1. Imports and Opened Namespaces

```
import Lean
import Batteries.Tactic.Exact
import Qq

open Lean Elab Command Term Meta Qq
```

- **Imports**: The code imports necessary modules:
  - `Lean`: The core Lean language functionalities.
  - `Batteries.Tactic.Exact`: Likely a custom or additional module providing enhanced tactics (assuming it's available in the environment).
  - `Qq`: A macro and metaprogramming utility for working with quoted expressions in Lean.
- **Namespaces**: Opens several Lean namespaces to make functions and types available without qualifying them:
  - `Lean`, `Elab`, `Command`, `Term`, `Meta`, `Qq`.

2. Theorems for Commutativity of Equality and Inequality

```
theorem Eq.comm'.{u} {α : Sort u} {a b : α} : (a = b) = (b = a) :=
propext Eq.comm
theorem Ne.comm'.{u} {α : Sort u} {a b : α} : (a ≠ b) = (b ≠ a) :=
propext ne_comm
```

- **Purpose**: These theorems express the commutativity of equality (`=`) and inequality (`≠`) in terms of propositional equality (`=`) rather than logical equivalence (`↔`).
- **Explanation**:
  - `Eq.comm` states that `a = b ↔ b = a`.
  - `propext` converts a logical equivalence into a propositional equality.
  - Thus, `Eq.comm'` states `(a = b) = (b = a)`.

3. Definition of `Lean.MVarId.congrWith`

```
partial def Lean.MVarId.congrWith (mvarId_ : MVarId) (eqs : List Expr)
: MetaM Unit := ...
```

- **Purpose**: Attempts to solve a metavariable (an unsolved goal in the proof) using congruence rules and provided equations, mirroring how Vampire uses congruence closure in its proofs.
- **Explanation**:
  - **Parameters**:
    - `mvarId_`: The metavariable (goal) to solve.
    - `eqs`: A list of equations that can be used in the congruence process.
  - **Process**:
    - Tries to solve the goal using `refl` (reflexivity).
    - If that fails, it tries to directly assign the goal to one of the provided equations (`eqs`).
    - If still unsuccessful, it breaks down the goal using congruence (`congrCore`) and recursively tries to solve the subgoals.
    - If necessary, it attempts to swap the sides of equalities or inequalities using `Eq.comm'` or `Ne.comm'` and tries again.

4. Definition of `Lean.Expr.eqswap`

```
def Lean.Expr.eqswap (e : Expr) : Expr := ...
```

- **Purpose**: Swaps the left-hand side (LHS) and right-hand side (RHS) of an equality or inequality expression.
- **Explanation**:
  - Checks if `e` is an equality (`a = b`) or inequality (`a ≠ b`).
  - If so, constructs a new expression with the LHS and RHS swapped (`b = a` or `b ≠ a`).
  - Returns the original expression if it's neither.

5. Definition of `mkEqNeSymm`

```
def mkEqNeSymm (h : Expr) : MetaM Expr := ...
```

- **Purpose**: Generates the symmetric version of an equality or inequality proof.
- **Explanation**:
  - If `h` is a proof of `a = b`, returns a proof of `b = a` using `Eq.symm`.
  - If `h` is a proof of `a ≠ b`, returns a proof of `b ≠ a` using `Ne.symm`.
  - Throws an error if `h` is neither.

6. Elaboration of `superpose`

```
elab "superpose " eq1:term:arg eq2:term:arg : term <= typ => do ...
```

- **Purpose**: Implements the superposition inference rule, which is a key rule in Vampire for handling equalities.
- **Explanation**:
  - **Process**:
    - Elaborates (`elabTerm`) and infers types for `eq1` and `eq2`.
    - Decomposes their types to instantiate any universal quantifiers.
    - Constructs a fresh metavariable `v` of type `e2 = typ`, where `e2` is the instantiated RHS of `eq2` and `typ` is the target type.
    - Uses `congrWith` to try to solve `v` using `eq1` and its symmetric version.
    - Constructs the final expression by applying the proven equality to `eq2`.
    - Ensures that the resulting expression has the correct type and that all metavariables are resolved (using `assumption`).

7. Definitions for Subsumption and Resolution

Subsumption Functions:

```
partial def doSubsumptionSingle {α : Type} (t1 : Q(Prop)) (target :
Q(Prop)) ...
partial def doSubsumptionWith {α : Type} (res : Q(Prop)) (t2 :
Q(Prop)) (target : Q(Prop)) ...
```

-
  - **Purpose**: These functions attempt to determine if one clause (logical formula) subsumes another, meaning one implies the other.
  - **Explanation**:
    - **`doSubsumptionSingle`**: Checks if a single term `t1` subsumes `target`, considering commutativity for equalities and inequalities.
    - **`doSubsumptionWith`**: Handles the case when `t2` is a disjunction (logical OR), recursively checking subsumption for each disjunct.

Resolution Functions:

```
partial def doResolutionSingle {α : Type} (t1 : Q(Prop)) (t2 :
Q(Prop)) ...
partial def doResolution (t1 : Q(Prop)) (t2 : Q(Prop)) ...
```

-
  - **Purpose**: Implement the resolution inference rule, which is fundamental in automated theorem proving for propositional and first-order logic.

- **Explanation**:
  - **`doResolutionSingle`**: Attempts to resolve a single term `t1` with `t2`, handling negations and potential commutativity.
  - **`doResolution`**: Recursively applies `doResolutionSingle` to disjunctive terms (logical ORs) in `t1`.

8. Elaboration of `resolve`

```
elab "resolve " eq1:term:arg eq2:term:arg : term <= typ => do ...
```

- **Purpose**: Provides a tactic to perform resolution between two clauses, similar to how Vampire applies resolution in its proofs.
- **Explanation**:
  - **Process**:
    - Elaborates `eq1` and `eq2`, infers their types, and instantiates any variables.
    - Attempts to resolve `eq1` with `eq2` to produce a term of type `typ`.
    - If the initial attempt fails, swaps `eq1` and `eq2` and tries again.
    - Uses the `doResolution` function to handle the resolution logic.
    - Applies the resulting function to the appropriate proof (`eq1` or `eq2`).

9. Elaboration of `subsumption`

```
elab "subsumption " eq1:term:arg eq2:term:arg : tactic => do ...
```

- **Purpose**: Implements a tactic to prove that one clause subsumes another, effectively showing that one implies the other.
- **Explanation**:
  - **Process**:
    - Retrieves the current goal (`goal`) and its type (`goalType`).
    - Attempts to match `eq1` or `eq2` to the goal, potentially swapping sides of equalities or inequalities.
    - Tries to resolve any remaining metavariables using `assumption`.
    - Assigns the proof to the goal if successful.

10. Macro Definition for `mod_symm`

```
macro "mod_symm " ar:term:arg : term => `(by first | exact $ar | exact
Eq.symm $ar | exact Ne.symm $ar)"
```

- **Purpose**: Provides a convenient way to attempt to apply a proof `ar` or its symmetric version to the current goal.
- **Explanation**:
    - The macro tries `exact $ar`, `exact Eq.symm $ar`, and `exact Ne.symm $ar` in order.
    - This is useful when the proof might require swapping sides of an equality or inequality.

11. Converting Vampire Proofs into Lean Proofs

- **Overall Strategy**:
    1. The code defines functions and tactics that correspond to the inference rules used by Vampire.
    2. By providing equivalents of Vampire's superposition, resolution, and subsumption in Lean, we can replicate the steps of a Vampire proof within Lean's proof environment.
- **Step-by-Step Conversion**:
    1. **Identify Proof Steps**: Break down the Vampire proof into individual inference steps (e.g., applications of superposition, resolution, subsumption).
    2. **Map to Lean Functions**: For each inference step, use the corresponding Lean function or tactic:
        - Use `superpose` for superposition steps.
        - Use `resolve` for resolution steps.
        - Use `subsumption` for subsumption steps.
    3. **Elaborate Terms**: Use `elabTerm` to elaborate terms and instantiate variables, aligning the Vampire proof terms with Lean's representation.
    4. **Handle Equations and Inequalities**: Use `Eq.comm'`, `Ne.comm'`, and functions like `mkEqNeSymm` to manage commutativity and symmetry as needed.
    5. **Solve Metavariables**: Use `congrWith` and other helper functions to resolve metavariables by matching them with known equations or applying congruence.
    6. **Construct Proofs**: Assemble the individual proof steps into a complete proof in Lean, ensuring that all types align and that the final proof matches the goal.
    7. **Verify and Optimize**: Use Lean's type checking and metavariable resolution to verify the correctness of each step and optimize the proof.

Example Workflow

Suppose Vampire has proven a theorem using the following steps:

1. **Superposition between Equations**: From `a = b` and `b = c`, derive `a = c`.
2. **Resolution with Negated Clause**: From `¬a = c` and `a = c`, derive a contradiction.

3. **Subsumption**: Show that a clause subsumes another.

Conversion into Lean Proof:

Step 1: Use `superpose` to derive `a = c`:
lean
Code kopieren
```
have h1 : a = b := ...
have h2 : b = c := ...
have h3 : a = c := superpose h1 h2
```

Step 2: Use `resolve` to derive a contradiction:
lean
Code kopieren
```
have h4 : ¬a = c := ...
have h5 : False := resolve h4 h3
```

Step 3: Use `subsumption` to show clause subsumption:
lean
Code kopieren
```
subsumption h_some_clause h_another_clause
```