

We can get a Binder from a typed reference by taking the reference itself as the second component:

```

toBinder :: (REF :<: INTM) → Binder
toBinder (r :<: t) = (r :<: t, NP r)

```

Extracting an element of Δ_1

Recall that, during simplification, we need to identify references belonging to the context Δ_1 and remove the corresponding Π in the simplified context Δ'_1 . However, in ProofState, we cannot really remove a Π once it has been made. Therefore, we delay the making of the Π s until we precisely know which ones are needed. Meanwhile, to carry out our analysis, we directly manipulate the binders computed from Δ_1 .

To symbolically remove a Π , we remove the corresponding Binder. When simplification ends, we simply introduce the Π s corresponding to the remaining binders.

Let us implement the “search and symbolic removal” operation *lookupBinders*: we are given a reference, which may or may not belong to the binders. If the reference belongs to the binders, we return the binders before it, and the binders after it (which might depend on it); if not, we return Nothing.

```

lookupBinders :: REF → Bwd Binder → Maybe (Bwd Binder, Fwd Binder)
lookupBinders p binders = help binders F0
  where
    help :: Bwd Binder → Fwd Binder → Maybe (Bwd Binder, Fwd Binder)
    help (binders :< b@(y :<: -, -)) zs
      | p ≡ y      = Just (binders, zs)
      | otherwise = help binders (b :> zs)
    help B0 _    = Nothing

```

Renaming references

As we have seen, we will need to carry a fair amount of renaming. A renaming operation consists in replacing some references by some other references, in a term in INTM form. In such a case, renaming is simply a matter of *fmap* over the term.

```
renameTM :: [(REF, REF)] → INTM → INTM
renameTM us = fmap (λr → maybe r id (lookup r us))
```

When renaming a forwards list of binders, we need to update the types of the corresponding references and update those references in later binders. Note that we never update the argument term (the second component of the binder) as it needs to remain in the scope of the original context.

```
renameBinders :: [(REF, REF)] → Bwd Binder → Fwd Binder
→ (Bwd Binder, [(REF, REF)])
renameBinders us bs ((y'@(n := DECL :< _):<: yt', a) :> xs) = do
  let yt'' = renameTM us yt'
      y'' = n := DECL :<: evTm yt''
      us' = (y', y'') : us
      renameBinders us' (bs :< (y'' :<: yt''), a)) xs
  renameBinders us bs F0 = (bs, us)
```

Representing equational constraints

Let us sum-up where we are in the construction of the motive. We are sitting in some internal context Δ . We have segregated this context in two parts, keeping only Δ_1 , the non-parametric hypotheses. Moreover, we have turned Δ_1 into a list of binders. Our mission here is to add a bunch of equational constraints to the binders, simplifying them wherever possible.

A Constraint represents an equation between a reference (in the indices Ξ) with its type, and a target in \vec{t} with its type.

```
type Constraint = (REF :<: INTM, (INTM : ~ >: INTM) :<: (INTM : ~ >: INTM))
```

We will be renaming references when we solve constraints, but we need to keep track of the original terms (without renaming) for use when constructing arguments to the eliminator (the second component of the binders, which are in the scope of the original context). We use the type $a : \sim >: b$ for a pair in which a is not updated and b is updated.

```
data a : ~ >: b = a : ~ >: b
  deriving (Eq, Show)
```

Note that there is no need to rename the left-hand sides of constraints, since they are fresh references that do not depend on the binders. Hence we can implement *renameConstraints* to apply a list of updates to the right-hand sides

```
renameConstraints :: [(REF, REF)] → Bwd Constraint → Fwd Constraint
→ Bwd Constraint
renameConstraints us bs ((yvt, (s : ~ >: s') :<: (st : ~ >: st')) :> xs) = do
  let s'' = renameTM us s'
      st'' = renameTM us st'
      renameConstraints us (bs :< (yvt, (s : ~ >: s'') :<: (st : ~ >: st''))) xs
  renameConstraints us bs F0 = bs
```

Acquiring constraints

The *introMotive* command starts with two copies of the motive type and a list of targets. It must be called inside the goal for the motive. It unfolds the types in parallel, introducing fresh *lambdaParams* on the left and working through the targets on the right; as it does so, it accumulates constraints between the introduced references (in Ξ) and the targets. It also returns the number of extra definitions created when simplifying the motive (e.g. splitting sigmas).

```
introMotive :: TY → TY → [INTM] → Bwd Constraint → Int
            → ProofState (Bwd Constraint, Int)
```

```
introMotive (PI (PRF p) t) (x : xs) xs = introMotive (t $$ A (evTm x)) xs cs
```

If the index and target are both pairs, and the target is not a variable, then we simplify the introduced constraints by splitting the pair. We make a new subgoal by currying the motive type, solve the current motive with the curried version, then continue with the target replaced by its projections. We exclude the case where the target is a variable, because if so we might be able to simplify its constraint.

```
introMotive (PI (SIGMA dFresh rFresh) tFresh) (PI (SIGMA dTarg rTarg) tTarg) (x : xs) cs n
  | ¬ · isVar $ evTm x = do
    let mtFresh = currySigma dFresh rFresh tFresh
        mtTarg  = currySigma dTarg rTarg tTarg
        mtFresh' ← bquoteHere mtFresh
```

```
b :=>: _ ← make ("sig" :<: mtFresh')
ref      ← lambdaParam (fortran tFresh)
give (N (b : $ A (N (P ref : $ Fst)) : $ A (N (P ref : $ Snd))))
goIn
```

```
sTarg' ← bquoteHere (SIGMA dTarg rTarg)
```

```
introMotive mtFresh mtTarg ((N (x ?? sTarg' : $ Fst)) : (N (x ?? sTarg' : $ Snd)) : xs) cs (n + 1)
where
  isVar :: VAL → Bool
  isVar (NP x) = True
  isVar _      = False
```

```
introMotive (PI sFresh tFresh) (PI sTarg tTarg) (x : xs) cs n = do
  ref      ← lambdaParam (fortran tFresh)
  sFresh' ← bquoteHere sFresh
  sTarg'  ← bquoteHere sTarg
  let c = (ref :<: sFresh', (x : ~>: x) :<: (sTarg' : ~>: sTarg'))
  elimTrace $ "CONSTRAINT: " ++ show c
  introMotive (tFresh $$ A (NP ref)) (tTarg $$ A (evTm x)) xs (cs :<: c) n
```

```
introMotive SET SET [] cs n = return (cs, n)
```

If $\text{PI (SIGMA } d \ r) \ t$ is the type of functions whose domain is a sigma-type, then *currySigma* $d \ r \ t$ is the curried function type that takes the projections separately.

```
currySigma :: VAL → VAL → VAL → VAL
currySigma d r t = PI d · L $ (fortran r) : . [·a ·
  PI (r - $ [NV a]) · L $ (fortran t) : . [·b ·
  t - $ [PAIR (NV a) (NV b)]]]
```

Simplifying constraints

The *simplifyMotive* command takes a list of binders, a list of constraints and a goal type. It computes an updated list of binders and an updated goal type.

To the left, we have a backwards list of binders: updated references and types, and non-updated argument terms.

To the right, we have a forwards list of constraints: references in Ξ together with the term representation of their type, and typed terms in Δ to which the references are equated.

```
simplifyMotive :: Bwd Binder → Fwd Constraint → INTM
                → ProofState (Bwd Binder, INTM)
```

For each constraint, we check if the term on the right is a reference. If so, we check the equation is homogeneous (so substitution is allowed) and look for the reference in Δ . If we find it, we can simplify by removing the equation, updating the binder and renaming the following binders, constraints and term.

```
simplifyMotive bs (c@(x :<: xt, (q : ~ >: q') :<: (pt : ~ >: pt')) :> cs) tm
| NP p' ← evTm q' = do
  eq ← withNSupply $ equal $ SET :>: (pty x, pty p')
  case (eq, lookupBinders p' bs) of
    (True, Just (pre, post)) → do
      let (post', us) = renameBinders [(p', x)] B0 post
          cs'         = renameConstraints us B0 cs
          tm'         = renameTM us tm
          simplifyMotive (pre < + > post') (cs' <>> F0) tm'
```

If the conditions do not hold, we simply have to go past the constraint by turning it into a binder:

```
_ → passConstraint bs c cs tm
```

```
simplifyMotive bs (c :> cs) tm = passConstraint bs c cs tm
```

Eventually, we run out of constraints, and we win:

```
simplifyMotive bs F0 tm = return (bs, tm)
```

To pass a constraint, we append a binder with a fresh reference whose type is the proof of the equation. When applying the motive, we will need to use reflexivity applied to the non-updated target.

```
passConstraint :: Bwd Binder → Constraint → Fwd Constraint → INTM
                → ProofState (Bwd Binder, INTM)
passConstraint bs (x :<: xt, (s : ~ >: s') :<: (st : ~ >: st')) cs tm = do
  let qt = PRF (EQBLUE (xt :>: NP x) (st' :>: s'))
      elimTrace $ "PASS: " ++ show qt
      freshRef ("qsm" :<: evTm qt)
      (λq → simplifyMotive
        (bs :< (q :<: qt, N (P refl : $ A st : $ A s))) cs tm)
```

Building the motive

Finally, we can make the motive, hence closing the subgoal. This simply consists in chaining the commands above, and give the computed term. Unless we've screwed things up, *giveOutBelow* should always be happy.

```

makeMotive :: TY → INTM → Bwd (REF <: INTM) → Bwd INTM → TY →
  ProofState (Bwd (REF <: INTM), [Binder])
makeMotive motiveType goal delta targets elimTy = do
  elimTrace $ "goal: " ++ show goal
  elimTrace $ "targets: " ++ show targets

```

Extract non-parametric, non-removable hypotheses Δ_1 from the context Δ :

```

elimTrace $ "delta: " ++ show (fmap (\(n := _) <: _ → n) delta)
(delta0, delta1) ← findNonParametricHyps delta elimTy
elimTrace $ "delta1: " ++ show (fmap (\(n := _) <: _ → n) delta1)

```

Transform Δ_1 into Binder form:

```

let binders = fmap toBinder delta1

```

Introduce Ξ and generate constraints between its references and the targets:

```

(constraints, n) ← introMotive motiveType motiveType (trail targets) B0 0
elimTrace $ "constraints: " ++ show constraints

```

Simplify the constraints to produce an updated list of binders and goal type:

```

(binders', goal') ← simplifyMotive binders (constraints <>> F0) goal
elimTrace $ "binders' : " ++ show binders'
elimTrace $ "goal' : " ++ show goal'

```

Discharge the binders over the goal type to produce the motive:

```

let goal'' = liftType' (fmap fst binders') goal'
elimTrace $ "goal'' : " ++ show goal''
giveOutBelow goal''

```

Return to the construction of the rebuilt eliminator, by going out the same number of times as *introMotive* went in:

```

replicateM_ n goOut
return (delta0, trail binders')

```

5.2.3 Putting things together

Now we can combine the pieces to produce the *elim* command:

```

elim :: Maybe REF → (TY >: EXT M) → ProofState [EXT M :=>: VAL]
elim comma (elimTy >: elim) = do

```

Here we go. First, we need to retrieve some information about our goal and its context, before we start modifying the development.

```

(goal :=>: _) ← getGoal "T"
delta ← getLocalContext comma

```

We call *introElim* to rebuild the eliminator as a definition, check that everything is correct, and make subgoals for the motive and methods.

```

(elimName, motiveType, targets) ← introElim (elimTy >: elim)

```

Then we call *makeMotive* to introduce the indices, build and simplify constraints, and solve the motive subgoal.

```
(delta0, binders) ← makeMotive motiveType goal delta targets elimTy
```

We leave the definition of the rebuilt eliminator, with the methods unimplemented, and return to the original problem.

```
goOut
```

We solve the problem by applying the eliminator. Since the binders already contain the information we need in their second components, it is straightforward to build the term we want and to give it. Note that we have to look up the latest version of the rebuilt eliminator because its definition will have been updated when the motive was defined.

```
Just (elim :=>: _) ← lookupName elimName
tt ← give $ N $ elim $## map snd binders
```

Now we have to move the methods. We use the usual trick: make new definitions and solve the old goals with the new ones. First we collect the types of the methods, quoting them (to expand the definition of the motive) and lifting them over Δ_0 :

```
toMotive
methodTypes ← many $ do
  goDown
  _ :=>: ty ← getHoleGoal
  ty' ← bquoteHere ty
  return (liftType' delta0 ty')
```

Next we move to the top of the original development, and make the lifted methods:

```
goOut -- to makeE
goOut -- to the original goal
cursorTop
liftedMethods ← traverse (make · ("1m":<:)) methodTypes
```

Then we return to the methods and solve them with the lifted versions:

```
cursorBottom
toMotive
let args = fmap (NP · fstEx) delta0
flip traverse liftedMethods $ \tt → do
  goDown
  give $ N $ termOf tt $## args
```

Finally we move back to the bottom of the original development:

```
goOut
goOut
return liftedMethods

toMotive :: ProofState ()
toMotive = goIn >> goIn >> goTop

toFirstMethod :: ProofState ()
toFirstMethod = goIn >> goTop
```

This leaves us on the same goal we started with. For interactive use, we will typically want to move to the first (lifted) method:

```
toFirstMethod :: ProofState ()
toFirstMethod = goIn >> goTop
```

The `getLocalContext` command takes a comma and returns the local context, by looking up the parameters above and dropping those before the comma, if one is supplied. Regardless of the comma, we only go back as far as a `CurrentEntry` with name `magicImplName` if one exists, so shared parameters for programming problems will always be excluded.

```
getLocalContext :: Maybe REF → ProofState (Bwd (REF :<: INTM))
getLocalContext comma = do
  delta ← getDefinitionsToImpl
  return · bwdList $ case comma of
    Nothing → delta
    Just c   → dropWhile (λ(r :<: _) → c ≠ r) delta
```

We make elimination accessible to the user by adding it as a Cochon tactic:

```
import → CochonTacticsCode where
elimCTactic :: Maybe RelName → DEXtmRN → ProofState String
elimCTactic c r = do
  c' ← traverse resolveDiscard c
  (e :=>: _ :<: elimTy) ← elabInferFully r
  elim c' (elimTy :=>: e)
  toFirstMethod
  return "Eliminated. Subgoals awaiting work..."
```

```
import → CochonTactics where
: (simpleCT
  "eliminate"
  (| (| (B0:<) (tokenOption tokenName) |) :< (| id tokenExTm
    | id tokenAscription |) |)
  (λ[n, e] → elimCTactic (argOption (unDP · argToEx) n) (argToEx e))
  "eliminate [<comma>] <eliminator> - eliminates with a motive.")
```

5.3 Propositional Simplification

5.3.1 Setting the Scene

A proposition is *nice* if it is:

- a neutral term of type `PROP`;
- a blue equation with (at least) one neutral side;
- $(x:S) \Rightarrow P$, with S not a proof and P nice or `FF`;
- $P \Rightarrow Q$, with P nice and Q nice or `FF`.

A proposition is *simple* if it is `FF` or a conjunction of zero or more nice propositions.

We write $\Gamma \vdash P \rightsquigarrow R$ to mean that the proposition P in context Γ simplifies to the simple proposition R . The rules in Figure 5.1 define this judgment, and the implementation follows these rules. The judgment $\Gamma \Vdash P$ is not yet defined, but means P can be proved from hypotheses in Γ by backchaining search.

Given $\Gamma \vdash P : \text{PROP}$, the propositional simplifier will either:

- discover that P is absurd and provide a proof $\Gamma \vdash f : (: - P \Rightarrow \text{FF})$, represented by `Left f`; or
- simplify P to a conjunction $\bigwedge_i P_i$ together with proofs $\Gamma \vdash g_i : (: - P \Rightarrow P_i)$ and $\Gamma, \vec{P}_i \vdash h : (: - P)$, represented by `Right (ps, gs, h)`.

$$\begin{array}{c}
\overline{\Gamma \vdash \top \rightsquigarrow \top} \quad \overline{\Gamma \vdash \text{FF} \rightsquigarrow \text{FF}} \\
\\
\frac{\Gamma \vdash P \rightsquigarrow \text{FF}}{\Gamma \vdash P \wedge Q \rightsquigarrow \text{FF}} \quad \frac{\Gamma \vdash Q \rightsquigarrow \text{FF}}{\Gamma \vdash P \wedge Q \rightsquigarrow \text{FF}} \quad \frac{\Gamma \vdash P \rightsquigarrow \bigwedge_i P_i \quad \Gamma \vdash Q \rightsquigarrow \bigwedge_j Q_j}{\Gamma \vdash P \wedge Q \rightsquigarrow \bigwedge_i P_i \wedge \bigwedge_j Q_j} \\
\\
\frac{\Gamma \vdash P \rightsquigarrow \text{FF}}{\Gamma \vdash P \Rightarrow Q \rightsquigarrow \top} \quad \frac{\Gamma \vdash P \rightsquigarrow \bigwedge_i P_i \quad \Gamma, \vec{P}_i \vdash Q \rightsquigarrow \top}{\Gamma \vdash P \Rightarrow Q \rightsquigarrow \top} \quad \frac{\Gamma \vdash P \rightsquigarrow \bigwedge_i P_i \quad \Gamma, \vec{P}_i \vdash Q \rightsquigarrow \text{FF}}{\Gamma \vdash P \Rightarrow Q \rightsquigarrow \vec{P}_i \Rightarrow \text{FF}} \\
\\
\frac{\Gamma \vdash P \rightsquigarrow \bigwedge_i P_i \quad \Gamma, \vec{P}_i \vdash Q \rightsquigarrow \bigwedge_j Q_j}{\Gamma \vdash P \Rightarrow Q \rightsquigarrow \bigwedge_j (\vec{P}_i \Rightarrow Q_j)} \\
\\
\frac{\Gamma, x:S \vdash Q \rightsquigarrow \text{FF}}{\Gamma \vdash (x:S) \Rightarrow Q \rightsquigarrow (x:S) \Rightarrow \text{FF}} \quad \frac{\Gamma, x:S \vdash Q \rightsquigarrow \top}{\Gamma \vdash (x:S) \Rightarrow Q \rightsquigarrow \top} \quad \frac{\Gamma, x:S \vdash Q \rightsquigarrow \bigwedge_j Q_j}{\Gamma \vdash (x:S) \Rightarrow Q \rightsquigarrow \bigwedge_j ((x:S) \Rightarrow Q_j)} \\
\\
\overline{\Gamma \vdash (s:S) = (s:S) \rightsquigarrow \top} \quad \frac{\Gamma \vdash (s:S) \leftrightarrow (t:T) \rightsquigarrow R}{\Gamma \vdash (s:S) = (t:T) \rightsquigarrow R} \quad \frac{\Gamma \Vdash P}{\Gamma \vdash P \rightsquigarrow \top}
\end{array}$$

Figure 5.1: Propositional simplification rules

type Simplify = Either (EXTM → INTM)
(Bwd (REF :<: INTM), Bwd (EXTM → INTM), INTM)

pattern Simply ps gs h = Right (ps, gs, h)
pattern SimplyAbsurd prf = Left prf
pattern SimplyTrivial prf = Simply B0 B0 prf
pattern SimplyOne p g h = Simply (B0 :< p) (B0 :< g) h

We need a name supply for simplification, and use the Maybe monad to allow failure. This could just as well be an arbitrary monad supporting these effects.

type Simplifier x = ReaderT NameSupply Maybe x

We can transform a simplification of a proposition P into a simplification of another proposition Q if we have functions from proofs of Q to proofs of P (first argument) and from proofs of P to proofs of Q (second argument).

simplifyTransform :: (EXTM → EXTM) → (INTM → INTM) → Simplify → Simplify
simplifyTransform e f (SimplyAbsurd prf) = SimplyAbsurd (prf · e)
simplifyTransform e f (Simply ps gs h) = Simply ps (fmap (·e) gs) (f h)

5.3.2 Simplification in Action

The *propSimplify* command takes a global context, local context and proposition; it attempts to simplify the proposition following the rules above. if the result is *SimplyAbsurd* or *SimplyTrivial* then no simplification is guaranteed to have taken place, but if it is *Simply* one or more new propositions then these will be simpler than the original proposition. Note that this may fail if no simplification is possible.

propSimplify :: Bwd REF → VAL → Simplifier Simplify

Simplifying TT and FF is remarkably easy.

```
propSimplify _ ABSURD = return (SimplyAbsurd N)
propSimplify _ TRIVIAL = return (SimplyTrivial VOID)
```

To simplify a conjunction $P \wedge Q$, we simplify each conjunct separately, then call the *simplifyAnd* helper function to combine the results. If either conjunct is absurd, then we can easily show that the conjunction is absurd. Otherwise, we append the lists of conjuncts and pre-compose the proofs with Fst or Snd as appropriate.

```
propSimplify delta (AND p q) = forkSimplify delta p $
  λpr → case fst pr of
    SimplyAbsurd px → return $ SimplyAbsurd (px · (: $Fst))
    Simply pis pgs ph → forkSimplify delta q $
      λqr → case fst qr of
        SimplyAbsurd qx → return $ SimplyAbsurd (qx · (: $Snd))
        Simply qis qgs qh → return $ Simply (pis < + > qis)
          (fmap (·(: $Fst)) pgs < + > (fmap (·(: $Snd)) qgs))
          (PAIR ph qh)
```

To simplify $(x :: - P) \Rightarrow Lx$, we first try to simplify P :

```
propSimplify delta (ALL (PRF p) l) =
  forkSimplify' (fortran l) delta p antecedent
  where
    antecedent :: (Simplify, Bool) → Simplifier Simplify
```

If P is absurd then the implication is trivial, which we can prove by absurdity elimination whenever someone gives us a proof of P :

```
antecedent (SimplyAbsurd px, _) = do
  l' ← bquote B0 l
  l'' ← annotate l' (ARR (PRF p) PROP)
  return · SimplyTrivial · L $ "absurd" : .
    (N (nEOp : @ [px (V 0), PRF (N (l'' : $ A (NV 0)))]))
```

If P is trivial, then we go under L by applying the proof and simplify the resulting proposition Q . The implication simplifies to the result of simplifying Q , with the proofs constructed by λ -abstracting in one direction and applying the proof of P in the other direction.

```
antecedent (SimplyTrivial pt, _) =
  forkSimplify delta (l $$ A (evTm pt))
  (return · simplifyTransform (: $A pt) LK · fst)
```

If P simplifies nontrivially, we have a bit more work to do. We add the simplified conjuncts of P to the context and apply L to the proof of P in the extended context, giving a new proposition Q . We then simplify Q . If P did not simplify and Q is syntactically equal to FF then we have to give up, as otherwise we would end up simplifying the proposition to itself.

```
antecedent x@(Simply pis pgs ph, simplifiedP) = do
  let q = l $$ A (evTm ph)
      guard (simplifiedP ∨ ¬ (q ≡ ABSURD))
  forkSimplify (delta < + > fmap fstEx pis) q (consequent x)
```

```
consequent :: (Simplify, Bool) → (Simplify, Bool) → Simplifier Simplify
```

If Q is absurd, then the simplified proposition is an implication from the simplified conjuncts of P to FF . The proof of the original implication is by absurdity elimination, applying the pgs to the proof of P to get proofs of the pis , then applying the simplified proposition to these.

```
consequent (Simply pis pgs ph, _) (SimplyAbsurd qx, _) = do
  let pisImPLYFF = dischargeAll pis (PRF ABSURD)
      freshRef ("ri" :<: evTm pisImPLYFF) $ λriRef → do
    l' ← bquote B0 l
    l'' ← annotate l' (ARR (PRF p) PROP)
    rh ← mkFun $ λpref →
      let piPrfs = fmap ($P pref) pgs
          in N (nEOp : @ [
            N (P riRef ### piPrfs),
            PRF (N (l'' : $ A (NP pref)))
          ])
    return $ SimplyOne (riRef :<: pisImPLYFF)
      (λrt → dischargeLam (fmap fstEx pis) (qx (rt : $ A ph)))
  rh
```

If the consequent Q is trivial, then the implication is trivial, which we can prove by applying the pgs to a hypothetical proof of P to get proofs of the pis , then substituting these for the pis in the proof of Q .

```
consequent (Simply pis pgs ph, _) (SimplyTrivial qt, _) = do
  rh ← mkFun $ λpref → substitute pis (fmap ($P pref) pgs) qt
  return $ SimplyTrivial rh
```

Otherwise, if Q simplifies, then the implication simplifies to a conjunction of implications. Each implication is from the simplified components of P to a single simplified component of Q . To prove the original implication, we assume a proof of P , then construct proofs of the pis from it and proofs of the qis by applying the proofs of the ris to these. We can then substitute these proofs for the pis and qis in the proof of Q .

```
consequent (Simply pis pgs ph, simpP) (Simply qis qgs qh, simpQ)
  | simpP ∨ simpQ = do
  let ris = fmap (dischargeAllREF pis) qis
      rgs = fmap wrapper qgs
      rh ← mkFun $ λpref →
        let piPrfs = fmap ($P pref) pgs
            qiPrfs = fmap (λ(ri :<: _) → N (P ri ### piPrfs)) ris
            in substitute pis piPrfs (substitute qis qiPrfs qh)
        return $ Simply ris rgs rh
  where
    wrapper :: (EXTM → INTM) → EXTM → INTM
    wrapper qg pv = dischargeLam (fmap fstEx pis) (qg (pv : $ A ph))
```

If we get to this point, neither the antecedent nor the consequent simplified, so we had better give up.

```
consequent (_, False) (_, False) = ()
```

To simplify a proposition that is universally quantified over a (completely canonical) enumeration, we can simplify it for each possible value.

```

propSimplify delta p@(ALL (ENUMT e) b) | Just ts ← getTags B0 e =
  process B0 B0 B0 (ZE :=>: ZE) ts
where
  getTags :: Bwd VAL → VAL → Maybe (Bwd VAL)
  getTags ts NILE = (| ts |)
  getTags ts (CONSE t e) = getTags (ts :< t) e
  getTags ts _ = (|)
  process :: Bwd (REF :<: INTM) → Bwd (EXTM → INTM) → Bwd INTM →
    INTM :=>: VAL → Bwd VAL → Simplifier Simplify
  process qs gs hs (n :=>: nv) B0 = do
    e' ← bquote B0 e
    b' ← bquote B0 b
    let b'' = b' ?? ARR (ENUMT e') PROP
    return $ Simply qs gs $
      L $ "xe" : . N (switchOp : @ [e', NV 0,
        L $ "yb" : . PRF (N (b'' : $ A (NV 0))),
        Prelude.foldr PAIR VOID (trail hs)])
  process qs1 gs1 hs1 (n :=>: nv) (ts :< t) =
    forkSimplify delta (b $$ A nv) $ λ(btSimp, _) → case btSimp of
      SimplyAbsurd prf → return $ SimplyAbsurd (prf · (: $ A n))
      Simply qs2 gs2 h2 → do
        let gs2' = fmap (·(: $ A n)) gs2
        process (qs1 < + > qs2) (gs1 < + > gs2') (hs1 :< h2)
          (SU n :=>: SU nv) ts

```

To simplify $(x : S) \Rightarrow Lx$ where S is not of the form $:- P$, we generate a fresh reference and apply L to it to get the proposition Q under the binder, which we can then simplify.

```

propSimplify delta p@(ALL s l) = freshRef (fortran l :<: s) $ λrefS → do
  let q = l $$ A (NP refS)
  guard (¬ (q ≡ ABSURD))
  forkPropSimplify (delta :< refS) q (consequent refS)
where
  consequent :: REF → Simplify → Simplifier Simplify

```

If Q is absurd, then the proposition simplifies to $(x : S) \Rightarrow \text{FF}$, with proofs by absurdity elimination in each direction.

```

consequent refS (SimplyAbsurd qx) = do
  freshRef ("psA" :<: PRF (ALLV (fortran l) s ABSURD)) $ λrefA → do
    l' ← bquote B0 l
    s' ← bquote B0 s
    let l'' = l' ?? ARR s' PROP
    return $
      SimplyOne (refA :<: PRF (ALLV (fortran l) s' ABSURD))
        (λpv → L $ "cab" : . qx ((inc 0 %%%# pv) : $ A (NV 0)))
        (L $ "cabs2" : . N (nEOp : @ [N (P refA : $ A (NV 0)),
          PRF (N (l'' : $ A (NV 0)))]))

```

If Q is trivial, then the proposition is also trivial, just by λ -binding the variable in the proof.

```

consequent refS (SimplyTrivial qt) =
  return $ SimplyTrivial (dischargeLam (B0 :< refS) qt)

```

Otherwise, Q simplifies to a conjunction of propositions $(x : S) \Rightarrow Q_i$ for each Q_i in the simplification of Q .

```

consequent refS (Simply qis qgs qh) = do
  s' ← bquote B0 s
  let pis    = fmap (dischargeAllREF (B0 :< (refS :<: s'))) qis
      pgs    = fmap (λqg pv → L $ "s" : . qg ((inc 0 %%# pv) : $ A (NV 0))) qgs
      qiPrfs = fmap (λ(pi :<: _) → N (P pi : $ A (NP refS))) pis
      ph     = dischargeLam (B0 :< refS) (substitute qis qiPrfs qh)
  return (Simply pis pgs ph)

```

To simplify a blue equation, we use *simplifyBlue*.

```

propSimplify delta (EQBLUE (sty :>: s) (tty :>: t)) =
  simplifyBlue True delta (sty :>: s) (tty :>: t)

```

To simplify a stuck green equation, we cannot unroll the corresponding blue equation because we would get an infinite loop, but we can use the other simplifications for blue equations. If we do not find a proof, we return the blue version as the simplification result because it is nicer than a green equation for the user.

```

propSimplify delta p@(N (op : @ [sty, s, tty, t]) | op ≡ eqGreen) = do
  m ← optional $ simplifyBlue False delta (sty :>: s) (tty :>: t)
  let q = PRF (EQBLUE (sty :>: s) (tty :>: t))
      q' ← bquote B0 q
  case m of
  Just (SimplyTrivial prf) →
    return · SimplyTrivial $ N (prf ?? q' : $ Out)
  _ → freshRef ("q" :<: q) $ λqRef →
    return $ SimplyOne (qRef :<: q') (CON · N) (N (P qRef : $ Out))

```

If nothing else matches, we can always try searching the context.

```

propSimplify delta p = propSearch delta p

```

The *simplifyBlue* command attempts to simplify a blue equation using *refl*, optionally unrolling it (calling *eqGreen* and simplifying the resulting pieces), or just searching the context. Note that if the *canUnroll* boolean is *False*, this will either find a proof of the equation and return *SimplyTrivial*, or it will fail.

```

simplifyBlue :: Bool → Bwd REF → TY :>: VAL → TY :>: VAL →
  Simplifier Simplify
simplifyBlue canUnroll delta (sty :>: s) (tty :>: t) =
  useRefl
  ⊙ unroll canUnroll
  ⊙ propSearch delta (EQBLUE (sty :>: s) (tty :>: t))
where
  useRefl :: Simplifier Simplify
  useRefl = do
    guard ≪≪ (asks · equal $ SET :>: (sty, tty))
    guard ≪≪ (asks · equal $ sty :>: (s, t))
    sty' ← bquote B0 sty
    s' ← bquote B0 s
    return · SimplyTrivial $ N (P refl : $ A sty' : $ A s')
  unroll :: Bool → Simplifier Simplify
  unroll False = ()
  unroll True = case opRun eqGreen [sty, s, tty, t] of
  Left _ → ()
  Right TRIVIAL → return $ SimplyTrivial (CON VOID)
  Right q → forkSimplify delta q
    (return · simplifyTransform (: $Out) CON · fst)

```

The *propSearch* operation searches the context for a proof of the proposition, and if it finds one, returns the trivial simplification. When *seekProof* finds a proof in the context, it calls *backchain* to go under any implications and test if the consequent matches the goal; if so, *backchain* then calls *seekProof* to attempt to prove the hypotheses, in the context with the backchained proposition removed.

```

propSearch :: Bwd REF → VAL → Simplifier Simplify
propSearch delta p = do
  prf ← seekProof delta F0 p
  prf' ← bquote B0 prf
  return $ SimplyTrivial prf'
where
  seekProof :: Bwd REF → Fwd REF → VAL → Simplifier VAL
  seekProof B0 _ _ = ()
  seekProof (rs < ref @ ( _ := DECL <: PRF q )) fs p =
    backchain (rs < ref) fs B0 p q ⊕ seekProof rs (ref > fs) p
  seekProof (rs < ref) fs p = seekProof rs (ref > fs) p
  backchain :: Bwd REF → Fwd REF → Bwd REF → VAL → VAL → Simplifier VAL
  backchain rs fs ss p (ALL (PRF s) l) = freshRef ("bc" <: PRF s) $ λsRef →
    backchain rs fs (ss < sRef) p (l $$$ A (NP sRef))
  backchain (rs < ref) fs ss p q = do
    guard ≪≪ (asks · equal $ PROP >: (p, q))
    ssPrfs ← traverse (seekProof (rs <>< fs) F0 · unPRF · pty) ss
    return $ pval ref $$$ fmap A ssPrfs
  unPRF :: VAL → VAL
  unPRF (PRF p) = p

```

The *forceSimplify* function is a variant of *propSimplify* that guarantees to give a result, by trying to simplify the proposition and yielding an identical copy if simplification fails. It also returns a boolean indicating whether simplification occurred. This is useful in cases such as \wedge , where we know we can do some simplification even if the conjuncts do not simplify. The first argument is an optional hint for the name of the reference.

```

forceSimplify :: String → Bwd REF → VAL → Simplifier (Simplify, Bool)
forceSimplify hint delta p =
  (propSimplify delta p ≫≫ return · (, True))
  ⊕ simplifyNone (PRF p)
where
  simplifyNone :: (NameSupplier m) ⇒ TY → m (Simplify, Bool)
  simplifyNone ty = do
    ty' ← bquote B0 ty
    freshRef (nameHint ty <: ty) $ λref →
      return (SimplyOne (ref <: ty') N (NP ref), False)
  nameHint :: VAL → String
  nameHint _ | ¬ (null hint) = hint
  nameHint (NP (n := _)) = fst (last n)
  nameHint (L (H _ s _)) = s
  nameHint (L (s : . _)) = s
  nameHint _ = "xnh"

```

To ensure correctness of fresh name generation, we need to fork the name supply before performing additional simplification, so we define helper functions to fork then call *propSimplify* or *forceSimplify*.

```

forkSimplify :: Bwd REF → VAL →
  ((Simplify, Bool) → Simplifier a) → Simplifier a
forkSimplify = forkSimplify' " "

```

```

forkSimplify' :: String → Bwd REF → VAL →
  ((Simplify, Bool) → Simplifier a) → Simplifier a
forkSimplify' hint delta p f = forkNSupply "fS"
  (forceSimplify hint delta p) f

```

```

forkPropSimplify :: Bwd REF → VAL →
  (Simplify → Simplifier a) → Simplifier a
forkPropSimplify delta p f = forkNSupply "fPS" (propSimplify delta p) f

```

5.3.3 Invoking Simplification

When in the ProofState, we can simplify a proposition using the current name supply and context:

```

runPropSimplify :: VAL → ProofState (Maybe Simplify)
runPropSimplify p = do
  nsupply ← askNSupply
  es      ← getParamsInScope
  return $ runReaderT (propSimplify (bwdList es) p) nsupply

```

The `propSimplifyHere` command attempts propositional simplification on the current location, which must be an open goal of type PRF p for some p . If it is unable to simplify p or simplifies it to FF, it will fail and throw an error. Otherwise, it will create zero or more new subgoals (from the conjuncts of the simplified proposition, if any), solve the current goal with the subgoals, and return a list of their types.

```

propSimplifyHere :: ProofState (Bwd INTM)
propSimplifyHere = do
  simpTrace "propSimplifyHere"
  (_ :=>: PRF p) ← getHoleGoal
  pSimp ← runPropSimplify p
  case pSimp of
    Nothing →
      throwError' $ err "propSimplifyHere: unable to simplify."
    Just (SimplyAbsurd _) →
      throwError' $ err "propSimplifyHere: oh no, goal is absurd!"
    Just (SimplyTrivial prf) → give prf >> return B0
    Just (Simply pis _ ph) → do
      subs ← traverse makeSubgoal pis
      give (substitute pis subs ph)
      return (fmap sndEx pis)
  where

```

The `makeSubgoal` command makes a new subgoal whose type corresponds to the type of the given reference, and returns its term representation.

```

makeSubgoal :: REF <: INTM → ProofState INTM
makeSubgoal (ref <: q') = do
  x      ← pickName "q" (fst (last (refName ref)))
  t :=>: _ ← make (x <: q')
  return (N t)

```

The *propSimplify* tactic attempts to simplify the type of the current goal, which should be propositional. Usually one will want to use *simplify* instead, or simplification will happen automatically (with the **let** and \leq tactics), but this is left for backwards compatibility.

```

import → CochonTacticsCode where
  propSimplifyTactic :: ProofState String
  propSimplifyTactic = do
    subs ← propSimplifyHere
    case subs of
      B0 → return "Solved."
      _  → do
        subStrs ← traverse prettyType subs
        nextGoal
        return ("Simplified to:\n" ++
              foldMap ( $\lambda s \rightarrow s ++ "\n"$ ) subStrs)
    where
      prettyType :: INTM → ProofState String
      prettyType ty = prettyHere (SET >: ty) >>= return · renderHouseStyle

import → CochonTactics where
  : nullaryCT "propsimplify" propSimplifyTactic
  "propsimplify - applies propositional simplification to the current goal."

```

5.4 Problem Simplification

Now that we have considered how to simplify propositions, we wish to use this technology to simplify programming problems. Suppose we have a goal of type $\Delta \rightarrow T$, where Δ is some telescope of hypotheses. There are various things we can do to simplify the problem, such as:

- splitting up Σ -types into their components;
- simplifying propositions in the hypotheses and goal;
- discarding uninformative hypotheses; and
- solving the problem completely if it is trivial (for example, if it depends on a proof of false).

The *problemSimplify* command performs these simplifications. It works by repeatedly transforming the proof state into a simpler version, one step at a time. It will fail if no simplification is possible. The real work is done in *simplifyGoal* below.

```

problemSimplify :: ProofState (EXTM :=> VAL)
problemSimplify = do
  simpTrace "problemSimplify"
  getHoleGoal >>= simplifyGoal True · valueOf >>= getCurrentDefinition

```

We say simplification is *at the top level* if we are simplifying exactly the current goal in the proof state. If this is not the case, we can still make some simplifications but others require us to quote the type being simplified and make a new goal so we are back at the top level. The Bool parameter to the following functions indicates whether simplification is at the top level.

When simplifying at the top level, we should *give* the simplified form once we have computed it. The *topWrap* command makes this easy.

```

topWrap :: Bool → INTM :=> VAL → ProofState (INTM :=> VAL)
topWrap True tt = give (termOf tt) >>= return tt
topWrap False tt = return tt

```

Once we have simplified the goal slightly, we use *trySimplifyGoal* to attempt to continue, but give back the current result if no more simplification is possible. If not at the top level, this has to create a new goal.

```

trySimplifyGoal :: Bool → TY → ProofState (INTM :=> VAL)
trySimplifyGoal True g = simplifyGoal True g ⊕
    (neutralise ≪≪ getCurrentDefinition)
trySimplifyGoal False g = simplifyGoal False g ⊕ (do
  g' ← bquoteHere g
  es ← getEntriesAbove
  cursorAboveLambdas
  make ("tsg" :<: liftType es g')
  goIn
  let rs = paramREFs es
      traverse (lambdaParam · refNameAdvice) rs
      x :=>: xv ← getCurrentDefinition
  goOut
  let z = x $$$ map NP rs
      return $ N z :=>: evTm z
  )

```

We implement the simplification steps in *simplifyGoal*. This takes a boolean parameter indicating whether simplification is at the top level, and a type being simplified. It will return a term and value of that type (which might be the current hole).

```

simplifyGoal :: Bool → TY → ProofState (INTM :=> VAL)

```

Functions from the unit type are simply constants, so we simplify them as such.

```

simplifyGoal b (PI UNIT t) = do
  simpTrace "PI UNIT"
  x :=>: xv ← trySimplifyGoal False (t $$$ A VOID)
  topWrap b $ LK x :=>: LK xv

```

Given a function from a Σ -type, we can split it into its components. **Adam:** we should not automatically split if this parameter belongs to the user (i.e. appears in a programming problem).

```

simplifyGoal b (PI (SIGMA d r) t) = do
  simpTrace "PI SIGMA"
  let mt = PI d · L $ (fortran r) : . [.a ·
    PI (r -$ [NV a]) · L $ (fortran t) : . [.b ·
    t -$ [PAIR (NV a) (NV b)]]]
  x :=>: xv ← simplifyGoal False mt
  ex ← annotate x mt
  let body = N (ex : $ A (N (V 0 : $ Fst)) : $ A (N (V 0 : $ Snd)))
  topWrap b $ L ("ps" : . body) :=>: L ("ps" : . body)

```

Similarly, if we have a function from an enumeration, we can split it into its branches. **Adam:** we should not do this automatically at all, but we need to modify the induction principles generated for data types first. For the moment, we check the enumeration is completely canonical, thereby avoiding the worst problems with this simplification step.

```

simplifyGoal b (PI (ENUMT e) t) | checkTags e = do
  simpTrace "PI ENUMT"
  x :=>: xv ← trySimplifyGoal False (branchesOp @@ [e, t])
  e' ← bquoteHere e
  t' ← bquoteHere t
  let body = N (switchOp : @ [e', NV 0, t', x])
  topWrap b $ L ("pe" : . body) :=>: L ("pe" : . body)
where
  checkTags :: VAL → Bool
  checkTags NILE = True
  checkTags (CONSE _ e) = checkTags e
  checkTags _ = False

```

If we have a function from a proof, we call on propositional simplification to help out. If the proposition is absurd we win, and if it simplifies into a conjunction we abstract over each conjunct individually. If the proposition cannot be simplified, we check to see if it is an equation with a variable on one side, and if so, eliminate it by *substEq*. Otherwise, we just put it in the context and carry on. Note that this assumes we are at the top level.

```

simplifyGoal True (PI (PRF p) t) = do
  simpTrace "PI PRF"
  pSimp ← runPropSimplify p
  maybe (elimEquation p t ⊕ passHypothesis t) (simplifyProp p t) pSimp
where
  elimEquation :: VAL → VAL → ProofState (INTM :=> VAL)
  elimEquation (EQBLUE (_X :=> x) (_Y :=> NP y@(yn := DECL :=> _))) t = do
    guard ≪≪ (withNSupply $ equal (SET :=> (_X, _Y)))
    t' ← bquoteHere t
    guard $ y ∈ t'
    simpTrace $ "elimSimp: " ++ show yn
    q ← lambdaParam "qe"
    _X' ← bquoteHere _X
    x' ← bquoteHere x
    let ety = PI (ARR _X SET) $ L $ "P" : . [_P .
      ARR (N (V _P : $ A x')) (N (V _P : $ A (NP y)))
    ]
    ex = P substEq : $ A _X' : $ A x' : $ A (NP y) : $ A (NP q)
    elimSimplify (ety :=> ex)
    neutralise ≪≪ getCurrentDefinition
  elimEquation (EQBLUE (_Y :=> NP y@(yn := DECL :=> _)) (_X :=> x)) t = do
    guard ≪≪ (withNSupply $ equal (SET :=> (_X, _Y)))
    t' ← bquoteHere t
    guard $ y ∈ t'
    simpTrace $ "elimSimp: " ++ show yn
    q ← lambdaParam "qe"
    _X' ← bquoteHere _X
    x' ← bquoteHere x
    let ety = PI (ARR _X SET) $ L $ "P" : . [_P .
      ARR (N (V _P : $ A x')) (N (V _P : $ A (NP y)))
    ]
    ex = P substEq : $ A _X' : $ A x' : $ A (NP y) : $ A
      (N (P symEq : $ A _X' : $ A (NP y) : $ A x' : $ A (NP q)))
    elimSimplify (ety :=> ex)
    neutralise ≪≪ getCurrentDefinition
  elimEquation _ _ = ()
  simplifyProp :: VAL → VAL → Simplify → ProofState (INTM :=> VAL)
  simplifyProp p t (SimplyAbsurd prf) = do
    r ← lambdaParam (fortran t)
    t' ← bquoteHere t
    t'' ← annotate t' (ARR (PRF p) SET)
    neutralise ≪≪ give (N (nEOp : @ [prf (P r), N (t'' : $ A (NP r))]))
  simplifyProp p t (SimplyTrivial prf) = do
    x :=>: xv ← trySimplifyGoal False (t $$ A (evTm prf))
    neutralise ≪≪ give (LK x)
  simplifyProp p t (SimplyOne (qr :=> qst) g h) = do
    t' ← bquoteHere t
    t'' ← annotate t' (ARR (PRF p) SET)
    let q = PIV (fortran t) qst (N (t'' : $ A ((B0 :=> qr) -|| h)))
    x :=>: xv ← trySimplifyGoal False (evTm q)
    let y = x ?? q
    r ← lambdaParam (fortran t)
    neutralise ≪≪ give (N (y : $ A (g (P r))))
  simplifyProp p t (Simply qs gs h) = do
    t' ← bquoteHere t
    t'' ← annotate t' (ARR (PRF p) SET)
    let q = dischargePi qs (N (t'' : $ A h))
    x :=>: xv ← trySimplifyGoal False (evTm q)
    let y = x ?? q
    r ← lambdaParam (fortran t)
    let as' = fmap ($ (P r)) qs

```

Otherwise, we simplify Π -types by introducing a hypothesis, provided we are at the top level.

```
simplifyGoal True (PI s t) = do
  simpTrace "PI top"
  passHypothesis t
```

To simplify a Π -type when not at the top level, we have to create a subgoal.

```
simplifyGoal False g@(PI _ _) = do
  simpTrace "PI not"
  g' ← bquoteHere g
  es ← getEntriesAbove
  cursorAboveLambdas
  make ("pig" :<: liftType es g')
  goIn
  let rs = paramREFs es
      traverse (lambdaParam · refNameAdvice) rs
  _ :=>: ty ← getHoleGoal
  simplifyGoal True ty
  x :=>: xv ← getCurrentDefinition
  goOut
  let z = x $$$ map NP rs
      return $ N z :=>: evTm z
```

When the goal is a proof of a proposition, and we are at the top level, we can just invoke propositional simplification...

```
simplifyGoal True (PRF p) = do
  simpTrace "PRF top"
  propSimplifyHere
  getCurrentDefinition >>= neutralise
```

...and if we are not at the top level, we create a subgoal.

```
simplifyGoal False g@(PRF _) = do
  simpTrace "PRF not"
  g' ← bquoteHere g
  make ("prg" :<: g')
  goIn
  x :=>: xv ← simplifyGoal True g
  goOut
  return $ x :=>: xv
```

If the goal is a programming problem to produce a proof, and the proposition is trivial, then we win. However, we cannot simplify non-trivial propositions as the user might not want us to. Similarly, we cannot simplify inside LABEL unless we know we are going to solve the goal completely.

```
simplifyGoal b (LABEL _ (PRF p)) = do
  pSimp ← runPropSimplify p
  case pSimp of
  Just (SimplyTrivial prf) → do
    topWrap b $ LRET prf :=>: LRET (evTm prf)
  _ → ()
```

If the goal is a Σ -type, we might as well split it into its components.

```

simplifyGoal b (SIGMA s t) = do
  simpTrace "SIGMA"
  stm :=>: sv ← trySimplifyGoal False s
  ttm :=>: tv ← trySimplifyGoal False (t $$ A sv)
  topWrap b $ PAIR stm ttm :=>: PAIR sv tv

```

If we are really lucky, the goal is trivial and we win.

```

simplifyGoal b UNIT = topWrap b $ VOID :=>: VOID
simplifyGoal b (LABEL _ UNIT) = topWrap b $ LRET VOID :=>: LRET VOID

```

Otherwise, we cannot simplify the problem.

```

simplifyGoal _ _ = throwError' $ err "simplifyGoal: cannot simplify"

```

When at the top level and simplifying a Π -type, *passHypothesis* introduces a hypothesis into the context and continues simplifying. Its argument is the codomain function of the Π -type.

```

passHypothesis :: VAL → ProofState (INTM :=>: VAL)
passHypothesis t = do
  ref ← lambdaParam (fortran t)
  trySimplifyGoal True (t $$ A (NP ref))

```

The *elimSimplify* command invokes elimination with a motive, simplifies the methods, then returns to the original goal.

```

elimSimplify :: (TY :=>: EXTM) → ProofState ()
elimSimplify tt = do
  methods ← elim Nothing tt
  simpTrace "Eliminated!"
  toFirstMethod
  replicateM_ (length methods) (optional problemSimplify >> goDown)
  goOut

```

```

cursorAboveLambdas :: ProofState ()
cursorAboveLambdas = do
  es ← getEntriesAbove
  case paramREFs es of
    [] → return ()
    _ → cursorUp >> cursorAboveLambdas

```

```

import → CochonTactics where
  : nullaryCT "simplify" (problemSimplify >> optional seekGoal >> return "Simplified.")
  "simplify - simplifies the current problem."

```

5.5 Record declaration

```

elabRecord :: String → [(String, DInTmRN)] → ProofState (EXTM :=>: VAL)
elabRecord name fields = ⊥ -- XXX: not yet implemented

```

```

import → CochonTactics where
  : CochonTactic
  { ctName = "record"
  , ctParse = do
    nom ← tokenString
    keyword KwDefn
    scs ← tokenListArgs (bracket Round $ tokenPairArgs
      tokenString
      (keyword KwAsc)
      tokenInTm)
      (keyword KwSemi)
    return $ B0 :< nom :< pars :< scs
  , ctIO = (λ[StrArg nom, pars, cons] → simpleOutput $
    elabRecord nom (argList (argPair argToStr argToIn) pars)
      (argList (argPair argToStr argToIn) cons)
    >> return "Record'd. ")
  , ctHelp = "record <name> [<para>]* := [(<label> : <ty>) ;]* - builds a record t
  }

```

5.6 Relabelling

The `partApplyREF` command takes a reference and list of argument values (as generated by `splitSpine`, and splits them into a term in local scope (i.e. the reference applied to the shared parameters) and a list of additional arguments. **Adam**: where should this live?

```

partApplyREF :: REF → [VAL] → ProofStateT e (EXTM :=>: VAL :<: TY, [VAL])
partApplyREF r@( _ := DECL :<: _ ) as = return (P r :=>: NP r :<: pty r, as)
partApplyREF r as = do
  es ← getGlobalScope
  help (pty r) B0 (paramREFs es) as
where
  help :: TY → Bwd REF → [REF] → [VAL] → ProofStateT e (EXTM :=>: VAL :<: TY, [VAL])
  help (PI s t) cs (r : rs) (NP x : as) | r ≡ x =
    help (t $$ A (NP x)) (cs :< r) rs as
  help ty cs [] as = do
    let t = P r $$$ fmap NP cs
    return (t :=>: evTm t :<: ty, as)
  help ty cs rs as = throwError' $ err "partApplyREF: failed on type " ++ errTyVal (ty :<: SET) +

```

A relabelling is a map from references to strings, giving a new name that should be used for the reference.

```
type Relabelling = Bwd (REF, String)
```

The `relabel` command changes the names of the pattern variables in a programming problem. It takes an unelaborated application corresponding to the programming problem, matches it against the existing arguments to determine the renaming, and refines the proof state appropriately.

```

relabel :: DEXtmRN → ProofState ()
relabel (DP [(f, Rel 0)] :$ts) = do
  tau' :=>: tau ← getHoleGoal
  case tau of
  LABEL (N l) ty → do
    let Just (r, as) = splitSpine l
        unless (f ≡ refNameAdvice r) $
          throwError' $ err "relabel: mismatched function name!"
        ts' ← traverse unA ts
        (_:<: rty, as') ← partApplyREF r as
        rl ← execStateT (relabelArgs rty ts' as') B0
        es ← getEntriesAbove
        refineProofState (liftType es tau') (N · ($ :$paramSpine es))
        introLambdas rl (paramREFs es)
    _ → throwError' $ err "relabel: goal is not a labelled type!"
  relabel _ = throwError' $ err "relabel: malformed relabel target!"

```

Once the refinement has been made, we need to introduce the hypotheses using their new names. The `introLambdas` command takes a relabelling and the references from the entries that were abstracted over, and introduces a hypothesis corresponding to each reference with the reference's new name.

```

introLambdas :: Relabelling → [REF] → ProofState ()
introLambdas rl [] = return ()
introLambdas rl (x : xs) = lambdaParam newName >> introLambdas rl xs
  where
    newName = case find ((x ≡) · fst) rl of
      Just (_, s) → s
      Nothing    → refNameAdvice x

```

```

unA :: MonadError (StackError t) m ⇒ Elim a → m a
unA (A a) = return a
unA _     = throwError' $ err "unA: not an A!"

```

```

extendRelabelling :: REF → String → StateT Relabelling (ProofStateT a) ()
extendRelabelling r s = do
  rl ← get
  case find ((r ≡) · fst) rl of
  Nothing          → put (rl :< (r, s))
  Just (_, t) | s ≡ t → return ()
               | otherwise → throwError' $
  err ("relabelValue: inconsistent names '" ++ s ++ "' and '" ++ t
      ++ "' for") ++ errRef r

```

```

relabelArgs :: TY → [DInTmRN] → [VAL] → StateT Relabelling ProofState ()
relabelArgs _ [] [] = return ()
relabelArgs _ [] _ = throwError' $ err "relabel: too few arguments!"
relabelArgs _ _ [] = throwError' $ err "relabel: too many arguments!"
relabelArgs (PI s t) (w : ws) (a : as) = do
  relabelValue (s :>: (w, a))
  relabelArgs (t $$ A a) ws as
relabelArgs ty ws as = throwError' $ err "relabel: unmatched\nty ="
  ++ errTyVal (ty :<: SET)
  ++ err "\nas =" ++ foldMap errVal as

```

relabelValue :: (TY :>: (DInTmRN, VAL)) → StateT Relabelling ProofState ()

If the value we are matching against is a stuck recursive call, we match against the user-friendly label (which is what the user would expect) rather than the horrible induction term.

relabelValue (ty :>: (w, N (n : \$ Call l))) = *relabelValue* (ty :>: (w, l))

If we are matching two parameters (applied to some arguments), we can extend the relabelling and matching the arguments.

```
relabelValue (ty :>: (DN (DP [(s, Rel 0)] $ws), N n))
| Just (r, as) ← splitSpine n = do
  (_ :<: ty, as') ← lift $ partApplyREF r as
  extendRelabelling r s
  ws' ← lift $ traverse unA ws
  relabelArgs ty ws' as'
```

If the display term is an underscore then we make no changes to the relabelling.

relabelValue (ty :>: (DU, _)) = *return* ()

If the display term and value are both canonical, we halfzip them together (ensuring the constructors match) and use *canTy* to match the pieces.

```
relabelValue (C cty :>: (DC w, C v)) = case halfZip w v of
  Nothing → throwError' $ err "relabelValue: mismatched constructors!"
  Just wv → (liftage fst $ canTy chev (cty :>: wv)) >> return ()
where
  chev :: (TY :>: (DInTmRN, VAL)) →
    StateT Relabelling (ProofStateT (DInTmRN, VAL)) (() :=>: VAL)
  chev (ty :>: (w, v)) = do
    liftage (λt → (t, error "erk")) $ relabelValue (ty :>: (w, v))
    return (() :=>: v)
  liftage :: (s → t) → StateT x (ProofStateT s) a
    → StateT x (ProofStateT t) a
  liftage = mapStateT · liftErrorState
```

If it is a tag (possibly applied to arguments) and needs to be matched against an element of an inductive type, we match the tags and values.

```
relabelValue (IMU l _I d i :>: (DTag s as, CON (PAIR t xs)))
| Just (e, f) ← sumilike _I (d $$ A i) = do
  ntm :=>: nv ← lift $ elaborate (Loc 0) (ENUMT e :>: DTAG s)
  sameTag ← lift $ withNSupply $ equal (ENUMT e :>: (nv, t))
  unless sameTag $ throwError' $ err "relabel: mismatched tags!"
  relabelValue (idescOp @@ [-I, f t,
    L $ "i" :. [.i · IMU (fmap (-$[]) l) (-I -$ []) (d -$ []) (NV i)]]
    :>: (foldr DPAIR DU as, xs))
```

Lest we forget, tags may also belong to enumerations!

```
relabelValue (ENUMT e :>: (DTag s [], t)) = do
  ntm :=>: nv ← lift $ elaborate (Loc 0) (ENUMT e :>: DTAG s)
  sameTag ← lift $ withNSupply $ equal (ENUMT e :>: (nv, t))
  unless sameTag $ lift $ throwError' $ err "relabel: mismatched tags!"
```

Nothing else matches? We had better give up.

```
relabelValue (ty :>: (w, v)) = lift $ throwError' $ err "relabel: can't match"
  ++ errTm w ++ err "with" ++ errTyVal (v :<: ty)
```

```
import → CochonTactics where
  : unaryExCT "relabel" (λex → relabel ex >> return "Relabelled.")
  "relabel <pattern> - changes names of arguments in label to pattern"
```

5.7 Programming Gadgets

5.7.1 The Return gadget

```
import → CochonTactics where
  : unaryInCT "=" (λtm → elabGiveNext (DLRET tm) >> return "Ta.")
  "= <term> - solves the programming problem by returning <term>."
```

5.7.2 The Define gadget

```
import → CochonTacticsCode where
  defineCTactic :: DExTmRN → DInTmRN → ProofState String
  defineCTactic rl tm = do
    relabel rl
    elabGiveNext (DLRET tm)
    return "Hurrah!"
```

```
import → CochonTactics where
  : (simpleCT
    "define"
    (| (| (B0:<) tokenExTm |) :< (%keyword KwDefn%) tokenInTm |)
    (λ[ExArg rl, InArg tm] → defineCTactic rl tm)
    "define <prob> := <term> - relabels and solves <prob> with <term>.")
```

5.7.3 The By gadget

The By gadget, written \leq , invokes elimination with a motive, then simplifies the methods and moves to the first subgoal remaining.

```
import → CochonTacticsCode where
  byCTactic :: Maybe RelName → DExTmRN → ProofState String
  byCTactic n e = do
    elimCTactic n e
    optional problemSimplify -- simplify first method
    many (goDown >> problemSimplify) -- simplify other methods
    many goUp -- go back up to motive
    optional seekGoal -- jump to goal
    return "Eliminated and simplified."
```

```
import → CochonTactics where
  : (simpleCT
    "<="
    (| (| (B0:<) (tokenOption tokenName) |) :< (| id tokenExTm
      | id tokenAscription |) |)
    (λ[n, e] → byCTactic (argOption (unDP · argToEx) n) (argToEx e))
    "<= [<comma>] <eliminator> - eliminates with a motive.")
```

5.7.4 The Refine gadget

The Refine gadget relabels the programming problem, then either defines it or eliminates with a motive.

```
import → CochonTactics where
  : (simpleCT
    "refine"
    (| (| (B0:<) tokenExTm |) :< (| id (%keyword KwEq%) tokenInTm
      | id (%keyword KwBy%) tokenExTm
      | id (%keyword KwBy%) tokenAscription
      |)
    |)
    (λ[ExArg rl, arg] → case arg of
      InArg tm → defineCTactic rl tm
      ExArg tm → relabel rl >> byCTactic Nothing tm)
    ("refine <prob> = <term> - relabels and solves <prob> with <term>.\n" ++
     "refine <prob> <= <eliminator> - relabels and eliminates with a motive."))
```

5.7.5 The Solve gadget

```
import → CochonTactics where
  : simpleCT
    "solve"
    (| (| (B0:<) tokenName |) :< tokenInTm |)
    (λ[ExArg (DP rn :$[]), InArg tm] → do
      (ref, spine, _) ← resolveHere rn
      _:<: ty ← inferHere (P ref $ : $ toSpine spine)
      _:=>: tv ← elaborate' (ty :=>: tm)
      tm' ← bquoteHere tv -- force definitional expansion
      solveHole ref tm'
      return "Solved."
    )
    "solve <name> <term> - solves the hole <name> with <term>."
```

5.8 Converting Epigram definitions to Haskell

```
dumpHaskell :: INTM :=>: VAL :<: TY → ProofState String
dumpHaskell (_:=>: v :<: ty) = do
  v' ← bquoteHere v
  ty' ← bquoteHere ty
  return $ "    ty    = " ++ showHaskell B0 ty' ++
    "\n    def    = " ++ showHaskell B0 v'
```

```
class ShowHaskell a where
  showHaskell :: Bwd String → a → String
```

```
instance ShowHaskell REF where
  showHaskell bs r = fst (mkLastName r)
```

```

instance ShowHaskell (Tm {d,p} REF) where
  showHaskell bs (L s)      = showHaskell bs s
  showHaskell bs (C c)      = showHaskell bs c
  showHaskell bs (NV i)     = "(NV " ++ maybe (show i) id (bs!.i) ++ ")"
  showHaskell bs (N n)      = "(N " ++ showHaskell bs n ++ ")"
  showHaskell bs (P r)      = showHaskell bs r
  showHaskell bs (V i)      = "(V " ++ maybe (show i) id (bs!.i) ++ ")"
  showHaskell bs (op :@ as) = "(" ++ opName op ++ "Op :@" ++ showHaskell bs as ++ ")"
  showHaskell bs (f :$ a)   = "(" ++ showHaskell bs f ++ " :$" ++ showHaskell bs a ++ ")"
  showHaskell bs (t :? ty) = "(" ++ showHaskell bs t ++ " :?" ++ showHaskell bs ty ++ ")"
  showHaskell bs x         = error "showHaskell: can't show " ++ show x

```

```

instance ShowHaskell (Scope p REF) where
  showHaskell bs (x :. t) = "(L $" ++ show x ++ " :. [" ++ x ++ ". " ++
    showHaskell (bs :< x) t ++ "]"
  showHaskell bs (K t)   = "(LK " ++ showHaskell bs t ++ ")"
  showHaskell bs x       = error "showHaskell: can't show " ++ show x

```

```

instance ShowHaskell (Can (Tm {d,p} REF)) where
  showHaskell bs Set      = "SET"
  showHaskell bs (Pi _S (LK _T)) = "(ARR " ++ showHaskell bs _S ++ " " ++
    showHaskell bs _T ++ ")"
  showHaskell bs (Pi _S _T) = "(PI " ++ showHaskell bs _S ++ " " ++
    showHaskell bs _T ++ ")"
  showHaskell bs (Con x)   = "(CON " ++ showHaskell bs x ++ ")"

```

```

showHaskell bs Uld      = "UID"
showHaskell bs (Tag s) = "(TAG " ++ show s ++ ")"
showHaskell bs (EnumT e) = "(ENUMT " ++ showHaskell bs e ++ ")"
showHaskell bs Ze      = "ZE"
showHaskell bs (Su t)  = "(SU " ++ showHaskell bs t ++ ")"
showHaskell bs Unit    = "UNIT"
showHaskell bs Void    = "VOID"
showHaskell bs (Sigma _S _T) = "(SIGMA " ++ showHaskell bs _S ++ " " ++ showHaskell bs _T ++ ")"
showHaskell bs (Pair a b) = "(PAIR " ++ showHaskell bs a ++ " " ++ showHaskell bs b ++ ")"
showHaskell bs Prop    = "PROP"
showHaskell bs (Prf p) = "(PRF " ++ showHaskell bs p ++ ")"
showHaskell bs (All s t) = "(ALL " ++ showHaskell bs s ++ " " ++ showHaskell bs t ++ ")"
showHaskell bs (And p q) = "(AND " ++ showHaskell bs p ++ " " ++ showHaskell bs q ++ ")"
showHaskell bs Trivial = "TRIVIAL"
showHaskell bs Absurd  = "ABSURD"
showHaskell bs (Inh t) = "(INH " ++ showHaskell bs t ++ ")"
showHaskell bs (Wit t) = "(WIT " ++ showHaskell bs t ++ ")"
  -- showHaskell bs (Mu (l := Id x)) = "(MU " ++ showHaskell bs l ++ " " ++ showHaskell bs x ++ ")"
showHaskell bs (IMu (l := (Id ii : & Id x)) i) = "(IMU " ++ showHaskell bs l ++ " " ++ showHaskell bs ii ++ " " ++ showHaskell bs i ++ ")"
showHaskell bs (EqBlue ss tt) = "(EQBLUE " ++ showHaskell bs ss ++ " " ++ showHaskell bs tt ++ ")"
  -- showHaskell bs (Monad s t) = "(MONAD " ++ showHaskell bs s ++ " " ++ showHaskell bs t ++ ")"
  -- showHaskell bs (Return x) = "(RETURN " ++ showHaskell bs x ++ ")"
  -- showHaskell bs (Composite t) = "(COMPOSITE " ++ showHaskell bs t ++ ")"
  -- showHaskell bs (Nu (l := Id t)) = "(NU " ++ showHaskell bs l ++ " " ++ showHaskell bs t ++ ")"
  -- showHaskell bs (CoIt d sty f s) = "(COIT " ++ showHaskell bs d ++ " " ++ showHaskell bs sty ++ " " ++ showHaskell bs f ++ " " ++ showHaskell bs s ++ ")"
showHaskell bs (Label l t) = "(LABEL " ++ showHaskell bs l ++ " " ++ showHaskell bs t ++ ")"
showHaskell bs (LRet t) = "(LRET " ++ showHaskell bs t ++ ")"
  -- showHaskell bs (Quotient x r p) = "(QUOTIENT " ++ showHaskell bs x ++ " " ++ showHaskell bs r ++ " " ++ showHaskell bs p ++ ")"
  -- showHaskell bs RSig = "RSIG"
  -- showHaskell bs REmpty = "REMPY"
  -- showHaskell bs (RCons s i t) = "(RCONS " ++ showHaskell bs s ++ " " ++ showHaskell bs i ++ " " ++ showHaskell bs t ++ ")"
  -- showHaskell bs (Record (l := Id s)) = "(RECORD " ++ showHaskell bs l ++ " " ++ showHaskell bs s ++ ")"
showHaskell bs x      = error "showHaskell: can't show " ++ show x

```

instance ShowHaskell (Elim (Tm {d, p} REF)) **where**

```

showHaskell bs (A a) = "(A " ++ showHaskell bs a ++ ")"
showHaskell bs Out   = "Out"
showHaskell bs Fst   = "Fst"
showHaskell bs Snd   = "Snd"
showHaskell bs (Call l) = "(CALL " ++ showHaskell bs l ++ ")"
showHaskell bs x      = error "showHaskell: can't show " ++ show x

```

instance ShowHaskell a ⇒ ShowHaskell [a] **where**

```

showHaskell bs xs = "[" ++ intercalate ", " (map (showHaskell bs) xs) ++ "]"

```

instance ShowHaskell a ⇒ ShowHaskell (Maybe a) **where**

```

showHaskell bs Nothing = "Nothing"
showHaskell bs (Just x) = "(Just " ++ showHaskell bs x ++ ")"

```

instance (ShowHaskell a, ShowHaskell b) ⇒ ShowHaskell (a >: b) **where**

```

showHaskell bs (ty >: t) = "(" ++ showHaskell bs ty ++ " >: " ++ showHaskell bs t ++ ")"

```

```

import → CochonTactics where
  : unaryExCT "haskell" (λt → elabInfer' t >>= dumpHaskell)
  "haskell - renders an Epigram term as a Haskell definition."

```

5.9 Unification

5.9.1 Solving flex-rigid problems

The *solveHole* command solves a flex-rigid problem by filling in the reference (which must be a hole) with the given term, which must contain no defined references. It records the current location in the proof state (but not the cursor position) and returns there afterwards.

```

solveHole :: REF → INTM → ProofState (EXTM :=> VAL)
solveHole ref tm = do
  here ← getCurrentName
  r ← solveHole' ref [] tm
  cursorBottom
  goTo here
  return r

```

The *solveHole'* command actually fills in the hole, accumulating a list of dependencies (references the solution depends on) as it passes them. It moves the dependencies to before the hole by creating new holes earlier in the proof state and inserting a news bulletin that solves the old dependency holes with the new ones.

```

solveHole' :: REF → [(REF, INTM)] → INTM → ProofState (EXTM :=>: VAL)
solveHole' ref@(name := HOLE _:<: _) deps tm = do
  es ← getEntriesAbove
  case es of
    B0 → goOutBelow >> cursorUp >> solveHole' ref deps tm
  _:<: e → pass e
where
  pass :: Entry Bwd → ProofState (EXTM :=>: VAL)
  pass (EDEF def@(defName := _) _ _ _ _ _)
  | name ≡ defName ∧ occurs def = throwError' $
    err "solveHole: you can't define something in terms of itself!"
  | name ≡ defName = do
    cursorUp
    news ← makeDeps deps []
    cursorDown
    goIn
    putNewsBelow news
    let (tm', _) = tellNews news tm
    tm'' ← bquoteHere (evTm tm')
    giveOutBelow tm''
  | occurs def = do
    goIn
    ty :=>: _ ← getGoal "solveHole"
    solveHole' ref ((def, ty) : deps) tm
  | otherwise = goIn >> solveHole' ref deps tm
  pass (EPARAM param _ _ _ _)
  | occurs param = throwError' $
    err "solveHole: param" † errRef param † err "occurs illegally."
  | otherwise = cursorUp >> solveHole' ref deps tm
  pass (EModule modName _) = goIn >> solveHole' ref deps tm
  occurs :: REF → Bool
  occurs ref = any (≡ ref) tm ∨ ala M.Any foldMap (any (≡ ref) · snd) deps

makeDeps :: [(REF, INTM)] → NewsBulletin → ProofState NewsBulletin
makeDeps [] news = return news
makeDeps ((name := HOLE k:<: tyv, ty) : deps) news = do
  let (ty', _) = tellNews news ty
  makeKinded Nothing k (fst (last name) :<: ty')
  EDEF ref _ _ _ _ ← getEntryAbove
  makeDeps deps ((name := DEFN (NP ref) :<: tyv, GoodNews) : news)
makeDeps _ _ = throwError' $ err "makeDeps: bad reference kind! Perhaps "
  † err "solveHole was called with a term containing unexpanded definitions?"

solveHole' ref _ _ = throwError' $ err "solveHole:" † errRef ref
  † err "is not a hole."

```

Adam: where should this live?

```

stripShared :: NEU → ProofState REF
stripShared n = getInScope >>= stripShared' n
  where
    stripShared' :: NEU → Entries → ProofState REF
    stripShared' (P ref@(_ := HOLE Hoping :<: _)) B0 = return ref
    stripShared' (n : $ A (NP r)) (es :<: EPARAM paramRef _ _ _ _)
      | r ≡ paramRef = stripShared' n es
    stripShared' n (es :<: EDEF _ _ _ _ _ _) = stripShared' n es
    stripShared' n (es :<: EModule _ _) = stripShared' n es
    stripShared' n es = do
      -- proofTrace $ "stripShared: fail on " ++ show n
      throwError' $ err "stripShared: fail on" ++ errVal (N n)

```

5.10 Matching

A *matching substitution* is a list of references with their values, if any.

```
type MatchSubst = Bwd (REF, Maybe VAL)
```

It is easy to decide if a reference is an element of such a substitution:

```

elemSubst :: REF → MatchSubst → Bool
elemSubst r = any ((r ≡) · fst)

```

When inserting a new reference-value pair into the substitution, we ensure that it is consistent with any value already given to the reference.

```

insertSubst :: REF → VAL → StateT MatchSubst ProofState ()
insertSubst x t = get >>= flip help F0
  where
    help :: MatchSubst → Fwd (REF, Maybe VAL) → StateT MatchSubst ProofState ()
    help B0 fs = error "insertSubst: reference not found!"
    help (rs :<: (y, m)) fs | x ≡ y = case m of
      Nothing → put (rs :<: (x, Just t) <>< fs)
      Just u → do
        guard ≡≡ (lift $ withNSupply (equal (pty x :>: (t, u))))
        put (rs :<: (y, m) <>< fs)
    help (rs :<: (y, m)) fs = help rs ((y, m) :> fs)

```

The matching commands, defined below, take a matching substitution (initially with no values for the references) and a pair of objects. The references must only exist in the first object, and each reference may only depend on those before it (in the usual telescopic style). Each command will produce an updated substitution, potentially with more references defined.

Note that the resulting values may contain earlier references that need to be substituted out. Any references left with no value at the end are unconstrained by the matching problem.

The *match Value* command requires the type of the values to be pushed in. It expands elements of Π -types by applying them to fresh references, which must not occur in solution values (this might otherwise happen when given a higher-order matching problem with no solutions). The fresh references are therefore collected in a list and *checkSafe* (defined below) is called to ensure none of the unsafe references occur.

Adam: This is broken, because it assumes all eliminators are injective (including projections). If you do something too complicated, it may end up matching references with things of the wrong type. A cheap improvement would be to check types before calling *insertSubst*, thereby giving a sound but incomplete matching algorithm. Really we should do proper higher-order matching.

```

matchValue :: Bwd REF → TY :> (VAL, VAL) → StateT MatchSubst ProofState ()
matchValue zs (ty :> (NP x, t)) = do
  rs ← get
  if x 'elemSubst' rs
  then lift (checkSafe zs t) >> insertSubst x t
  else matchValue' zs (ty :> (NP x, t))
matchValue zs tvv = matchValue' zs tvv

```

```

matchValue' :: Bwd REF → TY :> (VAL, VAL) → StateT MatchSubst ProofState ()
matchValue' zs (PI s t :> (v, w)) = do
  rs ← get
  rs' ← lift $ freshRef ("expand" :<: s) $ λsRef → do
    let sv = pval sRef
    execStateT (matchValue (zs :<: sRef) (t $$ A sv :> (v $$ A sv, w $$ A sv))) rs
  put rs'

```

```

matchValue' zs (C cty :> (C cs, C ct)) = case halfZip cs ct of
  Nothing → throwError' $ err "matchValue: unmatched constructors!"
  Just cst → do
    (mapStateT $ mapStateT $ liftError'
      (λv → convertErrorVALs (fmap fst v)))
      (canTy (chevMatchValue zs) (cty :>: cst))
    return ()

```

```

matchValue' zs (_ :> (N s, N t)) = matchNeutral zs s t >> return ()

```

```

matchValue' zs tvv = guard <<< (lift $ withNSupply $ equal tvv)

```

```

chevMatchValue :: Bwd REF → TY :> (VAL, VAL) →
  StateT MatchSubst (ProofStateT (VAL, VAL)) (() :=>: VAL)
chevMatchValue zs tvv@(_ :>: (v, _)) = do
  (mapStateT $ mapStateT $ liftError' (error "matchValue: unconvertable error!"))
  $ matchValue zs tvv
  return (() :=>: v)

```

The *matchNeutral* command matches two neutrals, and returns their type along with the matching substitution.

```

matchNeutral :: Bwd REF → NEU → NEU → StateT MatchSubst ProofState TY
matchNeutral zs (P x) t = do
  rs ← get
  if x 'elemSubst' rs
  then do
    lift $ checkSafe zs (N t)
    insertSubst x (N t)
    return (pty x)
  else matchNeutral' zs (P x) t
matchNeutral zs a b = matchNeutral' zs a b

```

```

matchNeutral' :: Bwd REF → NEU → NEU → StateT MatchSubst ProofState TY
matchNeutral' zs (P x) (P y) | x ≡ y = return (pty x)
matchNeutral' zs (f : $ e) (g : $ d) = do
  C ty ← matchNeutral zs f g
  case halfZip e d of
    Nothing → throwError' $ err "matchNeutral: unmatched eliminators!"
    Just ed → do
      (←, ty') ← (mapStateT $ mapStateT $ liftError' (error "matchNeutral: unconvertable error!")) $ do
        return ty'
matchNeutral' zs (fOp : @ as) (gOp : @ bs) | fOp ≡ gOp = do
  (←, ty) ← (mapStateT $ mapStateT $ liftError' (error "matchNeutral: unconvertable error!")) $ do
    return ty
matchNeutral' zs a b = throwError' $ err "matchNeutral: unmatched "
  ++ errVal (N a) ++ err "and" ++ errVal (N b)

```

As noted above, fresh references generated when expanding Π -types must not occur as solutions to matching problems. The `checkSafe` function throws an error if any of the references occur in the value.

```

checkSafe :: Bwd REF → VAL → ProofState ()
checkSafe zs t | any (∈ t) zs = throwError' $ err "checkSafe: unsafe!"
               | otherwise   = return ()

```

For testing purposes, we define a `match` tactic that takes a telescope of parameters to solve for, a neutral term for which those parameters are in scope, and another term of the same type. It prints out the resulting substitution.

```

import → CochonTacticsCode where
matchCTactic :: [(String, DInTmRN)] → DExTmRN → DInTmRN → ProofState String
matchCTactic xs a b = draftModule "__match" $ do
  rs ← traverse matchHyp xs
  (← :=>: av :<: ty) ← elabInfer' a
  cursorTop
  (← :=>: bv) ← elaborate' (ty :=>: b)
  rs' ← runStateT (matchValue B0 (ty :=>: (av, bv))) (bwdList rs)
  return (show rs')
where
  matchHyp :: (String, DInTmRN) → ProofState (REF, Maybe VAL)
  matchHyp (s, t) = do
    tt ← elaborate' (SET :=>: t)
    r ← assumeParam (s :<: tt)
    return (r, Nothing)

```

```

import → CochonTactics where
: (simpleCT
  "match"
  (do
    pars ← tokenListArgs (bracket Round $ tokenPairArgs
      tokenString
      (keyword KwAsc)
      tokenInTm) (| () |)
    keyword KwSemi
    tm1 ← tokenExTm
    keyword KwSemi
    tm2 ← tokenInTm
    return (B0 :< pars :< tm1 :< tm2)
  )
  (λ[pars, ExArg a, InArg b] →
    matchCTactic (argList (argPair argToStr argToIn) pars) a b)
  "match [<para>]* ; <term> ; <term> - match parameters in first term against
  )

```

Chapter 6

Elaboration

6.1 ElabProb: syntactic representation of elaboration problems

An `ElabProb` is a syntactic representation of an elaboration problem. Examples include elaborating a particular piece of display syntax into an evidence term and waiting for something to happen before elaboration can proceed. Crucially, an elaboration problem can be suspended by storing it in the proof state. This allows it to be left alone until more progress can be made (e.g. because it has been updated with news, or because the scheduler is free to make some non-local change to the proof state).

It caches the value representations of terms it contains. **Adam:** Is this optimisation actually worth the complication?

Pierre: I don't understand `ElabSchedule`. **Adam:** The idea is that `ElabSchedule` makes the scheduler work on other problems before coming back to work on this one. It doesn't seem to be used at the moment, so perhaps we can get rid of it. This data type may need to be redesigned to allow solution of hoping holes earlier in the elaboration process.

```
data ElabProb x
  = ElabDone (InTm x :=>: Maybe VAL)
    -- succeed with given term
  | ElabHope
    -- hope for a solution to turn up
  | ElabProb (DInTm x RelName)
    -- elaborate In display term
  | ElabInferProb (DExTm x RelName)
    -- elaborate and infer type of Ex display term
  | WaitCan (InTm x :=>: Maybe VAL) (ElabProb x)
    -- wait for value to become canonical
  | WaitSolve x (InTm x :=>: Maybe VAL) (ElabProb x)
    -- wait for reference to be solved with term
  | ElabSchedule (ElabProb x)
    -- kick off the scheduler
```

`ElabProb` is a traversable functor, parameterised by the type of references, which are typically `REFs`. Note that traversal will discard the cached values, but this is okay because the terms need to be re-evaluated after they have been updated anyway.

```
type EProb = ElabProb REF
```

An elaboration problem is said to be *unstable* if the scheduler can make progress on it, and *stable* if not. At present, the only kind of stable elaboration problem is waiting for a non-canonical term to become canonical.

Pierre: At this stage, the notion of scheduler has not been introduced. Therefore, stable or unstable doesn't speak to me. What is the scheduler doing? **Adam:** We need a general introduction to elaboration, explaining how all the various components fit together.

```

isUnstable :: EProb → Bool
isUnstable (ElabDone _)           = True
isUnstable ElabHope               = True
isUnstable (ElabProb _)          = True
isUnstable (ElabInferProb _)     = True
isUnstable (WaitCan (_ :=>: Just (C _)) _) = True
isUnstable (WaitCan (tm :=>: Nothing) _) | C _ ← evTm tm = True
isUnstable (WaitCan _ _)         = False
isUnstable (WaitSolve _ _ _)     = True
isUnstable (ElabSchedule _)      = True

```

Since `ElabProb` caches value representations of its terms, we define some handy functions for producing and manipulating these.

```

justEval :: INTM :=>: VAL → INTM :=>: Maybe VAL
justEval (tm :=>: v) = tm :=>: Just v
maybeEval :: INTM :=>: Maybe VAL → INTM :=>: VAL
maybeEval (tm :=>: Just v) = tm :=>: v
maybeEval (tm :=>: Nothing) = tm :=>: evTm tm

```

6.2 The `Elab` monad: a DSL for elaboration

Because writing elaborators is a tricky business, we would like to have a domain-specific language to write them with. We use the following set of instructions to define a monad that follows the syntax of this language, then write an interpreter to run the syntax in the `ProofState` monad.

```

eLambda    :: String → Elab REF
            -- create a λ and return its REF
eGoal      :: Elab TY
            -- return the type of the goal
eWait      :: String → TY → Elab (EXTM :=>: VAL)
            -- create a subgoal corresponding to a question mark
eCry       :: StackError DInTmRN → Elab a
            -- give up with an error
eElab      :: Loc → EProb → Elab a
            -- solve a suspendable elaboration problem and return the result
eCompute   :: (TY :=>: Elab (INTM :=>: VAL)) → Elab (INTM :=>: VAL)
            -- execute commands to produce an element of a given type
eFake      :: Elab (REF, Spine {TT} REF)
            -- return a fake reference to the current goal and the current spine
eResolve   :: RelName → Elab (INTM :=>: VAL, Maybe (Scheme INTM))
            -- resolve a name to a term and maybe a scheme
eAskNSupply :: Elab NameSupply
            -- return a fresh name supply

```

The instruction signature given above is implemented using the following monad.

```

data Elab x
  = EReturn x
  | ELambda String (REF → Elab x)
  | EGoal (TY → Elab x)
  | EWait String TY (EXTM :=>: VAL → Elab x)
  | ECry (StackError DInTmRN)
  | EElab Loc EProb
  | ECompute (TY :=>: Elab (INTM :=>: VAL)) (INTM :=>: VAL → Elab x)
  | EFake ((REF, Spine { TT } REF) → Elab x)
  | EAnchor (String → Elab x)
  | EResolve RelName ((INTM :=>: VAL, Maybe (Scheme INTM)) → Elab x)
  | EAskNSupply (NameSupply → Elab x)

```

Now we can define the instructions we wanted:

```

eLambda      = flip ELambda EReturn
eGoal        = EGoal EReturn
eWait x ty   = EWait x ty EReturn
eCry         = ECry
eElab loc p  = EElab loc p
eCompute     = flip ECompute EReturn
eFake        = EFake EReturn
eAnchor      = EAnchor EReturn
eResolve     = flip EResolve EReturn
eAskNSupply = EAskNSupply EReturn

```

```

eFaker :: Elab (EXTM :=>: VAL)
eFaker = do
  (r, sp) ← eFake
  let t = (P r) $ : $ sp
  return (t :=>: evTm t)

```

We will eventually need to keep track of which elaboration problems correspond to which source code locations. For the moment, Locs are just ignored.

```

newtype Loc = Loc Int deriving Show

```

6.3 Using the Elab language

6.3.1 Tools for writing elaborators

The *eCan* instruction asks for the current goal to be solved by the given elaboration problem when the supplied value is canonical.

```

eCan :: INTM :=>: VAL → EProb → Elab a
eCan ( _ :=>: C _ ) prob = eElab (Loc 0) prob
eCan tt                prob = eElab (Loc 0) (WaitCan (justEval tt) prob)

```

We can type-check a term using the *eCheck* instruction.

```

eCheck :: (TY :=>: INTM) → Elab (INTM :=>: VAL)
eCheck tytm = do
  nsupply ← eAskNSupply
  case liftError DTIN (typeCheck (check tytm) nsupply) of
    Left e  → throwError e
    Right tt → return tt

```

The *eCoerce* instruction attempts to coerce a value from the first type to the second type, either trivially (if the types are definitionally equal) or by hoping for a proof of the appropriate equation and inserting a coercion.

```

eCoerce :: INTM :=> VAL → INTM :=> VAL → INTM :=> VAL → Elab (INTM :=> VAL)
eCoerce (_S :=> _Sv) (_T :=> _Tv) (s :=> sv) = do
  eq ← eEqual $ SET :=> (_Sv, _Tv)
  if eq
  then return (s :=> sv)
  else do
    q :=> qv ← eHopeFor $ PRF (EQBLUE (SET :=> _Sv) (SET :=> _Tv))
    return $ N (coe : @ [_S, _T, q, s]) :=> coe @@ [_Sv, _Tv, qv, sv]

```

The *eEqual* instruction determines if two types are definitionally equal.

```

eEqual :: (TY :=> (VAL, VAL)) → Elab Bool
eEqual tyvv = do
  nsupply ← eAskNSupply
  return (equal tyvv nsupply)

```

The *eHope* instruction hopes that the current goal can be solved.

```

eHope :: Elab a
eHope = eElab (Loc 0) ElabHope

```

The *eHopeFor* instruction hopes for an element of a type.

```

eHopeFor :: TY → Elab (INTM :=> VAL)
eHopeFor ty = eCompute (ty :=> eHope)

```

The *eInfer* instruction infers the type of an evidence term.

```

eInfer :: EXT M → Elab (INTM :=> VAL)
eInfer tm = do
  nsupply ← eAskNSupply
  case liftError DTIN (typeCheck (infer tm) nsupply) of
  Left e → throwError e
  Right (tv <: ty) → do
    ty' :=> _ ← eQuote ty
    return $ PAIR ty' (N tm) :=> PAIR ty tv

```

The *eQuote* instruction β -quotes a value to produce a term representation.

```

eQuote :: VAL → Elab (INTM :=> VAL)
eQuote v = do
  nsupply ← eAskNSupply
  return (bquote B0 v nsupply :=> v)

```

The *eSchedule* instruction asks for the scheduler to deal with other problems before returning its result.

```

eSchedule :: (INTM :=> VAL) → Elab a
eSchedule (tm :=> tv) = eElab (Loc 0) (ElabSchedule (ElabDone (tm :=> Just tv)))

```

6.3.2 Elaborating DInTms

We use the `Elab` language to describe how to elaborate a display term to produce an evidence term. The `makeElab` and `makeElabInfer` functions read a display term and use the capabilities of the `Elab` monad to produce a corresponding evidence term.

When part of the display syntax needs to be elaborated as a subproblem, we call `subElab` or `subElabInfer` rather than `makeElab` or `makeElabInfer` to ensure that elaboration does not take place at the top level. This means that if the subproblem needs to modify the proof state (for example, to introduce a λ) it will create a new definition to work in. It also ensures that the subproblem can terminate with the `eElab` instruction, providing a syntactic representation.

```
subElab :: Loc → (TY :>: DInTmRN) → Elab (INTM :=>: VAL)
subElab loc (ty :>: tm) = eCompute (ty :>: makeElab loc tm)
```

```
subElabInfer :: Loc → DEXtMmRN → Elab (INTM :=>: VAL)
subElabInfer loc tm = eCompute (sigSetVAL :>: makeElabInfer loc tm)
```

Since we frequently pattern-match on the goal type when elaborating `In` terms, we abstract it out. Thus `makeElab'` actually implements elaboration.

```
makeElab :: Loc → DInTmRN → Elab (INTM :=>: VAL)
makeElab loc tm = makeElab' loc · (:>:tm) ≪≪ eGoal
```

```
makeElab' :: Loc → (TY :>: DInTmRN) → Elab (INTM :=>: VAL)
-- import {- MakeElabRules
-- [Feature = Enum]
makeElab' loc (PI (ENUMT e) t :>: m) | isTuply m = do
  t' :=>: _ ← eQuote t
  e' :=>: _ ← eQuote e
  tm :=>: tmv ← subElab loc (branchesOp @@ [e, t] :>: m)
  x ← eLambda (fortran t)
  return $ N (switchOp : @ [e', NP x, t', tm])
    :=>: switchOp @@ [e, NP x, t, tmv]
where
  isTuply :: DInTmRN → Bool
  isTuply DVOID      = True
  isTuply (DPAIR _ _) = True
  isTuply _          = False
```

To elaborate a tag with an enumeration as its type, we search for the tag in the enumeration to determine the appropriate index.

```
makeElab' loc (ENUMT t :>: DTAG a) = findTag a t 0
where
  findTag :: String → TY → Int → Elab (INTM :=>: VAL)
  findTag a (CONSE (TAG b) t) n
    | a ≡ b      = return (toNum n :=>: toNum n)
    | otherwise = findTag a t (succ n)
  findTag a _ n = throwError' · err $ "elaborate: tag ` "
    ++ a
    ++ " not found in enumeration."
  toNum :: Int → Tm {In, p} x
  toNum 0 = ZE
  toNum n = SU (toNum (n - 1))
```


We push types in to neutral terms by calling *subElabInfer* on the term, then coercing the result to the required type. (Note that *eCoerce* will check if the types are equal, and if so it will not insert a redundant coercion.)

```
makeElab' loc (w :>: DN n) = do
  w' :=>: _ ← eQuote w
  tt ← subElabInfer loc n
  let (yt :=>: yn <: ty :=>: tyv) = extractNeutral tt
  eCoerce (ty :=>: tyv) (w' :=>: w) (yt :=>: yn)
```

If we already have an evidence term, we just type-check it. This allows elaboration code to partially elaborate a display term then embed the resulting evidence term and call the elaborator again.

```
makeElab' loc (ty :>: DTIN tm) = eCheck (ty :>: tm)
```

If the type is neutral and none of the preceding cases match, there is nothing we can do but wait for the type to become canonical.

```
makeElab' loc (N ty :>: tm) = do
  tt ← eQuote (N ty)
  eCan tt (ElabProb tm)
```

If nothing else matches, give up and report an error.

```
makeElab' loc (ty :>: tm) = throwError' $ err "makeElab: can't push"
  ++ errTyVal (ty <: SET) ++ err "into" ++ errTm tm
```

6.3.3 Elaborating DEXtms

The *makeElabInfer* command is to *infer* in subsection 2.5.3 as *makeElab* is to *check*. It elaborates the display term and infers its type to produce a type-term pair in the evidence language.

The result of *makeElabInfer* is of type $(X : \text{SET}) \times X$, which we can represent as an evidence term or value (*sigSetTM* or *sigSetVAL*, respectively).

```
sigSetVAL :: Tm {ln, p} x
sigSetVAL = SIGMA SET (idVAL "ssv")
```

```
sigSetTM :: INTM
sigSetTM = sigSetVAL
```

The *extractNeutral* function separates type-term pairs in both term and value forms. It avoids clutter in the term representation by splitting it up if it happens to be a canonical pair, or applying the appropriate eliminators if not.

```
extractNeutral :: INTM :=>: VAL → INTM :=>: VAL <: INTM :=>: TY
extractNeutral (PAIR ty tm :=>: PAIR tyv tmv) = tm :=>: tmv <: ty :=>: tyv
extractNeutral (PAIR ty tm :=>: tv) = tm :=>: tv $$ Snd <: ty :=>: tv $$ Fst
extractNeutral (tm :=>: tv) = N (tm' : $ Snd) :=>: tv $$ Snd <: N (tm' : $ Fst) :=>: tv $$ Fst
  where tm' = tm ?? sigSetTM
```

Since we use a head-spine representation for display terms, we need to elaborate the head of an application. The *makeElabInferHead* function uses the *Elab* monad to produce a type-term pair for the head, and provides its scheme (if it has one) for argument synthesis. The head may be a parameter, which is resolved; an embedded evidence term, which is checked; or a type annotation, which is converted to the identity function at the given type.

```

makeElabInferHead :: Loc → DHEAD → Elab (INTM :=>: VAL, Maybe (Scheme INTM))
makeElabInferHead loc (DP rn) = eResolve rn
makeElabInferHead loc (DTEX tm) = (| (eInfer tm), ~Nothing |)
makeElabInferHead loc (DType ty) = do
  tm :=>: v ← subElab loc (SET :=>: ty)
  return (typeAnnotTM tm :=>: typeAnnotVAL v, Nothing)
where
  typeAnnotTM :: INTM → INTM
  typeAnnotTM tm = PAIR (ARR tm tm) (idTM "typeAnnot")
  typeAnnotVAL :: VAL → VAL
  typeAnnotVAL v = PAIR (ARR v v) (idVAL "typeAnnot")

```

Now we can implement *makeElabInfer*. We use *makeElabInferHead* to elaborate the head of the neutral term, then call *handleArgs* or *handleSchemeArgs* to process the spine of eliminators.

```

makeElabInfer :: Loc → DEXtMRN → Elab (INTM :=>: VAL)
makeElabInfer loc (t $ss) = do
  (tt, ms) ← makeElabInferHead loc t
  let (tm :=>: tmv <: ty :=>: tyv) = extractNeutral tt
  case ms of
    Just sch → handleSchemeArgs B0 sch (tm ?? ty :=>: tmv <: tyv) ss
    Nothing → handleArgs (tm ?? ty :=>: tmv <: tyv) ss
  where

```

The *handleSchemeArgs* function takes a list of terms (corresponding to de Bruijn-indexed variables), the scheme, term and type of the neutral, and a spine of eliminators in display syntax. It elaborates the eliminators and applies them to the neutral term, using the scheme to handle insertion of implicit arguments.

```

handleSchemeArgs :: Bwd (INTM :=>: VAL) → Scheme INTM →
  EXTM :=>: VAL <: TY → DSPINE → Elab (INTM :=>: VAL)

```

If the scheme is just a type, then we call on the non-scheme *handleArgs*.

```

handleSchemeArgs es (SchType _) ttt as = handleArgs ttt as

```

If the scheme has an implicit Π -binding, then we hope for a value of the relevant type and carry on. Note that we evaluate the type of the binding in the context *es*.

```

handleSchemeArgs es (SchImplicitPi (x <: s) schT)
  (tm :=>: tv <: PI sd t) as = do
  stm :=>: sv ← eHopeFor (eval s (fmap valueOf es, []))
  handleSchemeArgs (es <: (stm :=>: sv)) schT
  (tm : $ A stm :=>: tv $$ A sv <: t $$ A sv) as

```

If the scheme has an explicit Π -binding and we have an argument, then we can push the expected type into the argument and carry on. **question: Does this case need to be modified for higher-order schemes?**

```

handleSchemeArgs es (SchExplicitPi (x <: schS) schT)
  (tm :=>: tv <: PI sd t) (A a : as) = do
  let s' = schemeToInTm schS
  atm :=>: av ← subElab loc (eval s' (fmap valueOf es, [])) :=>: a
  handleSchemeArgs (es <: (atm :=>: av)) schT
  (tm : $ A atm :=>: tv $$ A av <: t $$ A av) as

```

If the scheme has an explicit Π -binding, but we have no more eliminators, then we go under the binder and continue processing the scheme in order to insert any implicit arguments that might be there. We then have to reconstruct the overall type-term pair from the result.

```

handleSchemeArgs es (SchExplicitPi (x <: schS) schT)
  (tm :=>: tv <: PI sd t) [] = do
  let sv = eval (schemeToInTm schS) (fmap valueOf es, [])
  tm :=>: tv ← eCompute
  (PI sv (L $ K sigSetVAL) :=>: do
    r ← eLambda x
    let rt = NP r
    handleSchemeArgs (es <: (rt :=>: rt)) schT
    (tm : $ A rt :=>: tv $$ A rt <: t $$ A rt) []
  )
  s' :=>: _ ← eQuote sv
  let atm = tm ?? PIV x s' sigSetTM : $ A (NV 0)
  rtm = PAIR (PIV x s' (N (atm : $ Fst))) (LAV x (N (atm : $ Snd)))
  return $ rtm :=>: evTm rtm

```

Otherwise, we probably have a scheme with an explicit Π -binding but an eliminator other than application, so we give up and throw an error.

```

handleSchemeArgs es sch (_ :=>: v <: ty) as = throwError' $
  err "handleSchemeArgs: cannot handle scheme" ++ errScheme sch ++
  err "with neutral term" ++ errTyVal (v <: ty) ++
  err "and eliminators" ++ map ErrorElim as

```

The *handleArgs* function is a simplified version of *handleSchemeArgs*, for neutral terms without schemes. It takes a typed neutral term and a spine of eliminators in display syntax, and produces a set-value pair in the *Elab* monad.

```

handleArgs :: (EXTM :=>: VAL <: TY) → DSPINE → Elab (INTM :=>: VAL)

```

If we have run out of eliminators, then we just give back the neutral term with its type.

```

handleArgs (tm :=>: tv <: ty) [] = do
  ty' :=>: _ ← eQuote ty
  return $ PAIR ty' (N tm) :=>: PAIR ty tv

```

If we have a term of a labelled type being eliminated with *Call*, we need to attach the appropriate label to the call and continue with the returned type.

```

handleArgs (t :=>: v <: LABEL l ty) (Call _ : as) = do
  l' :=>: _ ← eQuote l
  handleArgs (t : $ Call l' :=>: v $$ Call l <: ty) as

```

For all other eliminators, assuming the type is canonical we can use *elimTy*.

```

handleArgs (t :=>: v <: C cty) (a : as) = do
  (a', ty') ← elimTy (subElab loc) (v <: cty) a
  handleArgs (t : $ fmap termOf a' :=>: v $$ fmap valueOf a' <: ty') as

```

Otherwise, we cannot do anything apart from waiting for the type to become canonical, so we suspend elaboration and record the current problem.

```

handleArgs (tm :=>: tv <: ty) as = do
  tt ← eQuote ty
  eCan tt (ElabInferProb (DTEX tm :$as))

```

6.4 Implementing the Elab monad

6.4.1 Running elaboration processes

The `runElab` proof state command actually interprets an Elab x in the proof state. In other words, we define here the semantics of the Elab syntax.

```
runElab :: WorkTarget → (TY :>: Elab (INTM :=>: VAL)) →  
        ProofState (INTM :=>: VAL, ElabStatus)
```

This command is given a type and a program in the Elab DSL for creating an element of that type. It is also given a flag indicating whether elaboration is working on the current goal in the proof state. If the target is the current goal, the type pushed in must match the type of the goal.

```
data WorkTarget = WorkCurrentGoal | WorkElsewhere
```

It will return the term produced by elaboration, and a flag indicating whether elaboration was completely successful or had to suspend. If elaboration suspended when working on the current goal, the term will be a reference to that goal, so it cannot be solved.

```
data ElabStatus = ElabSuccess | ElabSuspended deriving Eq
```

Some Elab instructions can only be used when working on the current goal, for example introducing a lambda. We identify these so we can make a new subgoal when trying to execute them elsewhere.

```
currentGoalOnly :: Elab x → Bool  
currentGoalOnly (ELambda _ _) = True  
currentGoalOnly (ECry _)      = True  
currentGoalOnly (EFake _)     = True  
currentGoalOnly (EAnchor _)   = True  
currentGoalOnly _             = False
```

Now, let us give the semantics of each command in turn. First of all, we catch all commands that can only run on the current goal, and redirect them to the specially crafted `runElabNewGoal`.

```
runElab WorkElsewhere (ty :>: elab) | currentGoalOnly elab = runElabNewGoal (ty :>: elab)
```

`EReturn` is the `return` of the monad. It does nothing and always succeeds.

```
runElab _ (_ :>: EReturn x) = return (x, ElabSuccess)
```

`ELambda` creates a λ -parameter, if this is allowed by the type we are elaborating to.

```
runElab WorkCurrentGoal (ty :>: ELambda x f) = case lambdaable ty of  
  Just (_, s, tyf) → do  
    ref ← lambdaParam x  
    runElab WorkCurrentGoal (tyf (NP ref) :>: f ref)  
  Nothing → throwError' $ err "runElab: type" ++ errTyVal (ty :<: SET)  
    ++ err "is not lambdaable!"
```

`EGoal` retrieves the current goal and passes it to the elaboration task.

```
runElab wrk (ty :>: EGoal f) = runElab wrk (ty :>: f ty)
```

`EWait` makes a Waiting hole and pass it along to the next elaboration task.

```

runElab wrk (ty :>: EWait s tyWait f) = do
  tyWait' ← bquoteHere tyWait
  tt ← make (s :<: tyWait')
  runElab wrk (ty :>: f tt)

```

EElab contains a syntactic representation of an elaboration problem. This representation is interpreted and executed by `runElabProb`.

```

runElab wrk (ty :>: EElab l p) = runElabProb wrk l (ty :>: p)

```

ECompute allows us to combine elaboration tasks: we run a first task and pass its result to the next elaboration task.

```

runElab top (ty :>: ECompute (tyComp :>: elab) f) = do
  (e, _) ← runElab WorkElsewhere (tyComp :>: elab)
  runElab top (ty :>: f e)

```

ECry is used to report an error. It updates the current entry into a crying state.

```

runElab WorkCurrentGoal (ty :>: ECry e) = do
  e' ← distillErrors e
  let msg = show (prettyStackError e')
  mn ← getCurrentName
  elabTrace $ "Hole " ++ showName mn ++ " started crying:\n" ++ msg
  putHoleKind (Crying msg)
  t :=>: tv ← getCurrentDefinition
  return (N t :=>: tv, ElabSuspended)

```

EFake extracts the reference of the current entry and presents it as a fake reference. **Pierre**: This is an artifact of our current implementation of labels, this should go away when we label high-level objects with high-level names.

```

runElab WorkCurrentGoal (ty :>: EFake f) = do
  r ← getFakeRef
  inScope ← getInScope
  runElab WorkCurrentGoal · (ty :>:) $ f (r, paramSpine inScope)

```

EAnchor extracts the name of the current entry.

```

runElab WorkCurrentGoal (ty :>: EAnchor f) = do
  name ← getCurrentName
  runElab WorkCurrentGoal · (ty :>:) $ f (fst (last name))

```

EResolve provides a name-resolution service: given a relative name, it finds the term and potentially the scheme of the definition the name refers to. This is passed onto the next elaboration task.

```

runElab wrk (ty :>: EResolve rn f) = do
  (ref, as, ms) ← resolveHere rn
  let tm = P ref $ : $ toSpine as
      ms' = (| (flip applyScheme as) ms |)
  (tmv :<: tyv) ← inferHere tm
  tyv' ← bquoteHere tyv
  runElab wrk (ty :>: f (PAIR tyv' (N tm) :=>: PAIR tyv tmv, ms'))

```

EAskNSupply gives access to the name supply to the next elaboration task.



Read-only name supply. As often, we are giving here a read-only access to the name supply. Therefore, we must be careful not to let some generated names dangling into some definitions, or clashes could happen. □

```
runElab wrk (ty :>: EAskNSupply f) = do
  nsupply ← askNSupply
  runElab wrk (ty :>: f nsupply)
```

As mentioned above, some commands can only be used when elaboration is taking place in the current goal. This is the purpose of *runElabNewGoal*: it creates a dummy definition and hands back the elaboration task to *runElab*.

```
runElabNewGoal :: (TY :>: Elab (INTM :=>: VAL)) → ProofState (INTM :=>: VAL, ElabStatus)
runElabNewGoal (ty :>: elab) = do
  -- Make a dummy definition
  ty' ← bquoteHere ty
  x ← pickName "h" " "
  make (x :<: ty')
  -- Enter its development
  goIn
  (tm :=>: tmv, status) ← runElab WorkCurrentGoal (ty :>: elab)
  -- Leave the development, one way or the other
  case status of
    ElabSuccess → do
      -- By finalising it
      t ← giveOutBelow tm
      e ← neutralise t
      return (e, ElabSuccess)
    ElabSuspended → do
      -- By leaving it unfinished
      currentDef ← getCurrentDefinition
      e ← neutralise currentDef
      goOut
      return (e, ElabSuspended)
```

6.4.2 Interpreting elaboration problems

The *runElabProb* interprets the syntactic representation of an elaboration problem. In other words, this function defines the semantics of the EProb language.

```
runElabProb :: WorkTarget → Loc → (TY :>: EProb) →
  ProofState (INTM :=>: VAL, ElabStatus)
```

ElabDone *tt* always succeed at returning the given term *tt*.

```
runElabProb wrk loc (ty :>: ElabDone tt) =
  return (maybeEval tt, ElabSuccess)
```

ElabProb *tm* asks for the elaboration of the display term *tm* (for pushed-in terms).

```
runElabProb wrk loc (ty :>: ElabProb tm) =
  runElab wrk (ty :>: makeElab loc tm)
```

ElabInferProb *tm* asks for the elaboration and type inference of the display term *tm* (for pull-out terms).

```
runElabProb wrk loc (ty :>: ElabInferProb tm) =
  runElab wrk (ty :>: makeElabInfer loc tm)
```

`WaitCan tm prob` prevents the interpretation of the elaboration problem `prob` until `tm` is indeed a canonical object. Otherwise, the problem is indefinitely suspended.

Pierre: This fall-through pattern-match reminds me of Duff's devices.

```
runElabProb wrk loc (ty :>: WaitCan (_ :=>: Just (C _)) prob) =
  runElabProb wrk loc (ty :>: prob)
runElabProb wrk loc (ty :>: WaitCan (tm :=>: Nothing) prob) =
  runElabProb wrk loc (ty :>: WaitCan (tm :=>: Just (evTm tm)) prob)
```

The semantics of the `ElabHope` command is specifically given by the `runElabHope` interpreter in Section 6.4.3.

```
runElabProb wrk loc (ty :>: ElabHope) = runElabHope wrk ty
```

Any case that has not matched yet ends in a suspended state: we cannot make progress on it right now. The suspension of an elaboration problem is described in details in Section . Once in a suspended state, an elaboration problem might receive some care from the Scheduler (Section 6.6), which might be able to make some progress.

The following problems are suspended, for different reasons:

- a `WaitCan` command offering an object that is *not* canonical;
- a `WaitSolve` command, because we cannot solve references during elaboration, but the scheduler can do so later; and
- an `ElabSchedule` command, whose purpose is to suspend the current elaboration and call the scheduler.

Pierre: These are 3 different cases, getting suspended for 3 different reasons. Maybe it's ok, but maybe suspension is being abused. If not, there must be a nice way to present suspension that covers these 3 cases.

```
runElabProb top loc (ty :>: prob) = do
  ty' ← bquoteHere ty
  suspendThis top (name prob :<: ty' :=>: ty) prob
where
  name :: EProb → String
  name (WaitCan _ _) = "can"
  name (WaitSolve _ _ _) = "solve"
  name (ElabSchedule _) = "suspend"
  name _ = error "runElabProb: unexpected suspension."
```

6.4.3 Hoping, hoping, hoping

The `runElabHope` command interprets the `ElabHope` instruction, which hopes for an element of a given type. In a few cases, based on the type, we might be able to solve the problem immediately:

- An element of type `UNIT` is `VOID`;
- A proof of a proposition might be found or refined by the propositional simplifier (Section 5.3); and
- The solution of a programming is often an induction hypothesis that is sitting in our context

If these strategies do not match or fail to solve the problem, we just create a hole.

```

runElabHope :: WorkTarget → TY → ProofState (INTM :=>: VAL, ElabStatus)
runElabHope wrk UNIT = return (VOID :=>: VOID, ElabSuccess)
runElabHope wrk (PRF p) = simplifyProof wrk p
runElabHope wrk v@(LABEL (N l) ty) = seekLabel wrk l ty ⊕ lastHope wrk v
runElabHope wrk ty = lastHope wrk ty

```

Hoping for labelled types

If we are looking for a labelled type (e.g. to make a recursive call), we search the hypotheses for a value with the same label.

```

seekLabel :: WorkTarget → NEU → VAL → ProofState (INTM :=>: VAL, ElabStatus)
seekLabel wrk label ty = do
  es ← getInScope
  seekOn es
  where

```

The traversal of the hypotheses is carried by *seekOn*. It searches parameters and hands them to *seekIn*.

```

seekOn B0 = do
  label' ← bquoteHere label
  s ← prettyHere (ty :>: N label')
  proofTrace $ "Failed to resolve recursive call to "
    ++ renderHouseStyle s
  ()
seekOn (es' :< EPARAM param _ ParamLam _ _) =
  seekIn B0 (P param) (pty param) ⊕ seekOn es'
seekOn (es' :< _) = seekOn es'

```

Then, *seekIn* tries to match the label we are looking for with an hypothesis we have found. Recall that a label is a telescope targeting a label, hence we try to peel off this telescope to match the label.

```

seekIn :: Bwd REF → EXT M → VAL → ProofState (INTM :=>: VAL, ElabStatus)

```

On our way to the label, we instantiate the hypotheses with fresh references.

```

seekIn rs tm (PI s t) = freshRef (fortran t :<: s) $ λsRef →
  seekIn (rs :< sRef) (tm : $ A (NP sRef)) (t $$$ A (pval sRef))

```

We might have to go inside branches (essentially finite Π -types).

```

seekIn rs tm (N (op : @ [e, p])) | op ≡ branchesOp =
  freshRef (fortran p :<: e) $ λeRef → do
    e' ← bquoteHere e
    p' ← bquoteHere p
    seekIn (rs :< eRef) (switchOp : @ [e', NP eRef, p', N tm])
      (p $$$ A (pval eRef))

```

We have reached a label! The question is then “is this the one we are looking for?” First we call on the matcher (see section ??) to find values for the fresh references, then we generate a substitution from these values and apply it to the call term.

```

seekIn rs tm (LABEL (N foundLabel) u) = do
  ss ← execStateT (matchNeutral B0 foundLabel label) (fmap (, Nothing) rs)
  (xs, vs) ← processSubst ss
  let c = substitute xs vs (N tm)
  return (c :=>: evTm c, ElabSuccess)

```

If, in our way to the label the peeling fails, then we must give up.

```

seekIn rs tm ty = ()

```

To generate a substitution, we quote the value given to each reference and substitute out the preceding references. If a reference has no value, i.e. it is unconstrained by the matching problem, we hope for a solution. **Adam:** we could do some dependency analysis here to avoid cluttering the proof state with hopes that we don't make use of.

```

processSubst :: MatchSubst → ProofState (Bwd (REF :<: INTM), Bwd INTM)
processSubst B0 = return (B0, B0)
processSubst (rs :< (r, Just t)) = do
  (xs, vs) ← processSubst rs
  ty      ← bquoteHere (pty r)
  tm      ← bquoteHere t
  return (xs :< (r :<: substitute xs vs ty), vs :< substitute xs vs tm)
processSubst (rs :< (r, Nothing)) = do
  (xs, vs) ← processSubst rs
  ty      ← bquoteHere (pty r)
  let ty' = substitute xs vs ty
  (tm :=>: -, -) ← runElabHope WorkElsewhere (evTm ty')
  return (xs :< (r :<: ty'), vs :< tm)

```

Simplifying proofs

Pierre: To be reviewed.

If we are hoping for a proof of a proposition, we first try simplifying it using the propositional simplification machinery.

```

simplifyProof :: WorkTarget → VAL → ProofState (INTM :=>: VAL, ElabStatus)
simplifyProof wrk p = do
  pSimp ← runPropSimplify p
  case pSimp of
    Just (SimplyTrivial prf) → do
      return (prf :=>: evTm prf, ElabSuccess)
    Just (SimplyAbsurd _) → runElab wrk (PRF p :=>:
      ECry [err "simplifyProof: proposition is absurd:"
        ++ errTyVal (p :<: PROP)])
    Just (Simply qs _ h) → do
      qrs ← traverse partProof qs
      let prf = substitute qs qrs h
      return (prf :=>: evTm prf, ElabSuccess)
    Nothing → subProof wrk (PRF p)
  where
    partProof :: (REF :<: INTM) → ProofState INTM
    partProof (ref :<: _) = do
      ((tm :=>: -, -) ← subProof WorkElsewhere (pty ref))
      return tm

```

```

subProof :: WorkTarget → VAL → ProofState (INTM :=>: VAL, ElabStatus)
subProof wrk (PRF p) = flexiProof wrk p ⊕ lastHope wrk (PRF p)

```

After simplification has dealt with the easy stuff, it calls *flexiProof* to solve any flex-rigid equations (by suspending a solution process on a subgoal and returning the subgoal).

```

flexiProof :: WorkTarget → VAL → ProofState (INTM :=>: VAL, ElabStatus)

```

```

flexiProof wrk (EQBLUE (_S :=>: s) (_T :=>: t)) =
  flexiProofMatch      (_S :=>: s) (_T :=>: t)
  ⊕ flexiProofLeft  wrk (_S :=>: s) (_T :=>: t)
  ⊕ flexiProofRight wrk (_S :=>: s) (_T :=>: t)
flexiProof _ _ = ()

```

If we are trying to prove an equation between the same fake reference applied to two lists of parameters, we prove equality of the parameters and use reflexivity. This case arises frequently when proving label equality to make recursive calls. **question: Do we need this case, or are we better off using matching?**

```

flexiProofMatch :: (TY :=>: VAL) → (TY :=>: VAL)
  → ProofState (INTM :=>: VAL, ElabStatus)
flexiProofMatch (_S :=>: N s) (_T :=>: N t)
  | Just (ref, ps) ← pairSpines s t [] = do
  let ty = pty ref
      prfs ← proveBits ty ps B0
      let q = NP refl $$ A ty $$ A (NP ref) $$ Out
          r = CON (smash q (trail prfs))
          r' ← bquoteHere r
      return (r' :=>: r, ElabSuccess)
  where
    pairSpines :: NEU → NEU → [(VAL, VAL)] → Maybe (REF, [(VAL, VAL)])
    pairSpines (P ref@(sn := _:<: _)) (P (tn := _:<: _)) ps
      | sn ≡ tn = Just (ref, ps)
      | otherwise = Nothing
    pairSpines (s : $ A as) (t : $ A at) ps = pairSpines s t ((as, at) : ps)
    pairSpines _ _ _ = Nothing

```

```

proveBits :: TY → [(VAL, VAL)] → Bwd (VAL, VAL, VAL)
  → ProofState (Bwd (VAL, VAL, VAL))
proveBits ty [] prfs = return prfs
proveBits (PI s t) ((as, at) : ps) prfs = do
  (_ :=>: prf, _) ← simplifyProof WorkElsewhere (EQBLUE (s :=>: as) (s :=>: at))
  proveBits (t $$ A as) ps (prfs :< (as, at, prf))

```

```

smash :: VAL → [(VAL, VAL, VAL)] → VAL
smash q [] = q
smash q ((as, at, prf) : ps) = smash (q $$ A as $$ A at $$ A prf) ps

```

```

flexiProofMatch _ _ = ()

```

If one side of the equation is a hoping hole applied to the shared parameters of our current location, we can solve it with the other side of the equation. **question: How can we generalise this to cases where the hole is under a different list of parameters?**

question: Can we hope for a proof of equality and insert a coercion rather than demanding definitional equality of the sets? See `Elab.pig` for an example where this makes the elaboration process fragile, because we end up trying to move definitions with processes attached.

```

flexiProofLeft :: WorkTarget → (TY :>: VAL) → (TY :>: VAL)
  → ProofState (INTM :=>: VAL, ElabStatus)
flexiProofLeft wrk (_T :>: N t) (_S :>: s) = do
  guard ≪≪ withNSupply (equal (SET :>: (_S, _T)))

(q' :=>: q, _) ← simplifyProof False (EQBLUE (SET :>: _S) (SET :>: _T))

ref ← stripShared t
s' ← bquoteHere s
_S' ← bquoteHere _S
t' ← bquoteHere t
_T' ← bquoteHere _T
let p = EQBLUE (_T :>: N t) (_S :>: s)
    p' = EQBLUE (_T' :>: N t') (_S' :>: s')

N (coe : @ [_S', _T', q', s']) :=>: Just (coe @@ [_S, _T, q, s])

eprob = WaitSolve ref (s' :=>: Just s) ElabHope

suspendThis wrk ("eq" :<: PRF p' :=>: PRF p) eprob
flexiProofLeft _ _ _ = ()

flexiProofRight :: WorkTarget → (TY :>: VAL) → (TY :>: VAL)
  → ProofState (INTM :=>: VAL, ElabStatus)
flexiProofRight wrk (_S :>: s) (_T :>: N t) = do
  guard ≪≪ withNSupply (equal (SET :>: (_S, _T)))
  ref ← stripShared t
  s' ← bquoteHere s
  _S' ← bquoteHere _S
  t' ← bquoteHere t
  _T' ← bquoteHere _T
  let p = EQBLUE (_S :>: s) (_T :>: N t)
    p' = EQBLUE (_S' :>: s') (_T' :>: N t')
    eprob = WaitSolve ref (s' :=>: Just s) ElabHope
  suspendThis wrk ("eq" :<: PRF p' :=>: PRF p) eprob
flexiProofRight _ _ _ = ()

```

If all else fails, we can hope for anything we like by leaving a hoping subgoal, either using the current one (if we are working on it) or creating a new subgoal. Ideally we should cry rather than hoping for something patently absurd: at the moment we reject some impossible hopes but do not always spot them.

```

lastHope :: WorkTarget → TY → ProofState (INTM :=>: VAL, ElabStatus)
lastHope WorkCurrentGoal ty = do
  putHoleKind Hoping
  return · (, ElabSuspended) ≪≪ neutralise ≪≪ getCurrentDefinition
lastHope WorkElsewhere ty = do
  ty' ← bquoteHere ty
  return · (, ElabSuccess) ≪≪ neutralise ≪≪ makeKinded Nothing Hoping ("hope" :<: ty')

```

6.4.4 Suspending computation

Pierre: To be reviewed.

The *suspend* command can be used to delay elaboration, by creating a subgoal of the given type and attaching a suspended elaboration problem to its tip. When the scheduler hits the goal, the elaboration problem will restart if it is unstable.

```

suspend :: (String <: INTM ==>: TY) → EProb → ProofState (EXTM ==>: VAL)
suspend (x <: tt) prob = do
  -- Make a hole
  r ← make (x <: termOf tt)
  -- Store the suspended problem
  Just (EDEF ref xn dkind dev@(Dev { devTip = Unknown tt }) tm anchor) ← removeEntryAbove
  putEntryAbove (EDEF ref xn dkind (dev { devTip = Suspended tt prob }) tm anchor)
  -- Mark the Suspension state
  let ss = if isUnstable prob then SuspendUnstable else SuspendStable
  putDevSuspendState ss
  -- Mark for Scheduler action Pierre: right?
  suspendHierarchy ss
  return r

```

The *suspendMe* command attaches a suspended elaboration problem to the current location.

Pierre: We expect the tip to be in an Unknown state. That's an invariant.

```

suspendMe :: EProb → ProofState (EXTM ==>: VAL)
suspendMe prob = do
  -- Store the suspended problem in the Tip
  Unknown tt ← getDevTip
  putDevTip (Suspended tt prob)
  -- Mark for Scheduler action Pierre: right?
  let ss = if isUnstable prob then SuspendUnstable else SuspendStable
  suspendHierarchy ss
  getCurrentDefinition

```

The *suspendThis* command attaches the problem to the current goal if we are working on it, and creates a new subgoal otherwise.

```

suspendThis :: WorkTarget → (String <: INTM ==>: TY) → EProb →
  ProofState (INTM ==>: VAL, ElabStatus)
suspendThis WorkCurrentGoal _ ep =
  return · (, ElabSuspended) ≪≪ neutralise ≪≪ suspendMe ep
suspendThis WorkElsewhere stt ep =
  return · (, ElabSuccess) ≪≪ neutralise ≪≪ suspend stt ep

```

6.5 Invoking the Elaborator

6.5.1 Elaborating terms

The *elaborate* command elaborates a term in display syntax, given its type, to produce an elaborated term and its value representation. It behaves similarly to *check* from subsection 2.5.2, except that it operates in the Elab monad, so it can create subgoals and λ -lift terms.

```

elaborate :: Loc → (TY >: DInTmRN) → ProofState (INTM ==>: VAL)
elaborate loc (ty >: tm) = runElab WorkElsewhere (ty >: makeElab loc tm)
  ≫≫ return · fst

```

$elaborate' = elaborate \text{ (Loc 0)}$

$elaborateHere :: \text{Loc} \rightarrow \text{DInTmRN} \rightarrow \text{ProofState (INTM :=>: VAL, ElabStatus)}$

$elaborateHere \text{ loc } tm = \mathbf{do}$
 $_ :=>: ty \leftarrow getHoleGoal$
 $runElab \text{ WorkCurrentGoal } (ty :=>: makeElab \text{ loc } tm)$

$elaborateHere' = elaborateHere \text{ (Loc 0)}$

$elabInfer :: \text{Loc} \rightarrow \text{DEXTmRN} \rightarrow \text{ProofState (INTM :=>: VAL :<: TY)}$

$elabInfer \text{ loc } tm = \mathbf{do}$
 $(tt, _) \leftarrow runElab \text{ WorkElsewhere } (sigSetVAL :=>: makeElabInfer \text{ loc } tm)$
 $\mathbf{let} (tt' :<: _ :=>: ty) = extractNeutral \text{ tt}$
 $\text{return } (tt' :<: ty)$

$elabInfer' = elabInfer \text{ (Loc 0)}$

Sometimes (for example, if we are about to apply elimination with a motive) we really want elaboration to proceed as much as possible. The *elabInferFully* command creates a definition for the argument, elaborates it and runs the scheduler.

$elabInferFully :: \text{DEXTmRN} \rightarrow \text{ProofState (EXTM :=>: VAL :<: TY)}$

$elabInferFully \text{ tm} = \mathbf{do}$
 $make \text{ ("eif" :<: sigSetTM)}$
 $goIn$
 $(tm :=>: _, status) \leftarrow runElab \text{ WorkCurrentGoal } (sigSetVAL :=>: makeElabInfer \text{ (Loc 0) } tm)$
 $when (status \equiv \text{ElabSuccess}) (ignore (give \text{ tm}))$
 $startScheduler$
 $(tm :=>: v) \leftarrow getCurrentDefinition$
 $goOut$
 $\text{return } (tm : \$ \text{Snd} :=>: v \$\$ \text{Snd} :<: v \$\$ \text{Fst})$

6.5.2 Elaborating construction commands

The *elabGive* command elaborates the given display term in the appropriate type for the current goal, and calls the *give* command on the resulting term. If its argument is a nameless question mark, it avoids creating a pointless subgoal by simply returning a reference to the current goal (applied to the appropriate shared parameters).

$elabGive :: \text{DInTmRN} \rightarrow \text{ProofState (EXTM :=>: VAL)}$
 $elabGive \text{ tm} = elabGive' \text{ tm} < * \text{startScheduler} < * \text{goOut}$

$elabGiveNext :: \text{DInTmRN} \rightarrow \text{ProofState (EXTM :=>: VAL)}$
 $elabGiveNext \text{ tm} = elabGive' \text{ tm} < * \text{startScheduler} < * (\text{nextGoal} \oplus \text{goOut})$

```

elabGive' :: DInTmRN → ProofState (EXTM :=>: VAL)
elabGive' tm = do
  tip ← getDevTip
  case (tip, tm) of
    (Unknown _, DQ " ") → getDefn
    (Unknown _, _)      → do
      (tm' :=>: _, status) ← elaborateHere' tm
      case status of
        ElabSuccess → give tm'
        ElabSuspended → getDefn
    _ → throwError' $ err "elabGive: only possible for incomplete goals."
  where
    getDefn :: ProofState (EXTM :=>: VAL)
    getDefn = do
      CDefinition _ ref _ _ _ ← getCurrentEntry
      aus ← getGlobalScope
      return (applySpine ref aus)

```

The *elabMake* command elaborates the given display term in a module to produce a type, then converts the module to a goal with that type. Thus any subgoals produced by elaboration will be children of the resulting goal.

```

elabMake :: (String <: DInTmRN) → ProofState (EXTM :=>: VAL)
elabMake (s <: ty) = do
  makeModule s
  goIn
  ty' :=>: _ ← elaborate' (SET >: ty)
  tm ← moduleToGoal ty'
  goOutBelow
  return tm

```

The *elabPiParam* command elaborates the given display term to produce a type, and creates a Π with that type.

```

elabPiParam :: (String <: DInTmRN) → ProofState REF
elabPiParam (s <: ty) = do
  tt ← elaborate' (SET >: ty)
  piParamUnsafe (s <: tt)

```

```

elabLamParam :: (String <: DInTmRN) → ProofState REF
elabLamParam (s <: ty) = do
  tt ← elaborate' (SET >: ty)
  assumeParam (s <: tt)

```

6.5.3 Elaborating programming problems

The *elabLet* command sets up a programming problem, given a name and scheme. The command **let** *plus* (*m* : Nat) (*n* : Nat) : Nat should result in the following proof state:

```

plus
[ plus-type
  [ tau := Nat : Set ;
    (m : Nat) ->
    tau := Nat : Set ;

```

```

      (n : Nat) ->
    ] Nat : Set ;
  plus
    [ \ m : Nat ->
      \ n : Nat ->
      \ c : < plus^1 m n : Nat > ->
    ] c call : Nat ;
  plus-impl
    [ \ m : Nat ->
      \ n : Nat ->
    ] ? : < plus^1 m n : Nat > ;
  \ m : Nat ->
  \ n : Nat ->
] plus-impl m n call : Nat ;

```

```

elabLet :: (String <: Scheme DInTmRN) → ProofState (EXTM :=>: VAL)
elabLet (x <: sch) = do
  makeModule x
  goIn

```

First we need to elaborate the scheme so it contains evidence terms, then convert the module into a goal with the scheme assigned.

```

make (x ++ "-type" <: SET)
goIn
(sch', ty :=>: _) ← elabLiftedScheme sch
moduleToGoal (N ty)
putCurrentScheme sch'

```

Now we add a definition with the same name as the function being defined, to handle recursive calls. This has the same arguments as the function, plus an implicit labelled type that provides evidence for the recursive call.

```

CDefinition _ (mnom := HOLE _ <: ty) _ _ _ ← getCurrentEntry
pn :=>: _ ← getFakeCurrentEntry
let schCall = makeCall (P $ mnom := FAKE <: ty) 0 sch'
us ← getParamsInScope
let schCallLocal = applyScheme schCall us
make (x <: schemeToInTm schCallLocal)
goIn
putCurrentScheme schCall
refs ← traverse lambdaParam (schemeNames schCallLocal)
giveOutBelow (N (P (last refs) : $ Call (N (pn $## map NP (init refs))))))

```

For now we just call `elabProgram` to set up the remainder of the programming problem. This could be implemented more cleanly, but it works.

```

elabProgram (init $ schemeNames schCallLocal)
where

```

Sorry for the horrible de Bruijn index mangling. **question: Perhaps we should use something like TEL to represent schemes as telescopes of values?**

```

makeCall :: EXTM → Int → Scheme INTM → Scheme INTM
makeCall l n (SchType ty) =
  SchImplicitPi ("c" :<: LABEL (N (l $$$ fmap NV [n - 1, n - 2..0])) ty)
  (SchType (inc 0 %% ty))
makeCall l n (SchImplicitPi (x :<: s) schT) =
  SchImplicitPi (x :<: s) (makeCall l (n + 1) schT)
makeCall l n (SchExplicitPi (x :<: schS) schT) =
  SchExplicitPi (x :<: schS) (makeCall l (n + 1) schT)

```

The `elabProgram` command adds a label to a type, given a list of arguments. e.g. with a goal `plus : Nat → Nat → Nat`, `program x, y` will give a proof state of:

```

plus [
  plus := ? : (x : Nat) -> (y : Nat) -> <plus x y : Nat>
  \ x : Nat
  \ y : Nat
] plus x y call : Nat

```

```

elabProgram :: [String] → ProofState (EXTM :=>: VAL)
elabProgram args = do
  n ← getCurrentName
  pn ← getFakeCurrentEntry
  (gUnlifted :=>: _) ← getHoleGoal
  newty ← withNSupply $ pity (mkTel (unN $ valueOf pn) (evTm gUnlifted) [] args)
  newty' ← bquoteHere newty
  impl :=>: _ ← make (magicImplName :<: newty')
  argrefs ← traverse lambdaParam args
  let fcall = termOf pn $$$ (map NP argrefs)
      call = impl $$$ (map NP argrefs) : $ Call (N fcall)
  r ← give (N call)
  goIn
  return r
where
  mkTel :: NEU → TY → [VAL] → [String] → TEL TY
  mkTel n (PI s t) args (x : xs)
    = (x :<: s) : - : (λval → mkTel n (t $$ A val) (val : args) xs)
  mkTel n r args _ = Target (LABEL (mkL n (reverse args)) r)
  mkL :: NEU → [VAL] → VAL
  mkL n [] = N n
  mkL n (x : xs) = mkL (n : $ (A x)) xs

```

```

unN :: VAL → NEU
unN (N n) = n

```

6.5.4 Elaborating schemes

```

elabLiftedScheme :: Scheme DInTmRN → ProofState (Scheme INTM, EXTM :=>: VAL)
elabLiftedScheme sch = do
  inScope ← getInScope
  (sch', tt) ← elabScheme inScope sch
  return (liftScheme inScope sch', tt)

```

```

liftScheme :: Entries → Scheme INTM → Scheme INTM
liftScheme B0 sch = sch
liftScheme (es :< EPARAM _ (x, _) _ s _) sch =
  liftScheme es (SchExplicitPi (x :<: SchType (es -| s)) sch)
liftScheme (es :< _) sch = liftScheme es sch

```

```

elabScheme :: Entries → Scheme DInTmRN → ProofState (Scheme INTM, EXTM :=>: VAL)

```

```

elabScheme es (SchType ty) = do
  ty' :=>: _ ← elaborate' (SET :=>: ty)
  tt ← giveOutBelow ty'
  return (SchType (es -| ty'), tt)

```

```

elabScheme es (SchExplicitPi (x :<: s) t) = do
  make ("tau" :<: SET)
  goIn
  (s', ty :=>: _) ← elabScheme es s
  piParam (x :<: N ty)
  e ← getEntryAbove
  (t', tt) ← elabScheme (es :< e) t
  return (SchExplicitPi (x :<: s') t', tt)

```

```

elabScheme es (SchImplicitPi (x :<: s) t) = do
  ss ← elaborate' (SET :=>: s)
  piParam (x :<: termOf ss)
  e ← getEntryAbove
  (t', tt) ← elabScheme (es :< e) t
  return (SchImplicitPi (x :<: (es -| termOf ss)) t', tt)

```

6.6 The Scheduler

Handling elaboration essentially requires writing an operating system. Having defined how to execute processes in section 6.5, we now turn our attention to process scheduling. The scheduler is called when an elaboration process yields (either halting after solving its goal, halting with an error, or suspending work until later). It searches downwards in the proof state for unstable elaboration problems and executes any it finds.

When the scheduler is started, all problems before the working location should be stable, but there may be unstable problems in the current location and below it. The *startScheduler* command runs the scheduler from the current location, so it will stabilise the children and return to where it started.

```

startScheduler :: ProofState ()
startScheduler = getCurrentName >>= scheduler

```

In general, the scheduler might have to move non-locally (e.g. in order to solve goals elsewhere in the proof state), so it keeps track of a target location to return to. When *scheduler* is called, it checks to see if there might be any suspended problems below the current location. If so, it resets the suspend state and starts searching from the top of the development. If not, it calls *schedulerDone* to move out or terminate as appropriate.

```

scheduler :: Name → ProofState ()
scheduler n = do
  ss ← (| devSuspendState getAboveCursor |)
  case ss of
    SuspendUnstable → do putDevSuspendState SuspendNone
                          cursorTop
                          schedulerContinue n
    _                → schedulerDone n

```

The *schedulerContinue* command processes the entries below the cursor. The suspend state should describe the entries above the cursor. If there are no entries below, we are done. If we find a parameter, we simply move past it, because it cannot have a suspended problem attached. If we find a definition, we enter it, try to resume its current entry, then search its children from the top.

```

schedulerContinue :: Name → ProofState ()
schedulerContinue n = do
  cs ← getBelowCursor
  case cs of
    F0                → schedulerDone n
    EPARAM _ _ _ _ _ :> _ → cursorDown >> schedulerContinue n
    _ :> _            → do
      cursorDown
      goIn
      resumeCurrentEntry
      scheduler n

```

Once done, the *schedulerDone* command checks if this is the target location. If so, we stop; otherwise, we resume the current entry and continue searching.

```

schedulerDone :: Name → ProofState ()
schedulerDone n = do
  mn ← getCurrentName
  case mn of
    _ | mn ≡ n → cursorBottom
    []        → error "scheduler: got lost!"
    _        → do
      b ← resumeCurrentEntry
      if b then scheduler n
          else do
            goOutBelow
            schedulerContinue n

```

The *resumeCurrentEntry* command checks for an unstable elaboration problem on the current entry of the current location, and resumes elaboration if it finds one. If elaboration succeeds, it gives the resulting term. It returns whether an elaboration process was resumed (not whether the process succeeded).

```

resumeCurrentEntry :: ProofState Bool
resumeCurrentEntry = do
  tip ← getDevTip
  case tip of
    Suspended (ty :=>: tyv) prob | isUnstable prob → do
      putDevTip (Unknown (ty :=>: tyv))
      mn ← getCurrentName
      schedTrace $ "scheduler: resuming elaboration on " ++ showName mn
        ++ " : \n" ++ show prob
      mtt ← resume (ty :=>: tyv) prob
      case mtt of
        Just tt → give (termOf tt)
          >> schedTrace "scheduler: elaboration done."
        Nothing → schedTrace "scheduler: elaboration suspended."
      return True
    _ → return False

```

Given a (potentially, but not necessarily, unstable) elaboration problem for the current location, we can *resume* it to try to produce a term. If this succeeds, the cursor will be in the same location, but if it fails (i.e. the problem has been suspended) then the cursor could be anywhere earlier in the proof state.

```

resume :: (INTM :=>: VAL) → EProb → ProofState (Maybe (INTM :=>: VAL))
resume _ (ElabDone tt) = return · Just · maybeEval $ tt
resume (ty :=>: tyv) ElabHope =
  return · ifSnd ≪≪ runElabHope WorkCurrentGoal tyv
resume (ty :=>: tyv) (ElabProb tm) =
  return · ifSnd ≪≪ runElab WorkCurrentGoal (tyv >: makeElab (Loc 0) tm)
resume (ty :=>: tyv) (ElabInferProb tm) =
  return · ifSnd ≪≪ runElab WorkCurrentGoal (tyv >: makeElabInfer (Loc 0) tm)
resume (ty :=>: tyv) (WaitCan (tm :=>: Just (C v)) prob) =
  resume (ty :=>: tyv) prob
resume (ty :=>: tyv) (WaitCan (tm :=>: Nothing) prob) =
  resume (ty :=>: tyv) (WaitCan (tm :=>: Just (evTm tm)) prob)
resume _ prob@(WaitCan (tm :=>: _) _) = do
  schedTrace $ "Suspended waiting for " ++ show tm ++ " to become canonical."
  suspendMe prob
  return Nothing
resume _ (WaitSolve ref@(_ := HOLE _ <: _) stt prob) = do
  suspendMe prob
  mn ← getCurrentName
  tm ← bquoteHere (valueOf · maybeEval $ stt) -- force definitional expansion
  solveHole' ref [] tm -- takes us to who knows where
  return Nothing

```

If we have a WaitSolve problem where the hole has already been solved with something else, we need to ensure the solution is compatible. If the two solutions are definitionally equal, everything is fine, otherwise we hope for a proof of their equality. **Adam:** At the moment this proof isn't used, but hoping for it might cause things to be solved usefully anyway. Is there a better way to do this?

```

resume tt (WaitSolve ref@(_ := DEFN tmv' :<: ty) stt prob) = do
  aus ← getGlobalScope
  sibs ← getEntriesAbove
  let stt' = maybeEval stt
      stm = parBind aus sibs (termOf stt')
      stv = evTm stm
  eq ← withNSupply $ equal (ty :>: (stv, tmv'))
  if eq
  then resume tt prob

else runElabHope WorkElsewhere (PRF (EQBLUE (ty :>: tmv') (ty :>: stv))) >>
  schedTrace "resume: WaitSolve failed!" >> resume tt prob

else throwError' $ err "resume: hole" ++ errRef ref ++
  err "has been solved with" ++ errTyVal (tmv' :<: ty) ++
  err "but I wanted to solve it with" ++
  errTyVal (valueOf stt' :<: ty)

resume tt (ElabSchedule prob) = resume tt prob

```

```

ifSnd :: (a, ElabStatus) → Maybe a
ifSnd (a, ElabSuccess) = Just a
ifSnd (_, ElabSuspended) = Nothing

```

6.7 Wire Service

6.7.1 Updating a reference

Here we describe how to handle updates to references in the proof state, caused by refinement commands like *give*. The idea is to deal with updates lazily, to avoid unnecessary traversals of the proof tree. When *updateRef* is called to announce a changed reference (that the current development has already processed), it simply inserts a news bulletin below the current development.

```

updateRef :: REF → ProofState ()
updateRef ref = putNewsBelow [(ref, GoodNews)]

```

6.7.2 Committing news into the ProofState

The *propagateNews* function takes a current news bulletin and a list of entries to *add* to the current development. It applies the news bulletin to each entry in turn, picking up other bulletins along the way. This function is called when navigating to a development that may contain news bulletins, so they can be pushed out of sight.

```

propagateNews :: PropagateStatus → NewsBulletin → NewsyEntries →
  ProofState NewsBulletin

```

We need to keep track of whether news propagation was called normally or as a recursive call. If called normally, we will stash the news bulletin the proof state when done, but this is unnecessary if news propagation will continue outside the recursive call.

```

data PropagateStatus = NormalPropagate | RecursivePropagate

```

If we have nothing to say and nobody to tell, we might as well give up and go home. If we were called recursively and have nobody to listen to the news, we give up as well.

```
propagateNews _ [] (NF F0) = return []
propagateNews RecursivePropagate news (NF F0) = return news
```

If there are no entries to process, we should tell the current entry (there is one, as we are within a development), stash the bulletin after the current location and stop. Note that the insertion is optional because it will fail when we have reached the end of the module, at which point everyone knows the news anyway.

```
propagateNews NormalPropagate news (NF F0) = do
  news' ← tellCurrentEntry news
  optional (putNewsBelow news')
  return news'
```

To update a Parameter, we check to see if its type has become more defined, and pass on the good news if necessary.

```
propagateNews top news
  (NF (Right (EPARAM (name := DECL <: tv) sn k ty a) > es)) = do
  case tellNews news ty of
  (–, NoNews) → do
    let ref = name := DECL <: tv
        putEntryAbove (EPARAM ref sn k ty a)
        propagateNews top news (NF es)
  (ty', GoodNews) → do
    let ref = name := DECL <: evTm ty'
        putEntryAbove (EPARAM ref sn k ty' a)
        propagateNews top (addNews (ref, GoodNews) news) (NF es)
```

To update definitions or modules, we call on *propagateNewsWithin*.

```
propagateNews top news (NF (Right e > es)) = do
  news' ← propagateNewsWithin news e
  propagateNews top news' (NF es)
```

Finally, if we encounter an older news bulletin when propagating news, we can simply merge the two together.

```
propagateNews top news (NF (Left oldNews > es)) =
  propagateNews top (mergeNews news oldNews) (NF es)
```

The *propagateNewsWithin* command will:

1. add the definition to the proof state without its children;
2. recursively propagate the news to the children, adding them as it goes;
3. call *tellCurrentEntry* to update the definition itself; and
4. move the focus out of the definition.

```

propagateNewsWithin :: NewsBulletin → Entry NewsyFwd → ProofState NewsBulletin
propagateNewsWithin news e = do
  -- Get current context and insert it as a layer
  Dev es tip nsupply ss ← getAboveCursor
  below ← getBelowCursor
  putLayer (Layer es (mkCurrentEntry e) (reverseEntries below) tip nsupply ss)
  -- Extract new information and make it the current location
  let Just (Dev cs newTip newNSupply newSS) = entryDev e
      putAboveCursor (Dev B0 newTip newNSupply newSS)
      putBelowCursor F0
  -- Propagate news through children and current entry
  news' ← propagateNews RecursivePropagate news cs
  news'' ← tellCurrentEntry news'
  -- Go out to where we were before
  goOut
  return news''

```

6.7.3 Informing a current entry about its development

Pierre: To be more carefully reviewed.

The `tellEntry` function informs an entry about a news bulletin that its development (if any) have already received. It applies the news bulletin to the entry, returning the update entry together with (potentially) more news.



Invariant: `tellEntry` on a definition. If the entry is a definition, it must be the current entry of the current cursor position (i.e. the entry should come from `getLeaveCurrent`).

□

Pierre: There is something fishy with this function and this invariant. In reality, there are two function, one defined on Parameters and called only on Parameters (in `ProofState.Edition.Navigation`) (call it `tellParameterEntry`) and one defined on Definitions and only called by itself and `tellCurrentEntry` in a safe wrapper enforcing this invariant (call it `tellDefinitionEntry`).

If we do the split, on one hand, the invariant will always be enforced. On the other hand, we get two functions with a partial pattern-matching. At least, with two explicitly named functions, one can hardly ignore that one is for Parameters and the other for Definitions.

```
tellEntry :: NewsBulletin → Entry Bwd → ProofState (NewsBulletin, Entry Bwd)
```

Modules carry no type information, so they are easy:

```
tellEntry news (EModule n d) = return (news, EModule n d)
```

The update of a parameter consists in:

1. updating its type based on the news we received, and
2. adding to the news bulletin the fact that this parameter has been updated

```

tellEntry news (EPARAM (name := DECL <: tv) sn k ty anchor) = do
  let (ty' :=>: tv', n) = tellNewsEval news (ty :=>: tv)
      let ref = name := DECL <: tv'
          return (addNews (ref, n) news, EPARAM ref sn k ty' anchor)

```

To update a hole, we must first check to see if the news bulletin contains a definition for it. If so, we fill in the definition (and do not need to update the news bulletin). If not, we must **Pierre:** why?:

1. update the tip type;
2. update the overall type of the entry, as stored in the reference; and
3. update the news bulletin with news about this definition.

If the hole is Hoping and we have good news about its type, then we restart elaboration to see if it can make any progress.

```

tellEntry news (EDEF ref@(name := HOLE h :<: tyv) sn
                dkind dev@(Dev { devTip = Unknown tt }) ty anchor)
| Just (ref'@(_ := DEFN tm :<: _), GoodNews) ← getNews news ref = do
  -- We have a Definition for it
  es ← getInScope
  tm' ← bquoteHere (tm $$$ paramSpine es)
  let (tt', _) = tellNewsEval news tt
      (ty', _) = tellNews news ty
  -- Define the hole
  return (news, EDEF ref' sn dkind (dev { devTip = Defined tm' tt' }) ty' anchor)
| otherwise = do
  -- Not a Definition
  let (tt', n) = tellNewsEval news tt
      (ty' :=>: tyv', n') = tellNewsEval news (ty :=>: tyv)
      ref = name := HOLE h :<: tyv'
      tip = case (min n n', h) of
        (GoodNews, Hoping) → Suspended tt' ElabHope
        - → Unknown tt'
  return (addNews (ref, min n n') news,
          EDEF ref sn dkind (dev { devTip = tip }) ty' anchor)

```

To update a hole with a suspended elaboration problem attached, we proceed similarly to the previous case, but we also update the elaboration problem. If the news bulletin defines this hole, it had better just be hoping for a solution **Pierre**: Is this an invariant we are meant to enforce? Or something that might break one day? See bug #53., in which case we can safely ignore the attached ElabHope process.

```

tellEntry news (EDEF ref@(name := HOLE h <: tyv) sn
                dkind dev@(Dev { devTip = Suspended tt prob }) ty anchor)
| Just ne ← getNews news ref = do
  -- We have a Definition for it
  case prob of
  ElabHope → do
    -- The elaboration strategy has to be to Hope
    tellEntry news (EDEF ref sn dkind (dev { devTip = Unknown tt }) ty anchor)
  - → do
    -- Pierre: Is that a throwError or an error?
    throwError' · err · unlines $ [
      "tellEntry: news bulletin contains update", show ne,
      "for hole", show ref,
      "with suspended computation", show prob]
| otherwise = do
  -- We don't have a Definition
  let (tt', n)          = tellNewsEval news tt
      (ty' :=>: tyv', n') = tellNewsEval news (ty :=>: tyv)
      ref              = name := HOLE h <: tyv'
      prob'           = tellEProb news prob
      state            = if isUnstable prob'
                        then SuspendUnstable
                        else SuspendStable
      suspendHierarchy state
  return (addNews (ref, min n n') news,
          EDEF ref sn dkind (dev { devTip = Suspended tt' prob' }) ty' anchor)
  where tellEProb :: NewsBulletin → EProb → EProb
        tellEProb news = fmap (getLatest news)

```

To update a closed definition (Defined), we must:

1. update the tip type;
2. update the overall type of the entry, as stored in the reference;
3. update the definition and re-evaluate it (**question: could this be made more efficient?**); and
4. update the news bulletin with news about this definition.

```

tellEntry news (EDEF (name := DEFN tmL <: tyv) sn dkind
                    dev@(Dev { devTip = Defined tm tt }) ty anchor) = do
  let (tt', n)          = tellNewsEval news tt
      (ty' :=>: tyv', n') = tellNewsEval news (ty :=>: tyv)
      (tm', n'')        = tellNews news tm
      aus ← getGlobalScope
  let tmL' = parBind aus (devEntries dev) tm'

```

For paranoia purposes, the following test might be helpful:

```

mc ← withNSupply (inCheck $ check (tyv' :=>: tmL'))
mc 'catchEither' unlines ["tellEntry " ++ showName name ++ ":",
  show tmL', "is not of type", show ty']

```

```

let ref = name := DEFN (evTm tmL') <: tyv'
return (addNews (ref, GoodNews {-min (min n n') n'' -} ) news,
        EDEF ref sn dkind (dev { devTip = Defined tm' tt' }) ty' anchor)

```

The *tellCurrentEntry* function informs the current entry about a news bulletin that her children have already received, and returns the updated news.

```

tellCurrentEntry :: NewsBulletin → ProofState NewsBulletin
tellCurrentEntry news = do
  e ← getLeaveCurrent
  (news', e') ← tellEntry news e
  putEnterCurrent e'
  return news'

```

When the current location or one of its children has suspended, we need to update the outer layers.

```

suspendHierarchy :: SuspendState → ProofState ()
suspendHierarchy ss = getLayers ≫≡ putLayers · help ss
where
  help :: SuspendState → Bwd Layer → Bwd Layer
  help ss B0 = B0
  help ss (ls :< l) = help ss' ls :< l { laySuspendState = ss' }
    where ss' = min ss (laySuspendState l)

```

Chapter 7

Distillation

7.1 The distiller

The distiller, like the elaborator, is organized on a *check/infer* basis, following the type-checker implementation in Section 2.5.2. *distill* mirrors *check* — distilling INTMs, while *distillInfer* mirrors *infer* — distilling EXTMs.

7.1.1 Distilling INTMs

The *distill* command converts a typed INTM in the Evidence language to a term in the Display language; that is, it reverses *elaborate*. It performs christening at the same time, turning absolute names into relative names.

The distiller first tries to apply Feature-specific rules. These rules contain the intelligence of the distiller, aiming at making a concise Display term. If unsuccessful, the distiller falls back to the generic rules in *distillBase*.

When going under a binder, we have to introduce fresh names to distill further. When christening, these fresh names have to be dealt with separately (see *unresolve* in Section 4.18.2): indeed, they are actually bound variables. Hence, we collect this *local scope* as a list of Entries.

question: Does *unresolve* really need Entries, or can it cope with Bwd REF instead?

```
distill :: Entries → (TY :> INTM) → ProofStateT INTM (DInTmRN :=> VAL)

-- import j- DistillRules
-- [Feature = Anchor]
distill es (ANCHORS :> x@(ANCHOR (TAG u) t ts)) = do
  (displayTs :=> _) ← distill es (ALLOWEDBY (evTm t) :> ts)
  return (DANCHOR u displayTs :=> evTm x)
-- [/Feature = Anchor]
-- [Feature = Enum]
distill _ (ENUMT t :> tm) | Just r ← findIndex (t :> tm) = return r
where
  findIndex :: (VAL :> INTM) → Maybe (DInTmRN :=> VAL)
  findIndex (CONSE (TAG s) _ :> ZE) = Just (DTAG s :=> evTm tm)
  findIndex (CONSE _ a :> SU b) = findIndex (a :> b)
  findIndex _ = Nothing
```

Since elaboration turns lists into functions from enumerated types, we can do the reverse when distilling. This is slightly dubious.

```
distill es (PI (ENUMT e) t :> L (x : . N (op : @ [e', NV 0, t', b])))
  | op ≡ switchOp = distill es (branchesOp @@ [e, t] :> b)
```

The following cases turn vaguely list-like data into actual lists. We don't want this in general, but it is useful in special cases (when the data type is really supposed to be a list, as in EnumD).

question: When else should we use this representation?

```
distill _ (IMU (Just (L (K (ANCHOR (TAG r) _ _)))) _ _ _>: CON (PAIR ZE VOID)) | r ≡ "EnumU" =
  return (DVOID :=>: CON (PAIR ZE VOID))
```

```
distill es (C ty@(IMu (Just (LK (ANCHOR (TAG r) _ _)) :? =: _) _)>:
  C c@(Con (PAIR (SU ZE) (PAIR _ (PAIR _ VOID)))) | r ≡ "EnumU" = do
  Con (DPAIR _ (DPAIR s (DPAIR t _)) :=>: v) ← canTy (distill es)
  (ty :=>: c)
  return ((DPAIR s t) :=>: CON v)
-- [/Feature = Enum]
-- [Feature = Equality]
distill es (PROP :=>: tm@(EQBLUE (tty :=>: t) (uty :=>: u))) = do
  t' ← toDEXTm es (tty :=>: t)
  u' ← toDEXTm es (uty :=>: u)
  return $ DEqBlue t' u' :=>: evTm tm
```

When distilling a proof of an equation, we first check to see if the equation holds definitionally. If so, we avoid forcing the proof and return refl instead.

```
distill es (p@(PRF (EQBLUE (_S :=>: s) (_T :=>: t))) :=>: q) = do
  r ← askNSupply
  if equal (SET :=>: (_S, _T)) r ∧ equal (_S :=>: (s, t)) r
  then return (DU :=>: N (P refl : $ A _S : $ A s))
  else distillBase es (p :=>: q)
-- [/Feature = Equality]
-- [Feature = IDesc]
distill es (IMU l _I s i :=>: CON (PAIR t x))
| Just (e, f) ← sumilike _I (s $$ A i) = do
  m :=>: tv ← distill es (ENUMT e :=>: t)
  as :=>: xv ←
    distill es (idescOp @@ [_I, f tv
      ,
      L $ "i" : . [· i ·
        IMU (fmap (-$[]) l)
        (_I -$ []) (s -$ []) (NV i)
      ] :=>: x)
  case m of
    DTag s → return $ DTag s (unfold as) :=>: CON (PAIR tv xv)
    _ → return $ DCON (DPAIR m as) :=>: CON (PAIR tv xv)
where
  unfold :: DInTmRN → [DInTmRN]
  unfold DVOID = []
  unfold DU = []
  unfold (DPAIR s t) = s : unfold t
  unfold t = [t]
```

```

distill es (SET :=>: tm@(C (IMu ltm@(Just l :? =: (Id _I : & Id s)) i))) = do
  let lab = evTm ((l :? ARR _I ANCHORS) : $ A i)
      labTm ← bquoteHere lab
      (labDisplay :=>: _) ← distill es (ANCHORS :=>: labTm)
      _It :=>: _Iv ← distill es (SET :=>: _I)
      st :=>: sv ← distill es (ARR _Iv (idesc $$ A _Iv) :=>: s)
      it :=>: iv ← distill es (_Iv :=>: i)
      return $ (DIMU (Just labDisplay) _It st it :=>: evTm tm)
-- [/Feature = IDesc]
-- [Feature = Prop]
distill es (PRF TRIVIAL :=>: _) = return (DU :=>: VOID)
-- [/Feature = Prop]
-- [Feature = Sigma]
distill es (UNIT :=>: _) = return $ DVOID :=>: VOID
-- [/Feature = Sigma]

```

distill entries tt = distillBase entries tt

We separate out the standard distillation cases (without aspect extensions) so that aspect distill rules can give up and invoke the base cases.

```
distillBase :: Entries → (TY :=>: INTM) → ProofStateT INTM (DInTmRN :=>: VAL)
```

We distill structurally through canonical terms:

```

distillBase entries (C ty :=>: C c) = do
  cc ← canTy (distill entries) (ty :=>: c)
  return $ (DC $ fmap termOf cc) :=>: evTm (C c)

```

To distill a *lambda*-abstraction, we speculate a fresh reference and distill under the binder, then convert the scope appropriately. The INTM version of the entry type should never be used, so we can simply omit it. (Hopefully we will switch the entries to Bwd REF so this will not be necessary.)

```

distillBase entries (ty :=>: l@(L sc)) = do
  let x = fortran l
      (kind, dom, cod) ← lambdable ty 'catchMaybe' (err "distill: type "
        ++ errVal ty
        ++ err " is not lambdable.")
      tm' :=>: _ ← freshRef (x <: dom) $ λref →
        distill (entries :< EPARAM ref (mkLastName ref) kind
          (error "distill: type undefined") Nothing)
          (cod (pval ref) :=>: underScope sc ref)
      return $ DL (convScope sc x tm') :=>: (evTm $ L sc)
  where
    convScope :: Scope {TT} REF → String → DInTmRN → DSCOPE
    convScope (_ : . _) x tm = x :•tm
    convScope (K _) _ tm = DK tm

```

If we encounter a neutral term, we switch to *distillInfer*.

```

distillBase entries (_ :=>: N n) = do
  (n' <: _) ← distillInfer entries n []
  return $ DN n' :=>: evTm n

```

If none of the cases match, we complain loudly.

```

distillBase _ (ty :>: tm) = throwError' $
    err "distill: can't cope with\n" ++
    errTm tm ++ err " :<: " ++ errVal ty

```

7.1.2 Distilling EXTMs

The *distillInfer* command is the EXTm version of *distill*, which also yields the type of the term. Following *distill*, we maintain the local scope of fresh references.

Moreover, recall that the DExTm terms are in Spine form: they are composed of a DHead — either parameter, type annotation, or embedding of ExTm — and followed by a spine of eliminators. To perform this translation, we accumulate a *spine* and distill it when we reach the head. Doing so, shared parameters can be removed (see subsection 4.18.2).

```

distillInfer :: Entries → EXTm → Spine {TT} REF →
    ProofStateT INTM (DExTmRN :<: TY)
-- import j- DistillInferRules
-- [Feature = Labelled]
distillInfer es (t : $ Call (N l)) as = distillInfer es l as
-- [/Feature = Labelled]

```

To distill a parameter with a spine of eliminators, we use *unresolve* to determine a relative name for the reference, the number of shared parameters, and possibly a scheme attached to it. We then call on *distillSpine* to process the eliminators, and return the distilled elimination with the shared parameters and implicit arguments removed.

```

distillInfer entries tm@(P (name := kind :<: ty)) spine = do
    -- Compute a relative name from name
    proofCtxt ← get
    let (relName, argsToDrop, mSch) =
        unresolve name kind spine (inBScope proofCtxt) entries
    -- Distill the spine
    (e', ty') ← distillSpine entries (evTm tm :<: ty) spine
    -- Remove shared parameters and implicit arguments
    let spine1 = drop argsToDrop e'
        spine2 = maybe spine1 (hideImplicit spine1) mSch
    -- Ignore de Bruijn index on FAKE references (issue 87)
    let relName' = case (relName, kind) of
        [(s, _), FAKE] → [(s, Rel 0)]
        _                → relName
    -- Return the relative name applied to the simplified spine
    return $ (DP relName' :$spine2) :<: ty'
where

```

If the parameter has a scheme attached, we need to remove implicit arguments once we have distilled the spine and dropped the shared parameters. We proceed structurally through the scheme, removing arguments that should be implicit.

```

hideImplicit :: DSPINE → Scheme x → DSPINE
hideImplicit as (SchType _) = as
hideImplicit (a : as) (SchExplicitPi _ sch) = a : hideImplicit as sch
hideImplicit (a : as) (SchImplicitPi _ sch) = hideImplicit as sch
hideImplicit [] _ = []

```

To distill an elimination, we simply push the eliminator on to the spine.

```

distillInfer entries (t : $ e) spine = distillInfer entries t (e : spine)

```

Because there are no operators in the display syntax, we replace them with parameters containing the appropriate primitive references.

```
distillInfer entries (op : @ args) spine =
  distillInfer entries (P (lookupOpRef op)) (map A args ++ spine)
```

Unnecessary type ascriptions can simply be dropped. As we can always *infer* the type of a neutral term, there is no point in keeping its ascription. This also ensures that shared parameters are handled correctly when the head is a parameter under a type ascription, because distillation will proceed using the rule for parameters above.

```
distillInfer entries (N t :? _) spine = distillInfer entries t spine
```

Typed identity functions applied to an argument can simply be removed. We do this because they are inserted by elaboration of type annotations; if a user manually creates one, we can safely remove it anyway.

```
distillInfer entries (L (_ : . NV 0) :? PI ty _) ((A a) : spine) =
  distillInfer entries (a :? ty) spine
```

Otherwise, we have no choice but distilling both side of the type ascription (term and value). This gives a type annotation applied to the distilled term, together with the distilled spine.

```
distillInfer entries (t :? ty) spine = do
  -- Distill the type
  ty1 :=>: vty ← distill entries (SET :>: ty)
  -- Distill the term
  t1 :=>: vt ← distill entries (vty :>: t)
  -- Distill the spine
  (e, ty2) ← distillSpine entries (vt :<: vty) spine
  -- Annotate the term by the type, followed by its spine
  return $ DType ty1 :$(A t1 : e) :<: ty2
```

If nothing matches, we are unable to distill this term, so we complain loudly.

```
distillInfer _ tm _ = throwError' $
  err "distillInfer: can't cope with " ++ errTm (N tm)
```

7.1.3 Distillation support

The *distillSpine* command takes a list of entries in scope, a typed value and a spine of arguments for the value. It distills the spine, using *elimTy* to determine the appropriate type to push in at each step, and returns the distilled spine and the overall type of the application.

```
distillSpine :: Entries → (VAL :<: TY) → Spine {TT} REF →
  ProofStateT INTM (DSPINE, TY)
distillSpine _ (v :<: ty) [] = return ([], ty)
distillSpine entries (v :<: C ty) (a : spine) = do
  -- Distill structurally the eliminator a
  (e1, ty1) ← elimTy (distill entries) (v :<: ty) a
  -- Further distill the spine
  (es, ty2) ← distillSpine entries (v $$ (fmap valueOf e1) :<: ty1) spine
  -- Return distilled spine and type of the application
  return (fmap termOf e1 : es, ty2)
distillSpine entries (v :<: ty) spine = throwError' $
  err "distillSpine: cannot cope with" ++ errTyVal (v :<: ty)
  ++ err "which has non-canonical type" ++ errTyVal (ty :<: SET)
  ++ err "applied to spine" ++ map ErrorElim spine
```

The *toDExTm* helper function will distill a term to produce an Ex representation by applying a type annotation only if necessary.

```

toDExTm :: Entries → (INTM :>: INTM) → ProofStateT INTM DExTmRN
toDExTm entries ( _ :>: N tm ) = do
  (tm' :<: _) ← distillInfer entries tm []
  return tm'
toDExTm entries ( ty :>: tm ) = do
  (ty' :=>: tyv) ← distill entries (SET :>: ty)
  (tm' :=>: _) ← distill entries (tyv :>: tm)
  return $ DTY ty' tm'

```

7.1.4 Distillation interface

The *distillHere* command distills a term in the current context.

```

distillHere :: (TY :>: INTM) → ProofState (DInTmRN :=>: VAL)
distillHere tt = liftErrorState DTIN $ distill B0 tt

```

The *prettyHere* command distills a term in the current context, then passes it to the pretty-printer.

```

prettyHere :: (TY :>: INTM) → ProofState Doc
prettyHere = prettyHereAt maxBound

prettyHereAt :: Size → (TY :>: INTM) → ProofState Doc
prettyHereAt size tt = do
  dtm :=>: _ ← distillHere tt
  return (pretty dtm size)

```

7.2 The Scheme distiller

7.2.1 Distilling schemes

Distilling a scheme is similar in spirit to distilling a Π -type, in particular the λ -abstraction of its codomain (section 7.1.1). Provided a Scheme INTM, we compute the same scheme structure, with Display terms instead.

To do so, we proceed structurally, using *distill* on types and, recursively, *distillScheme* on schemes. Each time we go through a Π , we go under a binder; therefore we need to be careful to turn de Bruijn indices into the freshly introduced references.

This distiller takes the list of local entries we are working under, as well as the collected list of references we have made so far. It turns the INTM scheme into an a Display term scheme with relative names.

```

distillScheme :: Entries → Bwd REF → Scheme INTM →
  ProofStateT INTM (Scheme DInTmRN, INTM)

```

On a ground type, there is not much to be done: *distill* does the distillation job for us. However, we first have to turn the de Bruijn indices into references.

```

distillScheme entries refs (SchType ty) = do
  -- Instantiate de Bruijn indices with refs
  let ty1 = underneath refs ty
  -- Distill the type
  ty2 :=>: _ ← distill entries (SET :>: ty1)
  return (SchType ty2, ty1)

```

On an explicit Π , the domain is itself a scheme, so it needs to be distilled. Then, we go under the binder and distill the codomain, carrying the new *ref* and extending the local entries with it. The new reference is in local scope, so we need to extend the entries with it for later distillation. We know the INTM form of its type so we may as well supply it, though it should not be inspected.

```

distillScheme entries refs (SchExplicitPi (x <: schS) schT) = do
  -- Distill the domain
  (schS', s') ← distillScheme entries refs schS
  -- Under a fresh ref...
  freshRef (x <: evTm s') $ λref → do
    -- Distill the codomain
    (schT', t') ← distillScheme
      (entries <: EPARAM ref (mkLastName ref) ParamPi s' Nothing)
      (refs <: ref)
      schT
  return (SchExplicitPi (x <: schS') schT', PIV x s' t')

```

On an implicit Π , the operation is fairly similar. Instead of *distillScheme*-ing the domain, we proceed as for ground types — it is one.

```

distillScheme entries refs (SchImplicitPi (x <: s) schT) = do
  -- Distill the domain as a ground type
  let s' = underneath refs s
  sd :=>: sv ← distill entries (SET :=>: s')
  -- Under a fresh ref...
  freshRef (x <: sv) $ λref → do
    -- Distill the domain
    (schT', t') ← distillScheme
      (entries <: EPARAM ref (mkLastName ref) ParamPi s' Nothing)
      (refs <: ref)
      schT
  return (SchImplicitPi (x <: sd) schT', PIV x s' t')

```

We have been helped by *underneath*, which replaces (de Bruijn indexed) variables in a term with references from the given list.

```

underneath :: Bwd REF → INTM → INTM
underneath = underneath' 0
  where
    underneath' :: Int → Bwd REF → INTM → INTM
    underneath' _ B0 tm = tm
    underneath' n (rs <: ref) tm = underneath' (n + 1) rs (under n ref %% tm)

```

7.2.2 ProofState interface

For ease of use, *distillScheme* is packaged specially for easy ProofState usage.

```

distillSchemeHere :: Scheme INTM → ProofState (Scheme DInTmRN)
distillSchemeHere sch = do
  return · fst ≪≪ (liftErrorState DTIN $ distillScheme B0 B0 sch)
prettySchemeHere :: Scheme INTM → ProofState Doc
prettySchemeHere sch = do
  sch' ← distillSchemeHere sch
  return $ pretty sch' maxBound

```

7.3 The Moonshine distillery

7.3.1 Moonshining

The *moonshine* command attempts the dubious task of converting an Evidence term (possibly of dubious veracity) into a Display term. This is mostly for error-message generation. **question:** Presumably *moonshine* should accumulate Entries like *distill* and *friends*?

```
moonshine :: INTM → ProofStateT INTM DInTmRN
moonshine (LK t) = do
  t' ← moonshine t
  return $ DLK t'
moonshine (L (x : . t)) = do
  t' ← moonshine t
  return $ DL (x : •t')
moonshine (C c) = do
  c' ← traverse moonshine c
  return $ DC c'
moonshine (N n) = (do
  n' <: ty ← distillInfer B0 n []
  return $ DN n'
) ⊕ return (DTIN (N n))
moonshine t = return (DTIN t)
```

Chapter 8

Cochon

8.1 Loading Developments

8.1.1 Parsing a Development

To parse a development, we represent it as a list of DevLines, each of which corresponds to a Parameter or Definition entry and stores enough information to reconstruct it. Note that at this stage, we simply tag each definition with a list of commands to execute later.

```
data DevLine
  = DLParam ParamKind String DInTmRN
  | DLDef String [DevLine] (Maybe DInTmRN <: DInTmRN) [CTData]
  | DLModule String [DevLine] [CTData]
```

A module may have a list of definitions in square brackets, followed by an optional semicolon-separated list of commands.

```
parseDevelopment :: Parsley Token [DevLine]
parseDevelopment = bracket Square (many (pDef ⊕ pModule))
  ⊕ pure []
```

Parsing definitions

A definition is an identifier, followed by a list of children, a definition (which may be ?), and optionally a list of commands:

```
pDef :: Parsley Token DevLine
pDef = (| DLDef ident      -- identifier
        pLines           -- childrens (optional)
        pDefn            -- definition
        pCTSuffix       -- commands (optional)
        (%keyword KwSemi%)
  |)
```

Parsing children: Children, if any, are enclosed inside square brackets. They can be of several types: definitions, that we have already defined, or parameters. Parameters are defined below. The absence of sub-development is signaled by the := symbol.

```
pLines :: Parsley Token [DevLine]
pLines = bracket Square (many (pDef ⊕ pParam ⊕ pModule))
  ⊕ (keyword KwDefn * > pure [])
```

Parsing definitions: A definition is either a question mark or a term, ascripted by a type. The question mark corresponds to an open goal. On the other hand, giving a term corresponds to explicitly solving the goal.

```

pDefn :: Parsley Token (Maybe DInTmRN :<: DInTmRN)
pDefn = (| (%identEq " ? "%) (%keyword KwAsc%)
          ~Nothing :<: pDInTm
          | id pAsc
          |)
where pAsc = do
  tm :<: ty ← pAscription
  return $ Just tm :<: ty

```

Parsing commands: Commands can be typed directly in the developments by enclosing them inside [| . . . |] brackets. They are parsed in one go by *pCochonTactics*, so this is quite fragile. This is better used when we know things work.

```

pCTSuffix :: Parsley Token [CTData]
pCTSuffix = bracket (SquareB " ") pCochonTactics
  ⊕ pure []

```

Parsing Modules

A module is similar, but has no definition.

```

pModule :: Parsley Token DevLine
pModule = (| DLModule ident           -- identifier
            pLines                   -- children (optional)
            pCTSuffix                 -- commands (optional)
            (%keyword KwSemi%)
            |)

```

Parsing Parameters

A parameter is a λ -abstraction (represented by $\backslash x : T \rightarrow$) or a Π -abstraction (represented by $(x : S) \rightarrow$).

```

pParam :: Parsley Token DevLine
pParam = (| (%keyword KwLambda%) -- \
            (DLParam ParamLam)
            ident                 -- x
            (%keyword KwAsc%)      -- :
            (sizedDInTm (pred ArrSize)) -- T
            (%keyword KwArr%) |)   -- ->
  ⊕
  (bracket Round -- (
  (| (DLParam ParamPi)
    ident         -- x
    (%keyword KwAsc%) -- :
    pDInTm |)) < * -- S)
    keyword KwArr -- ->

```

8.1.2 Construction

Once we have parsed a development as a list of `DevLine`, we interpret it in the `ProofState` monad. This is the role of `makeDev`. It updates the proof state to represent the given list of `DevLines`, accumulating pairs of names and command lists along the way.

```

type NamedCommand = (Name, [CTData])
makeDev :: [DevLine] → [NamedCommand] → ProofState [NamedCommand]
makeDev [] ncs = return ncs
makeDev (l : ls) ncs = makeEntry l ncs >>= makeDev ls

```

Each line of development is processed by `makeEntry`. This is where the magic happens and the `ProofState` is updated.

```

makeEntry :: DevLine → [NamedCommand] → ProofState [NamedCommand]

```

Making a definition: To make a definition, we operate in 4 steps. First of all, we jump in a module in which we make our kids. Once this is done, we resolve our display syntax goal into a term: we are therefore able to turn the module in a definition. The third step consists in solving the problem if we were provided a solution, or give up if not. Finally, we accumulate the commands which might have been issued.

```

makeEntry (DLDef x kids (mtipTm <=: tipTys) commands) ncs = do
  -- Open a module named by her name
  n ← makeModule x
  goIn
  -- Recursively build the kids
  ncs' ← makeDev kids ncs
  -- Translate tipTys into a real INTM
  tipTy :=>: tipTyv ← elaborate' (SET :=>: tipTys)
  -- Turn the module into a definition of tipTy
  moduleToGoal tipTy
  -- Were we provided a solution?
case mtipTm of
  Nothing → do -- No.
    -- Leave
    goOut
  Just tms → do -- Yes!
    -- Give the solution tms
    elabGive tms
    return ()
  -- Is there any tactics to be executed?
case commands of
  [] → do -- No.
    -- Return the kids' commands
    return ncs'
  _ → do -- Yes!
    -- Accumulate our commands
    -- With the ones from the kids
    return $ (n, commands) : ncs'

```

Making a Module: Making a module involves much less effort than to make a definition. This is indeed a stripped-down version of the above code for definitions.

```

makeEntry (DLModule x kids commands) ncs = do
  -- Make the module
  n ← makeModule x
  goIn
  -- Recursively build the kids
  ncs' ← makeDev kids ncs
  -- Leave
  goOut
  -- Is there any tactics to be executed?
  case commands of
  [] → do -- No.
    -- Return the kids' commands
    return ncs'
  _ → do -- Yes!
    -- Accumulate our commands
    -- With the ones from the kids
    return $ (n, commands) : ncs'

```

Making a Parameter: To make a parameter, be it Lambda or Pi, is straightforward. First, we need to translate the type in display syntax to an INTM. Then, we make a fresh reference of that type. Finally, we store that reference in the development.

```

makeEntry (DLParam k x tys) ncs = do
  -- Translate the display tys into an INTM
  ty :=>: tyv ← elaborate' (SET :=>: tys)
  -- Make a fresh reference of that type
  freshRef (x <: tyv) (λref →
    -- Register ref as a Lambda
    putEntryAbove (EPARAM ref (mkLastName ref) k ty Nothing))
  -- Pass the accumulated commands
  return ncs

```

8.1.3 Loading the files

Once we have parsed a list of DevLines, we need to construct a Dev from them. The idea is to use commands defined in Section 4.5 to build up the proof state.

```

devLoad :: String → IO (Bwd ProofContext)
devLoad file = do
  -- Load the development file
  let devFile = dropExtension file ++ ".dev"
  devFileH ← (openFile devFile ReadMode >>= return · Just)
    'catchError' λ_ → return Nothing
  devLoad' devFileH $
    withFile file ReadMode readCommands

```

```

devLoad' :: Maybe Handle → IO [CTData] → IO (Bwd ProofContext)
devLoad' fileH commandLoad = do
  -- Load the development file
  dev ← case fileH of
    Just f → loadDevelopment f
    Nothing → return []
  -- Load the development in an empty proof state
  case runStateT (makeDev dev [] 'catchError' catchUnprettyErrors) emptyContext of
    Left errs → do
      putStrLn "Failed to load development:"
      putStrLn $ renderHouseStyle $ prettyStackError errs
      exitFailure
    Right (ncs, loc) → do
      -- Load the commands
      commands ← commandLoad
      -- Run them
      doCTacticsAt ([], commands) : ncs (B0 :< loc)

```

The following companion function takes care of the dirty details:

- Loading the content of the file;
- Tokenizing the file
- Parsing the development

```

loadDevelopment :: Handle → IO [DevLine]
loadDevelopment file = do
  f ← hGetContents file
  -- Tokenize the development file
  case parse tokenize f of
    Left err → do
      putStrLn $ "loadDevelopment: failed to tokenize:\n" ++
        show err
      exitFailure
    Right toks →
      -- Parse the development
      case parse parseDevelopment toks of
        Left err → do
          putStrLn $ "loadDevelopment: failed to parse:\n" ++
            show err
          exitFailure
        Right dev → do
          -- Return the result
          return dev

```

8.2 Cochon Command Lexer

Pierre: This needs some story.

8.2.1 Tokens

Because Cochon tactics can take different types of arguments, we need a tagging mechanism to distinguish them, together with projection functions.

```

data CochonArg = StrArg String
  | InArg DInTmRN
  | ExArg DExTmRN
  | SchemeArg (Scheme DInTmRN)
  | Optional CochonArg
  | NoCochonArg
  | ListArgs [CochonArg]
  | LeftArg CochonArg
  | RightArg CochonArg
  | PairArgs CochonArg CochonArg
deriving Show

```

8.2.2 Tokenizer combinators

```

tokenExTm :: Parsley Token CochonArg
tokenExTm = (| ExArg pDExTm |)

```

```

tokenAscription :: Parsley Token CochonArg
tokenAscription = (| ExArg pAscriptionTC |)

```

```

tokenInTm :: Parsley Token CochonArg
tokenInTm = (| InArg pDInTm |)

```

```

tokenAppInTm :: Parsley Token CochonArg
tokenAppInTm = (| InArg (sizedDInTm AppSize) |)

```

```

tokenName :: Parsley Token CochonArg
tokenName = (| (ExArg · (:\$[])) · DP) nameParse |)

```

```

tokenString :: Parsley Token CochonArg
tokenString = (| StrArg ident |)

```

```

tokenScheme :: Parsley Token CochonArg
tokenScheme = (| SchemeArg pScheme |)

```

```

tokenOption :: Parsley Token CochonArg → Parsley Token CochonArg
tokenOption p = (| Optional (bracket Square p)
  | NoCochonArg |)

```

```

tokenEither :: Parsley Token CochonArg → Parsley Token CochonArg
  → Parsley Token CochonArg
tokenEither p q = (| LeftArg p | RightArg q |)

```

```

tokenListArgs :: Parsley Token CochonArg → Parsley Token () → Parsley Token CochonArg
tokenListArgs p sep = (| ListArgs (pSep sep p) |)

```

```

tokenPairArgs :: Parsley Token CochonArg → Parsley Token () →
  Parsley Token CochonArg → Parsley Token CochonArg
tokenPairArgs p sep q = (| PairArgs p (%sep%) q |)

```

8.2.3 Printers

```
argToStr :: CochonArg → String
argToStr (StrArg s) = s
```

```
argToIn :: CochonArg → DInTmRN
argToIn (InArg a) = a
```

```
argToEx :: CochonArg → DExTmRN
argToEx (ExArg a) = a
```

```
argOption :: (CochonArg → a) → CochonArg → Maybe a
argOption p (Optional x) = Just $ p x
argOption _ NoCochonArg = Nothing
```

```
argList :: (CochonArg → a) → CochonArg → [a]
argList f (ListArgs as) = map f as
```

```
argEither :: (CochonArg → a) → (CochonArg → a) → CochonArg → a
argEither f g (LeftArg a) = f a
argEither f g (RightArg b) = g b
```

```
argPair :: (CochonArg → a) → (CochonArg → b) → CochonArg → (a, b)
argPair f g (PairArgs a b) = (f a, g b)
```

8.3 Cochon (Command-line Interface)

Here we have a very basic command-driven interface to the proof state monad.

```
cochon :: ProofContext → IO ()
cochon loc = cochon' (B0 :< loc)
```

```
cochon' :: Bwd ProofContext → IO ()
cochon' (locs :< loc) = do
  -- Safety belt: this *must* type-check!
  validateDevelopment (locs :< loc)
  -- Show goal and prompt
  putStr $ fst $ runProofState showPrompt loc
  hFlush stdout
  l ← getLine
  case parse tokenize l of
  Left pf → do
    putStrLn ("Tokenize failure: " ++ describePFailure pf id)
    cochon' (locs :< loc)
  Right ts →
    case parse pCochonTactics ts of
    Left pf → do
      putStrLn ("Parse failure: " ++ describePFailure pf (intercalate " " · map crushToken))
      cochon' (locs :< loc)
    Right cds → do
      locs' ← doCTactics cds (locs :< loc)
      cochon' locs'
```

```

paranoid = False
veryParanoid = False

```

```

validateDevelopment :: Bwd ProofContext → IO ()
validateDevelopment locs@(_ :< loc) = if veryParanoid
  then Data.Foldable.mapM_ validateCtxt locs -- XXX: there must be a better way to do that
  else if paranoid
    then validateCtxt loc
    else return ()
where validateCtxt loc = do
  case evalStateT (validateHere 'catchError' catchUnprettyErrors) loc of
    Left ss → do
      putStrLn "*** Warning: definition failed to type-check! ***"
      putStrLn $ renderHouseStyle $ prettyStackError ss
    _ → return ()

```

```

showPrompt :: ProofState String
showPrompt = do
  mty ← optional getHoleGoal
  case mty of
    Just (_ :=>: ty) → (| (showGoal ty) ++ showInputLine |)
    Nothing → showInputLine
  where
    showGoal :: TY → ProofState String
    showGoal ty@(LABEL _ _) = do
      h ← infoHypotheses
      s ← prettyHere · (SET:>:) ≪≪ bquoteHere ty
      return $ h ++ "\n" ++ "Programming: " ++ show s ++ "\n"
    showGoal ty = do
      s ← prettyHere · (SET:>:) ≪≪ bquoteHere ty
      return $ "Goal: " ++ show s ++ "\n"
    showInputLine :: ProofState String
    showInputLine = do
      mn ← optional getCurrentName
      case mn of
        Just n → return $ showName n ++ " > "
        Nothing → return "> "

```

```

describePFailure :: PFailure a → ([a] → String) → String
describePFailure (PFailure (ts, fail)) f = (case fail of
  Abort → "parser aborted."
  EndOfStream → "end of stream."
  EndOfParser → "end of parser."
  Expect t → "expected " ++ f [t] ++ "."
  Fail s → s
) ++ (if length ts > 0
  then ("\nSuccessfully parsed: ``" ++ f ts ++ "''.")
  else "")

```

A Cochon tactic consists of:

ctName the name of this tactic

ctParse a parser that parses the arguments for this tactic

ctIO an IO action to perform for a given list of arguments and current context

ctHelp the help text for this tactic

```
data CochonTactic =  
  CochonTactic { ctName :: String  
                , ctParse :: Parsley Token (Bwd CochonArg)  
                , ctIO   :: [CochonArg] → Bwd ProofContext → IO (Bwd ProofContext)  
                , ctHelp :: String  
                }
```

```
instance Show CochonTactic where
```

```
  show = ctName
```

```
instance Eq CochonTactic where
```

```
  ct1 ≡ ct2 = ctName ct1 ≡ ctName ct2
```

```
instance Ord CochonTactic where
```

```
  compare ct1 ct2 = compare (ctName ct1) (ctName ct2)
```

The *tacticsMatching* function identifies Cochon tactics that match the given string, either exactly or as a prefix.

```
tacticsMatching :: String → [CochonTactic]  
tacticsMatching x = case find ((x ≡) · ctName) cochonTactics of  
  Just ct → [ct]  
  Nothing → filter (isPrefixOf x · ctName) cochonTactics
```

```
tacticNames :: [CochonTactic] → String  
tacticNames = intercalate " , " · map ctName
```

Given a proof state command and a context, we can run the command with *runProofState* to produce a message (either the response from the command or the error message) and Maybe a new proof context.

```
runProofState :: ProofState String → ProofContext  
  → (String, Maybe ProofContext)  
runProofState m loc =  
  case runStateT (m 'catchError' catchUnprettyErrors) loc of  
    Right (s, loc') → (s, Just loc')  
    Left ss         → (renderHouseStyle $ prettyStackError ss, Nothing)  
  
simpleOutput :: ProofState String → Bwd ProofContext → IO (Bwd ProofContext)  
simpleOutput eval (locs :< loc) = do  
  case runProofState (eval < * startScheduler) loc of  
    (s, Just loc') → do  
      putStrLn s  
      return (locs :< loc :< loc')  
    (s, Nothing) → do  
      putStrLn "I'm sorry, Dave. I'm afraid I can't do that."  
      putStrLn s  
      return (locs :< loc)
```

We have some shortcuts for building common kinds of tactics: *simpleCT* builds a tactic that works in the proof state, and there are various specialised versions of it for nullary and unary tactics.

```

simpleCT :: String → Parsley Token (Bwd CochonArg)
          → ([CochonArg] → ProofState String) → String → CochonTactic
simpleCT name parser eval help = CochonTactic
  { ctName = name
  , ctParse = parser
  , ctIO = (λas → simpleOutput (eval as))
  , ctHelp = help
  }

```

```

nullaryCT :: String → ProofState String → String → CochonTactic
nullaryCT name eval help = simpleCT name (pure B0) (const eval) help

```

```

unaryExCT :: String → (DEXTmRN → ProofState String) → String → CochonTactic
unaryExCT name eval help = simpleCT
  name
  (| (B0:<) tokenExTm
   | (B0:<) tokenAscription |)
  (eval · argToEx · head)
  help

```

```

unaryInCT :: String → (DInTmRN → ProofState String) → String → CochonTactic
unaryInCT name eval help = simpleCT
  name
  (| (B0:<) tokenInTm |)
  (eval · argToIn · head)
  help

```

```

unDP :: DEXTm p x → x
unDP (DP ref :$[]) = ref

```

```

unaryNameCT :: String → (RelName → ProofState String) → String → CochonTactic
unaryNameCT name eval help = simpleCT
  name
  (| (B0:<) tokenName |)
  (eval · unDP · argToEx · head)
  help

```

```

unaryStringCT :: String → (String → ProofState String) → String → CochonTactic
unaryStringCT name eval help = simpleCT
  name
  (| (B0:<) tokenString |)
  (eval · argToStr · head)
  help

```

The master list of Cochon tactics.

```

cochonTactics :: [CochonTactic]
cochonTactics = sort (

```

Construction tactics:

```

nullaryCT "apply" (apply >> return "Applied.")
  "apply - applies the last entry in the development to a new subgoal."
: nullaryCT "done" (done >> return "Done.")
  "done - solves the goal with the last entry in the development."
: unaryInCT "give" (λtm → elabGiveNext tm >> return "Thank you.")
  "give <term> - solves the goal with <term>."
: simpleCT
  "lambda"
  (| (| bwdList (pSep (keyword KwComma) tokenString) (%keyword KwAsc%) |) :< tokenInTm
    | bwdList (pSep (keyword KwComma) tokenString)
    |)
  (λargs → case args of
    [] → return "This lambda needs no introduction!"
  - → case last args of
    InArg ty → Data.Traversable.mapM (elabLamParam · (:<:ty) · argToStr) (init args)
      >> return "Made lambda!"
  - → Data.Traversable.mapM (lambdaParam · argToStr) args
      >> return "Made lambda!"
  )
  ("lambda <labels> - introduces one or more hypotheses.\n" +
   "lambda <labels> : <type> - introduces new module parameters or hypotheses."

: simpleCT
  "let"
  (| (| (B0:<) tokenString |) :< tokenScheme |)
  (λ[StrArg x, SchemeArg s] → do
    elabLet (x :<: s)
    optional problemSimplify
    optional seekGoal
    return ("Let there be " ++ x ++ "."))
  "let <label> <scheme> : <type> - set up a programming problem with a scheme."

: simpleCT
  "make"
  (| (| (B0:<) tokenString (%keyword KwAsc%) |) :< tokenInTm
    | (| (B0:<) tokenString (%keyword KwDefn%) |) <><
    (| (λ(tm :<: ty) → InArg tm :> InArg ty :> F0) pAscription |)
    |)
  (λ(StrArg s : tyOrTm) → case tyOrTm of
    [InArg ty] → do
      elabMake (s :<: ty)
      goIn
      return "Appended goal!"
    [InArg tm, InArg ty] → do
      elabMake (s :<: ty)
      goIn
      elabGive tm
      return "Made."
  )
  ("make <x> : <type> - creates a new goal of the given type.\n"
   + "make <x> := <term> - adds a definition to the context.")

```

```
: unaryStringCT "module" ( $\lambda s \rightarrow \text{makeModule } s \gg \text{goIn} \gg \text{return "Made module."}$ )
  "module <x> - creates a module with name <x>."
```

```
: simpleCT
  "pi"
  (| (| (B0:<) tokenString (%keyword KwAsc%) |) :< tokenInTm |)
  ( $\lambda [\text{StrArg } s, \text{InArg } ty] \rightarrow \text{elabPiParam } (s :<: ty) \gg \text{return "Made pi!"}$ )
  "pi <x> : <type> - introduces a pi."
```

```
: simpleCT
  "program"
  (| bwdList (pSep (keyword KwComma) tokenString) |)
  ( $\lambda as \rightarrow \text{elabProgram } (\text{map } \text{argToStr } as) \gg \text{return "Programming."}$ )
  "program <labels>: set up a programming problem."
```

```
: nullaryCT "ungawa" ( $\text{ungawa} \gg \text{return "Ungawa!"}$ )
  "ungawa - tries to solve the current goal in a stupid way."
```

Navigation tactics:

```
: nullaryCT "in" ( $\text{goIn} \gg \text{return "Going in..."}$ )
  "in - moves to the bottom-most development within the current one."
: nullaryCT "out" ( $\text{goOutBelow} \gg \text{return "Going out..."}$ )
  "out - moves to the development containing the current one."
: nullaryCT "up" ( $\text{goUp} \gg \text{return "Going up..."}$ )
  "up - moves to the development above the current one."
: nullaryCT "down" ( $\text{goDown} \gg \text{return "Going down..."}$ )
  "down - moves to the development below the current one."
: nullaryCT "top" ( $\text{many goUp} \gg \text{return "Going to top..."}$ )
  "top - moves to the top-most development at the current level."
: nullaryCT "bottom" ( $\text{many goDown} \gg \text{return "Going to bottom..."}$ )
  "bottom - moves to the bottom-most development at the current level."
: nullaryCT "previous" ( $\text{prevGoal} \gg \text{return "Going to previous goal..."}$ )
  "previous - searches for the previous goal in the proof state."
: nullaryCT "root" ( $\text{many goOut} \gg \text{return "Going to root..."}$ )
  "root - moves to the root of the proof state."
: nullaryCT "next" ( $\text{nextGoal} \gg \text{return "Going to next goal..."}$ )
  "next - searches for the next goal in the proof state."
: nullaryCT "first" ( $\text{some prevGoal} \gg \text{return "Going to first goal..."}$ )
  "first - searches for the first goal in the proof state."
: nullaryCT "last" ( $\text{some nextGoal} \gg \text{return "Going to last goal..."}$ )
  "last - searches for the last goal in the proof state."
```

```
: unaryNameCT "jump" ( $\lambda x \rightarrow \text{do}$ 
  ( $n := \_$ )  $\leftarrow \text{resolveDiscard } x$ 
   $\text{goTo } n$ 
   $\text{return ("Jumping to " ++ showName } n \text{ ++ "...")}$ 
)
  "jump <name> - moves to the definition of <name>."
```

Miscellaneous tactics:

```

: CochonTactic
{ ctName = "execute"
, ctParse = (| (B0:<) tokenString |)
, ctIO = (λ[StrArg fn] (locs :< loc) → do
  exit ← system fn
  putStrLn $ if (exit ≡ ExitSuccess) then "Success." else "Failure."
  return (locs :< loc)
)
, ctHelp = "execute <command> - executes the given system command."
}

: CochonTactic
{ ctName = "help"
, ctParse = (| (B0:<) tokenString
  | B0
  |)
, ctIO = (λas locs → do
  case as of
  [] → putStrLn ("Tactics available: " ++ tacticNames cochonTactics)
  [StrArg s] → case tacticsMatching s of
  [] → putStrLn "There is no tactic by that name."
  cts → putStrLn (unlines (map ctHelp cts))
  return locs
)
, ctHelp = "help - displays a list of supported tactics.\n"
  ++ "help <tactic> - displays help about <tactic>.\n\n"
  ++ "What, you expected 'help help' to produce an amusing response?"
}

: CochonTactic
{ ctName = "quit"
, ctParse = pure B0
, ctIO = (λ_ _ → exitSuccess)
, ctHelp = "quit - terminates the program."
}

: CochonTactic
{ ctName = "save"
, ctParse = (| (B0:<) tokenString |)
, ctIO = (λ[StrArg fn] (locs :< loc) → do
  let Right s = evalStateT (much goOut ≫ prettyProofState) loc
  writeFile fn s
  putStrLn "Saved."
  return (locs :< loc)
)
, ctHelp = "save <file> - saves proof state to the given file."
}

```

```

: CochonTactic
{ ctName = "undo"
, ctParse = pure B0
, ctIO = (λ_ (locs :< loc) → case locs of
    B0 → putStrLn "Cannot undo." >> return (locs :< loc)
    _  → putStrLn "Undone."      >> return locs
    )
, ctHelp = "undo - goes back to a previous state."
}

: nullaryCT "validate" (validateHere >> return "Validated.")
"validate - re-checks the definition at the current location."

: CochonTactic
{ ctName = "load"
, ctParse = (| (B0:<) tokenString |)
, ctIO = (λ[StrArg file] locs → do
    commands ← withFile file ReadMode readCommands
    'catchError' \_ → do
        putStrLn $ "File " ++ file ++ " does not exist. Ignored."
        return []
    doCTactics commands locs)
, ctHelp = "load <f> - load the commands stored in <f>"
}

```

Import more tactics from an aspect:

```

import ← CochonTactics
: []

```

```

import ← CochonTacticsCode

```

```

pFileName :: Parsley Token String
pFileName = ident

```

```

type CTData = (CochonTactic, [CochonArg])

```

```

readCommands :: Handle → IO [CTData]
readCommands file = do
    f ← hGetContents file
    case parse tokenizeCommands f of
        Left err → do
            putStrLn $ "readCommands: failed to tokenize:\n" ++
                show err
            exitFailure
        Right lines → do
            t ← Data.Traversable.sequence $ map readCommand lines
            return $ Data.List.concat t

```

```

readCommand :: String → IO [CTData]
readCommand command =
  case parse tokenize command of
    Left err → do
      putStrLn $ "readCommand: failed to tokenize:\n" ++
        show err
      putStrLn $ "Input was: " ++ command
      exitFailure
    Right toks → do
      case parse pCochonTactics toks of
        Left err → do
          putStrLn $ "readCommand: failed to parse:\n" ++
            show err
          putStrLn $ "Input was: " ++ command
          exitFailure
        Right command → do
          return command

```

```

tokenizeCommands :: Parsley Char [String]
tokenizeCommands = (| id~[] (%pEndOfStream%)
  | id (%oneLineComment%)
    (%consumeUntil' endOfLine%)
    tokenizeCommands
  | id (%openBlockComment%)
    (%(eatNestedComments 0)%)
    tokenizeCommands
  | id (spaces * > endOfLine * > tokenizeCommands)
  | consumeUntil' endOfCommand :
    tokenizeCommands
|)
where endOfCommand = tokenEq ' ; ' * > spaces * > endOfLine
      ⊕ pEndOfStream * > pure ()
      endOfLine = tokenEq (head "\n") ⊕ pEndOfStream
      oneLineComment = tokenEq '-' * > tokenEq '-'
      openBlockComment = tokenEq '{' * > tokenEq '-'
      closeBlockComment = tokenEq '-' * > tokenEq '}'
      spaces = many $ tokenEq ' '
      eatNestedComments (-1) = (| id~() |)
      eatNestedComments i = (| id (%openBlockComment%)
        (eatNestedComments (i + 1))
      | id (%closeBlockComment%)
        (eatNestedComments (i - 1))
      | id (%nextToken%)
        (eatNestedComments i) |)

```

```

pCochonTactic :: Parsley Token CTDData
pCochonTactic = do
  x ← (| id ident
        | key anyKeyword
        |)
  case tacticsMatching x of
    [ct] → do
      args ← ctParse ct
      return (ct, trail args)
    [] → fail "unknown tactic name."
    cts → fail ("ambiguous tactic name (could be " ++ tacticNames cts ++ ").")

```

```

pCochonTactics :: Parsley Token [CTDData]
pCochonTactics = pSepTerminate (keyword KwSemi) pCochonTactic

```

```

doCTactic :: CTDData → Bwd ProofContext → IO (Bwd ProofContext)
doCTactic (ct, args) (locs :< loc) = ctIO ct args (locs :< loc)

```

```

doCTactics :: [CTDData] → Bwd ProofContext → IO (Bwd ProofContext)
doCTactics [] locs = return locs
doCTactics (cd : cds) locs = do
  locs' ← doCTactic cd locs
  doCTactics cds locs'

```

```

doCTacticsAt :: [(Name, [CTDData])] → Bwd ProofContext → IO (Bwd ProofContext)
doCTacticsAt [] locs = return locs
doCTacticsAt ((-, []) : ncs) locs = doCTacticsAt ncs locs
doCTacticsAt ((n, cs) : ncs) (locs :< loc) = do
  let Right loc' = execStateT (goTo n) loc
  locs' ← doCTactics cs (locs :< loc :< loc')
  doCTacticsAt ncs locs'

```

```

printChanges :: ProofContext → ProofContext → IO ()
printChanges from to = do
  let Right as = evalStateT getInScope from
      Right bs = evalStateT getInScope to
  let (lost, gained) = diff (as <>> F0) (bs <>> F0)
  if lost ≠ F0
    then putStrLn ("Left scope: " ++ showEntriesAbs (fmap reverseEntry (NF (fmap Right lost))))
    else return ()
  if gained ≠ F0
    then putStrLn ("Entered scope: " ++ showEntriesAbs (fmap reverseEntry (NF (fmap Right gained))))
    else return ()

```

```

diff :: (Eq a, Show a) ⇒ Fwd a → Fwd a → (Fwd a, Fwd a)
diff (x :> xs) (y :> ys)
  | x ≡ y = diff xs ys
  | otherwise = (x :> xs, y :> ys)
diff xs ys = (xs, ys)

```

8.4 Cochon error prettier

8.4.1 Catching the gremlins before they leave ProofState

```
catchUnprettyErrors :: StackError DInTmRN → ProofState a
catchUnprettyErrors e = do
  e' ← distillErrors e
  throwError e'
```

```
distillErrors :: StackError DInTmRN → ProofState (StackError DInTmRN)
distillErrors e = sequence $ fmap (sequence · fmap distillError) e
```

```
distillError :: ErrorTok DInTmRN → ProofState (ErrorTok DInTmRN)
distillError (ErrorVAL (v <: mt)) = do
  vTm ← bquoteHere v
  vDTm ← case mt of
    Just t → return · termOf ≪≪ distillHere (t >: vTm)
    Nothing → liftErrorState DTIN (moonshine vTm)
  return $ ErrorTm (vDTm <: Nothing)
distillError (ErrorTm (DTIN t <: mt)) = do
  d ← liftErrorState DTIN (moonshine t)
  return $ ErrorTm (d <: Nothing)
distillError e = return e
```

8.4.2 Pretty-printing the stack trace

```
prettyStackError :: StackError DInTmRN → Doc
prettyStackError e =
  vcat $
  fmap (text "Error: "< + >) $
  fmap hsep $
  fmap -- on the stack
  (fmap -- on the token
  prettyErrorTok) e
```

```
prettyErrorTok :: ErrorTok DInTmRN → Doc
prettyErrorTok (StrMsg s) = text s
prettyErrorTok (ErrorTm (v <: _)) = pretty v maxBound
prettyErrorTok (ErrorCan v) = pretty v maxBound
prettyErrorTok (ErrorElim e) = pretty e maxBound
```

The following cases should be avoided as much as possible:

```
prettyErrorTok (ErrorREF (name := _)) = text $ showName name
prettyErrorTok (ErrorVAL (v <: _)) = text "ErrorVAL" <>
  (brackets $ text $ show v)
```

8.5 Main

The following flags can be passed to the executable:

```

data Options = LoadFile FilePath
  | CheckFile FilePath
  | PrintFile FilePath
  | Interactive
  | Help

options :: [OptDescr Options]
options = [Option ['l'] ["load"] (ReqArg LoadFile "FILE") "Load the development "
, Option ['c'] ["check"] (ReqArg CheckFile "FILE") "Check the development "
, Option ['p'] ["print"] (ReqArg PrintFile "FILE") "Print the development "
, Option ['i'] ["interactive"] (NoArg Interactive) "Interactive mode"
, Option ['h'] ["help"] (NoArg Help) "Help! Help!"
]

```

Where CheckFile simply loads a development and terminates, whereas LoadFile drops to an interactive interface awaiting user's commands.

In case of error or explicit call to help, we display this nifty message:

```

message = "Epigram version (suc n)\n" ++
  "-----\n" ++
  "Usage:\n" ++
  "\tPig [options] [input file]\n\n" ++
  "Typing 'Pig --load FILE' has the same effect as 'Pig FILE'.\n" ++
  "If no input file is given, Pig starts in the empty context.\n" ++
  "Given the file name '-', Pig will read from standard input.\n\n" ++
  "Options: "

```

Finally, here is the *main*. Its role is simply to call *getOpt* and switch over the result. It's not extremely cute but there is no magic either.

```

main :: IO ()
main = do
  argv ← System.getArgs
  case getOpt RequireOrder options argv of
    -- Help:
    (Help : _, _, []) → do
      putStrLn $ usageInfo message options
    -- Load a development:
    (LoadFile file : _, _, []) → do
      loadDev file
    -- Check a development:
    (CheckFile file : _, _, []) → do
      withFile file (\loc → do
        validateDevelopment loc
        putStrLn "Loaded. ")
    -- Print a development:
    (PrintFile file : _, _, []) → do
      withFile file printTopDev
    -- Load a development (no flag provided):
    ([], (file : []), []) → do
      loadDev file
    -- Empty development:
    (Interactive : _, [], []) → do
      cochon emptyContext
    -- Empty development:
    ([], [], []) → do
      cochon emptyContext
    -- Error:
    (_, _, errs) → do
      ioError (userError (concat errs ++
        usageInfo message options))
  where
    withFile :: String → (Bwd ProofContext → IO a) → IO a
    withFile "-" g = devLoad' (Just stdin) (return []) ≧≧ g
    withFile file g = devLoad file ≧≧ g

```

```

loadDev :: String → IO ()
loadDev file = withFile file cochon'

```

```

printTopDev :: Bwd ProofContext → IO ()
printTopDev (_ :< loc) = do
  let Right s = evalStateT prettyProofState loc
  putStrLn s

```

Chapter 9

The Source Language

9.1 Structure

This is an implementation of the proposed `.epi` file structure, based on Conor’s notes. Everything is parameterised by a type for terms, so we get a bunch of traversable functors. This means we can’t easily enforce invariants on term direction, but the resulting simplification is probably worth it.

A *document* is a list of *constructions* (using slightly different terminology because we use “development” to mean something else in the proof state).

```
type EpiDoc t = [Construction t]

data Construction t = LetConstr (Decl t) (Maybe (Refinement t))
  deriving (Functor, Foldable, Traversable, Show)

data Decl t = Decl { declHyps      :: [Decl t]
                   , declTemplate :: String
                   , declType     :: t
                   }
  deriving (Functor, Foldable, Traversable, Show)

data Refinement t = Refinement { refProblem :: t
                                , refTactic  :: Tactic t
                                , refWhere   :: Block t
                                }
  deriving (Functor, Foldable, Traversable, Show)

data Tactic t = ReturnTac t
              | ByTac     t (Block t)
              | ShedTac

  deriving (Functor, Foldable, Traversable, Show)

type Block t = [Refinement t]
```

When we store terms, we potentially have three different representations: a `String` provided by the user, the result of parsing it to display syntax, and the result of elaborating it as an evidence term.

```
type Lexed      = String
type Parsed    = (String, DInTmRN)
type Elaborated = (String, DInTmRN, INTM)
```

9.2 Parser

For the moment, we set aside the question of how to get an EpiDoc Lexed and just explain how to parse the terms inside. Parsing documents has yet to be implemented: we should first lex the file to a string of tokens,

$$\text{lex} :: \text{String} \rightarrow \text{Either String [Token]}$$

then organise the tokens into a list of constructions,

$$\text{firstParse} :: [\text{Token}] \rightarrow \text{Either String (EpiDoc Lexed)}$$

and finally invoke the following to parse the terms within the constructions. (We might also want a better representation of parse errors.)

$$\begin{aligned} \text{parseEpiDoc} &:: \text{EpiDoc Lexed} \rightarrow \text{Either String (EpiDoc Parsed)} \\ \text{parseEpiDoc} &= \text{traverse parseConstr} \end{aligned}$$
$$\begin{aligned} \text{parseConstr} &:: \text{Construction Lexed} \rightarrow \text{Either String (Construction Parsed)} \\ \text{parseConstr} &= \text{traverse parseTerm} \end{aligned}$$

where

$$\begin{aligned} \text{parseTerm} &:: \text{String} \rightarrow \text{Either String (String, DInTmRN)} \\ \text{parseTerm } s &= \text{case parse tokenize } s \text{ of} \\ \text{Left } err &\rightarrow \text{Left (show err)} \\ \text{Right } toks &\rightarrow \text{case parse pDInTm } toks \text{ of} \\ \text{Left } err &\rightarrow \text{Left (show err)} \\ \text{Right } tm &\rightarrow \text{return (s, tm)} \end{aligned}$$

9.3 Elaborator

Here we explain how to elaborate constructions. The elaborator is going to need a lot of work in order to actually run these definitions.

First things first: to elaborate a term, we just use *subElab* defined before.

$$\begin{aligned} \text{elabTerm} &:: \text{Loc} \rightarrow \text{TY} \rightarrow \text{Parsed} \rightarrow \text{Elab Elaborated} \\ \text{elabTerm } loc \ ty \ (s, tm) &= \text{do} \\ \text{etm} :=>: _ &\leftarrow \text{subElab } loc \ (ty :>: tm) \\ \text{return } (s, tm, \text{etm}) \end{aligned}$$

The following cases need some thought about exactly what elaboration should do.

$$\begin{aligned} \text{elabConstr} &:: \text{Construction Parsed} \rightarrow \text{Elab (Construction Elaborated)} \\ \text{elabConstr (LetConstr decl mref)} &= \text{do} \\ \text{decl}' &\leftarrow \text{elabDecl decl} \\ \text{return (LetConstr decl}' \text{ Nothing)} \end{aligned}$$
$$\begin{aligned} \text{elabDecl} &:: \text{Decl Parsed} \rightarrow \text{Elab (Decl Elaborated)} \\ \text{elabDecl (Decl hyps } x \ ty) &= \\ &(| \text{Decl (traverse elabDecl hyps)} \sim x \ (\text{elabTerm (Loc 0) SET ty}) |) \end{aligned}$$
$$\begin{aligned} \text{elabRefinement} &:: \text{Refinement Parsed} \rightarrow \text{Elab (Refinement Elaborated)} \\ \text{elabRefinement (Refinement prob tac wbk)} &= \perp \end{aligned}$$

9.4 Example

```
plusC :: Construction Lexed
plusC = LetConstr
  (Decl [(Decl [] "x" "Nat"),
        (Decl [] "y" "Nat")
        ] "plus" "Nat")
  (Just (Refinement
        "plus x y"
        (ByTac "Nat.Ind x"
          [
            Refinement "plus 'zero y"
                      (ReturnTac "y")
                      [],
            Refinement "plus ('suc z) y"
                      ShedTac
                      []
          ]
        )
      )
  )
  []
))
```

```
parsePlusC :: Construction Parsed
parsePlusC = case parseConstr plusC of
  Right c → c
  Left e  → error e
```

```
elabPlusC :: Elab (Construction Elaborated)
elabPlusC = elabConstr parsePlusC
```

Chapter 10

Compiler

10.1 OpDef

Gadgets for building operators so that they will evaluate and compile coherently.

```
data OpDef = Arg (VAL → OpDef)      -- Any argument
           | ConArg (VAL → OpDef)   -- An argument that needs to be canonical
           | Body OpBody
```

```
data OpBody = OpCase OpBody [OpBody]
           | IsZero OpBody OpBody OpBody
           | Val VAL
           | Dec VAL (VAL → OpBody)
```

```
makeOpRun :: String → OpDef → [VAL] → Either NEU VAL
makeOpRun name (Arg fn) (v : vs) = makeOpRun name (fn v) vs
makeOpRun name (ConArg fn) (N t : vs) = Left t
makeOpRun name (ConArg fn) (CON v : vs) = makeOpRun name (fn v) vs
makeOpRun name (ConArg fn) (v : vs) = makeOpRun name (fn v) vs
makeOpRun name (Body b) [] = makeOpBody b where
  makeOpBody :: OpBody → Either NEU VAL
  makeOpBody (OpCase v vs) =
    case makeOpBody v of
      Right i@(C _) → if (num i < length vs) then
        makeOpBody (vs !! num i)
      else error $ "Missing case in " ++ name ++ ": " ++
        show i ++ ", " ++ show (length vs)
      Right (N t) → Left t
      Left t → Left t
  makeOpBody (IsZero v z s) =
    case makeOpBody v of
      Right ZE → makeOpBody z
      Right (SU _) → makeOpBody s
      Right (N t) → Left t
      Left t → Left t
  makeOpBody (Val v) = Right v
  makeOpBody (Dec (SU x) fn) = makeOpBody (fn x)
```

```

num :: VAL → Int
num ZE = 0
num (SU k) = (num k) + 1

```

```

makeOpRun name _ vs = error $ name ++ " stuck at " ++ show vs

```

Operator descriptions currently need to go here, with a signature in the .lhs-boot. The operator should also be added to OpCompile and OpGenerate. e.g.

```

import → OpCompile where
("switch", [e, x, p, b]) → App (Var "__switch") [Ignore, x, Ignore, b]

```

```

import → OpGenerate where
("switch", switchTest):

```

The the version to evaluate can be generated with 'makeOpRun':

```

switchOp = Op
  { opName = "switch"
  , opArity = 4
  , opTyTel = sOpTy
  , opRun = makeOpRun "switch" switchTest
  , opSimp = \_ → empty
  } where ...

```

The compiler decorates operator names with `__` (to prevent name clashes). Arguments which are not needed at run time should be replaced with 'Ignore' so that the compiler can erase them.

This is obviously not the neatest way of doing this. It would be better if all of this could be captured in the definition of Op.

```

switchTest :: OpDef
switchTest =
  ConArg (λarg → ConArg (λn →
    Arg (λp → Arg (λps →
      Body (IsZero (Val n)
        (Val (ps $$ Fst)))
        (Dec n (λk →
          Val (switchOp @@ [arg $$ Snd $$ Fst,
            k,
            L $ "x" : . [x · p - $ [SU (NV x)]],
            ps $$ Snd])))
          ))))

```

10.2 Compiler

This module uses Epic (`git clone git://github.com/edwinb/EpiVM.git`, or download from <http://github.com/edwinb/EpiVM>) to generate an executable from a collection of supercombinator definitions.

10.2.1 Representing Epic syntax

A CName, as one might expect, is a string name usable in C (or Epic). It should only contain alphanumeric characters and underscores.

```
type CName = String
```

The CNameable typeclass describes how to convert the Epigram representation of names into CNames.

```
class Show n => CNameable n where
  cname :: n -> CName
```

```
instance Show t => CNameable [(String, t)] where
  cname = concatMap (\(n, i) -> "_" ++ concatMap decorate n ++ "_" ++ show i)
  where decorate '_' = "___"
        decorate x | isAlphaNum x = [x]
                  | otherwise = '_' : (show (fromEnum x)) ++ "_"
```

```
instance CNameable REF where
  cname (x := d) = cname x
```

```
data CompileFn = Comp [CName] FnBody
```

```
data FnBody = Var CName
  | App FnBody [FnBody]
  | Case FnBody [FnBody] (Maybe FnBody) -- scrutinee, branches, default
  | Proj FnBody Int -- project from a tuple
  | CTag Int -- any tag
  | STag FnBody -- for Su
  | DTag FnBody -- decrement (for jump tables)
  | Let CName FnBody FnBody
  | Tuple [FnBody]
  | Lazy FnBody -- evaluate body lazily
  | Missing String
  | Ignore -- anything we can't inspect. Types, basically.
  | Error String
deriving Show
```

10.2.2 Compiling functions

The MakeBody typeclass describes how to convert a term into the body of a function (need to add the argument names elsewhere).

```
class MakeBody t where
  makeBody :: t -> FnBody
```

```
instance (CNameable n) => MakeBody (Tm {d, p} n) where
  makeBody (C can) = makeBody can
  makeBody (N t) = makeBody t
  makeBody (P x) = Var (cname x)
  makeBody (tm :$ elim) = makeBody (tm, elim)
  makeBody (op :@ args) = makeBody (op, args)
  makeBody (tm :? ty) = makeBody tm
  makeBody (L (K tm)) = App (Var "__const") [makeBody tm]
  makeBody x = error ("Please don't do this: makeBody " ++ show x)
```

Lots of canonical things are just there for the typechecker, and we don't care about them. So we'll just ignore everything that isn't otherwise explained.

```

instance CNameable n ⇒ MakeBody (Can (Tm {In, p} n)) where
  -- import j- CanCompile
  -- [Feature = Enum]
  makeBody Ze = CTag 0
  makeBody (Su x) = STag (makeBody x)
  -- [/Feature = Enum]
  -- [Feature = Labelled]
  makeBody (Label l t) = makeBody t
  makeBody (LRet t) = makeBody t
  -- [/Feature = Labelled]
  -- [Feature = Sigma]
  makeBody (Pair x y) = Tuple [makeBody x, makeBody y]
  makeBody Void = Tuple []
  -- [/Feature = Sigma]
  makeBody (Con t) = makeBody t
  makeBody _ = Ignore

instance CNameable n ⇒ MakeBody (Tm {Ex, p} n, Elim (Tm {In, p} n)) where
  makeBody (f, A arg) = appArgs f [makeBody arg]
  where appArgs :: Tm {d, p} n → [FnBody] → FnBody
    appArgs (f : $ (A a)) acc = appArgs f (makeBody a : acc)
    appArgs f acc = App (makeBody f) acc
  makeBody (f, Out) = makeBody f
  -- import j- ElimCompile
  -- [Feature = Labelled]
  makeBody (f, Call l) = makeBody f
  -- [/Feature = Labelled]
  -- [Feature = Sigma]
  makeBody (arg, Fst) = Proj (makeBody arg) 0
  makeBody (arg, Snd) = Proj (makeBody arg) 1
  -- [/Feature = Sigma]

```

Operators will, in many cases, just compile to an application of a function we write by hand in the Epic support file `epic/support.e`. **question: Why do we not report unknown operators sooner?**

```

instance CNameable n ⇒ MakeBody (Op, [Tm {In, p} n]) where
  makeBody (Op name arity _ _ , args)
    = case (name, map makeBody args) of
      -- import j- OpCompile
      -- [Feature = Enum]
      ("branches", _) → Ignore
      ("switch", [e, x, p, b]) → App (Var "__switch") [Ignore, x, Ignore, b]
      -- [/Feature = Enum]
      -- [Feature = IDesc]

  -- [/Feature = IDesc]
  -- [Feature = Sigma]
  ("split", [_ , _ , y, _ , f]) → App (Var "__split") [f, y]
  -- [/Feature = Sigma]
  _ → Lazy (Error ("Unknown operator" ++ show name))
  -- error ("Unknown operator" ++ show name)

```

10.2.3 Code generation

```
arglist = concat · (intersperse " , ")
```

The CodeGen typeclass describes things that are convertible to Epic code.

```
class CodeGen x where  
  codegen :: x → String
```

```
instance CodeGen (CName, CompileFn) where  
  codegen (n, def) = n ++ " " ++ codegen def
```

```
instance CodeGen CompileFn where  
  codegen (Comp args body) = "(" ++ arglist (map showarg args) ++ ")" -> Data = " ++  
    codegen body  
  where  
    showarg n = n ++ ":Data"
```

```
instance CodeGen FnBody where  
  codegen (Var x) = x  
  codegen (App f args) = codegen f ++ "(" ++ arglist (map codegen args) ++ ")"  
  codegen (Case sc opts def)  
    = "case " ++ codegen sc ++ " of { " ++  
      concat (intersperse " | "  
        (addDefault def (zipWith genOpt (map show [0..]) opts)))  
      ++ " } "  
  where addDefault Nothing opts = opts  
        addDefault (Just def) opts = opts ++ [genOpt "Default" def]  
        genOpt o c = o ++ " -> " ++ codegen c
```

```
codegen (Proj f i) = "(" ++ codegen f ++ "! " ++ show i ++ ")"  
codegen (CTag i) = show i  
codegen (STag n) = "1+" ++ codegen n  
codegen (DTag n) = codegen n ++ "-1"  
codegen (Let x v sc) = "(let " ++ x ++ ":Any = " ++ codegen v  
  ++ " in " ++ codegen sc ++ ")"  
codegen (Tuple xs) = "[" ++ arglist (map codegen xs) ++ "]"  
codegen (Lazy t) = "lazy(" ++ codegen t ++ ")"  
codegen (Missing m) = "error \"Missing definition " ++ m ++ "\""  
codegen Ignore = "42"  
codegen (Error s) = "error " ++ show s
```

10.2.4 Flattening and lambda-lifting

```
makeFns :: [(Name, Bwd Name, FnBody)] → [(CName, CompileFn)]  
makeFns xs = map (λ(n, args, tm) →  
  (cname n, Comp (map cname (trail args)) tm)) xs  
  ++ opGen
```

```

flatten :: ParamKind → Name → Bwd Name → Dev Fwd →
  [(Name, Bwd Name, FnBody)]
flatten b ma del (Dev F0 Module _ _) = []
flatten ParamLam ma del (Dev F0 (Unknown _) _ _) = [(ma, del, Missing (cname ma))]
flatten ParamLam ma del (Dev F0 (Defined tm _) _ _) =
  let (t, (_, defs)) = runState (lambdaLift ma del (fmap refName tm))
    (ma ++ [("lift", 0)], []) in
    (ma, del, makeBody t) : defs
flatten ParamAll ma del (Dev F0 _ _ _) = [(ma, del, Ignore)]
flatten ParamPi ma del (Dev F0 _ _ _) = [(ma, del, Ignore)]
flatten _ ma del dev@(Dev { devEntries = EPARAM (x := _) _ b _ _ :> es }) =
  flatten b ma (del :< x) dev { devEntries = es }
flatten b ma del dev@(Dev { devEntries = EDEF (her := _) _ _ herDev _ _ :> es }) =
  flatten ParamLam her del herDev ++ flatten b ma del dev { devEntries = es }

```

Lambda lifting: every lambda which is not at the top level is lifted out as a new top level definition. We keep track of the new top level functions we've added, and the next available name,

```

type LiftState = (Name, [(Name, Bwd Name, FnBody)])

```

```

nextName xs = reverse (nextName' (reverse xs))
where nextName' ((nm, i) : xs) = (nm, i + 1) : xs

```

```

addDef :: Name → (Bwd Name, InTm Name) →
  State LiftState ()
addDef nm (args, t) = do (n, fns) ← get
  put (n, (nm, args, makeBody t) : fns)

```

```

newName :: State LiftState Name
newName = do (nm, fns) ← get
  put (nextName nm, fns)
  return (nextName nm)

```

When we encounter a lambda, we create a new top level definition with all the arguments we've collected so far, plus the arguments to the lambda. Then replace the lambda with an application of the new function to all the names in scope.

```

lambdaLift :: Name → Bwd Name → Tm { d, TT } Name →
  State LiftState (Tm { d, TT } Name)
lambdaLift nm args l@(L (x : . t)) = lift args args l where
  lift :: Bwd Name → Bwd Name → (InTm Name) →
    State LiftState (InTm Name)
  lift tlargs args (L sc@(x : . t))
    = let name = nm ++ [(x, bwdLength args)] in
      lift tlargs (args :< name) (underScope sc name)
  lift tlargs args t = do t' ← lambdaLift nm args t
  name ← newName
  addDef name (args, t')
  return (N (apply (P name) tlargs))
apply f B0 = f
apply f (args :< a) = apply f args : $ (A (N (P a)))

```

Everything else is boring traversal of the term.

```

lambdaLift nm args (L (K t)) = do t' ← lambdaLift nm args t
  return (L (K t'))
lambdaLift nm args (C can) = (| C (traverse (lambdaLift nm args) can) |)
lambdaLift nm args (N t) = (| N (lambdaLift nm args t) |)
lambdaLift nm args (op : @ as) =
  (| ~op : @ (traverse (lambdaLift nm args) as) |)
lambdaLift nm args (t : $ el) =
  (| lambdaLift nm args t : $ traverse (lambdaLift nm args) el |)
lambdaLift nm args (v :? t) =
  (| lambdaLift nm args v :? (| (error "Can't happen") |) |)
lambdaLift nm args tm = (| tm |)

```

10.2.5 Invoking compilation

Where to look for support files. We'll need this to be a bit cleverer later. Only interested in `epic/support.e` for now (which is a good place to implement operators, for example).

```
libPath = [".", "./epic"]
```

The `compileCommand` takes the name of a main term to evaluate, a (forward) development containing the definitions, and an output file name. It calls Epic to produce an executable that evaluates the term, and returns whether compilation was successful.

```

compileCommand :: Name → Dev Fwd → String → IO Bool
compileCommand mainName dev outfile = do
  let flat = flatten ParamLam [] B0 dev
      output (makeFns flat) (cname mainName) outfile " "

```

Given a list of names and definitions, and the top level function to evaluate, write out an executable. This will evaluate the function and dump the result. Also take a list of extra options to give to Epic (e.g. for keeping intermediate code).

```

output :: [(CName, CompileFn)] → CName → FilePath → String → IO Bool
output defs mainfn outfile epicopts =
  do (epicFile, eh) ← tempfile
     Prelude.mapM_ ((hPutStrLn eh) . codegen) defs
     support ← readLibFile libPath "support.e"
     hPutStrLn eh support
     hPutStrLn eh (mainDef mainfn)
     hClose eh
     let cmd = "epic -checking 1 " ++ epicFile ++ " -o " ++ outfile ++ " " ++ epicopts
         exit ← system cmd
     return (exit ≡ ExitSuccess)

```

```

tempfile :: IO (FilePath, Handle)
tempfile =
  do env ← environment "TMPDIR"
     let dir = case env of
         Nothing → "/tmp"
         (Just d) → d
     openTempFile dir "Pig"

```

```

environment :: String → IO (Maybe String)
environment x = catch (do e ← getEnv x
  return (Just e))
  (\_ → return Nothing)

```

```

readLibFile :: [FilePath] → FilePath → IO String
readLibFile xs x = tryReads (map (λf → f ++ "/" ++ x) (". " : xs))
  where tryReads [] = fail $ "Can't find " ++ x
        tryReads (x : xs) = catch (readFile x)
          (λe → tryReads xs)

mainDef :: CName → String
mainDef m = "main () -> Unit = __dumpData(" ++ m ++ "())"

```

We add a Cochon tactic to invoke the compiler.

```

import → CochonTactics where
: CochonTactic
{ ctName = "compile"
, ctParse = (| (| (B0:<) tokenName |):< tokenString |)
, ctIO = (λ[ExArg (DP r :$[]), StrArg fn] (locs :< loc) → do
  let Right dev = evalStateT getAboveCursor loc
      Right (n := _) = evalStateT (resolveDiscard r) loc
      b ← compileCommand n (reverseDev dev) fn
      putStrLn (if b then "Compiled." else "EPIC FAIL")
      return (locs :< loc)
  )
, ctHelp = "compile <name> <file> - compiles the proof state with <name> as the
}

```

10.2.6 Operator definitions

Generating operator definitions from descriptions

```

makeOpCompile :: String → OpDef → CompileFn
makeOpCompile opname op = mkOp argNames [] op where
  mkOp (x : xs) args (Arg fn)
    = mkOp xs (args ++ [aname x])
      (fn (NP (aname x := fakeRef)))
  mkOp (x : xs) args (ConArg fn)
    = mkOp xs (args ++ [aname x])
      (fn (NP (aname x := fakeRef)))
  mkOp ns args (Body b) = Comp (map cname args) (mkDef ns b)

mkDef _ (Val v) = makeBody v
mkDef (x : xs) (Dec v scope)
  = Let (cname (aname x)) (DTag (makeBody v))
    (mkDef xs (scope (NP (aname x := fakeRef))))
mkDef xs (IsZero v z s)
  = Case (mkDef xs v) [mkDef xs z] (Just (mkDef xs s))

aname x = [(opname, 0), x]
fakeRef = error "Made up reference in makeOpCompile"

argNames = map (λi → ("x", i)) [1..]

```

```
opList :: [(String, OpDef)]
opList = (
  -- import j- OpGenerate
  -- [Feature = Enum]
  ("switch", switchTest):
  -- [/Feature = Enum]
  [])
```

```
opGen :: [(CName, CompileFn)]
opGen = map ( $\lambda(n, def) \rightarrow ("\_ " ++ n, makeOpCompile n def)$ ) opList
```

Appendix A

Kit

A.1 BwdFwd

Backward and forward lists, applicative with zipping.

```
data Bwd x = B0 | Bwd x <: x deriving (Show, Eq)
infixl 5 <:
data Fwd x = F0 | x >: Fwd x deriving (Show, Eq)
infixr 5 >:
```

```
bwdList :: [x] → Bwd x
bwdList = foldl (<:) B0
fwdList :: [x] → Fwd x
fwdList = foldr (>) F0
```

```
(<><) :: Bwd x → Fwd x → Bwd x
infixl 5 <><
xs <>< F0 = xs
xs <>< (y >: ys) = (xs <: y) <>< ys
```

```
(<>>) :: Bwd x → Fwd x → Fwd x
infixl 5 <>>
B0 <>> ys = ys
(xs <: x) <>> ys = xs <>> (x >: ys)
```

```
instance Applicative Bwd where
  pure x = pure x <: x
  (fs <: f) ⊗ (ss <: s) = (fs ⊗ ss) <: f s
  _ ⊗ _ = B0
```

```
instance Applicative Fwd where
  pure x = x >: pure x
  (f >: fs) ⊗ (s >: ss) = f s >: (fs ⊗ ss)
  _ ⊗ _ = F0
```

```
instance Monoid (Bwd x) where
  mempty = B0
  mappend xs B0 = xs
  mappend xs (ys <: y) = mappend xs ys <: y
```

```

instance Alternative Bwd where
  empty = mempty
  ( $\oplus$ ) = mappend

instance Monoid (Fwd x) where
  mempty = F0
  mappend F0      ys = ys
  mappend (x  $\succ$  xs) ys = x  $\succ$  mappend xs ys

class Applicative f  $\Rightarrow$  Naperian f where
  type Log f
  (!.) :: f x  $\rightarrow$  Log f  $\rightarrow$  Maybe x
  logTable :: f (Log f)

instance Naperian Bwd where -- cheeky!
  type Log Bwd = Int
  (xs < x) !. 0 = Just x
  (xs < x) !. i = xs !. (i - 1)
  _ !. _ = Nothing
  logTable = (1 + logTable) < 0

bwdLength :: Bwd x  $\rightarrow$  Int
bwdLength = getSum  $\cdot$  foldMap (\_  $\rightarrow$  Sum 1)

bwdNull :: Bwd x  $\rightarrow$  Bool
bwdNull B0      = True
bwdNull (_ < _) = False

bwdFoldCtxt :: (Fwd x  $\rightarrow$  t)  $\rightarrow$ 
  (t  $\rightarrow$  x  $\rightarrow$  Fwd x  $\rightarrow$  t)  $\rightarrow$ 
  Bwd x  $\rightarrow$  t
bwdFoldCtxt n s xs = help xs F0
  where help B0 ys = n ys
        help (xs < x) ys = s (help xs (x  $\succ$  ys)) x ys

elemIndex :: Eq x  $\Rightarrow$  x  $\rightarrow$  Bwd x  $\rightarrow$  Maybe Int
elemIndex x = bwdFoldCtxt (\_  $\rightarrow$  Nothing)
  ( $\lambda$ t y i  $\rightarrow$ 
    if y  $\equiv$  x then
      Just $ sum i
    else
      t)
  where sum = foldr' (\_ x  $\rightarrow$  x + 1) 0

```

A.2 Parsley

Here is a bargain basement parser combinator library. It does nothing fancy. It is hopelessly inefficient, but we can spend more effort when it becomes a more serious problem. In particular, we can easily represent extents numerically.

question: What are extents here?

A.2.1 Parsley's Semantics

A parser is represented as a function trying to transform a list of tokens t to:

- if it succeeds:
 - the tokens successfully consumed so far,
 - the lexeme generated by the tokenization, and
 - the remaining tokens to be read
- if it fails:
 - the tokens successfully consumed thus far,
 - the expected token

```
newtype PFailure  $t$  = PFailure ([ $t$ ], PError  $t$ ) deriving Show
```

```
data PError  $t$  = Abort
```

```
  | EndOfStream
```

```
  | EndOfParser
```

```
  | Expect  $t$ 
```

```
  | Fail String
```

```
  deriving Show
```

```
newtype Parsley  $t$   $x$  = Parsley { runParsley :: [ $t$ ] → Either (PFailure  $t$ ) ([ $t$ ],  $x$ , [ $t$ ]) }
```

The informal semantics given above is formalized by the *parse* function. A successful parse gives a lexeme, with no remaining tokens. In all other cases, something went wrong.

```
parse :: Parsley  $t$   $x$  → [ $t$ ] → Either (PFailure  $t$ )  $x$   
parse (Parsley  $p$ )  $ts$  = case  $p$   $ts$  of  
  Right ( $\_$ ,  $x$ , []) → Right  $x$   
  Right ( $ts$ ,  $\_$ ,  $cts$ ) → Left (PFailure ( $ts$ , EndOfParser))  
  Left  $e$  → Left  $e$ 
```

A.2.2 Structure

It's a Monad and all that.

```
instance Monad (Parsley  $t$ ) where
```

```
  return  $x$  = Parsley $  $\lambda ts$  → return ([],  $x$ ,  $ts$ )
```

```
  Parsley  $s$   $\gg=$   $f$  = Parsley $  $\lambda ts$  → do
```

```
    ( $sts$ ,  $s'$ ,  $ts$ ) ←  $s$   $ts$ 
```

```
    ( $tts$ ,  $t'$ ,  $ts$ ) ← runParsley ( $f$   $s'$ )  $ts$ 
```

```
    return ( $sts$  ++  $tts$ ,  $t'$ ,  $ts$ )
```

```
  fail  $s$  = Parsley $  $\lambda \_$  → fail  $s$ 
```

```
instance Functor (Parsley  $t$ ) where
```

```
  fmap =  $ap$  · return
```

```
instance Applicative (Parsley  $t$ ) where
```

```
  pure = return
```

```
  ( $\otimes$ ) =  $ap$ 
```

```
instance Alternative (Parsley  $t$ ) where
```

```
  empty = Parsley $  $\lambda \_$  → Left noMsg
```

```
   $p$   $\odot$   $q$  = Parsley $  $\lambda ts$  →
```

```
    either ( $\_$  → runParsley  $q$   $ts$ ) Right (runParsley  $p$   $ts$ )
```

```
instance MonadPlus (Parsley  $t$ ) where
```

```
  mzero = empty
```

```
  mplus = ( $\odot$ )
```

```

instance Error (PFailure t) where
  noMsg = PFailure ([], Abort)
  strMsg s = PFailure ([], Fail s)

```

A.2.3 Low-level combinators

You can consume the next token. Doing so, we could fail by hitting the end of the token stream.

```

nextToken :: Parsley t t
nextToken = Parsley $ \ ts → case ts of
  []      → Left (PFailure ([], EndOfStream))
  (t : ts) → Right ([t], t, ts)

```

You can consume everything! This always succeed.

```

pRest :: Parsley t [t]
pRest = Parsley $ \ ts → Right (ts, ts, [])

```

You can peek ahead, that is: we return a lexeme composed by all the tokens to come, but we do not consider them as consumed ([] on the left, *ts* on the right).

```

peekToken :: Parsley t [t]
peekToken = Parsley $ \ ts → Right ([], ts, ts)

```

You can make a parser give you the input extent it consumes as well as its output.

```

pExtent :: Parsley t x → Parsley t ([t], x)
pExtent (Parsley p) = Parsley $ \ ts → do
  (xts, x', ts) ← p ts
  return (xts, (xts, x'), ts)

```

A.2.4 High-level combinators

Based on the combinators defined in the previous section, we implement the combinators below without breaking the Parsley (...) abstraction.

You can test for the end of the token stream with *pEndOfStream*:

```

pEndOfStream :: Parsley t ()
pEndOfStream = guard ≪≪ (| null peekToken |)

```

Parsing separated sequences goes like this, purely applicative with She support. Either we can parse a *p* followed by many *sep* and *p*, or we return the empty list.

```

pSep :: Parsley t s → Parsley t x → Parsley t [x]
pSep sep p = (| p : (many $ sep * > p)
                | id ~ []
                |)

```

We can also allow an optional terminator for a separated sequence.

```

pSepTerminate :: Parsley t s → Parsley t x → Parsley t [x]
pSepTerminate sep p = pSep sep p < * optional sep

```

Similarly, one is often willing to consume some data up to some delimiter. This is the role of *consumeUntil*. It runs the parser *p* up to hitting a delimiter recognized by *delim*.

```

consumeUntil :: Parsley t a → Parsley t eol → Parsley t [a]
consumeUntil p delim = (| id~[] (%delim%)
  | p : (consumeUntil p delim) |)
consumeUntil' = consumeUntil nextToken

```

Thanks to the monadic nature of our parser, we can implement the following looping combinator. Hence, we can parse some input a with p and bind it. Then, we can try to use the dynamically generated parser l a . Failing that, we simply return a .

```

pLoop :: Parsley t a → (a → Parsley t a) → Parsley t a
pLoop p l = do
  a ← p
  pLoop (l a) l ⊕ pure a

```

Similarly, we can take advantage of the monadic nature of Parsley to do some post-processing on its output. Hence, $pMapFilter$ applies a *predicate* f to the result of p . The resulting parser is therefore a parser which recognizes valid p -inputs satisfying the predicate, and returning the result of f as tokens.

```

pFilter :: (a → Maybe b) → Parsley t a → Parsley t b
pFilter f p = do
  a ← p
  case f a of
    Just b → return b
    Nothing → empty

```

Here's a useful static operation:

```

pGuard :: Bool → Parsley t ()
pGuard True = (| () |)
pGuard False = (|)

```

A.2.5 Token manipulation

Based on the combinators above, we can already build some interesting parsers.

Hence, we can make a parser that matches the tokens satisfying a predicate, $tokenFilter$. Then, we can easily build a parser that matches a given token.

```

tokenFilter :: (t → Bool) → Parsley t t
tokenFilter p = pFilter (ok p) nextToken
  where ok :: (a → Bool) → a → Maybe a
    ok p a = (| a (%guard (p a)%) |)
tokenEq :: Eq t ⇒ t → Parsley t ()
tokenEq t = (| id~() (%tokenFilter (≡ t)%) |)

```

A.3 Missing Library

A.3.1 Renaming

```

trail :: (Applicative f, Foldable t, Monoid (f a)) ⇒ t a → f a
trail = foldMap pure

```

```

(⊕) :: Monoid x ⇒ x → x → x
(⊕) = mappend

```

$(\hat{\$}) :: (\text{Traversable } f, \text{Applicative } i) \Rightarrow (s \rightarrow i t) \rightarrow f s \rightarrow i (f t)$
 $(\hat{\$}) = \text{traverse}$

$\text{iter} :: (a \rightarrow b \rightarrow b) \rightarrow [a] \rightarrow b \rightarrow b$
 $\text{iter} = \text{flip} \cdot \text{Prelude.foldr}$

A.3.2 Indicator Function

$\text{indicator} :: (x \rightarrow \text{Bool}) \rightarrow x \rightarrow \text{Int}$
 $\text{indicator } p x = \text{if } p x \text{ then } 1 \text{ else } 0$

A.3.3 Newtype Unwrapping

class `Newtype n` **where**
type `Unwrap n`
 $\text{wrap} :: \text{Unwrap } n \rightarrow n$
 $\text{unwrap} :: n \rightarrow \text{Unwrap } n$

$\text{ala} :: \text{Newtype } v' \Rightarrow$
 $(t \rightarrow t') \rightarrow ((s \rightarrow t') \rightarrow u \rightarrow v') \rightarrow (s \rightarrow t) \rightarrow u \rightarrow \text{Unwrap } v'$
 $\text{ala } p h f u = \text{unwrap } (h (p \cdot f) u)$

instance `Newtype (Id a)` **where**
type `Unwrap (Id a) = a`
 $\text{wrap} = \text{Id}$
 $\text{unwrap} = \text{unId}$

instance `Newtype Any` **where**
type `Unwrap Any = Bool`
 $\text{wrap} = \text{Any}$
 $\text{unwrap} = \text{getAny}$

instance `Newtype (Sum a)` **where**
type `Unwrap (Sum a) = a`
 $\text{wrap} = \text{Sum}$
 $\text{unwrap} = \text{getSum}$

instance `Newtype (Endo a)` **where**
type `Unwrap (Endo a) = a → a`
 $\text{wrap} = \text{Endo}$
 $\text{unwrap} = \text{appEndo}$

newtype `AppLift a x = AppLift (a x)`

instance `(Applicative a, Monoid x) ⇒ Monoid (AppLift a x)` **where**
 $\text{mempty} = \text{AppLift } (\text{pure } \text{mempty})$
 $\text{mappend } (\text{AppLift } ax) (\text{AppLift } ay) = \text{AppLift } (\text{mappend } < \$ > ax \otimes ay)$

instance `Newtype (AppLift a x)` **where**
type `Unwrap (AppLift a x) = a x`
 $\text{wrap} = \text{AppLift}$
 $\text{unwrap } (\text{AppLift } ax) = ax$

A.3.4 Error Handling

```
catchMaybe :: MonadError [e] m ⇒ Maybe a → e → m a  
catchMaybe (Just x) _ = return x  
catchMaybe Nothing s = throwError [s]
```

```
catchEither :: MonadError [e] m ⇒ Either [e] a → e → m a  
catchEither (Right x) _ = return x  
catchEither (Left s) e = throwError (e : s)
```

```
throwError' :: MonadError [e] m ⇒ e → m a  
throwError' e = throwError [e]
```

```
pushError :: MonadError [e] m ⇒ m a → e → m a  
pushError c e = catchError c (λx → throwError (e : x))
```

A.3.5 Missing Applicatives

Ah, if only

```
instance Monad m ⇒ Applicative m
```

were possible. Unfortunately it isn't (at least without `UndecidableInstances`) so we have to do things the long way...

```
instance Applicative (State s) where  
  pure = return  
  (⊗) = ap
```

```
instance Applicative Identity where  
  pure = return  
  (⊗) = ap
```

```
instance Monad m ⇒ Applicative (ReaderT r m) where  
  pure = return  
  (⊗) = ap
```

```
instance MonadPlus m ⇒ Alternative (ReaderT r m) where  
  empty = mzero  
  (⊕) = mplus
```

```
instance Monad m ⇒ Applicative (StateT r m) where  
  pure = return  
  (⊗) = ap
```

```
instance MonadPlus m ⇒ Alternative (StateT r m) where  
  empty = mzero  
  (⊕) = mplus
```

```
instance Error x => Applicative (Either x) where
  pure = return
  (⊗) = ap
```

```
instance Error x => Alternative (Either x) where
  empty = Left $ strMsg "empty alternative"
  (Left _) ⊕ p = p
  p@(Right _) ⊕ _ = p
```

A.3.6 Missing Instances

```
instance Traversable (Either x) where
  traverse g (Left a) = pure (Left a)
  traverse g (Right b) = Right < $ > g b
```

```
instance Foldable (Either x) where
  foldMap = foldMapDefault
```

```
instance (Applicative f, Num x, Show (f x), Eq (f x)) => Num (f x) where
  x + y      = (| x + y |)
  x * y      = (| x * y |)
  x - y      = (| x - y |)
  abs x      = (| abs x |)
  negate x   = (| negate x |)
  signum x   = (| signum x |)
  fromInteger i = (| (fromInteger i) |)
```

```
instance Monoid o => Applicative (Writer o) where
  pure = return
  (⊗) = ap
```

Grr.

```
instance Monoid (IO ()) where
  mempty = return ()
  mappend x y = do x; y
```

A.3.7 HalfZip

```
class Functor f => HalfZip f where
  halfZip :: f x -> f y -> Maybe (f (x, y))
```

A.3.8 Functor Kit

```
newtype Id x = Id { unId :: x } deriving Show
newtype Ko a x = Ko { unKo :: a } deriving Show
data (p : + : q) x = Le (p x) | Ri (q x) deriving Show
data (p : * : q) x = p x : & q x deriving Show
```

```
instance Functor Id where
  fmap f (Id x) = Id (f x)
```

instance Functor (Ko a) **where**

fmap f (Ko a) = Ko a

instance (Functor p, Functor q) \Rightarrow Functor (p : + : q) **where**

fmap f (Le px) = Le (fmap f px)

fmap f (Ri qx) = Ri (fmap f qx)

instance (Functor p, Functor q) \Rightarrow Functor (p : * : q) **where**

fmap f (px : & qx) = fmap f px : & fmap f qx

instance Foldable Id **where**

foldMap = foldMapDefault

instance Foldable (Ko a) **where**

foldMap = foldMapDefault

instance (Traversable p, Traversable q, Foldable p, Foldable q) \Rightarrow Foldable (p : + : q) **where**

foldMap = foldMapDefault

instance (Traversable p, Traversable q, Foldable p, Foldable q) \Rightarrow Foldable (p : * : q) **where**

foldMap = foldMapDefault

newtype Fix f = InF (f (Fix f)) -- tying the knot

instance Show (f (Fix f)) \Rightarrow Show (Fix f) **where**

show (InF x) = "InF (" ++ show x ++ ") "

rec :: Functor f \Rightarrow (f v \rightarrow v) \rightarrow Fix f \rightarrow v

rec m (InF fd) = m

(fmap (rec m {-:: Fix f -> v -}))

(fd {-:: f (Fix f) -}))

{-:: f v -}))

instance Traversable Id **where**

traverse f (Id x) = Id < \$ > f x

instance Traversable (Ko a) **where**

traverse f (Ko c) = pure (Ko c)

instance (Traversable p, Traversable q) \Rightarrow Traversable (p : + : q) **where**

traverse f (Le px) = Le < \$ > traverse f px

traverse f (Ri qx) = Ri < \$ > traverse f qx

instance (Traversable p, Traversable q) \Rightarrow Traversable (p : * : q) **where**

traverse f (px : & qx) = (: &) < \$ > traverse f px \otimes traverse f qx

```

instance Applicative Id where -- makes fmap from traverse
  pure = Id
  Id f ⊗ Id s = Id (f s)

```

```

instance Monoid c ⇒ Applicative (Ko c) where -- makes crush from traverse
  -- pure :: x -> K c x
  pure x = Ko mempty
  -- (f*) :: K c (s -> t) -> K c s -> K c t
  Ko f ⊗ Ko s = Ko (mappend f s)

```

```

crush :: (Traversable f, Monoid c) ⇒ (x → c) → f x → c
crush m fx = unKo $ traverse (Ko · m) fx

```

```

reduce :: (Traversable f, Monoid c) ⇒ f c → c
reduce = crush id

```

```

instance Monoid Int where
  mempty = 0
  mappend = (+)

```

```

size :: (Functor f, Traversable f) ⇒ Fix f → Int
size = rec ((1+) · reduce)

```

```

instance HalfZip Id where
  halfZip (Id x) (Id y) = (| (Id (x, y)) |)

```

```

instance (Eq a) ⇒ HalfZip (Ko a) where
  halfZip (Ko x) (Ko y) | x ≡ y = (| (Ko x) |)
  halfZip _ _ = Nothing

```

```

instance (HalfZip p, HalfZip q) ⇒ HalfZip (p : + : q) where
  halfZip (Le x) (Le y) = (| Le (halfZip x y) |)
  halfZip (Ri x) (Ri y) = (| Ri (halfZip x y) |)
  halfZip _ _ = Nothing

```

```

instance (HalfZip p, HalfZip q) ⇒ HalfZip (p : * : q) where
  halfZip (x0 : & y0) (x1 : & y1) = (| (halfZip x0 x1) : & (halfZip y0 y1) |)

```

```

instance HalfZip [] where
  halfZip [] [] = (| [] |)
  halfZip (a : as) (b : bs) = (| ~ (a, b) : halfZip as bs |)

```

A.3.9 Applicative Kit

The *untilA* operator runs its first argument one or more times until its second argument succeeds, at which point it returns the result. If the first argument fails, the whole operation fails. If you have understood *untilA*, it won't take you long to understand *whileA*.

```

untilA :: Alternative f => f () -> f a -> f a
g 'untilA' test = g * > try
  where try = test ⊕ (g * > try)
whileA :: Alternative f => f () -> f a -> f a
g 'whileA' test = try
  where try = test ⊕ (g * > try)

```

The *much* operator runs its argument until it fails, then returns the state of its last success. It is very similar to *many*, except that it throws away the results.

```

much :: Alternative f => f () -> f ()
much f = (f * > much f) ⊕ pure ()

```

A.3.10 Monadic Kit

```

ignore :: Monad m => m a -> m ()
ignore f = do
  f
  return ()

```

A.4 Trace

Let us enumerate the different flavours of tracing available:

```

data Trace = ProofTrace
           | SimpTrace
           | ElimTrace
           | SchedTrace
           | ElabTrace
           deriving Show

```

We then can switch each one on or off individually:

```

traceEnabled :: Trace -> Bool
traceEnabled ProofTrace = True
traceEnabled SimpTrace = False
traceEnabled ElimTrace = False
traceEnabled SchedTrace = False
traceEnabled ElabTrace = True

```

That's fairly trivial, yet I'm pretty sure this goddamn laziness won't skip some traces (ML programmer speaking here).

```

monadTrace :: Monad m => Trace -> String -> m ()
monadTrace t s | traceEnabled t = do
  () <- trace ("[" ++ show t ++ "]" ++ s) $ return ()
  return ()
  | otherwise = return ()

```

Some handy aliases for the tracing function:

```

proofTrace = monadTrace ProofTrace
simpTrace = monadTrace SimpTrace
elimTrace = monadTrace ElimTrace
schedTrace = monadTrace SchedTrace
elabTrace = monadTrace ElabTrace

```

Bibliography

- [1] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, 2007.
- [2] Pierre Boutillier. Equality for λ -terms with list primitives. Technical report, Ecole Normale Supérieure de Lyon, 2009.
- [3] Ana Bove and Peter Dybjer. Dependent types at work. pages 57–99. 2009.
- [4] James Chapman. *Type checking and normalisation*. PhD thesis, University of Nottingham.
- [5] James Chapman, Thorsten Altenkirch, and Conor McBride. Epigram reloaded: a standalone typechecker for ett. In *6th Symposium on Trends in Functional Programming*, September 2005.
- [6] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. pages 75–114. 2002.
- [7] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. page 643. 1999.
- [8] Peter Dybjer and Anton Setzer. Induction-recursion and initial algebras. In *Annals of Pure and Applied Logic*, volume 124, 2000.
- [9] Peter Dybjer and Anton Setzer. Indexed induction-recursion. pages 93–113. 2001.
- [10] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning and Computation*. 2006.
- [11] Gérard Huet. The zipper. *J. Funct. Program.*
- [12] Conor McBride. Ornamental algebras, algebraic ornaments.
- [13] Conor McBride. Elimination with a motive. In *TYPES '00*, 2002.
- [14] Conor McBride and James McKinna. Functional pearl: i am not a number–i am a free variable. In *Haskell '04*, pages 1–9. ACM, 2004.
- [15] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
- [16] Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing strictly positive families. In *CATS '07*, pages 111–121. Australian Computer Society, Inc., 2007.
- [17] Benjamin C. Pierce and David N. Turner. Local type inference. In *POPL'98*, pages 252–265, New York, NY, 1998.

Index

elaboration problem, 167
 unstable, 167

simple proposition, 140