Zippy – Generic White-Box Proof Search with Zippers

Kevin Kappelmann

February 6, 2026

Abstract

This entry contains *Zippy*, a framework for tree-based searches. Zippy is largely independent of concrete search tree representations, search-algorithms, states and effects. It is designed to create analysable and navigable searches that are open to customisation and extensions by users. An accompanying arXiv preprint is available [2].

This entry also provides a concrete instantiation of the framework in the form of a general purpose white-box prover, called *zip*. The prover performs a proof tree search with customisable expansion actions and search strategies, including A*, breadth-first, depth-first, and best-first search. By default, it integrates the classical reasoner, simplifier, the blast and metis prover, and supports resolution with higher-order and proof-producing unification, conditional substitutions, case splitting, and induction, among other things. Users are free to extend the prover with additional expansion actions and search strategies. We demonstrate the capabilities of zip in an examples theory.

In most cases, zip can be used as a drop-in replacement for Isabelle's classical methods, including auto, fastforce, force, fast, etc. We demonstrate this with a benchmark containing 2267 method calls from Isabelle's standard library, where zip achieves a success rate of 99.82% (2263/2267).

The Zippy framework is founded on concepts from functional programming theory, particularly zippers, arrows, monads, lenses, and coroutines. This entry contains a library of mentioned concepts for Isabelle/ML.

Contents

0.1 ML Arguments Antiquotations

theory *ML-Args-Antiquotations*
imports
ML-Unification.ML-Functor-Instances
begin

Summary Antiquotation for lists of ML arguments

⟨*ML*⟩

end

Chapter 1

ML Typeclasses

1.1 Basic Setup for Generic Typeclasses

theory *Gen-ML-Typeclasses-Base*
imports *ML-Args-Antiquotations*
keywords *ML-gen-file* :: *thy-decl*
begin

⟨*ML*⟩

Setup alternative **ML-file** command to avoid errors when loading files twice. This is needed since we provide ML files whose source depends on context variables and that should be loadable in different contexts.

⟨*ML*⟩

end

1.2 ML Eval Antiquotation

theory *ML-Eval-Antiquotation*
imports
ML-Unification.ML-Functor-Instances
begin

Summary Antiquotation for ML evaluation.

⟨*ML*⟩

end

1.3 Typeclasses

theory *ML-Typeclasses-Base*
imports

Gen-ML-Typeclasses-Base
ML-Eval-Antiquotation
begin

declare [[*ParaT-args args*: [*'p1*] *sep*: , *encl*: , *encl-arg*: *stop*:]]

⟨*ML*⟩

ML-gen-file‹*typeclass-base.ML*›
ML-gen-file‹*typeclass-base-instance.ML*›

end

1.4 Bi-Typeclasses

theory *ML-Bi-Typeclasses*
imports *ML-Typeclasses-Base*
begin

ML-gen-file‹*bi-typeclass.ML*›

end

1.5 Categories

theory *ML-Categories*
imports
ML-Typeclasses-Base
ML-Unification.ML-General-Utils
begin

ML-gen-file‹*category.ML*›
ML-gen-file‹*category-instance.ML*›

ML-gen-file‹*category-util.ML*›

end

1.6 Coroutines

theory *ML-Coroutines*
imports
ML-Categories
begin

ML-gen-file‹*coroutine.ML*›
ML-gen-file‹*coroutine-util.ML*›

end

1.7 Indexed Typeclasses

theory *ML-ITypeclasses-Base*
imports
ML-Typeclasses-Base
begin

ML-gen-file‹*itypeclass-base.ML*›
ML-gen-file‹*itypeclass-base-instance.ML*›

end

1.8 Indexed Categories

theory *ML-ICategories*
imports
ML-Categories
ML-ITypeclasses-Base
begin

ML-gen-file‹*icategory.ML*›
ML-gen-file‹*icategory-instance.ML*›

ML-gen-file‹*icategory-util.ML*›

end

1.9 Lenses

theory *ML-Lenses*
imports
ML-ICategories
begin

ML-gen-file‹*lens.ML*›

⟨*ML*⟩

end

1.10 State

theory *ML-State-Base*
imports *Gen-ML-Typeclasses-Base*

begin

⟨*ML*⟩

end

theory *ML-State*
imports
ML-ICategories
ML-State-Base
begin

ML-gen-file‹*state.ML*›
ML-gen-file‹*istate.ML*›

end

theory *ML-Typeclasses*
imports
ML-Bi-Typeclasses
ML-Categories
ML-Coroutines
ML-Lenses
ML-State
ML-Typeclasses-Base
begin

end

theory *ML-ITypeclasses*
imports
ML-ICategories
ML-ITypeclasses-Base
ML-State
begin

end

1.10.1 Metis

theory *Extended-Metis-Data*
imports
HOL.Metis
ML-Unification.ML-Functor-Instances
ML-Unification.ML-Logger
SpecCheck.SpecCheck-Show
begin

⟨*ML*⟩

end

1.11 Tactic Utils

theory *Zippy-ML-Tactic-Utils*
imports
ML-Typeclasses-Base
begin

⟨*ML*⟩

end

1.11.1 Cases

theory *Cases-Tactics*
imports
ML-Unification.ML-Tactic-Utils
Zippy-ML-Tactic-Utils
begin

⟨*ML*⟩

end

theory *Cases-Tactics-HOL*
imports
Cases-Tactics
HOL.HOL
begin

⟨*ML*⟩

For a function $f :: T_1 \Rightarrow \ldots \Rightarrow T_n \Rightarrow T$ with multiple arguments, the function package creates a cases rule $f.cases$ where f's arguments are tupled and equated to a single variable. As a result, one has to supply the cases tactic a single instantiation $(t_1,\ldots,t_n)$ when using $f.cases$ while users would expect being able to supply n instantiations $t_1,\ldots,t_n$. Below attribute transforms such rules to the expected form.

⟨*ML*⟩

end

1.11.2 Blast

theory *Extended-Blast-Data*

imports
HOL.HOL
ML-Unification.ML-Functor-Instances
ML-Unification.ML-Logger
SpecCheck.SpecCheck-Show
begin

⟨*ML*⟩

end

1.12 Term Index Data

theory *Generic-Term-Index-Data*
imports
ML-Unification.ML-Functor-Instances
ML-Unification.ML-General-Utils
ML-Unification.ML-Logger
ML-Unification.ML-Term-Index
SpecCheck.SpecCheck-Show
begin

⟨*ML*⟩

end

1.13 Priority Queues

theory *ML-Priority-Queues*
imports
Pure
begin

⟨*ML*⟩

end

1.14 Actions

theory *Zippy-Actions-Base*
imports
SpecCheck.SpecCheck-Show
begin

⟨*ML*⟩

end

Chapter 2

Zippy

theory *Zippy-Base-Setup*
imports
ML-Unification.ML-Logger
ML-Unification.Setup-Result-Commands
begin

Summary Zippy is a tree-search framework based on (alternating) zippers.

setup-result *zippy-base-logger* = ‹*Logger.new-logger Logger.root Zippy-Base*›

end

2.1 ML Indexed-Map Antiquotation

theory *ML-IMap-Antiquotation*
imports
ML-Unification.ML-Functor-Instances
ML-Unification.ML-General-Utils
begin

Summary Antiquotation for indexed maps

⟨*ML*⟩

end

2.2 Generic Zippers Base Setup

theory *ML-Gen-Zippers-Base*
imports
Zippy-Base-Setup
ML-IMap-Antiquotation

Gen-ML-Typeclasses-Base
begin

The ML code is parametrised by the number of zippers *nzippers*, the number of type parameters of the underlying typeclasses *'p1*,...,*'pn*, and the number of additional type parameters for the zipper *'a1*,...,*'am*. All parameters of the underlying typeclasses are also put into the zipper type per default since zippers for search trees must be able to store moves in the zipper themselves, i.e. a zipper takes type parameters *'p1*,...,*'pn*,*'a1*,...,*'am*.

Note: due to a performance problem in Poly/ML's type checker, instantiation functors need to be carefully used (i.e. avoid deep type instantiation chains): https://github.com/polyml/polyml/issues/213

⟨*ML*⟩

end

2.3 Generic Zippers Setup

theory *ML-Gen-Zippers-Setup*
imports
ML-Gen-Zippers-Base
ML-Lenses
begin

declare [[*ParaT-args sep*: , *encl*: , *encl-arg*: *stop*:]]
declare [[*ParaT-args sep*: , *encl*: , *encl-arg*: *stop*:]]
and [[*ZipperT-args sep*: , *encl*: *encl-arg*: *stop*:]]
and [[*AllT-args sep*: , *encl*: ( ) *encl-arg*: *stop*:]]
and [[*imap start*: *1*]]

⟨*ML*⟩

end

2.4 Morphisms

theory *ML-Morphs*
imports
ML-Gen-Zippers-Setup
begin

ML-gen-file‹*morph-base.ML*›
ML-gen-file‹*morph.ML*›
ML-gen-file‹*modify-morph.ML*›
ML-gen-file‹*pair-morph.ML*›

end

2.5 Structured Lenses

theory *ML-Structured-Lenses*
imports
ML-Gen-Zippers-Setup
begin

ML-gen-file‹*structured-lens.ML*›
ML-gen-file‹*sstructured-lens.ML*›
ML-gen-file‹*comp-structured-lens.ML*›
ML-gen-file‹*modify-structured-lens.ML*›
ML-gen-file‹*pair-structured-lens.ML*›

⟨*ML*⟩

Note: we reload the ML files, just with different parameters.

ML-gen-file‹*structured-lens.ML*›
ML-gen-file‹*sstructured-lens.ML*›

⟨*ML*⟩

end

2.6 Zippers

theory *ML-Zippers*
imports
ML-Morphs
ML-Structured-Lenses
begin

ML-gen-file‹*zipper-morphs.ML*›
ML-gen-file‹*modify-zipper-morphs-zipper.ML*›
ML-gen-file‹*modify-zipper-morphs-container.ML*›
ML-gen-file‹*pair-zipper-morphs.ML*›

ML-gen-file‹*zipper-data.ML*›
ML-gen-file‹*modify-zipper-data-zipper.ML*›
ML-gen-file‹*modify-zipper-data-content.ML*›
ML-gen-file‹*pair-zipper-data.ML*›

ML-gen-file‹*zipper.ML*›
ML-gen-file‹*modify-zipper-zipper.ML*›
ML-gen-file‹*modify-zipper-content.ML*›
ML-gen-file‹*extend-zipper-context.ML*›
ML-gen-file‹*sub-zipper.ML*›
ML-gen-file‹*pair-zipper.ML*›

end

2.7 Linked Zippers

theory *ML-Linked-Zippers*
  imports
    ML-Zippers
begin

⟨*ML*⟩

ML-gen-file‹*linked-zipper-morphs.ML*›
ML-gen-file‹*linked-zipper.ML*›

end

2.8 Alternating Zippers

theory *ML-Alternating-Zippers*
  imports
    ML-Linked-Zippers
begin

⟨*ML*⟩

ML-gen-file‹*alternating-zipper-morphs.ML*›
ML-gen-file‹*alternating-zipper.ML*›
ML-gen-file‹*sub-alternating-zipper.ML*›
ML-gen-file‹*pair-alternating-zipper.ML*›
ML-gen-file‹*rotate-alternating-zipper.ML*›

end

2.9 Zipper Utils

theory *ML-Zipper-Utils*
  imports
    ML-Zippers
    ML-Coroutines
begin

ML-gen-file‹*enumerate-zipper.ML*›
ML-gen-file‹*df-preorder-enumerate-zipper.ML*›
ML-gen-file‹*df-postorder-enumerate-zipper.ML*›

end

theory *ML-Alternating-Zipper-Utils*
  imports
    ML-Alternating-Zippers

ML-Zipper-Utils
begin

ML-gen-file‹*enumerate-alternating-zipper.ML*›
ML-gen-file‹*df-postorder-enumerate-alternating-zipper.ML*›

ML-gen-file‹*alternating-zipper-util.ML*›

end

2.10 Exceptions

theory *Zippy-Exceptions*
imports
ML-Categories
ML-Morphs
begin

⟨*ML*⟩

end

2.11 Loggers

theory *Zippy-Loggers*
imports
ML-Unification.ML-Logger
begin

⟨*ML*⟩

end

2.12 Pretty Printing/Shows

theory *Zippy-Shows*
imports
ML-Gen-Zippers-Setup
SpecCheck.SpecCheck-Show
begin

⟨*ML*⟩

end

2.13 States

theory *Zippy-States*
  imports
    ML-State
    ML-Morphs
begin

⟨*ML*⟩

end

2.14 Zippy Base

theory *Zippy-Base*
  imports
    ML-Alternating-Zipper-Utils
    Zippy-Exceptions
    Zippy-Loggers
    Zippy-Shows
    Zippy-States
begin

⟨*ML*⟩

end

2.15 Coroutines

theory *Zippy-Coroutines*
  imports
    ML-Coroutines
    Zippy-Exceptions
begin

⟨*ML*⟩

end

2.16 Enums

theory *Zippy-Enums*
  imports
    Zippy-Base
    Zippy-Coroutines
begin

⟨*ML*⟩

end

2.17 Identifiers

theory *Zippy-Identifiers*
imports
Pure
begin

Summary Identifiers for Zippy

⟨*ML*⟩

end

theory *Zippy-Actions*
imports
ML-Priority-Queues
Zippy-Actions-Base
Zippy-Enums
Zippy-Identifiers
begin

⟨*ML*⟩

end

theory *ML-Alternating-Zipper-Nodes*
imports
ML-Alternating-Zippers
begin

ML-gen-file‹*node.ML*›
ML-gen-file‹*modify-node-content.ML*›
ML-gen-file‹*modify-node-next.ML*›

⟨*ML*⟩

Note: we reload the ML file `node.ML`, just with different parameters.

ML-gen-file‹*node.ML*›
⟨*ML*⟩

context
notes [[*imap stop*: ‹*ML-Gen.nzippers* () + *1*›]]
begin
ML-gen-file‹*instantiate-node-succ.ML*›
end

ML-gen-file‹*alternating-zipper-nodes.ML*›
ML-gen-file‹*alternating-zipper-nodes-zippers.ML*›
ML-gen-file‹*alternating-zipper-nodes-simple-zippers.ML*›

end

theory *ML-Zipper-Directions*
imports
Zippy-Base-Setup
begin

⟨*ML*⟩

end

theory *ML-Zipper-Positions*
imports
ML-Categories
Zippy-Base-Setup
begin

⟨*ML*⟩

end

2.18 Monadic Lists

theory *ML-Lists*
imports
ML-Typeclasses-Base
begin

Summary Lists with generic failure monad.

ML-gen-file‹*glist.ML*›

end

theory *ML-Zipper-Instances*
imports
ML-Zippers
ML-Zipper-Directions
ML-Zipper-Positions
ML-Lists
begin

ML-gen-file‹*content-zipper.ML*›
ML-gen-file‹*direction-zipper.ML*›
ML-gen-file‹*position-zipper.ML*›

ML-gen-file‹*list-zipper.ML*›
ML-gen-file‹*rose-zipper.ML*›

end

theory *ML-Alternating-Zipper-Instances*
imports
ML-Alternating-Zipper-Nodes
ML-Zipper-Instances
begin

ML-gen-file‹*alternating-local-position-zipper.ML*›
ML-gen-file‹*alternating-global-position-zipper.ML*›
ML-gen-file‹*alternating-depth-zipper.ML*›

end

theory *ML-Zipper-Position-Utils*
imports
ML-Gen-Zippers-Setup
ML-Zipper-Positions
begin

ML-gen-file‹*zipper-position-util.ML*›

end

theory *ML-Alternating-Zipper-Paths*
imports
ML-Alternating-Zipper-Instances
ML-Zipper-Position-Utils
begin

ML-gen-file‹*alternating-zipper-path.ML*›
ML-gen-file‹*alternating-zipper-path-util.ML*›
ML-gen-file‹*alternating-zipper-path-local-position-zipper.ML*›

end

2.19 Nodes

theory *Zippy-Nodes*
imports
ML-Alternating-Zipper-Nodes
Zippy-Base
begin

⟨*ML*⟩

end

2.20 Positions

theory *Zippy-Positions*
  imports
    ML-Alternating-Zipper-Paths
    Zippy-Nodes
begin

⟨*ML*⟩

end

theory *Zippy-Actions-Positions*
  imports
    Zippy-Actions
    Zippy-Positions
begin

⟨*ML*⟩

end

2.21 Union-Find

theory *ML-Union-Find*
  imports *Pure*
begin

⟨*ML*⟩

end

2.22 Goals

theory *Zippy-Goals-Base*
  imports
    ML-Typeclasses-Base
    ML-Unification.Unify-Resolve-Tactics-Base
    ML-Union-Find
    ML-Unification.ML-Unifiers
begin

⟨*ML*⟩

end

theory *Zippy-Goals*
imports
ML-Alternating-Zipper-Paths
Zippy-Base
Zippy-Goals-Base
begin

⟨*ML*⟩

end

2.23 Lists

theory *Zippy-Lists-Base*
imports
ML-Zipper-Instances
Zippy-Enums
Zippy-Nodes
begin

⟨*ML*⟩

end

theory *Zippy-Lists-Goals*
imports
Zippy-Goals
Zippy-Lists-Base
begin

⟨*ML*⟩

end

2.24 Table Data

theory *Generic-Table-Data*
imports
ML-Unification.ML-Functor-Instances
ML-Unification.ML-Logger
SpecCheck.SpecCheck-Show
begin

⟨*ML*⟩

end

2.25 Action Clusters

theory *Zippy-Action-Clusters*
imports
Zippy-Enums
Zippy-Identifiers
begin

⟨*ML*⟩

end

2.26 Goal Position Updates

theory *Zippy-Goal-Pos-Updates-Base*
imports
ML-Categories
Zippy-Goals-Base
begin

⟨*ML*⟩

end

theory *Zippy-Goal-Pos-Updates*
imports
Generic-Table-Data
Zippy-Actions
Zippy-Action-Clusters
Zippy-Goal-Pos-Updates-Base
Zippy-Goals
begin

⟨*ML*⟩

end

theory *Zippy-Lists-Goal-Pos-Updates*
imports
Zippy-Lists-Goals
Zippy-Goal-Pos-Updates
begin

⟨*ML*⟩

end

2.27 Collect

theory *Zippy-Collect*
  imports
    ML-Zipper-Instances
    Zippy-Nodes
begin

⟨*ML*⟩

end

theory *Zippy-Lists-Collect*
  imports
    Zippy-Lists-Base
    Zippy-Collect
begin

⟨*ML*⟩

end

theory *Zippy-Lists-Positions*
  imports
    SpecCheck.SpecCheck-Show
    Zippy-Lists-Base
    Zippy-Positions
begin

⟨*ML*⟩

end

theory *Zippy-Lists-Positions-Collect*
  imports
    Zippy-Lists-Collect
    Zippy-Lists-Positions
begin

⟨*ML*⟩

end

2.28 Action Applications

theory *Zippy-Action-Applications-Base*
  imports
    SpecCheck.SpecCheck-Show
begin

⟨*ML*⟩

end

theory *Zippy-Action-Applications*
  imports
    Zippy-Action-Applications-Base
    Zippy-Enums
begin

⟨*ML*⟩

end

2.29 Zippy Tactics

theory *Zippy-Tactics-Base*
  imports
    Zippy-Goal-Pos-Updates-Base
begin

⟨*ML*⟩

end

theory *Zippy-Tactics*
  imports
    Zippy-Action-Applications
    Zippy-Tactics-Base
begin

⟨*ML*⟩

end

2.30 Instance of Zippy for Proof Search

theory *Zippy-Instance*
  imports
    ML-Unification.ML-Costs-Priorities
    Zippy-Actions-Positions
    Zippy-Lists-Goal-Pos-Updates
    Zippy-Lists-Positions-Collect
    Zippy-Tactics
begin

⟨*ML*⟩

end

2.30.1 Homogenously Changed Goals Data

theory *Zippy-Instance-Hom-Changed-Goals-Data*
imports
Generic-Term-Index-Data
Zippy-Instance
begin

⟨*ML*⟩

end

2.30.2 Cases

theory *Zippy-Instance-Cases*
imports
Zippy-Instance-Hom-Changed-Goals-Data
Cases-Tactics
begin

⟨*ML*⟩

end

2.30.3 Classical Reasoner

theory *Zippy-Instance-Classical*
imports
HOL.HOL
Zippy-Instance
begin

⟨*ML*⟩

end

2.30.4 Induction

theory *Induction-Tactics*
imports
Cases-Tactics
HOL.HOL
begin

⟨*ML*⟩

end

2.30.5 Induction

theory *Zippy-Instance-Induction*
imports
Zippy-Instance-Hom-Changed-Goals-Data
Induction-Tactics
begin

⟨*ML*⟩

end

Substitution

theory *Zippy-Instance-Subst*
imports
Zippy-Instance-Hom-Changed-Goals-Data
begin

⟨*ML*⟩

end

2.31 Customisable Context Parser

theory *Context-Parsers*
imports
Generic-Table-Data
Zippy-Identifiers
begin

⟨*ML*⟩

end

2.31.1 Simplifier

theory *Extended-Simp-Data*
imports
ML-Unification.ML-Functor-Instances
ML-Unification.ML-Logger
begin

⟨*ML*⟩

end

Resolution

theory *Zippy-Instance-Resolve*
imports
Zippy-Instance-Hom-Changed-Goals-Data
begin

⟨*ML*⟩

end

2.31.2 Prover with Resolution and Simplification

theory *Zippy-Instance-Resolves-Simp*
imports
Extended-Simp-Data
Zippy-Instance-Resolve
begin

Summary Basic ingredients for a prover supporting resolution and simplification based on Zippy.

⟨*ML*⟩

end

theory *Zippy-Runs-Base*
imports
Pure
begin

⟨*ML*⟩

end

2.32 Sequences

theory *Zippy-Seqs*
imports
ML-State
ML-Morphs
begin

⟨*ML*⟩

end

2.33 Runs

theory *Zippy-Runs*
imports
Zippy-Action-Applications
Zippy-Actions
Zippy-Lists-Goals
Zippy-Lists-Positions
Zippy-Runs-Base
Zippy-Seqs
begin

⟨*ML*⟩

end

theory *Zippy-Instance-Pure*
imports
Zippy-Instance
Zippy-Runs
begin

Summary Setup the standard instance of Zippy for proof search with short name "zippy".

⟨*ML*⟩

end

Simplification

theory *Zippy-Instance-Simp*
imports
Zippy-Instance
begin

⟨*ML*⟩

end

2.33.1 Zip - Extensible White-Box Prover

theory *Zip-Pure*
imports
Context-Parsers
Zippy-Instance-Resolves-Simp
Zippy-Instance-Pure
Zippy-Instance-Simp
begin

Summary Setup the the extensible white-box prover called "zip" based on Zippy.

⟨*ML*⟩
declare [[*zip-parse add*: ‹(@{*binding run*}, *Zip.Run.Data.parse-context-update*)›]]

Method

⟨*ML*⟩

Resolution

⟨*ML*⟩

```
declare [[zip-init-gc ‹
  let
    open Zippy Zip.Rule.Resolve; open ZLPC MU; open SC A Mo
    val id = @{binding resolve-ho-unif-first}
    val meta = Base-Data.ACMeta.metadata (id,
      Lazy.value resolution with higher−order unification on first possible goal)
    val tac = resolve-tac
    fun ztac mk-meta thm - = Ctxt.with-ctxt (fn ctxt => tac ctxt [thm]
      |> Tac-AAM.lift-tac mk-meta
      |> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
      |> arr)
    val retrieval = Data.TI.unifiables
    fun lookup-goal ctxt = snd #> snd #> Data.TI.norm-term
      #> retrieval (Data.get-index (Context.Proof ctxt))
      #> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
    fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
      let fun lookup-cons-goals goals =
        lookup-each-focused-data (lookup-goal ctxt) goals focus
        |> map-index (fn (i, (focus, data)) =>
          cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
      in
        Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
        >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
      end)
    fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›
  ‹let
    open Zippy Zip.Match.Resolve; open ZLPC MU; open SC A Mo
    val id = @{binding resolve-ho-match-first}
    val descr = Lazy.value resolution with higher−order matching on first possible
goal
    val meta = Base-Data.ACMeta.metadata (id,
      Lazy.value resolution with higher−order matching on first possible goal)
    val tac = match-tac
    fun ztac mk-meta thm - = Ctxt.with-ctxt (fn ctxt => tac ctxt [thm]
```

```
      |> Tac-AAM.lift-tac mk-meta
      |> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
      |> arr)
    val retrieval = Data.TI.generalisations
    fun lookup-goal ctxt = snd #> snd #> Data.TI.norm-term
      #> retrieval (Data.get-index (Context.Proof ctxt))
      #> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
    fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
       let fun lookup-cons-goals goals = lookup-each-focused-data (lookup-goal ctxt)
goals focus
        |> map-index (fn (i, (focus, data)) =>
          cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
      in
        Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
        >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
      end)
    fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›
  ‹let
    open Zippy Zip.URule.Resolve; open ZLPC MU; open SC A Mo
    val id = @{binding resolve-proof-unif-first}
     val descr = Lazy.value resolution with proof−producing unification on first
possible goal
    val meta = Base-Data.ACMeta.metadata (id,
      Lazy.value resolution with proof−producing unification on first possible goal)
    val tac = Unify-Resolve-Base.unify-resolve-tac
    fun ztac normalisers unifier mk-meta thm - = Ctxt.with-ctxt (tac normalisers
unifier thm
      #> Tac-AAM.lift-tac mk-meta
      #> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
      #> arr)
    (∗Note: there is no complete, efficient retrieval other than taking all rules. In
case of a
     large rule set, one can use an incomplete retrieval returning only those rules
whose
    left−hand or right−hand side potentially higher−order unifies with a disagree-
ment term.
    Cf. the retrieval used in ML-Unification.ML-Unification-Hints∗)
    val retrieval = Data.TI.content
    fun lookup-goal ctxt - = retrieval (Data.get-index (Context.Proof ctxt))
      |> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
    fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
       let fun lookup-cons-goals goals = lookup-each-focused-data (lookup-goal ctxt)
goals focus
        |> map-index (fn (i, (focus, data)) =>
          cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
      in
        Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
```

```
      >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
    end)
  fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
    >>= AC.opt (K z) Up3.morph
in (id, init) end›]]

declare [[zip-init-gc
  ‹let
    open Zippy Zip.Rule.EResolve; open ZLPC MU; open SC A Mo
    val id = @{binding eresolve-ho-unif-first}
    val meta = Base-Data.ACMeta.metadata (id,
      Lazy.value e−resolution with higher−order unification on first possible goal)
    val tac = eresolve-tac
    fun ztac mk-meta thm - = Ctxt.with-ctxt (fn ctxt => tac ctxt [thm]
      |> Tac-AAM.lift-tac mk-meta
      |> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
      |> arr)
    val retrieval = Data.TI.unifiables
    fun lookup-goal ctxt = snd #> fst #>
     maps (Data.TI.norm-term #> retrieval (Data.get-index (Context.Proof ctxt)))
      #> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
    fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
       let fun lookup-cons-goals goals = lookup-each-focused-data (lookup-goal ctxt)
goals focus
        |> map-index (fn (i, (focus, data)) =>
          cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
      in
        Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
        >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
      end)
    fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›
  ‹let
    open Zippy Zip.Match.EResolve; open ZLPC MU; open SC A Mo
    val id = @{binding eresolve-ho-match-first}
    val meta = Base-Data.ACMeta.metadata (id,
      Lazy.value e−resolution with higher−order matching on first possible goal)
    val tac = ematch-tac
    fun ztac mk-meta thm - = Ctxt.with-ctxt (fn ctxt => tac ctxt [thm]
      |> Tac-AAM.lift-tac mk-meta
      |> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
      |> arr)
    val retrieval = Data.TI.generalisations
    fun lookup-goal ctxt = snd #> fst
      #> maps (Data.TI.norm-term #> retrieval (Data.get-index (Context.Proof
ctxt)))
      #> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
    fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
```

```
      let fun lookup-cons-goals goals = lookup-each-focused-data (lookup-goal ctxt)
goals focus
        |> map-index (fn (i, (focus, data)) =>
          cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
      in
        Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
        >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
      end)
    fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›
  ‹let
    open Zippy Zip.URule.EResolve; open ZLPC MU; open SC A Mo
    val id = @{binding eresolve-proof-unif-first}
    val meta = Base-Data.ACMeta.metadata (id,
     Lazy.value e−resolution with proof−producing unification on first possible goal)
    fun tac norms unify = Unify-Resolve-Base.unify-eresolve-tac norms unify norms
unify
     fun ztac normalisers unifier mk-meta thm - = Ctxt.with-ctxt (tac normalisers
unifier thm
      #> Tac-AAM.lift-tac mk-meta
      #> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
      #> arr)
    val retrieval = Data.TI.content
    fun lookup-goal ctxt - = retrieval (Data.get-index (Context.Proof ctxt))
      |> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
    fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
      let fun lookup-cons-goals goals = lookup-each-focused-data (lookup-goal ctxt)
goals focus
        |> map-index (fn (i, (focus, data)) =>
          cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
      in
        Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
        >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
      end)
    fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›]]
declare [[zip-init-gc
  ‹let
    open Zippy Zip.Rule.DResolve; open ZLPC MU; open SC A Mo
    val id = @{binding dresolve-ho-unif-first}
    val meta = Base-Data.ACMeta.metadata (id,
      Lazy.value d−resolution with higher−order unification on first possible goal)
    fun tac ctxt thms =
      let
        (∗Tactic.make-elim allows no context passing but Thm.biresolution fails to
certificate certain
       theorems without a context∗)
```

```
      fun make-elim ctxt thm =
          let val resolve = Thm.biresolution (SOME ctxt) false [(false, thm)] |>
HEADGOAL #> Seq.hd
        in zero-var-indexes (resolve revcut-rl) end
    in eresolve-tac ctxt (List.map (make-elim ctxt) thms) end
  fun ztac mk-meta thm - = Ctxt.with-ctxt (fn ctxt => tac ctxt [thm]
    |> Tac-AAM.lift-tac mk-meta
    |> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
    |> arr)
  val retrieval = Data.TI.unifiables
  fun lookup-goal ctxt = snd #> fst #>
   maps (Data.TI.norm-term #> retrieval (Data.get-index (Context.Proof ctxt)))
    #> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
  fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
    let fun lookup-cons-goals goals = lookup-each-focused-data (lookup-goal ctxt)
goals focus
      |> map-index (fn (i, (focus, data)) =>
        cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
    in
      Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
      >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
    end)
  fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
    >>= AC.opt (K z) Up3.morph
in (id, init) end›
‹let
  open Zippy Zip.Match.DResolve; open ZLPC MU; open SC A Mo
  val id = @{binding dresolve-ho-match-first}
  val meta = Base-Data.ACMeta.metadata (id,
    Lazy.value d−resolution with higher−order matching on first possible goal)
  fun tac ctxt thms =
    let
      fun make-elim ctxt thm =
          let val resolve = Thm.biresolution (SOME ctxt) false [(false, thm)] |>
HEADGOAL #> Seq.hd
        in zero-var-indexes (resolve revcut-rl) end
    in ematch-tac ctxt (List.map (make-elim ctxt) thms) end
  fun ztac mk-meta thm - = Ctxt.with-ctxt (fn ctxt => tac ctxt [thm]
    |> Tac-AAM.lift-tac mk-meta
    |> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
    |> arr)
  val retrieval = Data.TI.generalisations
  fun lookup-goal ctxt = snd #> fst #>
   maps (Data.TI.norm-term #> retrieval (Data.get-index (Context.Proof ctxt)))
    #> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
  fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
    let fun lookup-cons-goals goals = lookup-each-focused-data (lookup-goal ctxt)
goals focus
      |> map-index (fn (i, (focus, data)) =>
```

```
        cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
    in
      Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
      >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
    end)
  fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
    >>= AC.opt (K z) Up3.morph
in (id, init) end›
‹let
  open Zippy Zip.URule.DResolve; open ZLPC MU; open SC A Mo
  val id = @{binding dresolve-proof-unif-first}
  val meta = Base-Data.ACMeta.metadata (id,
   Lazy.value d−resolution with proof−producing unification on first possible goal)
  val tac = Unify-Resolve-Base.unify-dresolve-tac
  fun ztac normalisers unifier mk-meta thm - = Ctxt.with-ctxt (tac normalisers
unifier thm
    #> Tac-AAM.lift-tac mk-meta
    #> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
    #> arr)
  val retrieval = Data.TI.content
  fun lookup-goal ctxt - = retrieval (Data.get-index (Context.Proof ctxt))
    |> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
  fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
    let fun lookup-cons-goals goals = lookup-each-focused-data (lookup-goal ctxt)
goals focus
      |> map-index (fn (i, (focus, data)) =>
        cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
    in
      Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
      >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
    end)
  fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
    >>= AC.opt (K z) Up3.morph
in (id, init) end›]]
declare [[zip-init-gc
‹let
  open Zippy Zip.Rule.FResolve; open ZLPC MU; open SC A Mo
  val id = @{binding fresolve-ho-unif-first}
  val meta = Base-Data.ACMeta.metadata (id,
    Lazy.value f−resolution with higher−order unification on first possible goal)
  val tac = Unify-Resolve-Base.unify-fresolve-tac
    Higher-Order-Unification.norms Higher-Order-Unification.unify
  fun ztac mk-meta thm - = Ctxt.with-ctxt (fn ctxt => tac thm ctxt
    |> Tac-AAM.lift-tac mk-meta
    |> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
    |> arr)
  val retrieval = Data.TI.unifiables
  fun lookup-goal ctxt = snd #> fst
    #> maps (Data.TI.norm-term #> retrieval (Data.get-index (Context.Proof
```

```
ctxt)))
      #> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
    fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
      let fun lookup-cons-goals goals = lookup-each-focused-data (lookup-goal ctxt)
goals focus
        |> map-index (fn (i, (focus, data)) =>
          cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
      in
        Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
        >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
      end)
    fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›
  ‹let
    open Zippy Zip.Match.FResolve; open ZLPC MU; open SC A Mo
    val id = @{binding fresolve-ho-match-first}
    val meta = Base-Data.ACMeta.metadata (id,
      Lazy.value f−resolution with higher−order matching on first possible goal)
    (∗FIXME: use same matcher as in other match tactics∗)
    val tac = Unify-Resolve-Base.unify-fresolve-tac
      Mixed-Unification.norms-first-higherp-match
      (Mixed-Unification.first-higherp-e-match Unification-Combinator.fail-match)
    fun ztac mk-meta thm - = Ctxt.with-ctxt (fn ctxt => tac thm ctxt
      |> Tac-AAM.lift-tac mk-meta
      |> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
      |> arr)
    val retrieval = Data.TI.generalisations
    fun lookup-goal ctxt = snd #> fst
      #> maps (Data.TI.norm-term #> retrieval (Data.get-index (Context.Proof
ctxt)))
      #> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
     fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
      let fun lookup-cons-goals goals = lookup-each-focused-data (lookup-goal ctxt)
goals focus
        |> map-index (fn (i, (focus, data)) =>
          cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
      in
        Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
        >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
      end)
    fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›
  ‹let
    open Zippy Zip.URule.FResolve; open ZLPC MU; open SC A Mo
    val id = @{binding fresolve-proof-unif-first}
    val meta = Base-Data.ACMeta.metadata (id,
     Lazy.value f−resolution with proof−producing unification on first possible goal)
```

```
    val tac = Unify-Resolve-Base.unify-fresolve-tac
    fun ztac normalisers unifier mk-meta thm - = Ctxt.with-ctxt (tac normalisers
unifier thm
      #> Tac-AAM.lift-tac mk-meta
      #> Tac-AAM.Tac.zFIRST-GOAL-FOCUS
      #> arr)
    val retrieval = Data.TI.content
    fun lookup-goal ctxt - = retrieval (Data.get-index (Context.Proof ctxt))
      |> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
    fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => fn z =>
       let fun lookup-cons-goals goals = lookup-each-focused-data (lookup-goal ctxt)
goals focus
        |> map-index (fn (i, (focus, data)) =>
          cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
      in
        Up3.morph z >>= arr Mixin2.GCluster.get-stripped-goals
        >>= (fn goals => ZB.update-zipper3 (lookup-cons-goals goals) z)
      end)
    fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›]]

declare [[zip-parse ‹(@{binding rule}, Zip.Rule.parse-method)›]]
declare [[zip-parse ‹(@{binding match}, Zip.Match.parse-method)›]]
declare [[zip-parse ‹(@{binding urule}, Zip.URule.parse-method)›]]
```

Simplifier

```
declare [[zip-init-gc ‹
  let
    open Zippy; open ZLPC MU; open A Mo
    val name = asm-full-simp
    val id = Zippy-Identifier.make (SOME @{here}) name
    val tacs = (safe-asm-full-simp-tac, asm-full-simp-tac)
    fun f-timeout ctxt i state n time = (@{log Logger.WARN Zip.Simp.logger} ctxt
      (fn - => Pretty.breaks [
       Pretty.block [Pretty.str (name ^ timeout at pull number ), SpecCheck-Show.int
n,
           Pretty.str  after , Pretty.str (Time.print time), Pretty.str  seconds.],
            Pretty.block [Pretty.str Called on subgoal , SpecCheck-Show.term ctxt
(Thm.prem-of state i)],
         Pretty.str (implode [Consider removing , name,
             for this proof, increase/disable the timeout, or check for looping simp
rules.])
       ] |> Pretty.block0 |> Pretty.string-of);
      NONE)
    (*FIXME: why is the simplifier raising Option.Option and ERROR exceptions
in some cases?*)
    fun handle-exn ctxt exn = (@{log Logger.WARN Zip.Simp.logger} ctxt
```

```
      (fn - => Simplifier raised unexpected ^ exn ^ exception. Returning NONE
instead.);
      NONE)
    fun handle-exns-sq ctxt sq = Seq.make (fn - =>
      sq |> Seq.pull |> Option.map (apsnd (handle-exns-sq ctxt))
       handle Option.Option => handle-exn ctxt Option.Option | ERROR - =>
handle-exn ctxt ERROR)
    fun wrap-tac tac ctxt i state = Zip.Simp.Extended-Data.wrap-simp-tac
      (f-timeout ctxt i state) (fn ctxt => handle-exns-sq ctxt oo tac ctxt) ctxt i state
   val (safe-tac, tac) = apply2 wrap-tac (safe-asm-full-simp-tac, asm-full-simp-tac)
    val update = Library.maps snd
    #> LGoals-Pos-Copy.partition-update-gcposs-gclusters (Zip.Run.init-gposs
true)
    val mk-cud = Result-Action.copy-update-data-empty-changed
    open Base-Data
   val costs-progress = ((Cost.LOW, AAMeta.P.promising), (Cost.LOW3, AAMeta.P.promising))
    val madd-safe = fst
    fun mk-meta (cost, progress) = A.K (Library.K (Library.K (AAMeta.metadata
      {cost = cost, progress = progress})))
    val (mk-meta-safe, mk-meta-unsafe) = apply2 mk-meta costs-progress
    val (presultsq-safe, presultsq-unsafe) =
      apply2 (fst #> Zip.PResults.enum-scale-presultsq-default) costs-progress
    val data = Simp.gen-data Util.exn id name safe-tac tac update mk-cud
      madd-safe mk-meta-safe mk-meta-unsafe presultsq-safe presultsq-unsafe
    fun init - focus z =
        Tac.cons-action-cluster Util.exn (Base-Data.ACMeta.no-descr id) [(focus,
data)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›]]

declare [[zip-parse add: ‹(@{binding simp}, Zip.Simp.parse-extended [])›
  and default: ‹@{binding simp}›]]

end

theory Zip-HOL
  imports
    Cases-Tactics-HOL
    Extended-Blast-Data
    ML-Unification.ML-Unification-HOL-Setup
    Zippy-Instance-Cases
    Zippy-Instance-Classical
    Zippy-Instance-Induction
    Zippy-Instance-Subst
    Zip-Pure
begin
```

Simplifier

declare [[*zip-parse del*: ‹@{*binding simp*}›
and *add*: ‹(@{*binding simp*}, *Zip.Simp.parse-extended Splitter.split-modifiers*)›]]

Classical Reasoner

⟨*ML*⟩

declare [[*zip-init-gc* ‹
let
open Zippy; *open ZLPC MU*; *open A Mo*
val id = @{*binding classical-slow-step*}
val update = *Library.maps snd*
#> *LGoals-Pos-Copy.partition-update-gcposs-gclusters-gclusters* (*Zip.Run.init-gposs true*)
val mk-cud = *Result-Action.copy-update-data-empty-changed*
open Base-Data
val costs-progress = ((*Cost.VERY-LOW*, *AAMeta.P.promising*), (*Cost.LOW3*, *AAMeta.P.promising*),
(*Cost.MEDIUM*, *AAMeta.P.unclear*), (*Cost.MEDIUM*, *AAMeta.P.unclear*))
val madd-safe = *fst*
fun mk-meta (*cost*, *progress*) = *A.K* (*Library.K* (*Library.K* (*AAMeta.metadata* {*cost* = *cost*, *progress* = *progress*})))
val (*mk-meta-safe*, *mk-meta-inst0*, *mk-meta-instp*, *mk-meta-unsafe*) =
@{*apply 4*} *mk-meta costs-progress*
val (*presultsq-safe*, *presultsq-inst0*, *presultsq-instp*, *presultsq-unsafe*) =
@{*apply 4*} (*fst* #> *Zip.PResults.enum-scale-presultsq-default*) *costs-progress*
val data = *Classical.slow-step-data Util.exn id update mk-cud madd-safe mk-meta-safe mk-meta-inst0 mk-meta-instp mk-meta-unsafe presultsq-safe presultsq-inst0 presultsq-instp*
presultsq-unsafe
fun init - focus z =
Tac.cons-action-cluster Util.exn (*Base-Data.ACMeta.no-descr id*) [(*focus*, *data*)] *z*
>>= *AC.opt* (*K z*) *Up3.morph*
in (*id*, *init*) *end*›]]
declare [[*zip-init-gc* ‹
let
open Zippy; *open ZLPC MU*; *open A Mo*
val id = @{*binding atomize-prems*}
val update = *Library.maps snd*
#> *LGoals-Pos-Copy.partition-update-gcposs-gclusters-gclusters* (*Zip.Run.init-gposs true*)
val mk-cud = *Result-Action.copy-update-data-empty-changed*
open Base-Data
val (*cost*, *progress*) = (*Cost.LOW1*, *AAMeta.P.promising*)
val madd = *fst*
val mk-meta = *A.K* (*Library.K* (*Library.K* (*AAMeta.metadata* {*cost* = *cost*, *progress* = *progress*})))

```
    val presultsq-atomize-prems = Zip.PResults.enum-scale-presultsq-default cost
    val data = Classical.atomize-prems-data id update mk-cud madd mk-meta
      presultsq-atomize-prems
    fun init - focus z =
        Tac.cons-action-cluster Util.exn (Base-Data.ACMeta.no-descr id) [(focus,
data)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›]]
declare [[zip-parse add: ‹(@{binding clasimp}, Clasimp.clasimp-modifiers |> Method.sections)›
  ‹(@{binding cla}, Classical.cla-modifiers |> Method.sections)›
  and default: ‹@{binding clasimp}›]]

⟨ML⟩
declare [[zip-init-gc ‹
  let
    open Zippy Zip; open ZLPC MU; open A Mo Base-Data
    val id = @{binding blast}
   val (cost, progress, prio) = (Cost.VERY-LOW, AAMeta.P.promising, Cost.HIGH)
    val madd = fst
   val mk-meta = Library.K (Library.K (AAMeta.metadata {cost = cost, progress
= progress}))
    val tac = Blast.blast-tac
   fun ztac - = Ctxt.with-ctxt (fn ctxt => arr (Tac-AAM.Tac.zTRY-EVERY-FOCUS1
madd
      (Tac-AAM.lift-tac mk-meta (tac ctxt))))
    val presultsq = Zip.PResults.enum-scale-presultsq-default prio
    val data = {
      empty-action = Library.K Zippy.PAction.disable-action,
      meta = AMeta.metadata (id, Lazy.value blast with depth and timeout limit),
     result-action = Result-Action.action (Library.K (C.id ())) Result-Action.copy-update-data,
      presultsq = presultsq,
      tac = ztac}
     fun init - focus z = Tac.cons-action-cluster Util.exn (ACMeta.no-descr id)
[(focus, data)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›]]
declare [[zip-parse ‹(@{binding blast}, Scan.depend (fn context =>
  Zip.Blast.parse-attribute
  >> (fn attr => (ML-Attribute-Util.attribute-map-context attr context, ())))›]]
```

Substitution

⟨ML⟩

```
declare [[zip-init-gc ‹
  let
    open Zippy Zip.Subst.Concl; open ZLPC MU; open SC A Mo
    val id = @{binding subst-concl-some}
    val meta = Base-Data.ACMeta.metadata (id,
```

```
    Lazy.value substitution in conclusion on some goal)
  fun tac ctxt thms = SELECT-GOAL
    (let fun apply-occ-tac occ st = Seq.of-list thms |> Seq.maps (fn r =>
      EqSubst.eqsubst-tac' ctxt
        (EqSubst.skip-first-occs-search occ EqSubst.searchf-lr-unify-valid) r
        (Thm.nprems-of st) st)
    in Seq.EVERY (List.map apply-occ-tac [0]) end)
  fun ztac mk-meta thm - = Ctxt.with-ctxt (fn ctxt => tac ctxt [thm]
    |> Tac-AAM.lift-tac mk-meta
    |> Tac-AAM.Tac.zSOME-GOAL-FOCUS
    |> arr)
  val retrieval = Data.TI.content #> Library.K
  fun lookup-goal ctxt = retrieval (Data.get-index (Context.Proof ctxt))
    #> List.map (apsnd (transfer-data (Proof-Context.theory-of ctxt)))
  fun cons-actions focus = Ctxt.with-ctxt (fn ctxt =>
    Data.TI.content (Data.get-index (Context.Proof ctxt))
    |> List.map (snd #> transfer-data (Proof-Context.theory-of ctxt))
    |> map-index (fn (i, data) =>
      cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
    |> ZB.update-zipper3)
  fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
    >>= AC.opt (K z) Up3.morph
in (id, init) end›
‹let
  open Zippy Zip.Subst.Asm; open ZLPC MU; open SC A Mo
  val id = @{binding subst-asm-some}
  val meta = Base-Data.ACMeta.metadata (id,
    Lazy.value substitution in assumptions on some goal)
  fun tac ctxt thms = SELECT-GOAL
    (let fun apply-occ-tac occ st = Seq.of-list thms |> Seq.maps (fn r =>
      EqSubst.eqsubst-asm-tac' ctxt
        (EqSubst.skip-first-asm-occs-search EqSubst.searchf-lr-unify-valid) occ r
        (Thm.nprems-of st) st)
    in Seq.EVERY (List.map apply-occ-tac [0]) end)
  fun ztac mk-meta thm - = Ctxt.with-ctxt (fn ctxt => tac ctxt [thm]
    |> Tac-AAM.lift-tac mk-meta
    |> Tac-AAM.Tac.zSOME-GOAL-FOCUS
    |> arr)
  fun cons-actions focus = Ctxt.with-ctxt (fn ctxt =>
    Data.TI.content (Data.get-index (Context.Proof ctxt))
    |> List.map (snd #> transfer-data (Proof-Context.theory-of ctxt))
    |> map-index (fn (i, data) =>
      cons-nth-action Util.exn meta ztac ctxt i data focus >>> Up4.morph)
    |> ZB.update-zipper3)
  fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
    >>= AC.opt (K z) Up3.morph
in (id, init) end›]]

declare [[zip-parse ‹(@{binding subst}, Zip.Subst.parse-method)›]]
```

Cases and Induction

⟨*ML*⟩

```
declare [[zip-init-gc ‹
  let open Zippy Zip.Cases; open ZLPC MU; open SC A Mo
    val id = @{binding cases-some}
    val meta = Base-Data.ACMeta.metadata (id, Lazy.value cases on some goal)
    val tac = Cases-Data-Args-Tactic-HOL.cases-tac (fn simp => fn opt-rule =>
fn insts =>
      fn facts => fn ctxt => Induct.cases-tac ctxt simp [insts] opt-rule facts)
    fun ztac mk-meta data - = Ctxt.with-ctxt (fn ctxt => tac data ctxt
      |> Tac-AAM.lift-tac mk-meta
      |> Tac-AAM.Tac.zSOME-GOAL-FOCUS
      |> arr)
    val opt-default-update-action = NONE
    fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => Data.get (Context.Proof
ctxt)
      |> List.map (transfer-data (Proof-Context.theory-of ctxt))
      |> map-index (fn (i, data) =>
       cons-nth-action Util.exn meta ztac opt-default-update-action ctxt i data focus
>>> Up4.morph)
      |> ZB.update-zipper3)
    fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›]]
declare [[zip-init-gc ‹let open Zippy Zip.Induction; open ZLPC MU; open SC A
Mo
    val id = @{binding induct-some}
    val meta = Base-Data.ACMeta.metadata (id, Lazy.value induction on some
goal)
    val tac = Induction-Data-Args-Tactic-HOL.induct-tac false
    fun ztac mk-meta data - = Ctxt.with-ctxt (fn ctxt => tac data ctxt
      |> Tac-AAM.lift-tac mk-meta
      |> Tac-AAM.Tac.zSOME-GOAL-FOCUS
      |> arr)
    val opt-default-update-action = NONE
    fun cons-actions focus = Ctxt.with-ctxt (fn ctxt => Data.get (Context.Proof
ctxt)
      |> List.map (transfer-data (Proof-Context.theory-of ctxt))
      |> map-index (fn (i, data) =>
       cons-nth-action Util.exn meta ztac opt-default-update-action ctxt i data focus
>>> Up4.morph)
      |> ZB.update-zipper3)
    fun init - focus z = Node.cons3 Util.exn meta [(focus, cons-actions)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›]]
declare [[zip-parse ‹(@{binding cases}, Zip.Cases.parse-context-update)›]]
declare [[zip-parse ‹(@{binding induct}, Zip.Induction.parse-context-update)›]]
```

end

Metis

theory *Zip-Metis*
imports
Extended-Metis-Data
Zip-HOL
begin

⟨*ML*⟩

declare [[*zip-init-gc* ‹
let
open Zippy Zip; *open ZLPC MU*; *open A Mo Base-Data*
val id = @{*binding metis*}
val guard-name = *FI.id*
val descr = *Lazy.lazy* (*fn* - => *implode-space* [
metis in parallel if no promising progress by, *guard-name*, *is expected*])
val logger = *Zip.Metis.logger*
val (*cost*, *progress*, *prio*) = (*Cost.VERY-LOW*, *AAMeta.P.promising*, *Cost.VERY-LOW*)
val madd = *fst*
val mk-meta = *Library.K* (*Library.K* (*AAMeta.metadata* {*cost* = *cost*, *progress*
= *progress*}))
fun tac args ctxt i state =
let
(∗*Calling metis unconditionally often leads to loops. Before calling metis*, *we*
hence first
gauge if zip would likely be able to make some promising progress without it.
If it
can, *we skip the metis call.*∗)
val zip-progress-tac =
let
val steps = *20* (∗*should be sufficient to get an estimation*∗)
val - = @{*log Logger.TRACE*} *ctxt* (*fn* - => *implode-space* [*Checking if*,
guard-name, *can make any promising progress in*, *string-of-int steps*,
steps.])
in
Context.proof-map (
(∗*remove metis from test steps to avoid loops*∗)
Run.Init-Goal-Cluster.Data.map-table (*Run.Init-Goal-Cluster.Data.Table.delete-safe*
id)
#> *Run.Data.map-exec* (*Library.K Zippy.Run.AStar.promising′*))
#> *Run.tac* (*SOME steps*) #> *CHANGED* #> *SELECT-GOAL*
end
fun f-timeout n time = (@{*log Logger.WARN*} *ctxt* (*fn* - => *Pretty.breaks* [
Pretty.block [*Pretty.str metis timeout at pull number* , *SpecCheck-Show.int*
n,
Pretty.str after , *Pretty.str* (*Time.print time*), *Pretty.str seconds.*],

```
            Pretty.block [Pretty.str Called on subgoal , SpecCheck-Show.term ctxt
(Thm.prem-of state i)],
          Pretty.str Consider removing metis for this proof or increase/disable the
timeout.
        ] |> Pretty.block0 |> Pretty.string-of);
        NONE)
    in
      (if can (zip-progress-tac ctxt i #> Seq.hd) state
      then (@{log Logger.TRACE} ctxt (fn - => Pretty.breaks [
          Pretty.str (guard-name ^ made promising progress.),
          Pretty.str Skipping metis.
        ] |> Pretty.block |> Pretty.string-of);
        no-tac)
      else (@{log Logger.TRACE} ctxt (fn - => Pretty.breaks [
          Pretty.str (guard-name ^ did not make promising progress.),
          Pretty.str Calling metis.
        ] |> Pretty.block |> Pretty.string-of);
        Extended-Metis-Data-Args.metis-tac f-timeout args ctxt i)) state
    end
  fun ztac args - = Ctxt.with-ctxt (tac args
    #> Tac-AAM.lift-tac mk-meta
    #> Tac-AAM.Tac.zTRY-EVERY-FOCUS1 madd
    #> arr)
  val presultsq = Zip.PResults.enum-scale-presultsq-default prio
  fun data args = {
    empty-action = Library.K Zippy.PAction.disable-action,
    meta = AMeta.metadata (id, descr),
   result-action = Result-Action.action (Library.K (C.id ())) Result-Action.copy-update-data,
    presultsq = presultsq,
    tac = ztac args}
  fun init - focus z = Ctxt.get-ctxt () >>= (fn ctxt =>
    let
      val args = Zip.Metis.get-args (Context.Proof ctxt)
     val focus-data = if null (Extended-Metis-Data-Args.PA.get-runs args) then []
        else [(focus, data args)]
    in
     Tac.cons-action-cluster Util.exn (ACMeta.metadata (id, descr)) focus-data z
      >>= AC.opt (K z) Up3.morph
    end)
 in (id, init) end›]]
declare [[zip-parse ‹(@{binding metis}, Zip.Metis.parse-attribute
 :|-- (fn attr => Scan.depend (ML-Attribute-Util.attribute-map-context attr
   #> rpair () #> Scan.succeed)))›]]

end
```

Try0

theory *Zip-Try0*

imports
HOL.Sledgehammer
Zip-Metis
begin

Try0 Integration

⟨*ML*⟩

end

2.34 Examples and Brief Technical Overview for Zip

theory *Zip-Examples*
imports
Zip-Try0
HOL.List
begin

Summary The *zip* method is a customisable general-purpose white-box prover based on the Zippy framework. This theory begins with examples demonstrating some key features and concludes with a brief technical overview for users interested in customising the method.

On a high-level, *zip* performs a proof tree search with customisable expansion actions and search strategies. By default, it uses an A^* search and integrates the classical reasoner, simplifier, the blast and metis prover, and supports resolution with higher-order and proof-producing unification [1], conditional substitutions, case splitting and induction, among other things.

In most cases, *zip* can be used as a drop-in replacement for Isabelle's classical methods like *auto*, *fastforce*, *force*, *fast*, etc., as demonstrated in `Benchmarks`. Note, however, that *zip* can be slower than those methods due to a more general search procedure.

Like *auto*, *zip* supports non-terminal calls and interactive proof exploration.

zip comes with **try0** integration in `../Zip_Try0.thy`. Import that theory as a first file to obtain the integration. If you want to omit the integration, import `../Zip_Metis.thy` instead.

2.34.1 Examples

Note: some examples in this files are adapted from *HOL.List*. Some original proofs from *HOL.List* are left in comments and tagged with "ORIG" for comparison.

experiment

begin

You can use it like *auto*:

lemma *sorted-wrt* (>) $l \longleftrightarrow$ *sorted* (*rev* l) $\wedge$ *distinct* l
⟨*proof*⟩

lemma *sorted-wrt* (>) $l \longleftrightarrow$ *sorted* (*rev* l) $\wedge$ *distinct* l
⟨*proof*⟩

You can use it for proof exploration (i.e. the method returns incomplete attempts):

lemma
assumes [*intro*]: $P \Longrightarrow Q$
and [*simp*]: $A = B$
shows $A \longrightarrow Q$
⟨*proof*⟩

Note that the method returned the goal $B \Longrightarrow Q$ but not $B \Longrightarrow P$. The reason is that, by default, the method only returns those incomplete attempts that only use "promising" expansions on its search path, as further elaborated in the technical overview. The simplifier, for instance, is marked as a "promising" expansion action. For the classical reasoner, expansions with unsafe (introduction) rules are not marked as promising while safe rules are.

One can instruct the method to return all attempts by changing its default strategy from `Zip.AStar.promising'` to `Zip.AStar.all'`:

lemma
assumes [*intro*]: $P \Longrightarrow Q$
and [*simp*]: $A = B$
shows $A \longrightarrow Q$
⟨*proof*⟩

Alternatively, the introduction rule can be marked as safe:

lemma
assumes [*intro*!]: $P \Longrightarrow Q$
and [*simp*]: $A = B$
shows $A \longrightarrow Q$
⟨*proof*⟩

Many explorations are possibly infinite or too large for an exhaustive search. In such cases, one may limit the number of expansion steps. Below, we fuel the method with 20 steps:

lemma
assumes [*intro*]: $P \wedge P \Longrightarrow P$
shows P
⟨*proof*⟩

One can also limit the maximum search depth, e.g. to depth 2:

lemma
assumes [*intro*]: $P \wedge P \Longrightarrow P$
shows P
⟨*proof*⟩

You can perform case splits:

lemma *tl xs* $\neq$ [] $\longleftrightarrow$ *xs* $\neq$ [] $\wedge$ $\neg(\exists\, x.\ xs = [x])$
⟨*proof*⟩

fun *foo* :: *nat* ⇒ *nat* **where**
foo 0 = 0
| *foo* (*Suc 0*) = *1*
| *foo* (*Suc* (*Suc n*)) = *1*

lemma *foo n + foo m < 4*
⟨*proof*⟩

You may also use patterns of the shape (pattern - anti-patterns). All terms occurring in the goal that (1) satisfy the pattern and (2) do not satisfy any of the anti-patterns are then taken as instantiation candidates:

lemma *foo n + foo m < 4*
— matches natural numbers, but no applications (e.g. *foo n*)
⟨*proof*⟩

Note that for a function f with multiple arguments, the function package creates a cases rule $f.cases$ where f's arguments are tupled and equated to a single variable. Example:

fun *bar* :: *nat* ⇒ *bool* ⇒ *nat* **where**
bar 0 - = 0
| *bar* (*Suc 0*) *True* = *1*
| *bar* (*Suc 0*) *False* = *0*
| *bar* (*Suc* (*Suc n*)) *b* = *1*

thm *bar.cases*

As a result *cases n* :: *nat b* :: *bool rule*: *bar.cases* will raise an error: The cases rule requires a *nat* × *bool* pair for instantiation. The right invocation is *cases* (*n*, *b*) *rule*: *bar.cases*.

Moreover, the invocation *cases* (*pat*) (*?n* :: *nat*, *?b* :: *nat*) *rule*: *bar.cases* will not find any matches in a goal term *bar n b < 2* since no *nat* × *bool* pair occurs in the goal. The solution is to transform the cases rule to the desired form with *deprod-cases*:

thm *bar.cases*[*deprod-cases*]

lemma *bar n b < 2*
⟨*proof*⟩

You may even use predicates on term zippers (see `Term_Zipper`):

lemma *foo n + foo m < 4*
⟨*proof*⟩

You can use induction:

lemma *foo n + foo m < 4*
⟨*proof*⟩

Again, you may also use patterns and predicates:

lemma *foo n + foo m < 4*
⟨*proof*⟩

Here are some more complex combinations (the original code from the standard library is marked with ORIG in the following). Note that configurations for different actions are separated by *where*.

lemma *list-induct2*:
length xs = length ys ⟹ *P* [] [] ⟹
(⋀*x xs y ys*. *length xs = length ys* ⟹ *P xs ys* ⟹ *P* (*x*#*xs*) (*y*#*ys*))
⟹ *P xs ys*
⟨*proof*⟩

lemma *list-induct2'*:
⟦ *P* [] [];
⋀*x xs*. *P* (*x*#*xs*) [];
⋀*y ys*. *P* [] (*y*#*ys*);
⋀*x xs y ys*. *P xs ys* ⟹ *P* (*x*#*xs*) (*y*#*ys*) ⟧
⟹ *P xs ys*
⟨*proof*⟩

Data passed as method modifiers can also be stored in the context more generally:

fun *gauss* :: *nat* ⇒ *nat* **where**
gauss 0 = 0
| *gauss* (*Suc n*) = *n + 1 + gauss n*

context notes *gauss.induct*[*zip-induct* (*pat*) (- :: *nat* − - -)]
begin
lemma *gauss n* = (*n* ∗ (*n + 1*)) *div 2* ⟨*proof*⟩

lemma *gauss n < gauss* (*Suc n*) ⟨*proof*⟩

lemma *gauss n > 0* ⟷ *n > 0* ⟨*proof*⟩
end

In some cases, it is necessary (or advisable for performance reasons) to change the search strategy from the default A^* (`Zip.AStar`) search to

breadth-first (`Zip.Breadth_First`), depth-first (`Zip.Depth_First`), or best-first (`Zip.Best_First`) search. You can either try them individually or use `Zip.Try` to search for the fastest one in parallel. Note that `Zip.Try` is only meant for exploration. It should be replaced by the discovered, most efficient strategy in the final proof document!

lemma *list-induct3*:
length xs = length ys $\Longrightarrow$ *length ys = length zs* $\Longrightarrow$ *P* [] [] [] $\Longrightarrow$
($\bigwedge$*x xs y ys z zs. length xs = length ys* $\Longrightarrow$ *length ys = length zs* $\Longrightarrow$ *P xs ys zs* $\Longrightarrow$ *P* (*x*#*xs*) (*y*#*ys*) (*z*#*zs*))
$\Longrightarrow$ *P xs ys zs*
⟨*proof*⟩

zip is also registered to **try0** for each search strategy:

lemma *map f xs = map g ys* $\longleftrightarrow$ *length xs = length ys* $\wedge$ ($\forall$ *i*<*length ys. f* (*xs*!*i*) = *g* (*ys*!*i*))
— use the try0 command to see all successful attempts
⟨*proof*⟩

One can use conditional substitution rules:

lemma *filter-insort*:
sorted (*map f xs*) $\Longrightarrow$ *P x* $\Longrightarrow$ *filter P* (*insort-key f x xs*) = *insort-key f x* (*filter P xs*)
⟨*proof*⟩

lemma *rev-eq-append-conv*: *rev xs = ys* @ *zs* $\longleftrightarrow$ *xs = rev zs* @ *rev ys*
⟨*proof*⟩

lemma *dropWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ $\Longrightarrow$
dropWhile (λ*y. y* ≠ *x*) (*rev xs*) = *x* # *rev* (*takeWhile* (λ*y. y* ≠ *x*) *xs*)
⟨*proof*⟩

Zip integrates the *blast* prover. In cases where *blast* loops or takes too long, users may specify a limit on its search depth or disable it completely:

lemma ($\forall$ *X. X* ⊆ *X*) $\Longrightarrow$ *1 + 1 = 2*

⟨*proof*⟩

Zip also integrates the *metis* provers. Since *metis* easily loops, it is only activated if (1) users explicitly provide it some options and/or rules and (2) *zip* is not expected to make any promising progress on the given goal node without using *metis* (cf. the setup in *Zippy.Zip-Metis*). Users may pass several options/runs to metis using the separator *and*.

A typical workflow with *zip* looks as follows:

1. Check if *zip* is successful.
2. If it is unsuccessful but terminates:
   (a) Check if there is a relevant lemma missing based on the returned goals. It might also be helpful to check non-promising attempts by switching from `Zip.AStar.promising'` to `Zip.AStar.all'` (or analogously for another search strategy).
   (b) Call **sledgehammer** on the remaining goals and either add the metis calls to *zip* directly or check if there are some helpful lemmas in the *metis* call that you could add another way (e.g. as a simp or classical rule) such that the *metis* call becomes unnecessary. Hint: you may also want to use *metis-instantiate* for this step.
3. If it is unsuccessful and does not terminate:
   (a) Check if *zip* is successful with a different strategy (e.g. by using `Zip.Try` or trying the strategies manually) and/or depth limit.
   (b) If none of the above is successful:
      i. Limit *zip*'s search steps to a number such that the set of returned subgoals looks reasonable to you.
      ii. Check which of the returned subgoals were not solvable by sequentially applying *zip*[*1*] to each subgoal. For each subgoal that is not solved: continue with step 2 of this workflow.

lemma *extract-Cons-code*: *List.extract P* (*x* # *xs*) = (*if P x then Some* ([], *x*, *xs*) *else* (*case List.extract P xs of None* ⇒ *None* | *Some* (*ys*, *y*, *zs*) ⇒ *Some* (*x*#*ys*, *y*, *zs*)))

⟨*proof*⟩

lemma *longest-common-prefix*:
∃ *ps xs′ ys′*. *xs* = *ps* @ *xs′* ∧ *ys* = *ps* @ *ys′* ∧ (*xs′* = [] ∨ *ys′* = [] ∨ *hd xs′* ≠ *hd ys′*)
⟨*proof*⟩

lemma *not-distinct-decomp*: ¬ *distinct ws* ⟹ ∃ *xs ys zs y*. *ws* = *xs*@[*y*]@*ys*@[*y*]@*zs*

⟨*proof*⟩

lemma *lexord-cons-cons*:
(*a* # *x*, *b* # *y*) ∈ *lexord r* ⟷ (*a* = *b* ∧ (*x*,*y*)∈ *lexord r*) ∨ (*a*,*b*)∈ *r* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

One can pass (elim-/dest-/forward-)rules that should be resolved by Isabelle's standard higher-order unifier, matcher, or a customisable proof-producing unifier (see `ML_Unification`). In contrast to the rules passed to the classical reasoner, each such rule can be annotated with individual data, e.g. priority, cost, and proof-producing unifier to be used.

Resolving rules with a proof-producing unifier is particularly useful in situations where equations do not hold up to $\alpha\beta\eta$-equality but some stronger, provable equality (see the examples theories in `ML_Unification` for more details). By default, the proof-producing unifier `Standard_Mixed_Comb_Unification.first_higherp_` is used, which uses the simplifier and unification hints (cf. *ML-Unification.ML-Unification-Hints*), among other things.

lemma
assumes *Q*
and [*simp*]: *Q* = *P*
shows *P*
⟨*proof*⟩

Passing rules to *urule* is particularly useful when dealing with definitions. In such cases, theorems for the abbreviated concept can re-used for the new definition (without making the definition opaque in general, as is the case with **abbreviation**):

definition *my-refl P* ≡ *reflp-on* {*x*. *P x*} — some derived concept

lemma *my-refl-uhint* [*uhint*]:
assumes {*x*. *P x*} ≡ *S*
shows *my-refl P* ≡ *reflp-on S*
⟨*proof*⟩

lemma *my-refl P Q* ⟹ *P x* ⟹ ∃ *x*. *Q x x*
⟨*proof*⟩

lemma
assumes ⋀*Q*. *P* (*reflp-on* {*x*. *Q x* ∧ *True*})
shows *True* ∧ *P* (*my-refl Q*)
⟨*proof*⟩

For very fine-grained control, one can even specify individual functions to initialise the proof trees linked to the rule's side conditions. By default, each such tree is again solved by all expansion actions registered to *zip*. Below, we override that behaviour and let the rule's second premise be solved by

reflexivity instead. You may check the technical overview section for more details about below code.

lemma
assumes $P \Longrightarrow U = U \Longrightarrow Q$
shows $A \longrightarrow Q$
⟨*proof*⟩

Zippy and zip both use the `Logger` from *ML-Unification.ML-Logger*. You can use it to trace its search. Check the logger's examples theory in `ML_Unification` for a demonstration how to filter and modify the logger's output.

lemma
assumes [*intro*]: $P \Longrightarrow Q$
and [*simp*]: $A = B$
shows $A \longrightarrow Q$
⟨*proof*⟩

2.34.2 Technical Overview

The method *zip* is based on the Zippy framework. For a preprint about the latter see [2]. On a high-level, the method has three phases:

1. Initialise the proof tree for a given goal.
2. Repeatedly expand and modify nodes of the proof tree.
3. Extract discovered theorems from the proof tree.

   The particularities of the expansion (e.g. order of expansion, expansion rules, search bounds) are largely customisable. Some configuration possibilities are demonstrated above.

   During initialisation, the proof tree is (typically) augmented with action clusters. Each action cluster stores a set of prioritised actions (short: *pactions*; cf. `Zippy_PAction_Mixin_Base`). A paction consists of a priority and a function modifying the proof tree, called an *action*. The paction's priority can be used to select action candidates during search.

   By default, the tree is initialised with the set of initialisation functions registered in `Zip.Run.Init_Goal_Cluster`. The current registrations can be seen as follows:

⟨*ML*⟩

A registration requires an identifier and an initialisation function modifying the proof tree in the desired way. Below, we register and delete an initialisation that adds an action cluster with a single paction, capable of closing goals by reflexivity:

```
declare [[zip-init-gc ‹
  let open Zippy; open ZLPC Base-Data MU; open A Mo
    val id = @{binding refl}
    (*action cluster metadata*)
    val ac-meta = Mixin-Base3.Meta.Meta.metadata (id, Lazy.value reflexivity ac-
tion cluster)
    (*action metadata*)
    val a-meta = Mixin-Base4.Meta.Meta.metadata (id, Lazy.value proof by reflex-
ivity)
    (*action application metadata*)
    fun mk-aa-meta - - = AAMeta.metadata {cost = Cost.VERY-LOW, (*cost of
the action's result*)
      progress = AAMeta.P.Promising} (*is it a promising expansion?*)
    fun ztac - = Ctxt.with-ctxt (fn ctxt => arr (resolve-tac ctxt @{thms refl}
     |> Tac-AAM.lift-tac mk-aa-meta |> Tac-AAM.Tac.zSOME-GOAL-FOCUS))
    val data = {
      (*disable the action once the tactic returns no results*)
      empty-action = Library.K Zippy.PAction.disable-action,
      meta = a-meta,
      (*attach a new node from the tactic's result and, for every remaining subgoal,
copy the actions
      registered for those subgoals from the node's parent*)
      result-action = Result-Action.action (Library.K (C.id ()))
        Result-Action.copy-update-data-empty-changed,
      (*the sequence of priorities for each pull from the tactic's result sequence*)
      presultsq = Zip.PResults.enum-scale-presultsq-default Cost.LOW,
      tac = ztac}
    (*attach the action cluster and step back to the parent node*)
     fun init - focus z = Tac.cons-action-cluster Util.exn (ACMeta.no-descr id)
[(focus, data)] z
      >>= AC.opt (K z) Up3.morph
  in (id, init) end›]]
declare [[zip-init-gc del: @{binding refl}]]
```

Since the kind of pactions tried by zip is extensible, the parser of *zip* is also extensible. Each parser has to apply its desired changes to the context and return a unit:

```
declare [[zip-parse add: ‹(@{binding refl},
  apfst (Config.put-generic Unify.search-bound 5) (*change the search bound*)
  #> tap (fn - => writeln I got parsed!) #> Scan.succeed ())›]]

lemma x = x
  ⟨proof⟩

declare [[zip-parse del: ‹@{binding refl}›]]
```

The initialised proof tree can then be expanded. By default, an A^* search is performed, taking into consideration the pactions' user supplied priorities and the sum of costs of the path leading to a paction. Other available search

strategies are depth-first and breadth-first search with A^*-cost tiebreakers, and best-first search on the pactions' priorities. Other search strategies may at will.

For more details, check the sources of the setup in *Zippy.Zip-Pure* and *Zippy.Zip-HOL*.

end
end

2.35 Zippy Paper Guide

theory *Zippy-Paper*
imports
Pure
begin

Summary Guide for the preprint[2]

- General Information
  - Unfortunately, employing the polymorphic record extension mechanism described in the paper hits severe performance problems of the Poly/ML compiler. To get around minute-long compilation times, all data fields of the zipper have to be instantiated in one go rather than by repeated instantiation of polymorphic fields, cf. `Zippy/Instances/zippy_instance_base.ML`. Relevant issue on GitHub: https://github.com/polyml/polyml/issues/213
- Section 2
  - Nodes `Gen_Zippers/Zippers5/Alternating_Zippers/Instances/Nodes/node.ML`
- Section 2.1
  - Categories and Arrows `ML_Typeclasses/Gen_Typeclasses/Typeclasses_1/Categories/category.ML`
  - Morphs
    - Morphism Base `Gen_Zippers/Zippers5/Morphs/morph_base.ML`
    - Morphisms `Gen_Zippers/Zippers5/Morphs/morph.ML`
  - Zippers
    - Zipper Morphs `Gen_Zippers/Zippers5/Zippers/zipper_morphs.ML`
    - Zipper Data `Gen_Zippers/Zippers5/Zippers/zipper_data.ML`
    - Zipper `Gen_Zippers/Zippers5/Zippers/zipper.ML`

  - Linked Zippers
    * Linked Zipper Morphs `Gen_Zippers/Zippers5/Linked_Zippers/linked_zipper_morphs.ML`
    * Linked Zippers `Gen_Zippers/Zippers5/Linked_Zippers/linked_zipper.ML`
  - Alternating Zippers
    * Alternating Zipper Morphs `Gen_Zippers/Zippers5/Alternating_Zippers/alternating_zipper_morphs.ML`
    * Alternating Zippers `Gen_Zippers/Zippers5/Alternating_Zippers/alternating_zipper.ML`
    * Alternating Zipper Product `Gen_Zippers/Zippers5/Alternating_Zippers/pair_alternating_zipper.ML`

- Section 2.1.1
  - Monads `ML_Typeclasses/Gen_Typeclasses/Typeclasses_1/typeclass_base.ML`
  - Kleisli Category `ML_Typeclasses/Gen_Typeclasses/Typeclasses_1/Categories/category_instance.ML`
  - Generating Alternating Zippers from Node Zippers
    * Extending the Alternating Zipper `Gen_Zippers/Zippers5/Alternating_Zippers/Instances/Nodes/alternating_zipper_nodes.ML`
    * Extending and Lifting the Input Zippers `Gen_Zippers/Zippers5/Zippers/extend_zipper_context.ML`
  - Generating Node Zippers `Gen_Zippers/Zippers5/Alternating_Zippers/Instances/Nodes/alternating_zipper_nodes_simple_zippers.ML`

- Section 2.2
  - Lenses `ML_Typeclasses/Gen_Typeclasses/Typeclasses_1/Lenses/lens.ML`

- Section 3
  - We implemented a generalisation of the state monad that also allows the state type to change during computation. Such states are not monads but (Atkey) indexed monads.
    * Atkey Indexed Monads `ML_Typeclasses/Gen_Typeclasses/Typeclasses_1/itypeclass_base.ML`
    * Indexed State Monad `ML_Typeclasses/Gen_Typeclasses/Typeclasses_1/State/istate.ML`
    * State Monad `ML_Typeclasses/Gen_Typeclasses/Typeclasses_1/State/istate.ML`

  - Antiquotations
    * Sources `ML_Typeclasses/Antiquotations/ML_Eval_Antiquotation.thy` `ML_Typeclasses/Antiquotations/ML_Args_Antiquotations.thy` `Antiquotations/ML_IMap_Antiquotation.thy`
    * Example Configuration and Follow-Up ML-Code Generation `Gen_Zippers/Zippers5/Morphs/ML_Morphs.thy`

- Section 4. We highlight the differences/extensions to the paper description
  - The zipper uses an additional "action cluster" layer that sits between a goal cluster and an action. Action clusters collect related actions, e.g. one could create an action cluster for classical reasoners, one for simplification actions, etc. This gives the search tree some more structure but is not strictly necessary (it is thus omitted in the paper).
  - Adding Actions `Zippy/Actions/zippy_paction_mixin_base.ML`
    * Action nodes do not store a static cost and action but, more generally, an "action with priority" (paction) that dynamically computes a priority, action pair.
    * Action clusters store a "copy" morphism such that actions generating new children can move their action siblings to the newly created child while updating their siblings' goal focuses (since the number and order of goals may have changed in the new child).
  - Adding Goals `Zippy/Goals/zippy_goals_mixin_base.ML`
    * Goal Clusters `Zippy/Goals/Base/zippy_goal_clusters.ML`
    * Goal Cluster `Zippy/Goals/Base/zippy_goal_cluster.ML`
    * Goal Focus `Zippy/Goals/Base/zippy_goal_focus.ML`
    * Union Find `Union_Find//imperative_union_find.ML`
  - Lifting Tactics
    * Lifting Isabelle Tactics to Zippy Tactics `Zippy/Tactics/Base/zippy_ztactic.ML`
    * Lifting Zippy Tactics to Actions `Zippy/Instances/zippy_instance_tactic.ML`
  - The Basic Search Tree Model `Zippy/Instances/zippy_instance_base.ML` Since there are is always exactly one goal clusters node, we do not use lists for the topmost layer.
    * List Zipper `Gen_Zippers/Zippers5/Zippers/Instances/list_zipper.ML`
  - Adding Failure and State `Zippy/Instances/Zippy_Instance_Pure.thy`

    - Option Monad and Transformers `ML_Typeclasses/Gen_Typeclasses/Typeclasses_1/typeclass_base_instance.ML`
  - Adding Positional Information
    - Extending the Alternating Zipper `Zippy/Positions/zippy_positions_mixin_base.`
    - Alternating (Global) Position Zipper `Gen_Zippers/Zippers5/Alternating_Zippers/Instances/alternating_global_position_zipper.ML`
  - Running a Search
    - Postorder Depth-First Enumeration for Zippers `Gen_Zippers/Zippers5/Zippers/Utils/df_postorder_enumerate_zipper.ML`
    - Postorder Depth-First Enumeration for Alternating Zippers `Gen_Zippers/Zippers5/Alternating_Zippers/Utils/df_postorder_enumerate_a`
    - Runs `Zippy/Runs/zippy_run_mixin.ML`
  - Retrieving Theorems from the Tree `Zippy/Goals/Lists/zippy_lists_goals_results_t`
  - Final example usages can be found here `Zippy/Instances/Zip/Examples/Zip_Examples.thy`.

end

Bibliography

[1] K. Kappelmann. Unification Utilities for Isabelle/ML. *Archive of Formal Proofs*, September 2023. https://isa-afp.org/entries/ML_Unification.html, Formal proof development.

[2] K. Kappelmann. Zippy – Generic White-Box Proof Search with Zippers. 2025. Work In Progress.