Core search

$$
\begin{aligned}
P, Q &\in \text{Atom} \\
F, G &\in \text{Formula} ::= P \mid P \sim Q \mid P \wedge Q \mid P \vee Q \mid R \to F \mid \forall x.F \mid \exists x.F \\
R, S &\in \text{Rule} ::= P \mid R \wedge S \mid R \vee S \mid F \to R \mid \forall x.R \mid \exists x.R \\
f, g &\in \text{Process} ::= \mathbf{add}\ P \mid \mathbf{add}\ P \sim Q \mid f + g \mid f \times g
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{sym} &= \text{fresh symbol} \\
\mathbf{var} &= \text{fresh variable}
\end{aligned}
$$

$$
\begin{aligned}
[\![F]\!]_R &: \text{Process} = \text{a search for a proof of } F \text{ given } R \\
[\![P]\!]_R &= \mathbf{add}\ P + [\![[\![R]\!]_P]\!]_R \\
[\![P \sim Q]\!]_R &= \mathbf{add}\ P \sim Q \\
[\![F \wedge G]\!]_R &= [\![F]\!]_R \times [\![G]\!]_R \\
[\![F \vee G]\!]_R &= [\![F]\!]_R + [\![G]\!]_R \\
[\![S \to F]\!]_R &= [\![F]\!]_{R \wedge S} \\
[\![\forall x.F]\!]_R &= [\![F[\mathbf{sym}/x]]\!]_R \\
[\![\exists x.F]\!]_R &= [\![F[\mathbf{var}/x]]\!]_R
\end{aligned}
$$

$$
\begin{aligned}
[\![R]\!]_P &: \text{Formula} = \text{subgoal produced by applying } R \text{ to } P \\
[\![Q]\!]_P &= (P \sim Q) \vee (Q \to P) \\
[\![R \wedge S]\!]_P &= [\![R]\!]_P \vee [\![S]\!]_P \\
[\![R \vee S]\!]_P &= [\![R]\!]_P \wedge [\![S]\!]_P \\
[\![F \to R]\!]_P &= F \wedge [\![R]\!]_P \\
[\![\forall x.R]\!]_P &= [\![R[\mathbf{var}/x]]\!]_P \\
[\![\exists x.R]\!]_P &= [\![R[\mathbf{sym}/x]]\!]_P
\end{aligned}
$$

Idea: $R \to P \Longleftarrow [\![R]\!]_P$.
Idea for $[\![R \wedge S]\!]_P$: $R \wedge S \to P \iff (R \to P) \vee (S \to P) \Longleftarrow [\![R]\!]_P \vee [\![S]\!]_P$.
Idea for $[\![R \vee S]\!]_P$: $R \vee S \to P \iff (R \to P) \wedge (S \to P) \Longleftarrow [\![R]\!]_P \wedge [\![S]\!]_P$.
Idea for $[\![F \to R]\!]_P$: $(F \to R) \to P \iff (R \to P) \wedge (F \vee P) \Longleftarrow F \wedge [\![R]\!]_P$.

- P is dropped because it would lead to useless searches: if

  $$[\![F \to R]\!]_P = [\![R]\!]_P \wedge (F \vee P)$$

  then

  $$
  \begin{aligned}
  [\![P]\!]_{F \to R} &= \mathbf{add}\ P + [\![[\![F \to R]\!]_P]\!]_{F \to R} \\
  &= \mathbf{add}\ P + [\![[\![R]\!]_P \wedge (F \vee P)]\!]_{F \to R} \\
  &= \mathbf{add}\ P + [\![[\![R]\!]_P]\!]_{F \to R} \times ([\![F]\!]_{F \to R} + [\![P]\!]_{F \to R})
  \end{aligned}
  $$

  and the nested $[\![P]\!]_{F \to R}$ can't do any better than the original query.

- In Prolog, R is a conjunction of inference rules of the form $\forall v_1 \cdots v_n.F_1 \to \cdots \to F_m \to P$ and this definition for $[\![F \to R]\!]_P$ agrees with SLD resolution.

Solvers

We want to be able to hook the core search algorithm up to a bunch of solvers that can solve constraints over specific domains (for example, if we hook things up to a solver for equality of s-expressions using first-order unification, we get Prolog). Solvers will often have typed variables (e.g. Int vs Bitvector in SMT). To handle this, extend Formula with $\forall(x : T), F \mid \exists(x : T), F$ and similar for Rule. T can be any expression the solver wants—it's just a way to pass extra information about a variable to a solver.

Solvers must support:

- (**add** P) to add constraints
- **push**, **pop** for backtracking
- At least one of {**sat?**, **unsat?**}
- **zonk** to grab satisfying assignments

Solvers can optionally support the special constraint $(P \sim Q)$ (presumably some kind of equality).

Solvers can optionally support variables:

- To use $\forall$ rule / $\exists$ goal, need **var**.
- To use $\exists$ rule / $\forall$ goal, need **sym**.
- In both cases, need substitution $e[y/x]$.

There are two ways in which solvers should be able to communicate with each other:

1. A constraint in one domain gives rise to constraints in other domains. For example, the Prolog constraint $(N + 5 \in \text{SMT}) :: XS \sim (1 \in \text{SMT}) :: nil$ should generate the SMT constraint $N + 5 = 1$. To implement this, we can simply allow solvers to add constraints to other solvers (e.g. in this case, the Prolog unification engine can add the constraint $N + 5 = 1$ to the SMT solver). When users define a new domain D, they can also specify how constraints in D give rise to constraints in other domains.
2. A solver for one domain can request a "translation" of a term from another domain. For example, suppose an SMT solver is given the constraint

$$\underbrace{\left(\int_0^{N+1\in\text{SMT}} x^2 \, \mathrm{d}x\right)}_{\text{Mathematica}} + 1 = M$$

   with a Mathematica expression with an SMT expression in it in it. The following exchange allows for the SMT solver to extract useful information from the Mathematica expression, even though it's from a completely different domain:

   - SMT to Mathematica: what is $\int_0^{N+1\in\text{SMT}} x^2 \, \mathrm{d}x$?
   - Mathematica to SMT: what is $N + 1$?
   - SMT to Mathematica: $N + 1$ is $N' + 1$.
   - Mathematica to SMT: $\int_0^{N+1\in\text{SMT}} x^2 \, \mathrm{d}x = \int_0^{N'+1} x^2 \, \mathrm{d}x = (N' + 1)^3/3$ is $(N + 1)^3/3$.
   - SMT adds constraint $(N + 1)^3/3 + 1 = M$.

   Notes:

   - The Mathematica expression $(N' + 1)^3/3$ got converted into an SMT expression $(N + 1)^3/3$.

- The exchange generated a fresh variable N' and remembered the association between $N \in$ SMT and $N' \in$ Mathematica.

These steps work only if the user has defined converters between Mathematica and SMT that work this way.

In the first case, constraints in Prolog are being converted into constraints in SMT. In the second case, terms are being converted back and forth between SMT and Mathematica. In both cases, the user has to somehow specify how all these conversions are supposed to work. If there are n domains then naively specifying every conversion requires $O(n^2)$ work. Thankfully things are easier than that: if there are conversions $g : A \to B$ and $f : B \to C$, then $f \circ g$ is a conversion $A \to C$, so computing the transitive closure of user-defined conversions can automatically derive many others. For example, if the user specifies conversions $g_i : D_i \to \text{SExp}, f_i : \text{SExp} \to D_i$ to/from a domain called SExp (which has no nontrivial constraint solving ability of its own) for each of n domains $D_1, \ldots, D_n$, we can automatically derive conversions to/from each pair of domains: for all $i, j \in \{1, \ldots, n\}$, $f_j \circ g_i : D_i \to D_j$.