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-file‹*args-antiquotations.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‹
functor-instance ‹*struct-name*: *Para-Type-Args-Antiquotations*
  functor-name: *Args-Antiquotations*
  id: ‹*ParaT*›
  more-args: ‹*val init-args* = {
    args = *SOME* [*'p1*],
    sep = *SOME* , ,
    encl = *SOME* (,),
    encl-arg = *SOME* (,),
    start = *SOME 0*,
    stop = *SOME NONE*}››
›
local-setup ‹*Para-Type-Args-Antiquotations.setup-args-attribute*
  (*SOME set parameter type args antiquotation arguments*)›
setup ‹*Para-Type-Args-Antiquotations.setup-args-antiquotation*›
setup ‹*Para-Type-Args-Antiquotations.setup-arg-antiquotation*›

ML‹
structure ML-Gen =
struct
  structure ParaT = *Para-Type-Args-Antiquotations*
  val ParaT-nargs = *Context.the-generic-context #> ParaT.nargs*
  val ParaT-nargs' = *ParaT-nargs #> string-of-int*
  val mk-name = *space-implode -*
  fun sfx-ParaT-nargs s = *mk-name* [*s*, *ParaT-nargs'* ()]
end

›

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‹
  let
    (*adapted from Pure/ML/ml-file.ML (removed duplicated file−loading check by
skipping provide)*)
    fun command environment catch-all debug get-file = Toplevel.generic-theory (fn
gthy =>
      let
        val file = get-file (Context.theory-of gthy)
        (* val provide = Resources.provide-file file; *)
        val source = Token.file-source file

      val - = Document-Output.check-comments (Context.proof-of gthy) (Input.source-explode
source)

        val flags: ML-Compiler.flags =
          {environment = environment, redirect = true, verbose = true, catch-all =
catch-all,
            debug = debug, writeln = writeln, warning = warning}
      in
        gthy
        |> Local-Theory.touch-ml-env
        |> ML-Context.exec (fn () => ML-Context.eval-source flags source)
        |> Local-Theory.propagate-ml-env
        (* |> Context.mapping provide (Local-Theory.background-theory provide) *)
      end)
    val ML = command  false
  in
    Outer-Syntax.command command-keyword ‹ML-gen-file›
      read and evaluate Isabelle/ML file without duplication check.
      (Resources.parse-file −−| Scan.option keyword ‹;› >> ML NONE)
  end
›

end
```

1.2 ML Eval Antiquotation

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

Summary Antiquotation for ML evaluation.

ML-file‹*eval-antiquotation.ML*›

ML‹
  functor-instance ‹*struct-name*: *Standard-Eval-Antiquotation*
    functor-name: *Eval-Antiquotation*
    id: ‹›
    more-args: ‹*val init-args* = {
      parser = *SOME* (*Parse-Util.ML-string* (*K eval string must be non−empty.*))
    }››
›
local-setup ‹*Standard-Eval-Antiquotation.setup-attribute NONE*›
setup ‹*Standard-Eval-Antiquotation.setup-antiquotation*›

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‹*val sfx-ParaT-nargs* = *ML-Gen.sfx-ParaT-nargs*›

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‹
structure ***eval***‹*sfx-ParaT-nargs SLens*› =
eval‹*sfx-ParaT-nargs Lens*›(
structure L = ***eval***‹*sfx-ParaT-nargs Lens-Base*›(
eval‹*sfx-ParaT-nargs Arrow-Apply*›(
eval‹*sfx-ParaT-nargs SArrow-Apply*›))
structure A = ***eval***‹*sfx-ParaT-nargs Arrow*›(***eval***‹*sfx-ParaT-nargs SArrow-Apply*›))

signature ***eval***‹*sfx-ParaT-nargs SLENS*› =
eval‹*sfx-ParaT-nargs LENS*›
where type (@{*ParaT-args*} *'a*, *'b*) *C.morph* = *'a* −> *'b*
where type (@{*ParaT-args*} *'t*, *'o*, *'s*, *'i*) *lens* =
(@{*ParaT-args*} *'t*, *'o*, *'s*, *'i*) ***eval***‹*sfx-ParaT-nargs SLens* ^ *.lens*›
›

end

1.10 State

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

ML-file‹*state-result.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-file‹*extended-metis-data.ML*›

end

1.11 Tactic Utils

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

ML-file ‹~~/*src*/*Tools*/*IsaPlanner*/*zipper.ML*›

ML‹
structure Term-Zipper = *Zipper*
structure Term-Zipper-Search = *ZipperSearch*
›

ML-file‹*zippy-ml-tactic-util.ML*›

end

1.11.1 Cases

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

ML-file‹*cases-tactic.ML*›
ML-file‹*cases-data.ML*›

end

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

ML‹
structure Cases-Tactic-HOL = Cases-Tactic(open Induct
fun get-casesP ctxt (fact :: -) = find-casesP ctxt (Thm.concl-of fact)
| get-casesP - - = []
fun get-casesT ctxt binderTs (SOME t :: -) = find-casesT ctxt (Term.fastype-of1
(binderTs, t))
| get-casesT - - - = [])
structure Cases-Data-Args-Tactic-HOL = Cases-Data-Args-Tactic(Cases-Tactic-HOL)
›

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,...,t_n) when using *f.cases* while users would expect being able to supply n instantiations t_1,...,t_n. Below attribute transforms such rules to the expected form.

local-setup ‹
let
fun add-prodTs T acc = try HOLogic.dest-prodT T
|> (fn NONE => T :: acc | SOME (T, T') => T :: add-prodTs T' acc)
fun inst-prod-cases ctxt thm = case Thm.prop-of thm |> Term.add-vars |> build
|> rev of
[] => error No schematic variable in passed cases theorem.
| ((n, -), T) :: - =>
let
val maxidx = Thm.maxidx-of thm

```
        val inst = build (add-prodTs T)
          |> map-index (fn (idx, T) => Var ((n, maxidx + idx + 1), T))
          |> rev
          |> (fn t :: ts => fold (curry HOLogic.mk-prod) ts t)
          |> Thm.cterm-of ctxt
      in Thm.instantiate' [] [SOME inst] thm end
  in
    Attrib.local-setup @{binding deprod-cases}
      (Scan.succeed (Thm.rule-attribute [] (Context.proof-of #> inst-prod-cases)))
      (Transform cases rule that uses a single product for instantiation to a cases
rule using
       ^ multiple instantiations.)
  end
›

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-file‹extended-blast-data.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-file‹generic-term-index-data.ML›

end
```

1.13 Priority Queues

theory *ML-Priority-Queues*
  imports
    Pure
begin

ML-file‹*priority-queue.ML*›
ML-file‹*leftist-heap.ML*›

end

1.14 Actions

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

ML-file‹*zippy-action-result.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-file‹*imap-antiquotation.ML*›

ML‹
functor-instance ‹*struct-name*: *Standard-IMap-Antiquotation*
  functor-name: *IMap-Antiquotation*
  id: ‹›
  more-args: ‹*val init-args* = {
    sep = *SOME* *n*,
    encl = *SOME* (,),
    encl-inner = *SOME* (,),

start = SOME 1,
stop = SOME 2}››
›
local-setup ‹*Standard-IMap-Antiquotation.setup-attribute NONE*›
setup ‹*Standard-IMap-Antiquotation.setup-antiquotation*›

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‹
functor-instance ‹*struct-name*: *Zipper-Type-Args-Antiquotations*
functor-name: *Args-Antiquotations*
id: ‹*ZipperT*›
more-args: ‹*val init-args* = {
args = SOME ['*a1*],
sep = SOME , ,
encl = SOME (,),
encl-arg = SOME (,),
start = SOME 0,
stop = SOME NONE}››
›
local-setup ‹*Zipper-Type-Args-Antiquotations.setup-args-attribute*
(*SOME set zipper type args antiquotation arguments*)›
setup ‹*Zipper-Type-Args-Antiquotations.setup-args-antiquotation*›
setup ‹*Zipper-Type-Args-Antiquotations.setup-arg-antiquotation*›

ML‹
functor-instance ‹*struct-name*: *All-Type-Args-Antiquotations*
functor-name: *Args-Antiquotations*
id: ‹*AllT*›
more-args: ‹*val init-args* = {

```
    args = SOME ['p1, 'a1],
    sep = SOME , ,
    encl = SOME ((, )),
    encl-arg = SOME (, ),
    start = SOME 0,
    stop = SOME NONE}››
›
local-setup ‹All-Type-Args-Antiquotations.setup-args-attribute
  (SOME set all type args antiquotation arguments)›
setup ‹All-Type-Args-Antiquotations.setup-args-antiquotation›
setup ‹All-Type-Args-Antiquotations.setup-arg-antiquotation›


ML‹
structure ML-Gen =
struct
  open ML-Gen
  structure AllT = All-Type-Args-Antiquotations
  structure ZipperT = Zipper-Type-Args-Antiquotations
  fun nzippers () = ZipperT.nargs (Context.the-generic-context ())
  val nzippers' = nzippers #> string-of-int

  fun setup-zipper-args paraT-args zipperT-args =
    ParaT.map-args (K paraT-args)
    #> ZipperT.map-args (K zipperT-args)
    #> AllT.map-args (K (paraT-args @ zipperT-args))
    #> Standard-IMap-Antiquotation.map-stop (K (length zipperT-args))
 fun setup-zipper-args' (opt-ParaT-nargs, opt-ParaT-arg) (opt-nzippers, opt-zipperT-arg)
context =
    let
      val ParaT-nargs = if-none ‹ParaT.nargs context› opt-ParaT-nargs
      val ParaT-arg = if-none ‹string-of-int #> prefix 'p› opt-ParaT-arg
      val ParaT-args = map-range ParaT-arg ParaT-nargs
      val nzippers = if-none ‹nzippers ()› opt-nzippers
      val zipperT-arg = if-none ‹string-of-int #> prefix 'a› opt-zipperT-arg
      val zipperT-args = map-range zipperT-arg nzippers
    in setup-zipper-args ParaT-args zipperT-args context end

  (*ML structure names may not begin with a digit; hence we add a n prefix for
indexed names*)
  val nprefix = prefix n
  fun pfx-nzippers s = mk-name [nprefix (nzippers' ()), s]

  (*modular arithmetic for domain [1,...,nzippers]*)
  fun add-mod-nzippers i j = ((i + j - 1) mod nzippers ()) + 1
  fun add-mod-nzippers' i = add-mod-nzippers i #> string-of-int
  fun sub-mod-nzippers i j = ((i - j - 1) mod nzippers ()) + 1
  fun sub-mod-nzippers' i = sub-mod-nzippers i #> string-of-int
  val succ-mod-nzippers = add-mod-nzippers 1
```

```
  val succ-mod-nzippers' = add-mod-nzippers' 1
  fun pred-mod-nzippers i = sub-mod-nzippers i 1
  fun pred-mod-nzippers' i = sub-mod-nzippers' i 1

  fun sfx-T-nargs s = mk-name [s, ParaT-nargs' (), nzippers' ()]

  val pfx-sfx-nargs = pfx-nzippers #> sfx-T-nargs

  (*instantiate a zipper type*)
  fun sfx-inst s i = mk-name [s, string-of-int i]

  fun sfx-inst-T-nargs s = sfx-inst s #> sfx-T-nargs
  val pfx-sfx-inst-nargs = pfx-nzippers #> sfx-inst-T-nargs

  fun inst-zipperT instT i =
    let val ctxt = Context.the-local-context ()
    in
      nzippers ()
       |> map-range (fn j => if i = j + 1 then instT else ZipperT.mk-arg-code j
ctxt)
      |> commas
    end
end
›

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]]

setup‹Context.theory-map (ML-Gen.setup-zipper-args' (NONE, NONE) (SOME
5, NONE))›

ML‹
  val sfx-ParaT-nargs = ML-Gen.sfx-ParaT-nargs
  val sfx-T-nargs = ML-Gen.sfx-T-nargs
  val sfx-inst-T-nargs = ML-Gen.sfx-inst-T-nargs
›
```

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*›

setup‹*fn theory => let val nzippers = ML-Gen.nzippers () + 1*
in Context.theory-map (ML-Gen.setup-zipper-args' (NONE, NONE) (SOME nzippers, NONE)) theory end›

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

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

setup‹*fn theory => let val nzippers = ML-Gen.nzippers () − 1*
in Context.theory-map (ML-Gen.setup-zipper-args' (NONE, NONE) (SOME nzippers, NONE)) theory end›

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‹
val mk-name = ML-Gen.mk-name
›

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‹
val pfx-sfx-nargs = ML-Gen.pfx-sfx-nargs

val succ-mod-nzippers = ML-Gen.succ-mod-nzippers′
val pred-mod-nzippers = ML-Gen.pred-mod-nzippers′
›

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-file‹*zippy-exception-mixin-base.ML*›

ML-file‹*zippy-exception-mixin.ML*›

end

2.11 Loggers

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

ML-file‹*zippy-logger-mixin-base.ML*›

end

2.12 Pretty Printing/Shows

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

ML-file‹*zippy-show-mixin-base.ML*›

end

2.13 States

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

ML-file‹*zippy-ctxt-state-mixin-base.ML*›
ML-file‹*zippy-ctxt-state-mixin.ML*›

ML-file‹*zippy-state-mixin-base.ML*›
ML-file‹*zippy-state-mixin.ML*›

end

2.14 Zippy Base

theory *Zippy-Base*
imports

ML-Alternating-Zipper-Utils
Zippy-Exceptions
Zippy-Loggers
Zippy-Shows
Zippy-States
begin

ML-file‹*zippy-monad-util.ML*›

ML-file‹*zippy-base-base.ML*›
ML-file‹*zippy-base.ML*›

ML‹
(∗*Grounding utilities needed to store zippy data/functions as context data* (*since only monomorphic*
data can be stored in the generic context). *Note that both ParaT and ZipperT arguments are grounded.*

FIXME: *generalise loading of ML code dependent on the grounding such that it can be re−loaded with*
different ground types.
∗)
structure ML-Gen =
struct
open ML-Gen
val ground-zipper-types =
  let val mk-groundT = *K unit*
 in ML-Gen.setup-zipper-args′ (*NONE*, *SOME mk-groundT*) (*NONE*, *SOME mk-groundT*)
end
val reset-zipper-types = *ML-Gen.setup-zipper-args*′ (*NONE*, *NONE*) (*NONE*, *NONE*)
end
›

end

2.15 Coroutines

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

ML-file‹*zippy-coroutines-mixin-base.ML*›

end

2.16 Enums

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

ML-file‹*zippy-enum-mixin-base.ML*›
ML-file‹*zippy-enum-mixin.ML*›

end

2.17 Identifiers

theory *Zippy-Identifiers*
imports
Pure
begin

Summary Identifiers for Zippy

ML-file‹*zippy-identifier.ML*›

end

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

ML-file‹*zippy-paction-mixin-base.ML*›
ML-file‹*zippy-paction-mixin.ML*›

ML-file‹*zippy-paction-queue-mixin-base.ML*›
ML-file‹*zippy-paction-queue-mixin.ML*›

ML-file‹*zippy-presults-mixin-base.ML*›
ML-file‹*zippy-presults-mixin.ML*›

ML-file‹*zippy-paction-presults-mixin-base.ML*›
ML-file‹*zippy-paction-presults-mixin.ML*›

ML-file‹*zippy-action-metadata.ML*›
ML-file‹*zippy-action-metadata-mixin-base.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*›

setup‹*fn theory =>*
let val nzippers = ML-Gen.nzippers () + 1
in Context.theory-map (ML-Gen.setup-zipper-args' (NONE, NONE) (SOME nzippers, NONE)) theory end›

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

ML-gen-file‹*node.ML*›
ML‹
  val succ-node-sig = sfx-T-nargs NODE
  val succ-node-functor = sfx-T-nargs Node
›
setup‹*fn theory =>*
let val nzippers = ML-Gen.nzippers () − 1
in Context.theory-map (ML-Gen.setup-zipper-args' (NONE, NONE) (SOME nzippers, NONE)) theory end›

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-file‹*zipper-direction.ML*›

end

theory *ML-Zipper-Positions*

imports
ML-Categories
Zippy-Base-Setup
begin

ML-file‹*zipper-position.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-file‹*zippy-node-base.ML*›
ML-file‹*zippy-node.ML*›

end

2.20 Positions

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

ML-file‹*zippy-positions-mixin-base.ML*›
ML-file‹*zippy-positions-mixin.ML*›

ML-file‹*zippy-node-positions-mixin-base.ML*›

end

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

ML-file‹*zippy-presults-positions-mixin.ML*›

end

2.21 Union-Find

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

ML-file‹*imperative-union-find.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-file‹*zippy-thm-state.ML*›
ML-file‹*zippy-goal-clusters.ML*›
ML-file‹*zippy-goal-cluster.ML*›

ML-file‹*zippy-goal-focus.ML*›

ML-file‹*zippy-goal-results.ML*›

ML-file‹*zippy-top-meta-vars.ML*›

end

theory *Zippy-Goals*
imports

ML-Alternating-Zipper-Paths
Zippy-Base
Zippy-Goals-Base
begin

ML-file‹*zippy-goal-clusters-mixin-base.ML*›
ML-file‹*zippy-goal-clusters-mixin.ML*›

ML-file‹*zippy-goal-cluster-mixin-base.ML*›
ML-file‹*zippy-goal-cluster-mixin.ML*›

ML-file‹*zippy-goals-mixin-base.ML*›

ML-file‹*zippy-goal-focus-mixin-base.ML*›

ML-file‹*zippy-goal-results-mixin-base.ML*›
ML-file‹*zippy-goal-results-mixin.ML*›

ML-file‹*zippy-goals-results-mixin-base.ML*›

ML-file‹*zippy-top-meta-vars-mixin-base.ML*›
ML-file‹*zippy-top-meta-vars-mixin.ML*›

ML-file‹*zippy-goal-results-top-meta-vars-mixin-base.ML*›
ML-file‹*zippy-goal-results-top-meta-vars-mixin.ML*›

ML-file‹*zippy-goals-results-top-meta-vars-mixin-base.ML*›

end

2.23 Lists

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

ML-file‹*zippy-lists-base.ML*›
ML-file‹*zippy-lists.ML*›

end

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

ML-file‹*zippy-lists-goals-mixin.ML*›
ML-file‹*zippy-lists-goals-results-mixin.ML*›
ML-file‹*zippy-lists-goals-results-top-meta-vars-mixin.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-file‹*generic-table-data.ML*›

end

2.25 Action Clusters

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

ML-file‹*zippy-copy-mixin-base.ML*›
ML-file‹*zippy-copy-mixin.ML*›
ML-file‹*zippy-enum-copy-mixin.ML*›

ML-file‹*zippy-action-cluster-metadata.ML*›
ML-file‹*zippy-action-cluster-metadata-mixin-base.ML*›

end

2.26 Goal Position Updates

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

ML-file‹*zippy-goal-pos-update.ML*›
ML-file‹*zippy-goal-pos-update-util.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-file‹*zippy-goals-pos-mixin-base.ML*›
ML-file‹*zippy-goals-pos-mixin.ML*›

setup‹*Context.theory-map ML-Gen.ground-zipper-types*›
ML-file‹*zippy-update-goal-cluster-mixin-base.ML*›
ML-file‹*zippy-update-goal-cluster-mixin.ML*›

setup‹*Context.theory-map ML-Gen.reset-zipper-types*›

ML-file‹*zippy-goals-pos-copy-mixin-base.ML*›
ML-file‹*zippy-goals-pos-copy-mixin.ML*›

end

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

ML-file‹*zippy-lists-goals-pos-copy-mixin.ML*›

end

2.27 Collect

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

ML-file‹*zippy-collect-trace-mixin-base.ML*›
ML-file‹*zippy-collect-trace-mixin.ML*›
ML-file‹*zippy-node-collect-trace-mixin-base.ML*›

end

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

ML-file‹*zippy-lists-collect-mixin-base.ML*›

end

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

ML-file‹*zippy-lists-positions-mixin-base.ML*›
ML-file‹*zippy-lists-positions-mixin.ML*›

end

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

ML-file‹*zippy-lists-positions-collect-mixin-base.ML*›

end

2.28 Action Applications

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

ML-file‹*zippy-action-app-num.ML*›
ML-file‹*zippy-action-app-progress.ML*›

end

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

begin

ML-file‹*zippy-action-app-num-mixin-base.ML*›
ML-file‹*zippy-action-app-num-mixin.ML*›

ML-file‹*zippy-action-app-metadata.ML*›
ML-file‹*zippy-action-app-metadata-mixin-base.ML*›
ML-file‹*zippy-action-app-metadata-mixin.ML*›
ML-file‹*zippy-enum-action-app-metadata-mixin.ML*›

ML-file‹*zippy-prio-mixin-base.ML*›

end

2.29 Zippy Tactics

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

ML-file‹*zippy-rtactic-result.ML*›
ML-file‹*zippy-rtactic.ML*›

ML-file‹*zippy-ztactic-result.ML*›
ML-file‹*zippy-ztactic.ML*›

end

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

ML-file‹*zippy-tactic-action-app-metadata-mixin-base.ML*›
ML-file‹*zippy-tactic-action-app-metadata-mixin.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-file‹*zippy-instance-base.ML*›
ML-file‹*zippy-instance.ML*›
ML-file‹*zippy-instance-paction.ML*›
ML-file‹*zippy-instance-presults.ML*›
ML-file‹*zippy-instance-tactic.ML*›

end

2.30.1 Homogenously Changed Goals Data

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

setup‹*Context.theory-map ML-Gen.ground-zipper-types*›
ML-file‹*zippy-instance-hom-changed-goals-data.ML*›

setup‹*Context.theory-map ML-Gen.reset-zipper-types*›

end

2.30.2 Cases

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

setup‹*Context.theory-map ML-Gen.ground-zipper-types*›
ML-file‹*zippy-instance-cases-data.ML*›
setup‹*Context.theory-map ML-Gen.reset-zipper-types*›

end

2.30.3 Classical Reasoner

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

ML-file‹*zippy-instance-classical.ML*›

end

2.30.4 Induction

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

ML-file‹*induction-tactic.ML*›
ML-file‹*induction-data.ML*›

ML‹
structure Induction-Tactic-HOL = Induction-Tactic(*Induct*)
structure Induction-Data-Args-Tactic-HOL = Induction-Data-Args-Tactic(*Induction-Tactic-HOL*)
›

end

2.30.5 Induction

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

setup‹*Context.theory-map ML-Gen.ground-zipper-types*›
ML-file‹*zippy-instance-induction-data.ML*›
setup‹*Context.theory-map ML-Gen.reset-zipper-types*›

end

Substitution

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

setup‹*Context.theory-map ML-Gen.ground-zipper-types*›
ML-file‹*zippy-instance-subst-data.ML*›
setup‹*Context.theory-map ML-Gen.reset-zipper-types*›

end

2.31 Customisable Context Parser

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

ML-file‹*context-parsers.ML*›

end

2.31.1 Simplifier

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

ML-file‹*extended-simp-data.ML*›

end

Resolution

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

setup‹*Context.theory-map ML-Gen.ground-zipper-types*›
ML-file‹*zippy-instance-uresolve-data.ML*›
ML-file‹*zippy-instance-resolves-data.ML*›
ML-file‹*zippy-instance-uresolves-data.ML*›
setup‹*Context.theory-map ML-Gen.reset-zipper-types*›

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.

setup‹*Context.theory-map ML-Gen.ground-zipper-types*›
ML-file‹*zippy-instance-resolves-simp.ML*›
setup‹*Context.theory-map ML-Gen.reset-zipper-types*›

end

theory *Zippy-Runs-Base*
imports
Pure
begin

ML-file‹*zippy-run-result.ML*›

end

2.32 Sequences

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

ML-file‹*zippy-seq-from-monad-mixin-base.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-file‹*zippy-step-mixin-base.ML*›
ML-file‹*zippy-step-mixin.ML*›

ML-file‹*zippy-run-mixin-base.ML*›
ML-file‹*zippy-run-mixin.ML*›

setup‹*Context.theory-map ML-Gen.ground-zipper-types*›
ML-file‹*zippy-run-data.ML*›
setup‹*Context.theory-map ML-Gen.reset-zipper-types*›

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‹
(∗ *create monad* ∗)
local
(∗∗ *exceptions* ∗∗)
structure ME = ***eval***‹*sfx-ParaT-nargs Monad-Exception-Monad-Or*›(
eval‹*sfx-ParaT-nargs Option-Monad-Or-Trans*›(
eval‹*sfx-ParaT-nargs Identity-Monad*›))

(∗∗ *proof context* ∗∗)
structure Ctxt-MSTrans = ***eval***‹*sfx-ParaT-nargs State-Trans*›(
type s = *Proof.context*; *structure M* = *ME*; *structure SR* = *Pair-State-Result-Base*)
structure ME = ***eval***‹*sfx-ParaT-nargs Monad-Exception-State-Trans*›(
structure M = *ME*; *structure S* = *Ctxt-MSTrans*)

(∗∗ *arbitrary state (not needed for now)* ∗∗)
(∗ *structure MS* = ***eval***‹*sfx-ParaT-nargs IState-Trans*›(
structure M = *Ctxt-MSTrans*; *structure SR* = *Pair-State-Result-Base*)
structure ME = ***eval***‹*sfx-ParaT-nargs IMonad-Exception-State-Trans*›(
structure M = *ME*; *structure S* = *MS*)
structure ME : ***eval***‹*sfx-ParaT-nargs MONAD-EXCEPTION-BASE*› =
struct open ME; *type* (@{*ParaT-args*} *'a*) *t* = (*unit*, @{*ParaT-arg 0*}, @{*ParaT-arg 0*}, *'a*) *t end*
structure MCtxt = ***eval***‹*sfx-ParaT-nargs IMonad-State-State-Trans*›(
type ParaT = *unit*; *structure M* = *Ctxt-MSTrans*; *structure S* = *ME*) ∗)

structure Ctxt = *Zippy-Ctxt-State-Mixin*(*Zippy-Ctxt-State-Mixin-Base*(*Ctxt-MSTrans*))

(∗∗ *possibility to extract sequences from monad* ∗∗)
structure Seq-From-Monad = *Zippy-Seq-From-Monad-Mixin-Base*(*structure M* = *ME*
type @{*AllT-args*} *state* = {*ctxt* : *Proof.context*(∗, *state* : @{*ParaT-arg 0*}∗)}
fun seq-from-monad {*ctxt*(∗, *state*∗)} *m* = *Seq.make* (*fn* - =>
(∗*State-MSTrans.eval state m* |>∗) *Ctxt-MSTrans.eval ctxt m* |> *the-default Seq.empty*
|> *Seq.pull*))

(∗ *instance and utilities* ∗)
val exn : @{*ParaT-args encl*: ( )} *ME.exn* = ()

```
  structure Zippy-Base = Zippy-Instance-Base(structure ME = ME
    type prio = Cost.cost
    type cost = Cost.cost
    val eq-cost = Cost.eq
    val pretty-cost = Cost.pretty
    fun update-cost c NONE = c
      | update-cost c (SOME (c', -)) = Cost.add (c, c'))
  structure Logging =
  struct
    val logger = Logger.setup-new-logger zippy-base-logger Zippy
    local structure Shared = struct val parent-logger = logger end
    in
    structure Base = Zippy-Logger-Mixin-Base(open Shared; val name = Base)
    structure Copy = Zippy-Logger-Mixin-Base(open Shared; val name = Copy)
     structure Enum-Copy = Zippy-Logger-Mixin-Base(open Shared; val name =
Enum-Copy)
   structure PAction = Zippy-Logger-Mixin-Base(open Shared; val name = PAction)
    structure LGoals = Zippy-Logger-Mixin-Base(open Shared; val name = LGoals)
    structure LGoals-Pos-Copy = Zippy-Logger-Mixin-Base(open Shared; val name
= LGoals-Pos-Copy)
    structure PAction-PResults = Zippy-Logger-Mixin-Base(open Shared; val name
= PAction-PResults)
     structure Result-Action = Zippy-Logger-Mixin-Base(open Shared; val name =
Result-Action)
    end
  end
  structure Zippy = Zippy-Instance(
     structure Z = Zippy-Base; structure Ctxt = Ctxt; structure Log-Base = Log-
ging.Base
    structure Log-LGoals = Logging.LGoals; structure Log-LGoals-Pos-Copy = Log-
ging.LGoals-Pos-Copy
    structure Show-Prio = Zippy-Show-Mixin-Base(
      type @{AllT-args} t = Zippy-Base.Base-Data.PAction.prio
      val pretty = Library.K Cost.pretty))
  structure Zippy-PAction = Zippy-Instance-PAction(
    structure Z = Zippy
    val mk-exn = Library.K exn
    structure Ctxt = Ctxt; structure Log-Base = Logging.Base;
     structure Log-PAction = Logging.PAction; structure Log-Result-Action = Log-
ging.Result-Action
    structure Log-LGoals = Logging.LGoals; structure Log-LGoals-Pos-Copy = Log-
ging.LGoals-Pos-Copy
   structure Log-Copy = Logging.Copy; structure Log-Enum-Copy = Logging.Enum-Copy)
  structure Zippy-PResults = Zippy-Instance-PResults(
    structure Z = Zippy-PAction
    structure Ctxt = Ctxt; structure Log-PAction = Logging.PAction
    structure Log-PAction-PResults = Logging.PAction-PResults)
  structure Zippy-Tactic = Zippy-Instance-Tactic(
    structure Z = Zippy-PResults; structure Ctxt = Ctxt; structure Log-PAction =
```

```
Logging.PAction)
in
(* final creation of structure *)
structure Zippy =
struct
open Zippy-Base Zippy Zippy-PAction Zippy-PResults Zippy-Tactic
(** add loggers **)
structure Logging = Logging
(** add monads **)
(* structure State-MSTrans = MS
structure State = Zippy-State-Mixin(Zippy-State-Mixin-Base(MS)) *)
structure Ctxt-MSTrans = Ctxt-MSTrans
structure Ctxt = Ctxt
structure Seq-From-Monad = Seq-From-Monad
(** add base data **)
structure Base-Data = struct open Base-Data; structure Cost = Cost end
(** extend some basic and add some compound mixins **)
local
 structure Base-Mixins = struct structure Exn = Exn; structure Co = Co; structure
Ctxt = Ctxt end
 structure Goals = Zippy-Goals-Mixin-Base(
  structure GClusters = Mixin-Base1.GClusters; structure GCluster = Mixin-Base2.GCluster)
 structure Goals-Pos = Zippy-Goals-Pos-Mixin(structure Z = ZLPC
  structure Goals-Pos = Zippy-Goals-Pos-Mixin-Base(open Goals; structure GPU
= Base-Data.Tac-Res.GPU))
 structure Goals-Pos-Copy = Zippy-Goals-Pos-Copy-Mixin(Zippy-Goals-Pos-Copy-Mixin-Base(open
Goals-Pos
   structure Copy = Mixin-Base3.Copy))
in
structure ZB = Zippy-Base(open Base-Mixins; structure Z = ZLPC; structure Log
= Logging.Base
 imap ‹‹{i}› => ‹structure Show{i} = Show.Zipper{i}››)
structure ZN = Zippy-Node(open Base-Mixins; structure Z = ZLPC)
structure ZL = Zippy-Lists(open Base-Mixins; structure Z = ZLPC)
structure ZE = Zippy-Enum-Mixin(open Base-Mixins; open ZLPC)
structure ZP = Zippy-Positions-Mixin(open Base-Mixins; structure Z = ZLPC.ZP)
structure Mixin2 =
struct structure GCluster = Zippy-Goal-Cluster-Mixin(Zippy.Mixin-Base2.GCluster)
end
structure LGoals = Zippy-Lists-Goals-Mixin(structure Z = ZL; structure Goals =
Goals
 structure Ctxt = Ctxt; structure Log = Logging.LGoals)
structure LGoals-Pos-Copy = Zippy-Lists-Goals-Pos-Copy-Mixin(open Base-Mixins;
structure Z = ZL
 structure GPC = Goals-Pos-Copy;
 structure Log = Logging.LGoals-Pos-Copy; structure Log-Base = Logging.Base;
 structure Log-LGoals = Logging.LGoals; imap ‹‹{i}› => ‹structure Show{i} =
Show.Zipper{i}››)
structure Goals-Results = Zippy-Goals-Results-Mixin-Base(open LGoals-Pos-Copy
```

```
  structure GClusters-Results = Mixin-Base1.Results; structure GCluster-Results
= Mixin-Base2.Results)
structure Goals-Results-TMV = Zippy-Goals-Results-Top-Meta-Vars-Mixin-Base(open
Goals-Results
  structure Top-Meta-Vars = Mixin-Base2.Top-Meta-Vars)
structure PAction =
struct
  local structure PAction-More = Zippy-PAction-Mixin(open Base-Mixins
    structure PAction = Mixin-Base4.PAction; structure Log = Logging.PAction
    structure Show = Show.Zipper4)
  in open PAction-More PAction end
end
structure PResults =
struct
  local structure PResults-More = Zippy-PResults-Mixin(PResults)
  in open PResults-More PResults end
  (*exponential backoff with base s*)
  fun enum-scale-presultsq s = enum-presultsq (MU.A.arr (Cost.scale s))
end
end
(** add instance specific utilities **)
structure Util =
struct
  val exn = exn
  fun mk-exn - = exn
  local
    open ZLPC
    structure ZN = Zippy-Node(structure Z = ZLPC; structure Exn = Exn)
    fun node-no-next1 gclusters = ZN.node-no-next1 exn (Node.co1 gclusters)
    fun node-no-next2 parent-top-meta-vars gcluster = ZN.node-no-next2 exn
      (Node.co2 parent-top-meta-vars gcluster)
  in
  fun run-monad {ctxt : Proof.context(*, state : @{ParaT-arg 0}*)} :
    (@{ParaT-args} 'a) M.t -> {ctxt : Proof.context(*, state : @{ParaT-arg 0}*),
result : 'a} option =
    (*State-MSTrans.run state #>*) Ctxt-MSTrans.run ctxt #> Option.map (fn x
=>
      let
        val ctxt = Ctxt-MSTrans.SR.state x
        (* val x = Ctxt-MSTrans.SR.value x
        val state = State-MSTrans.SR.state x
        val result = State-MSTrans.SR.value x *)
        val result = Ctxt-MSTrans.SR.value x
      in {ctxt = ctxt(*, state = state*), result = result} end)
  fun init-thm-state (st : Zippy-Thm-State.state) : (@{ParaT-args} @{AllT-args}
Z1.zipper) M.t =
    LGoals.init-state
    (node-no-next1 #> ZN.container-no-parent exn #> Container.init-container1
#> Z1.ZM.Zip.morph)
```

```
    (node-no-next2 (Zippy-Thm-State.meta-vars st)) st
  end
end
end
end
›

ML‹
(*add run and steps*)
structure Zippy =
struct open Zippy
local open MU; open SC A Mo
  structure Base-Mixins =
  struct
    structure Z = ZLPC
    structure Exn = Exn; structure Co = Co; structure Ctxt = Ctxt
    imap ‹‹{i}› => ‹
    structure Show{i} = Show.Zipper{i}
    structure Show-Container{i} = Show.Container{i}››
  end
in
(* steps *)
(** base **)
structure Step =
struct
 local structure Log = Zippy-Logger-Mixin-Base(val parent-logger = Logging.logger;
val name =  Step)
  in open Log end
  local structure ZCost = Zippy-Collect-Trace-Mixin(ZLPC.ZCollect)
  in
  fun check-depth-limit opt-limit = ZP.ZZDepth-Co4.getter
    #> (fn depth => case opt-limit of
        NONE => pure depth
      | SOME l => if depth <= l then pure depth else Exn.ME.throw Util.exn)
  structure AStar =
  struct
    structure Logging =
    struct
      val logger = Logger.setup-new-logger logger AStar
      local structure Base = struct val parent-logger = logger end
      in
      structure PAction-Queue = Zippy-Logger-Mixin-Base(open Base; val name =
PAction-Queue)
      structure Step = Zippy-Logger-Mixin-Base(open Base; val name = Step)
      end
    end
    structure PAction-Queue = Zippy-PAction-Queue-Mixin-Base(
      structure PAction = PAction
     structure Queue = Leftist-Heap(type prio = PAction.prio; val ord = Cost.ord))
```

```
    structure Show =
    struct
      structure Queue-Entry = Zippy-Show-Mixin-Base(
        type @{AllT-args} t = @{AllT-args} PAction-Queue.entry
        fun pretty ctxt {prio, zipper,...} = SpecCheck-Show.record [
          (Priority, Show.Prio.pretty ctxt prio),
          (Zipper, Show.Zipper4.pretty ctxt zipper)])
      structure Prio = Show.Prio
    end
    structure PAction-Queue = Zippy-PAction-Queue-Mixin(open Base-Mixins
      structure Z = ZE; structure PAction-Queue = PAction-Queue
      val mk-exn = Util.mk-exn; structure Log = Logging.PAction-Queue
       structure Show-Queue-Entry = Show.Queue-Entry; structure Show-Prio =
Show.Prio)
    structure Step = Zippy-Step-Mixin(open Base-Mixins
      structure Step = Zippy-Step-Mixin-Base(open Goals-Results-TMV
        structure PAction-Queue = PAction-Queue)
      val mk-exn = Util.mk-exn; structure Log = Logging.Step
      structure Log-LGoals = Zippy.Logging.LGoals
      structure Log-PAction-Queue = Logging.PAction-Queue
       structure Show-Queue-Entry = Show.Queue-Entry; structure Show-Prio =
Show.Prio)
    fun mk-prio ({prio, zipper,...} : @{AllT-args} PAction-Queue.entry) =
      ZCost.ZZCollect-Co4.getter zipper |> ZCost.get-current
      |> Option.map (Library.curry Cost.add prio) |> the-default prio
    fun mk-prio-depth-limit opt-depth-limit (entry as {zipper,...}) =
      check-depth-limit opt-depth-limit zipper >>= arr (fn - => mk-prio entry)
  end
  end

  structure Best-First =
  struct
    structure Logging =
    struct
      val logger = Logger.setup-new-logger logger Best-First
      local structure Base = struct val parent-logger = logger end
      in
      structure PAction-Queue = Zippy-Logger-Mixin-Base(open Base; val name =
PAction-Queue)
      structure Step = Zippy-Logger-Mixin-Base(open Base; val name = Step)
      end
    end
    structure PAction-Queue = Zippy-PAction-Queue-Mixin-Base(
      structure PAction = PAction
      structure Queue = AStar.PAction-Queue.Queue)
    structure Show =
    struct
      structure Queue-Entry = Zippy-Show-Mixin-Base(
        type @{AllT-args} t = @{AllT-args} PAction-Queue.entry
```

```
      fun pretty ctxt {prio, zipper,...} = SpecCheck-Show.record [
        (Priority, Show.Prio.pretty ctxt prio),
        (Zipper, Show.Zipper4.pretty ctxt zipper)])
    structure Prio = AStar.Show.Prio
  end
  structure PAction-Queue = Zippy-PAction-Queue-Mixin(open Base-Mixins
    structure Z = ZE; structure PAction-Queue = PAction-Queue
    val mk-exn = Util.mk-exn; structure Log = Logging.PAction-Queue
     structure Show-Queue-Entry = Show.Queue-Entry; structure Show-Prio =
Show.Prio)
  structure Step = Zippy-Step-Mixin(open Base-Mixins
    structure Step = Zippy-Step-Mixin-Base(open Goals-Results-TMV
     structure PAction-Queue = PAction-Queue)
    val mk-exn = Util.mk-exn; structure Log = Logging.Step
    structure Log-LGoals = Zippy.Logging.LGoals
    structure Log-PAction-Queue = Logging.PAction-Queue
     structure Show-Queue-Entry = Show.Queue-Entry; structure Show-Prio =
Show.Prio)
  fun mk-prio ({prio,...} : @{AllT-args} PAction-Queue.entry) = prio
  fun mk-prio-depth-limit opt-depth-limit (entry as {zipper,...}) =
    check-depth-limit opt-depth-limit zipper >>= arr (fn - => mk-prio entry)
end

local
  type prio = {depth : int, prio : PAction.prio}
  fun prio-ord depth-ord ({depth = depth1, prio = prio1}, {depth = depth2, prio
= prio2}) =
  prod-ord depth-ord Cost.ord ((depth1, prio1), (depth2, prio2))
in
structure Depth-First =
struct
  structure Logging =
  struct
    val logger = Logger.setup-new-logger logger Depth-First
    local structure Base = struct val parent-logger = logger end
    in
    structure PAction-Queue = Zippy-Logger-Mixin-Base(open Base; val name =
PAction-Queue)
    structure Step = Zippy-Logger-Mixin-Base(open Base; val name = Step)
    end
  end
  structure PAction-Queue = Zippy-PAction-Queue-Mixin-Base(
    structure PAction = PAction
    structure Queue = Leftist-Heap(type prio = prio; val ord = prio-ord (int-ord
#> rev-order)))
  structure Show =
  struct
    structure Queue-Entry = Zippy-Show-Mixin-Base(
      type @{AllT-args} t = @{AllT-args} PAction-Queue.entry
```

```
      fun pretty ctxt {prio, zipper,...} = SpecCheck-Show.record [
        (Priority, Show.Prio.pretty ctxt prio),
        (Zipper, Show.Zipper4.pretty ctxt zipper)])
    structure Prio = Zippy-Show-Mixin-Base(
      type @{AllT-args} t = PAction-Queue.Queue.prio
      fun pretty ctxt {depth, prio} = SpecCheck-Show.record [
        (Depth, SpecCheck-Show.int depth),
        (Priority, Show.Prio.pretty ctxt prio)])
  end
  structure PAction-Queue = Zippy-PAction-Queue-Mixin(open Base-Mixins
    structure Z = ZE; structure PAction-Queue = PAction-Queue
    val mk-exn = Util.mk-exn; structure Log = Logging.PAction-Queue
     structure Show-Queue-Entry = Show.Queue-Entry; structure Show-Prio =
Show.Prio)
  structure Step = Zippy-Step-Mixin(open Base-Mixins
    structure Step = Zippy-Step-Mixin-Base(open Goals-Results-TMV
      structure PAction-Queue = PAction-Queue)
    val mk-exn = Util.mk-exn; structure Log = Logging.Step
    structure Log-LGoals = Zippy.Logging.LGoals
    structure Log-PAction-Queue = Logging.PAction-Queue
     structure Show-Queue-Entry = Show.Queue-Entry; structure Show-Prio =
Show.Prio)
  fun mk-prio-depth-limit opt-depth-limit (entry as {zipper,...}) =
    check-depth-limit opt-depth-limit zipper
    >>= arr (fn depth => AStar.mk-prio entry |> (fn prio => {depth = depth,
prio = prio}))
 end

 structure Breadth-First =
 struct
  structure Logging =
  struct
    val logger = Logger.setup-new-logger logger Breadth-First
    local structure Base = struct val parent-logger = logger end
    in
    structure PAction-Queue = Zippy-Logger-Mixin-Base(open Base; val name =
PAction-Queue)
    structure Step = Zippy-Logger-Mixin-Base(open Base; val name = Step)
    end
  end
 structure PAction-Queue = Zippy-PAction-Queue-Mixin-Base(structure PAction
= PAction
  structure Queue = Leftist-Heap(type prio = Depth-First.PAction-Queue.Queue.prio
      val ord = prio-ord int-ord))
  structure Show =
  struct
    structure Queue-Entry = Zippy-Show-Mixin-Base(
      type @{AllT-args} t = @{AllT-args} PAction-Queue.entry
      fun pretty ctxt {prio, zipper,...} = SpecCheck-Show.record [
```

```
        (Priority, Show.Prio.pretty ctxt prio),
        (Zipper, Show.Zipper4.pretty ctxt zipper)])
    structure Prio = Depth-First.Show.Prio
  end
  structure PAction-Queue = Zippy-PAction-Queue-Mixin(open Base-Mixins
    structure Z = ZE; structure PAction-Queue = PAction-Queue
    val mk-exn = Util.mk-exn; structure Log = Logging.PAction-Queue
     structure Show-Queue-Entry = Show.Queue-Entry; structure Show-Prio =
Show.Prio)
  structure Step = Zippy-Step-Mixin(open Base-Mixins
    structure Step = Zippy-Step-Mixin-Base(open Goals-Results-TMV
      structure PAction-Queue = PAction-Queue)
    val mk-exn = Util.mk-exn; structure Log = Logging.Step
    structure Log-LGoals = Zippy.Logging.LGoals
    structure Log-PAction-Queue = Logging.PAction-Queue
     structure Show-Queue-Entry = Show.Queue-Entry; structure Show-Prio =
Show.Prio)
  val mk-prio-depth-limit = Depth-First.mk-prio-depth-limit
 end
 end
 end

(* run *)
structure Run =
struct
(** base **)
fun with-state f = Ctxt.with-ctxt (fn ctxt => (*State.with-state (fn state =>*)
 f {ctxt = ctxt(*, state = state*)})
local structure Run = Zippy-Run-Mixin(open Base-Mixins
 structure Run = Zippy-Run-Mixin-Base(open Goals-Results
  structure Seq-From-Monad = Seq-From-Monad)
 val with-state = with-state
 structure Log = Zippy-Logger-Mixin-Base(val parent-logger = Logging.logger; val
name =  Run))
in open Run end
(** utility functions **)
local
 structure GClusters = Zippy-Goal-Clusters-Mixin(GClusters)
 structure GClusters-Results = Zippy-Goal-Results-Mixin(GClusters-Results)
 structure LGoals-Results-TMV = Zippy-Lists-Goals-Results-Top-Meta-Vars-Mixin(open
Base-Mixins
  structure Z = Zippy-Lists(open Base-Mixins)
  structure Goals-Results-Top-Meta-Vars = Goals-Results-TMV; structure Log-LGoals
= Logging.LGoals)
 structure EAction-App-Meta = Zippy-Enum-Action-App-Metadata-Mixin(
  structure Z = ZE
  structure Meta = Zippy-Action-App-Metadata-Mixin(Zippy.Mixin-Base5.Meta))
in
fun run-statesq init-sstate step finish fuel c = init-sstate c
```

```
  >>= with-state (fn ms => arr (fn ss => repeat-step step finish finish fuel ss ms
c
    |> Seq.maps (Zippy-Run-Result.cases #thm-states I)))
fun run-statesq' init-sstate step mk-unreturned-statesq = run-statesq init-sstate step
(fn - => fn - =>
  ZLPC.Z1.ZM.Zip.morph >>> mk-unreturned-statesq >>> arr (Zippy-Run-Result.Unfinished
#> Seq.single))
fun mk-df-post-unreturned-statesq x = mk-unreturned-statesq (Ctxt.with-ctxt
   (LGoals-Results-TMV.mk-statesq (LGoals-Results-TMV.enum-df-post-children2
Util.mk-exn))) x
fun mk-df-post-unreturned-promising-statesq x = mk-unreturned-statesq (Ctxt.with-ctxt
  (LGoals-Results-TMV.mk-statesq (EAction-App-Meta.enum-df-post-promising-children2
Util.mk-exn))) x

local
  fun mk-funs mk-prio-depth-limit init-pactions-queue step-queue =
  let
    fun gen finish opt-depth-limit =
      let val mk-prio = mk-prio-depth-limit opt-depth-limit
      in run-statesq' (init-pactions-queue mk-prio) (step-queue mk-prio) finish end
    fun mk-limit-variants f = (gen f, SOME #> gen f, gen f NONE)
  in
    (gen,
    mk-limit-variants mk-df-post-unreturned-statesq,
    mk-limit-variants mk-df-post-unreturned-promising-statesq)
  end
in
structure AStar =
struct
  local open Step.AStar
  in
  val (gen, (gen-all, all, all'), (gen-promising, promising, promising')) =
    mk-funs mk-prio-depth-limit PAction-Queue.init-pactions-queue Step.step-queue
  end
end
structure Best-First =
struct
  local open Step.Best-First
  in
  val (gen, (gen-all, all, all'), (gen-promising, promising, promising')) =
    mk-funs mk-prio-depth-limit PAction-Queue.init-pactions-queue Step.step-queue
  end
end
structure Depth-First =
struct
  local open Step.Depth-First
  in
  val (gen, (gen-all, all, all'), (gen-promising, promising, promising')) =
    mk-funs mk-prio-depth-limit PAction-Queue.init-pactions-queue Step.step-queue
```

end
end
structure Breadth-First =
struct
local open Step.Breadth-First
in
val (gen, (gen-all, all, all′), (gen-promising, promising, promising′)) =
mk-funs mk-prio-depth-limit PAction-Queue.init-pactions-queue Step.step-queue
end
end
end
end
end
end
end
›

end

Simplification

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

ML-file‹*zippy-instance-simp.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‹
local structure Zippy-Simp = Zippy-Instance-Simp(structure Z = Zippy; structure Ctxt = Z.Ctxt)
in structure Zippy = struct open Zippy-Simp Zippy end end›

ML‹
local open Zippy

```
  val default-presultsq-scale = Rat.make (12, 10)
  structure Resolve-Base =
  struct
    structure Z = Zippy
    val cost = Cost.MEDIUM
    structure TI = Discrimination-Tree
    open Base-Data.AAMeta
    fun init-args get-thm = {
      empty-action = SOME (Library.K PAction.disable-action),
       default-update = SOME (fn - => @{undefined}), (*just a temporary place-
holder*)
       mk-cud = SOME Result-Action.copy-update-data-empty-changed,
     presultsq = SOME (PResults.enum-scale-presultsq default-presultsq-scale cost),
       mk-meta = SOME (Library.K (Library.K (metadata {cost = cost, progress
= P.promising}))),
       del-select = SOME (apsnd (snd #> get-thm) #> Thm.eq-thm)}
  end
in
functor-instance ‹struct-name: Zip
  functor-name: Zippy-Instance-Resolves-Simp
  id: ‹zip›
  more-args: ‹open Resolve-Base; open Z
   val resolve-init-args = init-args Zippy-Instance-Hom-Changed-Goals-Data-Args.PD.get-thm
    val simp-init-args = {timeout = SOME 10.0, depth = NONE}
    structure Log-Base = Z.Logging.Base››
structure Zip =
struct open Zip
(*add resolution with proof−producing unification*)
structure Logging =
struct open Logging
  structure URule = Zippy-Logger-Mixin-Base(val parent-logger = logger; val name
= URule)
end

local val default-update = Run.init-gpos
in
val - = Theory.setup (Rule.Resolve.map-default-update (K default-update)
  #> Rule.EResolve.map-default-update (K default-update)
  #> Rule.DResolve.map-default-update (K default-update)
  #> Rule.FResolve.map-default-update (K default-update)
  #> Match.Resolve.map-default-update (K default-update)
  #> Match.EResolve.map-default-update (K default-update)
  #> Match.DResolve.map-default-update (K default-update)
  #> Match.FResolve.map-default-update (K default-update)
  |> Context.theory-map)

functor-instance ‹struct-name: URule
  functor-name: Zippy-Instance-UResolves-Data
  id: ‹FI.prefix-id urule›
```

```
  path: ‹FI.long-name›
  more-args: ‹open Resolve-Base
    structure PDC = Zippy-Instance-Hom-Changed-Goals-Data-Args.PDC
    val init-args = init-args Zippy-Instance-UResolve-Data-Args.PD.get-rule |> (fn
args => {
    normalisers = SOME Standard-Mixed-Comb-Unification.norms-first-higherp-comb-unify,
      unifier = SOME Standard-Mixed-Comb-Unification.first-higherp-comb-unify,
      mk-meta = SOME (PDC.get-mk-meta args),
      empty-action = SOME (PDC.get-empty-action args),
      default-update = SOME default-update,
      mk-cud = SOME (PDC.get-mk-cud args),
      presultsq = SOME (PDC.get-presultsq args),
      del-select = SOME (PDC.get-del-select args)})
    structure Log = Logging.URule››
end

structure PResults =
struct
val default-presultsq-scale = default-presultsq-scale
val enum-scale-presultsq-default = Zippy.PResults.enum-scale-presultsq default-presultsq-scale
end
end
end
›
local-setup‹Zip.Run.Init-Goal-Cluster.Data.setup-attribute
  (Either.Right goal cluster initialisation)›
local-setup‹Zip.Simp.Extended-Data.setup-attribute (Either.Right extended simp)›

ML‹
structure Zip =
struct open Zip(* add parsers *)
functor-instance ‹struct-name: Context-Parsers
  functor-name: Context-Parsers
  id: ‹FI.prefix-id parse›
  path: ‹FI.long-name›
  more-args: ‹
    val parent-logger = Logging.logger
    val parsers-separator = where››

(* add instance specific utilities *)
structure Run =
struct open Run
local open Zippy; open ZLPC MU; open SC A
in
fun init st = st |>
  (Util.init-thm-state >>> Down1.morph >>> Z2.ZM.Unzip.morph
  >>> Init-Goal-Cluster.update-all (Library.K Util.exn)
    (arr (Mixin2.GCluster.get-ngoals #> Base-Data.Tac-Res.GPU.F.all-upto))
  >>> top2 >>> Z1.ZM.Unzip.morph)
```

```
val are-thm-variants = apply2 Thm.prop-of #> Term-Util.are-term-variants
fun changed-uniquesq st = Seq.filter (fn st' => not (are-thm-variants (st, st')))
  #> Tactic-Util.unique-thmsq are-thm-variants

functor-instance ‹struct-name: Data
  functor-name: Zippy-Run-Data
  id: ‹FI.prefix-id run›
  path: ‹FI.struct-op Run›
  more-args: ‹
    structure Z = ZLPC
    structure Ctxt = Ctxt
    structure Seq-From-Monad = Seq-From-Monad
    type exec-config = int option (*fuel*)
    val init-args = {
      init = SOME init,
      exec = SOME Run.AStar.promising',
      post = SOME (fn st => Ctxt.with-ctxt (fn ctxt =>
        arr (changed-uniquesq st #> Seq.maps (prune-params-tac ctxt))))}››
fun tac fuel ctxt = Data.tac fuel {ctxt = ctxt}
end
end

(** some convenience abbreviations **)
structure AStar = Zippy.Run.AStar
structure Depth-First = Zippy.Run.Depth-First
structure Breadth-First = Zippy.Run.Breadth-First
structure Best-First = Zippy.Run.Best-First
(** try all executors in parallel **)
structure Try =
struct
local open Zippy; open MU
  fun exec-all fs ctxt =
    let fun run (who, f) = Timing.timing
      (Seq-From-Monad.seq-from-monad {ctxt = ctxt} #> Seq.filter Thm.no-prems
#> Seq.pull) f
      |> (fn (-, NONE) => NONE
        | (timing, x) => (warning (who ^ finished. Timing: ^ Timing.message
timing); x))
      |> Library.K |> Seq.make
    in Par-List.map run fs |> Seq.of-list |> Seq.flat |> (fn sq => Mo.pure sq ctxt)
end
in
val execs = [AStar, Depth-First, Breadth-First, Best-First]
fun gen post depth fuel c = [AStar.gen, Depth-First.gen, Breadth-First.gen, Best-First.gen]
  |> List.map (fn f => f post depth fuel c) |> curry (op ~~) execs |> exec-all
fun gen-all depth fuel c = [AStar.gen-all, Depth-First.gen-all, Breadth-First.gen-all,
Best-First.gen-all]
  |> List.map (fn f => f depth fuel c) |> curry (op ~~) execs |> exec-all
```

```
fun all depth fuel c = [AStar.all, Depth-First.all, Breadth-First.all, Best-First.all]
  |> List.map (fn f => f depth fuel c) |> curry (op ~~) execs |> exec-all
fun all' fuel c = [AStar.all', Depth-First.all', Breadth-First.all', Best-First.all']
  |> List.map (fn f => f fuel c) |> curry (op ~~) execs |> exec-all
end
end
end
›
local-setup‹Zip.Context-Parsers.setup-attribute NONE›
local-setup‹Zip.Run.Data.setup-attribute NONE›
declare [[zip-parse add: ‹(@{binding run}, Zip.Run.Data.parse-context-update)›]]
```

Method

```
local-setup ‹
  let open Zippy Zip.Run
    val parse-fuel = Parse-Util.option Parse.nat
    val parse = Scan.lift parse-fuel --| Zip.Context-Parsers.parse
      >> (Method.SIMPLE-METHOD oo tac)
   in Method.local-setup Zip.binding parse Extensible white-box prover based on
Zippy end›
```

Resolution

```
local-setup‹Zip.Rule.setup-attribute
  (Either.Right (e/d/f-)resolution with higher-order unification)›
local-setup‹Zip.Match.setup-attribute
  (Either.Right (e/d/f-)resolution with higher-order matching)›
local-setup‹Zip.URule.setup-attribute
  (Either.Right (e/d/f-)resolution with proof-producing unification)›

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 disagreement 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-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‹
local structure Zippy-Classical = Zippy-Instance-Classical(*structure Z = Zippy*;
structure Ctxt = Z.Ctxt)
in
structure Zippy = struct open Zippy-Classical Zippy end
structure Zip =
struct open Zip
functor-instance ‹*struct-name*: *Blast*
functor-name: *Extended-Blast-Data*
id: ‹*FI.prefix-id blast*›
path: ‹*FI.long-name*›
more-args: ‹
val init-args = {depth = SOME 4, timeout = SOME 10.0}
val parent-logger = Logging.logger
››
end
end
›

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}›]]

local-setup‹Zip.Blast.setup-attribute NONE›
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‹
structure Zip =
struct open Zip
local open Zippy
in
functor-instance ‹struct-name: Subst
 functor-name: Zippy-Instance-Subst-Data
 id: ‹FI.id›
 path: ‹FI.long-name›
 more-args: ‹
   structure Z = Zippy
   structure TI = Discrimination-Tree
   val cost = Cost.MEDIUM
   open Base-Data.AAMeta
   val init-args = {
     empty-action = SOME (Library.K PAction.disable-action),
     default-update = SOME Zip.Run.init-gpos,
     mk-cud = SOME Result-Action.copy-update-data-empty-changed,
     presultsq = SOME (Zip.PResults.enum-scale-presultsq-default cost),
      mk-meta = SOME (Library.K (Library.K (metadata {cost = cost, progress
= P.promising}))),
     del-select = SOME (apsnd (snd #> #thm #> the) #> Thm.eq-thm)}
   structure Log = Logging››
end
```

```
end
›
local-setup‹Zip.Subst.setup-attribute NONE›

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‹
structure Zip =
struct open Zip
local open Zippy
  structure Base-Args =
  struct
    open Base-Data
    structure Z = Zippy
    structure Ctxt = Ctxt
    fun mk-init-args cost = {
      simp = SOME true,
     match = SOME (can Seq.hd oooo Type-Unification.e-unify Unification-Util.unify-types
      (Mixed-Unification.first-higherp-e-match Unification-Combinator.fail-match)),
      empty-action = SOME (Library.K Zippy.PAction.disable-action),
      default-update = SOME Zip.Run.init-gpos,
      mk-cud = SOME Zippy.Result-Action.copy-update-data-empty-changed,
      presultsq = SOME (Zip.PResults.enum-scale-presultsq-default cost),
      mk-meta = SOME (Library.K (Library.K (AAMeta.metadata
        {cost = cost, progress = AAMeta.P.promising})))}
    structure Log = Logging
    structure Log-Base = Logging.Base
    structure Log-LGoals-Pos-Copy = Logging.LGoals-Pos-Copy
    structure Log-LGoals = Logging.LGoals
  end
in
functor-instance ‹struct-name: Cases
  functor-name: Zippy-Instance-Cases-Data
  FI-struct-name: FI-Cases-Data
  id: ‹FI.id›
  path: ‹FI.long-name›
  more-args: ‹open Base-Args
    val init-args = mk-init-args Cost.MEDIUM
  ››
structure Cases = Cases.Cases-Data
functor-instance ‹struct-name: Induction
  functor-name: Zippy-Instance-Induction-Data
  FI-struct-name: FI-Induction-Data
```

```
  id: ‹FI.id›
  path: ‹FI.long-name›
  more-args: ‹open Base-Args
    val init-args = mk-init-args Cost.HIGH
  ››
structure Induction = Induction.Induction-Data
end
end
›
local-setup ‹Zip.Cases.setup-attribute NONE›
local-setup ‹Zip.Induction.setup-attribute NONE›

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‹
structure Zip =
struct open Zip
functor-instance ‹struct-name: Metis
 functor-name: Extended-Metis-Data
 id: ‹FI.prefix-id metis›
 path: ‹FI.long-name›
 more-args: ‹
  val init-args = {runs = SOME [], timeout = SOME 10.0}
  val parent-logger = Logging.logger
  ››
end
›

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 ‹
structure Zip =
struct open Zip
structure Try0 =
struct
 local val method-name = FI.id
 in
 fun add-run-config exec s =
  let
    val (pfx, sfx) = first-field method-name s |> the
     |> apfst (fn pfx => (not (String.isSuffix ( pfx) ? suffix () pfx)
    val (mid, sfxs) = Substring.full sfx |> (if String.isSuffix ] sfx
     then Substring.splitr (not o equal #[)
       #> apfst (Substring.trimr 1) #> apsnd (single #> cons (Substring.full
[))
     else rpair [])
    val (mid, close-parenth) = Substring.splitr (equal #)) mid |> apsnd (K ))
    val config-prefix = if Substring.isEmpty mid then
     else Zip.Context-Parsers.parsers-separator ^
    val exec = Zip.FI.struct-op (ML-Syntax-Util.mk-struct-access exec all')
    val run-config = implode [config-prefix, run exec: , exec]
  in
```

```
    implode [pfx, method-name, Substring.string mid, " ", run-config, close-parenth]
    ^ Substring.concat sfxs
  end
 fun gen-register (name, update-command)=
  Try0.register-proof-method name {run-if-auto-try = true}
  (Option.map (fn {command, time, state,...} =>
    {name = name, command = update-command command, time = time, state
= state})
    ooo Try0-Util.apply-raw-named-method method-name true Try0-Util.full-attrs
    (Try0-HOL.silence-methods
    #> Context.proof-map (Logger.set-log-levels Logger.root Logger.OFF)))
 val mk-registrations = map (fn exec => (method-name ^ "-" ^ exec, add-run-config
exec))
  #> cons (method-name, I)
end
end
end

local val registrations = Zip.Try0.mk-registrations Zip.Try.execs
in
val - = map Zip.Try0.gen-register registrations
val - = Theory.setup (Config.map-global Try0.schedule
 (prefix (implode-space (map fst registrations) ^ " ")))
end
›

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* ⟷ *sorted* (*rev l*) ∧ *distinct l*
by (*induction l*) (*zip iff*: *antisym-conv1 simp*: *sorted-wrt-append*)

lemma *sorted-wrt* (>) *l* ⟷ *sorted* (*rev l*) ∧ *distinct l*
by (*induction l*) (*zip iff*: *antisym-conv1 simp*: *sorted-wrt-append*)

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 $A \longrightarrow Q$
apply *zip*
back
back
oops

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 $A \longrightarrow Q$
apply (*zip run exec*: *Zip.AStar.all'*)
oops

Alternatively, the introduction rule can be marked as safe:

lemma
assumes [*intro*!]: $P \Longrightarrow Q$
and [*simp*]: $A = B$
shows $A \longrightarrow Q$
apply *zip*
oops

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
apply (*zip 20 run exec*: *Zip.AStar.all'*) — Note: loops if no limit is passed
oops

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

lemma
assumes [*intro*]: $P \wedge P \Longrightarrow P$
shows P
apply (*zip run exec*: *Zip.AStar.all 2*) — Note: loops if no depth limit is passed
oops

You can perform case splits:

lemma *tl xs* $\neq$ [] $\longleftrightarrow$ *xs* $\neq$ [] $\wedge$ $\neg(\exists\, x.\ xs = [x])$
by (*zip cases xs*)

fun *foo* :: *nat* $\Rightarrow$ *nat* **where**
foo 0 = 0
| *foo* (*Suc 0*) = *1*
| *foo* (*Suc* (*Suc n*)) = *1*

lemma *foo n + foo m < 4*
by (*zip cases n rule*: *foo.cases* **and** *m rule*: *foo.cases*)

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*)
by (*zip cases* (*pat*) (- :: *nat* − - -) *rule*: *foo.cases*)

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 $nat \times bool$ 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 $nat \times bool$ 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
  by (zip cases (pat) - :: nat - :: bool rule: bar.cases[deprod-cases])
```

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

```
lemma foo n + foo m < 4
  by (zip cases (pred) fst #> member (op =) [@{term n}, @{term m}] rule:
foo.cases)
```

You can use induction:

```
lemma foo n + foo m < 4
  by (zip induct rule: foo.induct)
```

Again, you may also use patterns and predicates:

```
lemma foo n + foo m < 4
  by (zip induct (pat) (- :: nat − - -) rule: foo.induct)
```

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
  by (zip induct xs arbitrary: ys where cases (pat) (- :: - list − [] - -))
```

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*
by (*zip induct xs arbitrary*: *ys* **where** *cases* (*pat*) (- :: - *list* − [] - -))

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* **by** *zip*

lemma *gauss n* < *gauss* (*Suc n*) **by** *zip*

lemma *gauss n* > *0* ⟷ *n* > *0* **by** *zip*
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* ⟹ *length ys* = *length zs* ⟹ *P* [] [] [] ⟹
(⋀*x xs y ys z zs*. *length xs* = *length ys* ⟹ *length ys* = *length zs* ⟹ *P xs ys zs*
⟹ *P* (*x*#*xs*) (*y*#*ys*) (*z*#*zs*))
⟹ *P xs ys zs*
by (*induct xs arbitrary*: *ys zs*)
— this suggests Breadth_First
(*zip cases* (*pat*) (- :: - *list* − []) **where** *run exec*: *Zip.Breadth-First.all′*)

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

lemma *map f xs* = *map g ys* ⟷ *length xs* = *length ys* ∧ (∀ *i*<*length ys*. *f* (*xs*!*i*)
= *g* (*ys*!*i*))
— use the try0 command to see all successful attempts
by (*zip simp*: *list-eq-iff-nth-eq*)

One can use conditional substitution rules:

lemma *filter-insort*:

sorted (*map f xs*) ⟹ *P x* ⟹ *filter P* (*insort-key f x xs*) = *insort-key f x* (*filter P xs*)
by (*induct xs*)
(*zip subst insort-is-Cons* **where** *run exec*: *Zip.Best-First.all'*)
— this also works, but it is slower

lemma *rev-eq-append-conv*: *rev xs* = *ys* @ *zs* ⟷ *xs* = *rev zs* @ *rev ys*
by (*zip subst rev-rev-ident*[*symmetric*])

lemma *dropWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ ⟹
dropWhile (λ*y*. *y* ≠ *x*) (*rev xs*) = *x* # *rev* (*takeWhile* (λ*y*. *y* ≠ *x*) *xs*)
by (*zip induct xs* **where** *subst dropWhile-append2*)

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 (∀ *X*. *X* ⊆ *X*) ⟹ *1* + *1* = *2*

by (*zip blast depth*: *0*)

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)))*

apply (*zip simp add*: *extract-def comp-def split*: *list.splits*
where *metis dropWhile-eq-Nil-conv list.distinct*(*1*))
done

lemma *longest-common-prefix*:
∃ *ps xs' ys'*. *xs = ps @ xs' ∧ ys = ps @ ys' ∧ (xs' = [] ∨ ys' = [] ∨ hd xs' ≠ hd ys')*
apply (*induct xs ys rule*: *list-induct2'*)

apply (*zip metis append-eq-Cons-conv list.sel*(*1*) *self-append-conv*)

done

lemma *not-distinct-decomp*: ¬ *distinct ws* ⟹ ∃ *xs ys zs y*. *ws = xs@[y]@ys@[y]@zs*
proof (*induct n* ≡ *length ws arbitrary*:*ws*)
case (*Suc n ws*)
then show *?case* **using** *length-Suc-conv* [*of ws n*]

apply (*zip metis append-Cons* **and** *split-list append-eq-append-conv2*)
done

qed *simp*

lemma *lexord-cons-cons*:
(*a* # *x*, *b* # *y*) ∈ *lexord r* ⟷ (*a* = *b* ∧ (*x*,*y*)∈ *lexord r*) ∨ (*a*,*b*)∈ *r* (**is** *?lhs* = *?rhs*)
by (*zip metis hd-append list.sel*(*1*,*3*) *tl-append2* **and** *Cons-eq-append-conv*)

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*
by (*zip urule assms*)

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*
using *assms* **unfolding** *my-refl-def* **by** *simp*

lemma *my-refl P Q* ⟹ *P x* ⟹ ∃ *x*. *Q x x*

by (*zip urule* (*d*) *reflp-onD*) — we can directly use ⟦*reflp-on ?A ?R*; *?x* ∈ *?A*⟧ ⟹ *?R ?x ?x* as a destruction rule

lemma
assumes ⋀*Q*. *P* (*reflp-on* {*x*. *Q x* ∧ *True*})
shows *True* ∧ *P* (*my-refl Q*)
by (*zip urule assms*)

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 ⟹ U = U ⟹ Q
  shows A ⟶ Q
  apply (zip rule assms updates: [2: ‹fn i =>
    let open Zippy; open ZLPC Base-Data MU; open SC A
      val id = @{binding refl}
      val a-meta = AMeta.metadata (id, Lazy.value proof by reflexivity)
      fun mk-aa-meta - - = AAMeta.metadata {cost = Cost.VERY-LOW, progress
= AAMeta.P.Promising}
      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))
    in
      Tac.cons-action-cluster Util.exn (ACMeta.no-descr id) [(GFocus.single i, {
        empty-action = Library.K Zippy.PAction.disable-action,
        meta = a-meta,
        result-action = Result-Action.action (Library.K (C.id ()))
          Result-Action.copy-update-data-empty-changed,
        presultsq = Zip.PResults.enum-scale-presultsq-default Cost.LOW,
        tac = ztac})]
      >>> (the #> Up3.morph)
    end›])
  back
  oops
```

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 ⟹ Q
  and [simp]: A = B
  shows A ⟶ Q
 supply [[ML-map-context ‹Logger.set-log-levels Zippy.Logging.logger Logger.DEBUG›]]
  apply zip
  oops
```

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-val‹*Zip.Run.Init-Goal-Cluster.Data.get-table* (*Context.the-generic-context* ())›

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 action cluster*)
(∗*action metadata*∗)
val a-meta = *Mixin-Base4.Meta.Meta.metadata* (*id*, *Lazy.value proof by reflexivity*)
(∗*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*
by (*zip refl*)

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.