An Introduction to the Ergo Theorem Prover

Peter Robinson

pjr@itee.uq.edu.au

School of Information Technology and Electrical Engineering,

The University of Queensland

Topics

- Ergo overview
- Sequent Calculus
- Simple Ergo proofs
- Tactics
- Rule Browsing
- Proofs involving quantifiers
- Window Inference
- Defining Theories
- Equality, Types, Undefinedness
- Future Directions

Ergo Overview

- Ergo is an interactive theorem prover based on Sequent Calculus.
- Ergo is implemented in Qu-Prolog - an extension of Prolog that provides built-in support for object variables, quantifiers and substitutions.
- The Ergo release comes with about 50 predefined theories, approximately 2000 theorems and many predefined tactics.
- It contains the predefined Gumtree proof interface which comes with its own Gumtree tactic language and tactic compiler.
- Ergo (and Qu-Prolog) can be downloaded from `http://www.itee.uq.edu.au/~pjr`

Ergo Architecture I

- Ergo maintains soundness while providing flexibility by using a layered architecture.
- The inner layer (the Ergo kernel) is responsible for managing proofs and the theory database.
- The theory database stores information about the theories: axioms, definitions, symbol declarations, theory inheritance, etc..
- The proof engine constructs proof trees by applying rules supplied by the user directly or via a tactic.

Ergo Architecture II

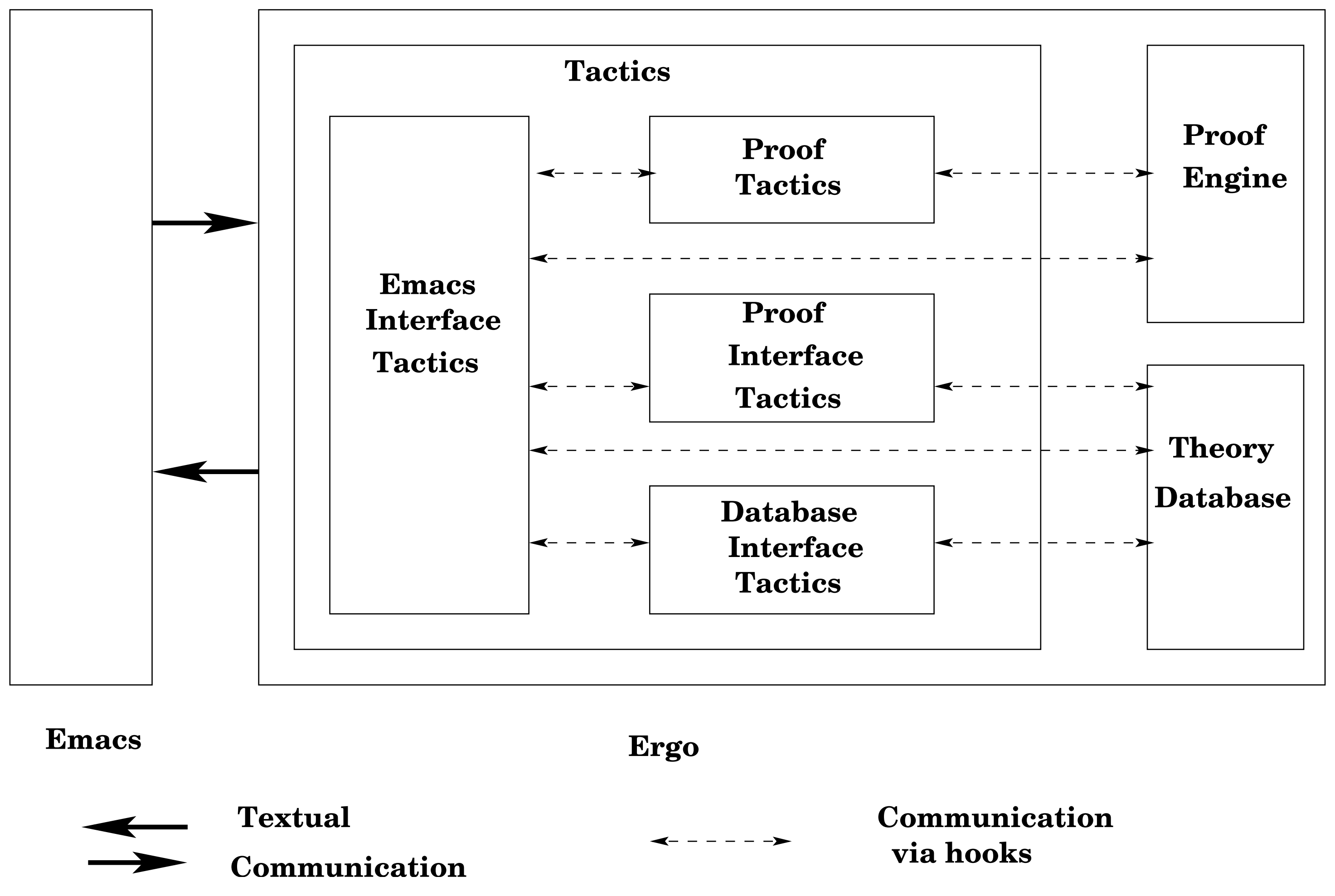
- The outer layer is where user interfaces and tactics live.
- Users are free to write and use their own interfaces and tactics without fear of producing unsound inferences.
- As an example at this level, Ergo comes with the predefined Gumtree proof interface and tactic language.

Ergo Architecture III

- The implementation contains several "hooks" which allow users to modify the behavior of some (non-critical) parts of the system.
- The implementation allows "documentation nodes" to be attached to code. This documentation is added to the database. Users are able to use a hook to determine how this documentation is to be displayed.
- In the emacs interface, a predefined hook causes the documentation to be displayed as hypertext within emacs. Another hook causes the documentation to be translated into latex form, and yet another causes the documentation to be translated to HTML form. The system makefile uses the last two hooks to produce both a latex and HTML reference manual.

Ergo Architecture IV

The architecture of Ergo showing the emacs interface



Sequent Calculus I

- In a sequent calculus proof, the main data structure is a sequent.
- A sequent consists of a ’goal’ (i.e. a formula to be proved) and hypotheses (assumed to be true for the proof of the goal).
- A proof in sequent calculus is typically done in a ’goal directed’ or ’backward’ style starting with what is to be proved and applying inference rules to reduce the problem to simpler problems until all (sub) problems have been proved.

Sequent Calculus II

- A sequent calculus proof constructs a ’proof tree’.
- A proof tree is a tree with each node being a sequent, where the root node is the sequent representing the theorem being proved.
- The children of each node is related to the node by the application of an inference rule.
- A node of a proof tree is ’closed’ if an inference rule has been applied to the node, and ’open’ otherwise.

Sequent Calculus - Example Rules I

We will use the notation
$\Delta \longrightarrow \Gamma$
to represent the sequent where Γ is the goal (conclusion) and Δ is the collection of hypotheses (antecedents).
In Ergo Γ is a single formula, but in some variations of Sequent Calculus Γ can also be a collection of formulae.

$$\frac{\Delta \longrightarrow A, \Delta \longrightarrow B}{\Delta \longrightarrow A \wedge B}$$

This rule is often called *and-r* or *and-intro*.

Sequent Calculus - Example Rules II

$$\frac{\Delta, A, B \longrightarrow \Gamma}{\Delta, A \wedge B \longrightarrow \Gamma}$$ (*and-l* or *and-elim*)

$$\frac{\Delta, A \longrightarrow B}{\Delta \longrightarrow A \Rightarrow B}$$ (*implies-r* or *implies-intro*).

$$\frac{\Delta \longrightarrow A}{\Delta \longrightarrow \forall x\ A}$$ provided x is not free in Δ

$$\frac{\Delta, \forall x\ A, [B/x]A \longrightarrow \Gamma}{\Delta, \forall x\ A \longrightarrow \Gamma}$$

where the notation $[B/x]A$ represents the term A with all free occurrences of x replaced by B.

Sequent Calculus - Semantics

To get the semantics of a sequent you conjoin the hypotheses, turn the sequent arrow into an implication, and universally quantify all the free variables.
The axioms of a given theory can then be justified by appealing to the semantics of the sequents by arguing that the conclusion of the rule follows from the premises of the rule at the semantics level.
For example, to justify the *all-intro* rule, we argue that the formula
$\Delta \Rightarrow \forall x\ A$
follows from the formula
$\forall x\ (\Delta \Rightarrow A)$

(given that x is not free in Δ)

Variables in Ergo

Before giving examples of Ergo rules we need to look at the kinds of variables that can occur in Ergo rules. (These are the same kinds of variables that appear in Qu-Prolog.)

- Variables we call *schematic variables* (also often called *meta-variables*) – these variables ‘range over’ **terms** within the object logic. These variables appear as strings starting with a capital letter.
- Variables we call *object variables* – these are also meta-level variables but they are restricted to range over the **variables** of the object logic. These variables appear as strings starting with a lower-case letter. They are distinguished from Qu-Prolog atoms by declaration (within the theory).

Context and Rule Constraints

- Ergo has been designed to allow for different kinds of hypotheses – each kind is called a *context* in Ergo. However, the only context used so far is called the `hyp` context and corresponds to the usual idea of hypotheses within Sequent Calculus.
- Ergo rules can have associated constraints, the most common being `not_free_in` constraints and `context_search` constraints that determines if a hypothesis of a given form is present.
- Other constrains are also possible and include user-defined programs (ofter referred to as *oracles*).

Ergo Rules

Ergo contains two kinds of rules:

- *Axioms* – (really primitive inference rules)
- *Theorems* – (really derived inference rules). Users declare theorems in their theory files but Ergo refers to them as *postulates* until they are proved.

Ergo Rule Example: and_intro

```
axiom and_intro ===
    A, B
    -------
    A and B.
```

The keyword `axiom` introduces an axiom. This is followed by the name of the axiom, then `===` which can be read as ‘is defined as’ and finally the statement of the axiom.

In this example the hypotheses are not mentioned and means that the current collection of hypotheses is unchanged by the application of the rule.

Ergo Rule Example: and_elim_c

```
theorem and_elim_c(A:thy(term), B:thy(term)) ===
    hyp---[CId ::: A and B]+++[A,B] ---> C
    --------------------------------------
    Node :::                             C
  provided
    context_search(hyp, Node, CId, A and B).
```

There are several things to notice about this rule:

- The rule contains arguments that have associated types – in this case the type says that the arguments are theory terms.
- The conclusion of the rule has a node identifier labelled `Node`.

rgo Rule Example: and_elim_c continue

```
theorem and_elim_c(A:thy(term), B:thy(term)) ===
    hyp---[CId ::: A and B]+++[A,B] ---> C
    --------------------------------------
    Node :::                             C
  provided
    context_search(hyp, Node, CId, A and B).
```

- The rule has an associated `context_search` constraint that requires that the proof node the rule is applied to must have an hypothesis that matches `A and B` (with context identifier `CId`)
- When the rule is applied the hypothesis `A and B` (with context identifier `CId`) is removed and the hypotheses `A` and `B` are added.

Ergo Rule Example: imp_intro

```
axiom imp_intro ===
    hyp+++[A] ---> B
    ----------------
    A => B.
```

Ergo Rule Example: all_intro

```
axiom all_intro(x:obvar) ===
    hyp ??? nfi(x) ---> A
    ---------------------
    all x A.
```

- This rule has an argument (of type `obvar` – i.e. it is an object variable) that is the variable bound by `all`.
- It also shows the application of a `not_free_in` filter to the hypotheses – it filters out all hypotheses that contains free occurrences of `x`.

Ergo Rule Example: all_elim_c, ex_elim_c

```
theorem all_elim_c(A:thy(term),x:obvar,T:thy(term)) ===
    hyp +++ [[T/x]A] ---> C
    ------------------------------------
    Node ::: C
  provided
    context_search(hyp, Node, CId2, all x A).

theorem ex_elim_c(x:obvar,A:thy(term)) ===
    hyp --- [CId ::: ex x A] ??? nfi(x) +++ [A] ---> C
    ------------------------------------
    Node ::: C
  provided
    context_search(hyp, Node, CId, ex x A),
    x not_free_in C.
```

Ergo Rule Example: do_arith_lt1

```
axiom do_arith_lt1(Exp1:thy(term), Exp2:thy(term)) ===
     Exp1 < Exp2
      provided  delay_until(ground(Exp1 < Exp2),
                                  arith_oracle(Exp1 < Exp2, true)).
```

- This is an example of a rule that uses a user-defined oracle.
- In this case `arith_oracle` is a user-defined Qu-Prolog predicate that interfaces with an infinite precision calculator.
- Because the calculator requires numbers (and not variables) the call on the oracle is delayed until no more variables remain in the expressions.

Proofs in Ergo

- For the example proofs we will use the Gumtree interface with the (experimental) Ergo GUI xergo
- The Gumtree interface is mostly interested in two kinds of proof nodes:
  - Open nodes – nodes which have not yet proved.
  - Current nodes – a subset of the open nodes that the user is currently working on.
- All proof commands the user supplies apply to the current nodes.
- If there is a mismatch between the number of current nodes and the number of nodes required by the proof command then the command fails.

Proof Commands in Ergo

There are four kinds of proof commands:

- Information commands that display information about the proof or the theory database.
- Proof management commands – save proofs, quit proofs.
- Tactic calls – these commands extend the proof tree by applying a tactic. A tactic could be as simple as a rule application or as complicated as a tactic that searches for a complete proof.
- Backtracking commands: retry – find another solution to the previous tactic call; undo – undo the effect of the previous tactic call.

Ergo Proof Example

We will now start up Ergo and do a few very simple proofs. The examples used throughout are based on the self-guided emacs tutorial for Ergo.

Start Ergo

Gumtree Tactics - Modes and Arities

- Modes (probably should be called types) describe the types of arguments to tactics.
- Arities describe how the tactic transforms proof nodes – the input arity is the number of nodes the tactic applies to; the output arity is the number of nodes the tactic produces.
- Gumtree tactics have a declaration part and definition part:
  - The declaration part gives the name of the tactic, the mode and arity and optional documentation.
  - The definition part gives the code for the tactic.

Gumtree Tactics - tacticals.gum

- The file `tacticals.gum` contains several useful tactics that combine other tactics – could be called higher-order tactics.
- The name comes from Isabelle where there is a distinction between tactics and tacticals that ‘glue’ tactics together in some way.
- Ergo makes no distinction between them – they are all tactics – its just that some may take tactics as arguments.

Gumtree Tactics - repeat I

```
tactic
  repeat(Tactic)
doc
  'Repeatedly call Tactic until it fails.',nl,
  'repeat(Tactic) always succeeds and all alternatives are',nl,
  'removed on termination.'
mode
  repeat(tactic)
arity
  _.

tactic repeat(Tactic) ===
    !(
        Tactic;
        repeat(Tactic)
        '|'
        skip
    ).
```

Gumtree Tactics - repeat II

- The `!` removes alternatives and is similar to `once` in Prolog.
- The `'|'` is tactic disjunction – it provides alternatives.
- `skip` is the tactic that matches against any number of open nodes and always succeeds.
- `;` is tactic composition

Gumtree Tactics - deep I

```
tactic
  deep(Tac)
doc
  'Try applying Tactic to each open node, and then apply deep(Tac)',nl,
  'to each successful result. Alternatives are removed from',nl,
  'each successful application of Tactic.'
mode
  deep(tactic)
arity
  _.

tactic deep(Tac) ===
    !(
        zero
        '|'
        [!(one;Tac;deep(Tac) '|' one), deep(Tac)]
    ).
```

Gumtree Tactics - deep II

- `zero` is the tactic that matches against zero open nodes (and succeeds in this case).
- `one` is the tactic that matches against exactly one open node.
- Note that the supplied tactic `Tac` must have input arity one.
- The initial `one` is there to enforce this constraint, but also to help the gumtree compiler produce efficient code.
- The Prolog list notation is used for parallel tactics where the tactics in the list partition the open nodes (in order) with each tactic applying to the corresponding partition.

Table Tactics

- Gumtree comes with a predefined mechanism for supporting user-defined collections of tactics.
- Users can declare the name of a tactic table.
- Users can then add entries into the table.
- The entries are typically rule applications but can be any tactic of input arity one.
- Users can then call the table as a tactic (with input arity one).

Table Tactics - Example I

```
declare_tactic_table
  simplify
doc
  'A table of rules for attempting a propositional proof.'.

erase_tactic_table(simplify).

in_tactic_table(simplify, context(hyp, ID, H);term(T);
                          {'=='(H, T)};rule(assump(ID))).
in_tactic_table(simplify, rule(true), true).
in_tactic_table(simplify, rule(not_not_true), not _).
in_tactic_table(simplify, rule(imp__imp__iff), _ <=> _).
in_tactic_table(simplify, imp_to_rule(and_intro_r), X and X).
in_tactic_table(simplify, rule(imp_intro), _ => _).
in_tactic_table(simplify, rule(hyp_iff_intro)).
in_tactic_table(simplify, rule(and_elim_c)).
.....
```

Table Tactics - Example II

- The optional third argument is for matching against the current goal.
- `context` is used to extract information form the context of the current open node.
- `term` is used to extract the goal of the current open node.
- `{}` is used to escape to the Qu-Prolog level.

Elves

- The `elves` tactic is a special (user-defined) tactic that is called before each user interaction with the Gumtree interpreter.
- It is typically used to carry out proof steps that might be considered to always be a good idea.
- It can be turned on and off by setting an Ergo parameter.
- This parameter can also be used to change the behavior of the `elves` tactic.

The init_proof tactic

- Ergo can prove derived inference rules that have premises.
- Ergo requires that the remaining open nodes in such proofs be in exactly the same order as in the statement of the theorem.
- This can be tricky to organize.
- The `init_proof` tactic gets around this problem by transforming the problem so that the premises are generated immediately (as open nodes) in the correct order, and are added to the hypotheses of the goal of the theorem.
- `init_proof` takes care of modifications to the hyps (including not-free-in filters)

Rule Browsing

- Ergo comes with a large number of predefined rules.
- Without help, it can be difficult to find rules that might be useful in a given situation.
- The rule browser is called as `rules(Query)` where `Query` is a term of the rule query language.
- The command `use(Query)` uses the rule browser to find a rule matching `Query` and then tries to apply the rule.

Back To Ergo

Proofs Involving Quantifiers

The following issues relate to proofs involving quantifiers and substitutions.

- Unification involving quantified terms can generate not-free-in constraints and substitutions (from a change of bound variable).
- Unification involving substitutions can generate delayed unification problems.
- Such constraints can lead to overspecialization in proofs. Ergo will reject proofs that contain relevant constraints that were not stated explicitly in the statement of the theorem being proved.

Using Rules Involving Quantifiers

In order to minimize the number of substitutions and not-free-in constraints it is often a good idea to use a variant of `rule` called `srule`.

- `srule` first tries ‘structural unification‘ – if that fails it reverts to normal unification (and so is the same as `rule`)
- Structural unification treats quantified terms and terms with substitutions in the same way as compound terms.
- If two terms structurally unify then they unify but typically don’t produce the most general unifier (at least when quantifiers and substitutions are involved).
- Because `srule` often reduces the number of substitutions and not-free-in constraints it can have a dramatic improvement on the performance of Ergo.

Solving Delayed Unification Problems

Ergo comes with a predefined Qu-Prolog predicate called `incomplete_retry_delays` that attempts to find solutions to the collection of delayed unification problems.

- Mostly, delayed unification problems are created because the problem has more than one solution.
- `incomplete_retry_delays` is a heuristic that tries to find (on backtracking)different solutions to a collection of delayed problems.

Dealing with Constraints

- If you attempt to save a proof where the constraints don't match the constraints specified in the statement of the theorem then Ergo will display a message showing the problem and then fail.
- Users can look at the constraints at any time during the proof by using the command `show_constraints`
- At any time users can call the tactic `{incomplete_retry_delays}` – i.e. escape to Qu-Prolog.
- Sometimes it is simpler for the user to help Ergo by supplying more information – e.g.
  - `{ x not_free_in A }` may simplify a delayed unification problem
  - `unify_terms(T1, T2)` may also help.

Window Inference I

- Window Inference is a form of transformational reasoning that tries to maximize the amount of context information that can be used to transform (sub)terms.
- It was the basic reasoning style for earlier versions of Ergo.
- In the current Ergo Window Inference is carried out using a combination of rules and tactics (that apply these rules).
- WI can be used to simplify expressions and to carry out proofs that are transformational in style - e.g. program refinement.
- Sequent Calculus and WI styles can be freely mixed within a single proof.

Window Inference II

There are three groups of WI tactics:

- Opening a ‘window’ – a single tactic supported by opening rules – gives access to subexpressions for transformation.
- Transformations – supported by transitivity rules and transformation rules.
- Closing a window – a single tactic supported by reflexivity rules.

Window Inference III

- A window is represented by a sequent whose goal is of the form `Expr WindowRelation SimpExpr` where
  - `Expr` is the expression to be transformed;
  - `WindowRelation` is the (transitive, reflexive) relation to be used; and
  - `SimpExpr` is the result.
  - Throughout, schematic variables are used for ‘placeholders’ that are incrementally instantiated as rules are applied.
- Typically `SimpExpr` is initially a schematic variable.

Opening Rule Example

```
theorem open_and_1_imp ===
    hyp+++[A] ---> C => B
    ------------------------------
    (C and A) => (B and A).
```

- Each rule has a standard form for the name.
- In this case `=>` is the window relation.
- This rule says that we can transform `C and A` to `B and A` (relative to `=>`) if we can transform `C` to `B` with the added hypothesis `A`.

Opening Rule Tactic

- `op(PosList)` is the tactic that, given a list of argument positions `PosList`, opens a window at the subexpression specified by the position list.
- The tactic first applies the required transitivity rule and then applies a sequence of opening rules that correspond to the positions given in the position list.

For example, below shows the rules applied by opening a window at position `[1,2]` in `(A => B) => C`

```
 not C, A --> B <=> Y
----------------------    (rule open_imp_2_iff)
not C --> (A => B) <=> A => Y
 ---------------------    (rule open_imp_1_iff)
(A => B) => C <=> (A => Y) => C   (A => Y) => C <=> X
----------------------------------------------    (rule iff__iff__iff)
           (A => B) => C <=> X
```

Transformational Tactics

- Transformational tactics use rules (or tactics) to transform one expression to another relative to the window relation.
- The simplest variant is `trans(Tactic)` where `Tactic` proves a theorem of the form `Expr WindowRelation SimpExpr`
- The tactic can strengthen or reverse the window relation to make the supplied tactic applicable.

Example transformation

Below is an example transformation using the following rule

```
theorem false_imp ===
    (false => A) <=> true.
```

```
  ------------------------    (rule false_imp)
   (false => B and C) <=> true
  -----------------------     (rule iff__imp)
  (false => B and C) => true  true => X
   ----------------------------------  (rule imp__imp__imp)
     (false => B and C) => X
```

Window Closing Tactic

- There is a single window closing tactic called `close_win`
- It simply applies the transitivity rule for the window relation.
- This typically instantiates a placeholder variable on the RHS to the expression on the LHS.

Building Theories I

Things to consider when building our theory:

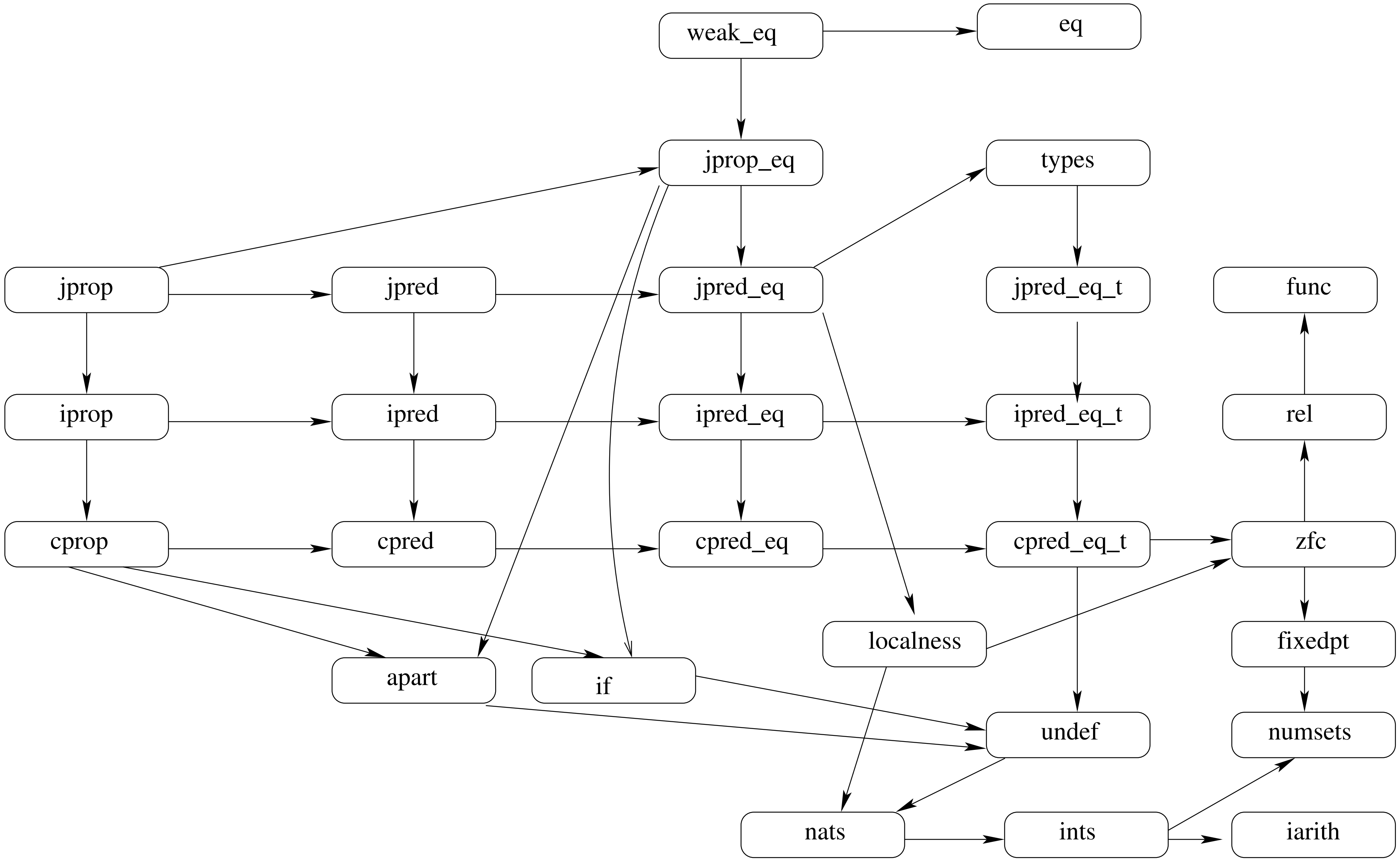
- What existing theories should we inherit?
- What are the ‘primitive‘ constants?
- What are the axioms?
- What are the other constants (definitions)?
- What are the theorems?

Building Theories II

Once we have constructed our theory then we might want to

- Prove the theorems!
- Write some tactics to help us prove those theorems.
- Write some tactics to help others use our theory.
- We might possibly provide a ‘tactic’ that describes how the current open nodes should be displayed to others (not so common).

Theory Graph for Predefined Theories



Weak Equality v. Strong Equality

- Strong equality allows replacement by equal terms at any subterm of a term.
- This makes strong equality easier to use - e.g. no extra conditions on rules for fixed points, induction, etc.
- Opening rules for equality follow directly from properties of strong equality.
- Weak equality requires rules about localness to get similar replacement properties.
- Strong equality is not suitable for modal logics such as those used in program refinement or theories about temporal properties.
- We want Ergo to directly support these kinds of theories so it relies mostly on weak equality.

Types and Undefinedness I

- Theorem provers implemented in a functional programming language typically require types to be associated with the declaration of each constant.
- For example, in a theory about integers, the declaration of the constant $+$ is accompanied by a type declaration that says that $+$ maps a pair of integers to an integer.
- By doing this the prover gets the same benefits from types as the underlying implementation language (and gets to use its typechecker!).

Types and Undefinedness II

- Ergo does not use types in this way. Constant declarations do not come with associated types – just arities.
- A given theory can, of course, include constants that represent types and provide type inference rules – e.g.

```
theorem add_ints_is_int ===
 X:ints, Y:ints
 ---------------
 X + Y : ints.
```

- This mean Ergo does type inference by rule application – which is less efficient than the other approach BUT makes type inference explicit.

Types and Undefinedness III

- A major advantage of the Ergo approach is in managing undefinedness (and partial functions).
- The `undef` theory of Ergo declares the constant `?` to represent undefinedness (often called ‘bottom’).
- We can then write down rules that produce or process undefinedness – e.g.

```
theorem add_undef === not (X:ints and Y:ints) <=> X + Y = ? .
theorem undef_add_undef === ? + X = ? .
theorem div0_undef === X div 0 = ? .
```

- Dealing with undefinedness in provers that require constants to have types is nowhere near as straightforward.

Future Directions

- Development of theories for the B specification language.
- Adding support for theory interpretation and instantiation – existed in earlier versions but not ported yet.
- Populating Ergo with more tactics.
- Further development of xergo
  - Support for exploring/annotating existing proofs – useful for giving example proofs for new users and for auditing proofs.
  - Dumping mathematical textbook style summaries of proof.
  - Adding a GUI interface to the rule browser.