System Requirements

The docker image is large (10.4 GB), as is the data output by a complete run of the benchmarks (an additional 3.1 GB), thus we recommend a computer with a decent amount of free disk space.

Note also that we have only tested the image on Linux and Mac (ARM), and that Mac runs docker images through a VM, thus its runs will be *significantly* slower.

Statically Resolvable Ambiguity (Artifact)

This artifact contains everything related to the Statically Resolvable Ambiguity paper:

- The mechanized proof of statically resolvable ambiguity.
- The modified versions of the OCaml compiler.
- The library implementing our grouper.
- The benchmarking script.
- The benchmarking data used in the paper.
- The Jupyter notebook analyzing the data and producing the plots.

These are available in two forms:

- As a docker image, with all of the above and all relevant executables and dependencies built and installed.
- As a compressed archive, only containing the source code for all of the above.

The latter is smaller and useful to just look at the code without downloading a very large docker image, but requires setting up an appropriate environment if you wish to run anything. We thus strongly recommend using the docker image if you wish to run any of the experiments or try out our modified OCaml compiler.

Kick the Tires - Basic Instructions for Setup

First, make sure you have Docker installed and configured (see docs.docker.com/get-docker/ for instructions).

Next, download the docker image and run the container:

```
# Note that depending on how your docker installation is configured
# you may need to add 'sudo' before all 'docker' commands.

# This first command may take a few minutes, the image is large
docker load --input static-resolvable.tar.gz

# The second command should be quick
docker run -p 8888:8888 -it --name static-resolvable-container static-resolvable
```

At this point you should have a shell open in the folder `/root` in the container. You can exit either using `Ctrl+D` or `exit`.

When you later want to resume the container you can use:

```
docker start -ia static-resolvable-container
```

The container starts with the shell in directory `/root`, which contains all relevant source code, previously produced benchmark data, and a `Makefile` to assist in running the various components. We suggest these commands be run during the "kick the tires" phase:

```
make coq            # Compile the coq proof, found under /root/coq,
                    # takes approx. 7min.

make coq-check      # Check that there are no 'admit's in the proofs

make ast-check      # Check that 'unamb' parses the same as 'original',
                    # takes approx. 2 minutes. Each equal parse prints
                    # a '.', each non-equal parse prints a '!'.

make bench-small    # Run a small subset of the benchmarks, takes approx. 9min.
                    # Output is placed in /root/data/data-log-small.csv, which
                    # will be read by a Jupyter notebook later. Progress will
                    # be printed in the form of a '.' for a successful parse
                    # and 'F' for a failed parse. We expect a number of parse
                    # failures, see reasoning in the Jupyter notebook later,
                    # under the heading 'Parse failures, totals'.

make jupyter        # Open the Jupyter notebook to generate and examine plots.
                    # Copy the link printed starting with '127.0.0.1' and open
                    # it in your web browser; the 'docker run' command above
                    # exposes the port outside the container. If the notebook
                    # does not open immediately, double-click 'Analysis.ipynb'
                    # in the file tree to the left.
```

The Jupyter notebook contains further instructions on how to run it and generate the plots. To close the notebook either hit `Ctrl+C` in the terminal followed by `y`, or select `File > Shut Down` in the notebook in the browser.

NOTE FOR ARTIFACT REVIEWERS: One of the changes requested by the paper reviewers was to include benchmarks that directly compare parse-times, instead of end-to-end compile times, as well as include a fix to a performance bug we discovered between the initial submission and the author response. This artifact thus contains plots and data that is markedly different from the paper. We discuss this in more detail in section "Connecting Paper Claims and the Artifact".

Claims: Formalism and Mechanization

This section explains how the formalism presented in the paper corresponds to the mechanization. Unless otherwise noted, all definitions are found in the file `New.v`.

Section 3

Section 3 introduces the syntax and semantics of the formalism. The corresponding mechanization is mostly straightforward.

Figure 7

Syntax

Paper	Coq
Op	`op_label : Type`
π	`par_label : op_label`
g	`n : node` where `is_grouped n`

Paper	Coq
p	`n : node` where `is_grouped_or_par n`
h	No exact correspondence (see below)
e	`n : node` where `is_node n`
$\langle \square\ o]$	`l o : node -> node` where `lopen_ctx l`
$[o\ \square\ \rangle$	`r o : node -> node` where `ropen_ctx r`
$-$, $\leftarrow$, $\rightarrow$, $\leftrightarrow$	`DNone`, `DL`, `DR`, `DLR` : dirs

The semantics is parameterized over a type `op_label` (corresponding to Op) containing at least a value `par_label` (corresponding to π). This type is the allowed set of labels o of nodes.

The syntax for expressions is represented as a single type `node`, corresponding to e extended with holes `_`. We use different predicates to distinguish between the different kinds of expressions, e.g., `is_node` for nodes in e and `is_grouped` for nodes in g. Note that there is no predicate for nodes in h; instead it is baked into the definition of `is_node`.

Left and right-focused frames are represented as functions of type `op_label -> node -> node` whose structure is constrained by the predicates `lopen_ctx` and `ropen_ctx` respectively to only allow producing the kinds of nodes expected by the syntax. For example, the rule `NLEdge` steps `r o NHole` to `r o g` (where `ropen_ctx r`), just as the rule L-Edge in the paper steps $[o\ \_\ \rangle$ to $[o\ g\rangle$.

The representation of directions in `dirs` (in `Old.v`) is straightforward.

Helpers

Paper	Coq
op	`op : node -> op_label`
bind	`precedence : op_label -> op_label -> dirs`
$o_1 \in A_{\swarrow}(o_2)$	`allowed_left o2 o1 : bool`
$o_1 \in A_{\searrow}(o_2)$	`allowed_right o2 o1 : bool`
$o \in A_{root}$	`allowed_top o : bool`
groupings	`gallowed : op_label -> op_label -> op_label -> dirs`

The semantics is parameterized over a function `precedence`, corresponding to the *bind* function, and functions to `bool` `allowed_left`, `allowed_right` and `allowed_top`, corresponding to the sets $A_{\swarrow}(o)$, $A_{\searrow}(o)$ (for $o \in Op$) and A_{root}.

A minor difference is that the mechanized `op` is defined for holes, which *op* in the paper is not. This is not an issue as `op` is only ever used on nodes g (for which `is_grouped` hold) or p (for which `is_grouped_or_par` hold), which excludes holes.

The function `gallowed` directly mirrors the definition of *groupings*.

Semantics

Paper	Coq
$S \vdash \bar{e} \Rightarrow g$	`GSteps (fun _ => true) allowed es \[g]`
$\bar{e}_1 \rightarrow \bar{e}_2$	`LRStep (fun _ => true) es1 es2`

The operational semantics of grouping is defined by the mutually recursive relations `GSteps`, corre-

sponding to the relation $S \vdash \bar{e} \Rightarrow g$, and `LRStep`, corresponding to the relation $\bar{e}_1 \rightarrow \bar{e}_2$. Both relations are parameterized over a predicate `Pre` which must hold for the sequence of expressions to the left of the expressions being grouped (used later in the definition of left-most derivations). As the rules in Figure 7 have no such requirements, `Pre` can be chosen as `(fun _ => true)`. The relation `NGSteps` further takes a function of type `op_label -> bool` which corresponds to the set S of allowed labels in the two top rules of Figure 7.

The helpers `dir_includes_left` and `dir_includes_right` are used to check if a direction is in the set $\{\leftarrow, \leftrightarrow\}$ or respectively $\{\rightarrow, \leftrightarrow\}$.

The running example

This example (found in Figures 3, 5, and 6, plus the first derivation in Section 3) is mechanized in `Example.v`. We also show that the grouping assumption holds (`Example grouping_assumption_holds`).

Definition 3.1

Definition 3.1 is mechanized as `GroupingAssumption`.

Section 4

Section 4 presents the proof of statically resolvable ambiguity, together with the necessary definitions. Parts of the mechanized proof was done for an older version of the semantics (in the file `Old.v`), but by proving equivalence of the two versions we are able to transfer the results to the semantics presented above. We start by listing the definitions and lemmas that use the new semantics, and then move to the old semantics.

Definitions and lemmas using the new semantics

Paper	Coq
"e is fully ungrouped"	`is_fully_open e`
$\Rightarrow_{LM}$	`GSteps (Forall is_ropen)`
$\rightarrow_{LM}$	`LRStep (Forall is_ropen)`
Lemma 4.6	`Theorem leftmost_to_gsteps` and `Theorem gsteps_to_leftmost`
Theorem 4.7	`Theorem static_resolvability`

(note that the mechanization uses `Theorem` throughout, even for the propositions called "lemma" in the paper)

The starting input to the grouping semantics is a sequence of nodes which are fully ungrouped, corresponding to `is_fully_open`. The relations for left-most groupings $S \vdash \bar{e} \Rightarrow_{LM} g$ and $\bar{e}_1 \rightarrow_{LM} \bar{e}_2$ are the same as the standard relations for grouping, with the difference that we require all expressions to the left of the expressions being grouped to be right-focused frames with holes. In the mechanization we reuse the `GSteps` and `LRStep` relations but set the predicate `Pre` as `Forall is_ropen` (remember that `Pre` holds for the sequence of expressions to the left of the expressions being grouped).

The equivalence between left-most and regular grouping is divided into two lemmas, one for each direction of the equivalence. The main theorem is stated in `static_resolvability`, which uses the helper predicate `Resolvable` (defined in `Resolvable.v`) as explained in the paper.

Equivalence between new and old semantics

Paper	Coq
"node"	`prec_node`
"a node with label o has children in directions $\bar{d}$"	`edges o = ds`
translation from old to new	`old_to_new : prec_node -> node`
translation from new to old	`new_to_old : node -> prec_node`
well-formedness of old nodes	`old_wf : prec_node -> Prop`
equivalence of old and new semantics	`Theorem steps_equiv`

The nodes in the old semantics is represented by the datatype `prec_node`. The main difference from `node` is that there is no explicit representation of holes. Instead, the old semantics is parameterized by a function `edges : op_label -> dirs` denoting in which directions a node with a given label has children. This means that a node with holes to the left and right is represented as a node without children but with specified `edges` to the left and right.

In showing the equivalence between the two semantics, we use translation functions `old_to_new` and `new_to_old` (lifted to sequences of nodes in `olds_to_news` and `news_to_olds`). Because it is possible to represent an ill-formed `prec_node`, e.g. by creating a node with a left child when `edges` only specifies a child to the right, we define well-formedness of nodes in the old semantics in the property `old_wf`. Note that it is not possible to represent an ill-formed node in the new semantics.

There are relations similar to `GSteps` and `LRStep` in the old semantics (called `GStep` and `LRStep`), which are also parameterized over a predicate `Pre` to allow expressing left-most groupings in the same relation. In order to prove static resolvability, we only need to connect the new semantics to the left-most grouping semantics of the old semantics. This is done in the theorem `steps_equiv`, which states that given well-formed `prec_node`s `ns` and `n`, the old left-most grouping semantics steps `ns` to `n` if and only if the new grouping semantics steps `olds_to_news ns` to `old_to_new n`.

Definitions and lemmas using the old semantics

(These definitions are all in `Old.v`)

Paper	Coq
`flatten`	`fullpar_flatten : prec_node -> list prec_node`
"$\bar{e}$ is fully ungrouped"	`Forall FullyOpen es`
parform($\bar{e}$)	`FullParForm allowed es`, where `allowed : op_label -> bool`
"`flatten` gives a fully ungrouped sequence"	`Theorem fullpar_flatten_open`
Lemma 4.1	`Theorem flatten_with_restriction_form`
Lemma 4.2	`Theorem flatten_with_restriction_preserves_steps`
Lemma 4.3	`Theorem lrstep_preserves_fullparform`
Lemma 4.4	Implied by `Theorem fullparform_unamb_step`
Lemma 4.5	`Theorem fullparform_unamb`

A fully grouped expression can be flattened (`fullpar_flatten`) to produce a sequence in *parform*. The mechanized variant `FullParForm` further takes a set (function to `bool`) which is used internally to ensure that nodes only appear where they are allowed to appear, in particular ensuring that we only put parentheses where they are allowed.

The lemmas from the paper are mostly direct translations. Lemma 4.4 does not have a direct

equivalent, but is proved as part of the helper lemma `fullparform_unamb_step`. Note that `flatten_with_restriction_preserves_steps` is written using a general form that allows proving it for both the left-most and the regular grouping semantics.

Claims: Benchmarks

The artifact contains scripts for two styles of benchmarks: those used by the revised version of the paper, as well as those used for the initial submission of the paper, for completeness.

New Benchmarks

To run the new benchmarks and look at the generated plots:

```
make bench    # This takes a *long* time, approx. 2 days and 10h
              # on the testing machine. However, most packages
              # produce very similar results, thus running
              # overnight and then interrupting with 'Ctrl+C'
              # should give a reasonable dataset for comparison.

make jupyter  # Copy the link beginning with '127.0.0.1' and
              # open it in your web browser, then double-click
              # 'Analysis.ipynb' and follow the instructions
```

Figure 10 can be seen as the final cell in the notebook, the data for Table 1 is found in the output of the first cell under "Analysis and Plots". Other graphs and tables in the notebook provide additional views of the data, some of which are referenced in the body text of the evaluation.

Old Benchmarks

To run the old benchmarks and look at the generated plots:

```
make bench-old  # This takes a *long* time, the data used in the
                # paper came from running the benchmarks for
                # approx 17.5h, then interrupting with 'Ctrl+C'.

make jupyter    # Copy the link beginning with '127.0.0.1' and
                # open it in your web browser, then double-click
                # 'OldAnalysis.ipynb' and follow the instructions.
```

Each plot and datapoint used in the paper has its own cell in the notebook.

Claims: Accuracy of `unamb`

This can be checked by running the following:

```
make ast-check
```

This parses each `.ml` and `.mli` file in the OCaml compiler repository, comparing the debug parse tree output (via `-dparsetree`) with both `original` and `unamb`, comparing the output.

The summary at the end of the output should be as follows (whitespace added for clarity in the readme):

```
# Done
Total files:     2286
Differing files:
```

```
testsuite/tests/exotic-syntax/exotic.ml
testsuite/tests/parsetree/locations_test.ml
```

Inside the Container

This section gives general information about the docker image, along with how to run benchmarks on a different set of packages and how to run the various OCaml compiler versions.

Make Targets

The makefile in the container (`/root/Makefile`) contains a number of targets for common tasks:

- `make coq`, build the Coq project.
- `make coq-check`, check that there are no admits in the Coq project.
- `make ast-check`, check that `unamb` agrees with `original` on how to parse the files in the OCaml repository.
- `make bench`, run the benchmarks. This takes a *long* time (approx. 2 days and 10 hours for 2125 packages on the test machine) and gives its primary output in `data/data-log.csv`. Additional logging information is also output and can be found in `data/opam-log.txt` (Opam output from downloading the packages) and `data/build-log.txt` (Output by the compilers when parsing). Aborting benchmarking part-way through can be done with an interrupt (`Ctrl+C`). The data produced is written continuously, thus the plots can still be produced in this case, albeit with less data.
- `make bench-small` is a smaller version of `make bench`; it only benchmarks the 10 most used Opam packages. Data is output to a different set of files, e.g., `data/data-log-small.csv` instead of `data/data-log.csv` and `data/opam-log-small.txt` instead of `data/opam-log.txt`.
- `make bench-old`, run benchmarks in the style used for the submitted version of the paper.
- `make jupyter` starts Jupyter with the notebook used to generate the plots in the paper. The notebook also contains code to plot new data produced by `make bench` and `make bench-small`, as well as instructions on how to update it.
- `make original-switch`, `make unamb-switch`, `make ambif-switch`. These targets rebuild each of the three OCaml switches. Each of these take quite a while (~40min to run all three).

Contents

The `/root` directory (which the container starts in) contains the following:

- `README.md` and`README.pdf`, this document.
- `Makefile`, which contains targets for the most common tasks we expect.
- `coq/`, which contains the mechanized proof.
- `ocaml/`, the source code with our modified OCaml compiler.
  - The four versions in the paper can be accessed as `git` tags (after `cd ocaml`):
    - `original` (the unmodified compiler) can be seen with `git checkout broken-original`.
    - `unamb` (our version without ambiguity) can be seen with `git checkout broken-unamb`.
    - `ambif` (our version with dangling else) can be seen with `git checkout broken-ambif`.
    - `ambmatch` (our version with dangling else and nested match ambiguity) can be seen with `git checkout broken-ambmatch`. Note that this version does not bootstrap, and is thus not available as an Opam switch, due to the large number of nested matches in the source code.
  - The majority of our changes are in `ocaml/parsing/parser.mly`, i.e., the Menhir grammar defining the parser.
- `miking/` contains the source code of the Miking compiler and standard library. Our grouper is implemented as a library in Miking which we then compile to OCaml code. The library itself can

be found in `miking/stdlib/parser/breakable.mc`. Everything else under `miking/` is included for completeness sake, but is otherwise not relevant; it is merely a dependency.
- `breakable-ml/` contains the code wrapping the `breakable.mc` library as an OCaml module. In the docker image the OCaml library has already been built and can be found in `ocaml/parsing/brokensyntax.ml` and `ocaml/parsing/brokensyntax.mli` (when one of the modified `ocaml/` versions is checked out).
- `Analysis.ipynb`, the Jupyter notebook responsible for the analysis and plots intended for the revised version of the paper.
- `data/` contains the data from our most recent benchmarks for the plots, as well as any data produced by new benchmark runs.
- `single-bench.fish` is the script that runs the benchmarks intended for the revised version.
- `bench.fish` is the script that runs the benchmarks for the submitted version of the paper.
- `check-asts.fish` compares the parse results for `original` and `unamb`.
- `sorted-deps` is a list of Opam packages to use in the benchmarks, sorted by number of transitive dependents, i.e., the most used packages are first.
- `examples/`, a few OCaml example programs demonstrating ambiguities.

Running the Modified OCaml Compiler

The modified versions of the OCaml compiler are already built and available as Opam switches. To switch between them, run:

```
# To use the unmodified OCaml compiler
eval (opam env --switch=broken-original --set-switch)

# To use our unambiguous version
eval (opam env --switch=broken-unamb --set-switch)

# To use our ambiguous version with dangling else
eval (opam env --switch=broken-ambif --set-switch)
```

After that you can use the various tools as normal, for example:

```
# Start a REPL
ocaml

# Run a file
ocaml examples/dangling.ml
```

In particular, to see an example of an ambiguity error, run the following:

```
eval (opam env --switch=broken-ambif --set-switch)
ocaml examples/dangling.ml
```

This is the expected output:

```
File "./examples/dangling.ml", lines 5-9, characters 2-31:
5 | ..if public then
6 |     if color = "red" then
7 |       print_endline "It's red"
8 |   else
9 |     print_endline "It's secret"
Error: The program is ambiguous:

if public then
```

```
    ( if color = "red" then
      print_endline "It's red"
  else
    print_endline "It's secret" )
if public then
    ( if color = "red" then
      print_endline "It's red" )
  else
    print_endline "It's secret"
```

Note that the actual output will have some coloring not visible above; parentheses to be added are shown in red, syntax that is relevant to the ambiguity is shown in white, and everything else is unhighlighted.

Benchmarking Other Packages

The benchmarking script (`single-bench.fish`) takes a list of Opam packages for which to benchmark parsing on STDIN, thus it is easy to run with a different dataset. For example, if `input.txt` has the following contents:

```
dune
result
odoc
```

. . . then we can run the benchmarks parsing all files in the packages `dune`, `result`, and `odoc` as follows:

```
./single-bench.fish data/opam-log-custom.txt data/build-log-custom.txt \
  data/data-log-custom.csv < input.txt
```

The three arguments designate output files. All three are mandatory. The first two are for debugging, the third (`data/data-log-custom.csv`) contains the actual data. To tell the Jupyter notebook about this new dataset, follow the instructions in the notebook itself, under “Data Sources and Pivot Tables”.