SyMP: The Hacker's Manual

Sergey Berezin and Alex Groce

May 12, 2001

Abstract

This manual is a high level description of the SyMP implementation. The algorithms are presented in the form of inference rules.

Contents

1 Configuration Mechanism

SyMP (Symbolic Model Prover) is a very modular system. The main components are the prover management core, interactive user interface (abstract and concrete layers), and proof system modules. The prover manager and the abstract user interface comprise the core of the system, and concrete user interfaces and proof systems are add-on modules.

To facilitate implementation of extensions in SyMP, we provide a flexible configuration mechanism for adding new proof system modules and encourage the developers to follow the guidelines described below.

1.1 Top level configuration.

SyMP uses *autoconf* and *GNU make* to configure, compile and install the tool. All these steps should be as automatic and transparent to the end user as possible, and follow the standard pattern used by most of GNU software, which consists of the three automagical steps:

```
./configure
make
make install
```

Since the tool can include new proof system modules, the `configure` script should have a way to find them and configure properly. We use a special configuration file `proof-systems.conf` on the top level that lists the names of the proof systems you want to configure and compile in, and their respective directories. The file is in the `name = directory` format (one line per entry). The name can be any alpha-numerical string with no spaces, starting with a letter, and must be different from the other proof systems' names. The name is case-sensitive. The directory can be relative to the current directory, or it can be an absolute path. The former is always preferred, but if you are building a module outside of the main source tree, the latter might be more convenient to you.

The file *`proof-systems.conf` is the only file you need to modify on the top level to add your own proof system module. In theory, you should* **not** *need to touch other files unless you know what you are doing.* All the rest of the configuration information must be provided in the proof system's directory. If you comply, this will ensure that your proof system will peacefully co-exist with other proof systems and can later be included in the main SyMP distribution. Also, this will make it easier for you to upgrade the SyMP core to newer versions with minimum hassle. However, if you really want some to add a new feature to the core, you are more than welcome to discuss it with the main developers.

Now, some notes on what happens in the `configure` script and at `make` time. Again, normally *we do not expect you to change anything in these scripts, this information is for your reference only*. First, the `configure` script scans `proof-systems.conf` for the names and directories of the proof systems to configure, looks into these directories for additional configuration information (lists of various source and target files), and builds the master `Makefile`. Then, at `make` time, the master `sources.cm` file is generated which includes all the ML sources from all the configured proof systems, and `provers-join.sml` is generated from the template `provers-join.sml.in`. This file instantiates and connects the proof systems with the common layer of the abstract user interface. In addition, a pair of files `sequent-common.{sig,sml}` is created in *each of the proof system's directory* customised to each of the proof system, as well as the top-level pair. The customized versions are almost identical to the top-level one, except that the `SequentBase` structure is exported with the `SEQUENT_BASE_`*Name* signature, where *Name* is the name of the proof system. This allows you to design a custom sequent type that would be visible to your implementation through the structure with common definitions `sequent-common.sml`, and at the same time, it will be guaranteed to match the corresponding top-level signature.[1] The names for the proof system structures are derived from the proof systems' respective names in the `proof-systems.conf` file, and that's why the names must be alpha-numeric and different from each other.

1.2 Other `make`-ables.

To compile documentation from source, use `make doc`. This will also recurse into all the proof system subdirectories and compile local documentation for each module. In the "all source" distribution, this rule is executed automatically with `make install`. In the "standard" distribution (where documentation is already compiled) `make doc` does nothing.

`make tar` produces a tarball of "normal" source distribution with compiled documentation. It automatically runs `make doc` if necessary. This is a good way to pack a distribution for end users. Note, that since documentation sources are not included in the tarball, it is a good idea to edit the `Makefile.in` file to remove the dependency of the `make doc` rule on the sources and delete the body of the rule. If you need to make an "all source" distribution, including the documentation sources, issue `make tarsrc` command instead.

`make clean` deletes most of the automatically generated files, but leaves the compiled ML core and, possibly, some other junk like backup files. To clean up the distribution from those files, use `make veryclean`.

1.3 Proof system local configuration.

Every proof system is placed in its own separate directory. It does not matter whether the directory is a part of the main source tree or not. Each such a directory must contain 3 configuration files: `proof-system.conf`, `Makefile`, and `sources.cm`. For concrete examples of these files, look into the default proof system's directory in the distribution.

The `proof-system.conf` file defines the way the proof system is instantiated and referred to in the prover core. The file is in the `key = value` format, no line breaks are allowed in the definitions. Currently, it must define two keys: `functor` and `name`. The `functor` value is a full instantiation of the functor that implements the `proof-system.sig` interface, as it should appear in SML code. The `name` value is a string that uniquely identifies the proof system for the user interface; for example, emacs interface passes this string to SyMP to identify which proof system should be used.

`Makefile` is a usual GNU make specification file. The minimal `Makefile` must contain two rules: `all` and `doc`, for compiling the proof system and its documentation respectively. Note, that the `all` rule does not need to compile any ML files; it will be done on the top level (since the proof system depends on some top-level SyMP libraries). It is used only to generate all necessary components that CM (ML compilation manager) will not be able to. This `Makefile` file will be scanned by the top-level `configure` script and checked for the values of the variables listed in Figure 1. These values will then be included in the master `Makefile` generated by the `configure` script.

The `sources.cm` file defines a *group* of (local) ML files that ML compilation manager has to know about. This file will be included as a subgroup to the master `sources.cm` file. According to the CM rules, a subgroup can export its top-level declarations to the supergroup (the *client*), but cannot import anything from it. In particular, it means that you cannot use any top-level declarations from other proof systems or the SyMP core, unless you explicitly include the appropriate `*.cm` files as your own subgroups. Since there are some libraries you want to include from the system core, their interfaces are moved to the separate top-level `common.cm` file that any proof system can (and, in fact, has to) include. All the rest of the proof system can be organized in any way you like, as long as it is correctly reflected by the configuration files.

In theory, you can also include the `source.cm` file(s) of other proof system(s) and it will work as you expect as long as all the imported proof systems are compiled in together with yours. The system may also compile even if you do not configure the imported proof systems, however, the master `Makefile` will not be aware of your hack, and will not even know about the existence of those proof systems, which may result in (un)expected surprises. This would also tie your proof system to other external modules that you might not have control of, which in turn reduces

[1] This is all really a hack around the restriction of SML/NJ on definitional structure sharing and absence of higher-order functors.

MLSRC	The list of source files relevant to ML code. These include all the `*.sml` and `*.sig` files, ML-lex and ML-yacc sources, and other source files from which ML files are generated. In particular, the local `sources.cm` file should be there.
ELISPSRC	The list of `*.el` files used by the emacs user interface. Normally, it will be one file that defines the major mode for the proof system's input language. See `emacs-interface/elisp/symp-mode.el` for an example.
DOCSRC	The list of local documentation sources. These files are used to generate documentation (with `make doc`), but are not final documentation files themselves. They are not included in the tarball made with `make tar`.
OTHERSRC	The list of all the other source files. These could be sources of internally used executables, local configuration files, etc. In particular, the `Makefile` itself should be here.
MLGEN	The list of files relevant to ML code that are generated automatically; These files will be deleted with `make clean`; do not list any source files here.
CMGEN	The list of files generated by the compilation manager. In particular, `CM` (the directory with bytecode files) should go here. These files will be deleted with `make clean`.
OTHERGEN	The list of all the other automatically generated files (local object files and executables, generated documentation files, etc.). These files will be deleted with `make clean` and are not included in the tarball made with `make tar`.
MANPAGES	The list of files in the UNIX manpage format. These will be installed in a directory where the `man` program can find them, and also included in the tarball made with `make tar`.
DOCS	The list of all the other documentation files in their final format. These may be source files (like `README` file), or Postscript or HTML files generated from other sources. These files are installed in the documentation directory and included in the tarball made with `make tar`. Note, that `make clean` will not erase them, and if you want to erase some of them, add those to the `OTHERGEN` list as well.
INTERNALEXE	The list of local executables (binaries and scripts). These files will be installed in the SyMP installation directory and will be accessible to the proof system from the executable path. They will not be installed as global executables, so even if your OS already has a global executable with a conflicting name, the Right Thing should happen and everything should still work.

Figure 1: Local `Makefile` variables for a proof system module.

robustness and portability of your code. In short, try to make your proof system as independent as possible, rely only on the modules mentioned in `common.cm`, and your live will be happier ever after.

2 Proof Management Core

The module structure of the prover is summarized in Figure 2, and Figure 3 illustrates the way these modules interconnect. The prover is designed in as general way as we could think of, and has a way to plug in different proof systems with custom sequents and input languages. This is achieved by stacking a number of layers of modules that implement different parts of the prover's functionality. These modules are parameterized by the modules of the upper layers with a fixed API, which makes the implementation flexible at any level in the hierarchy.

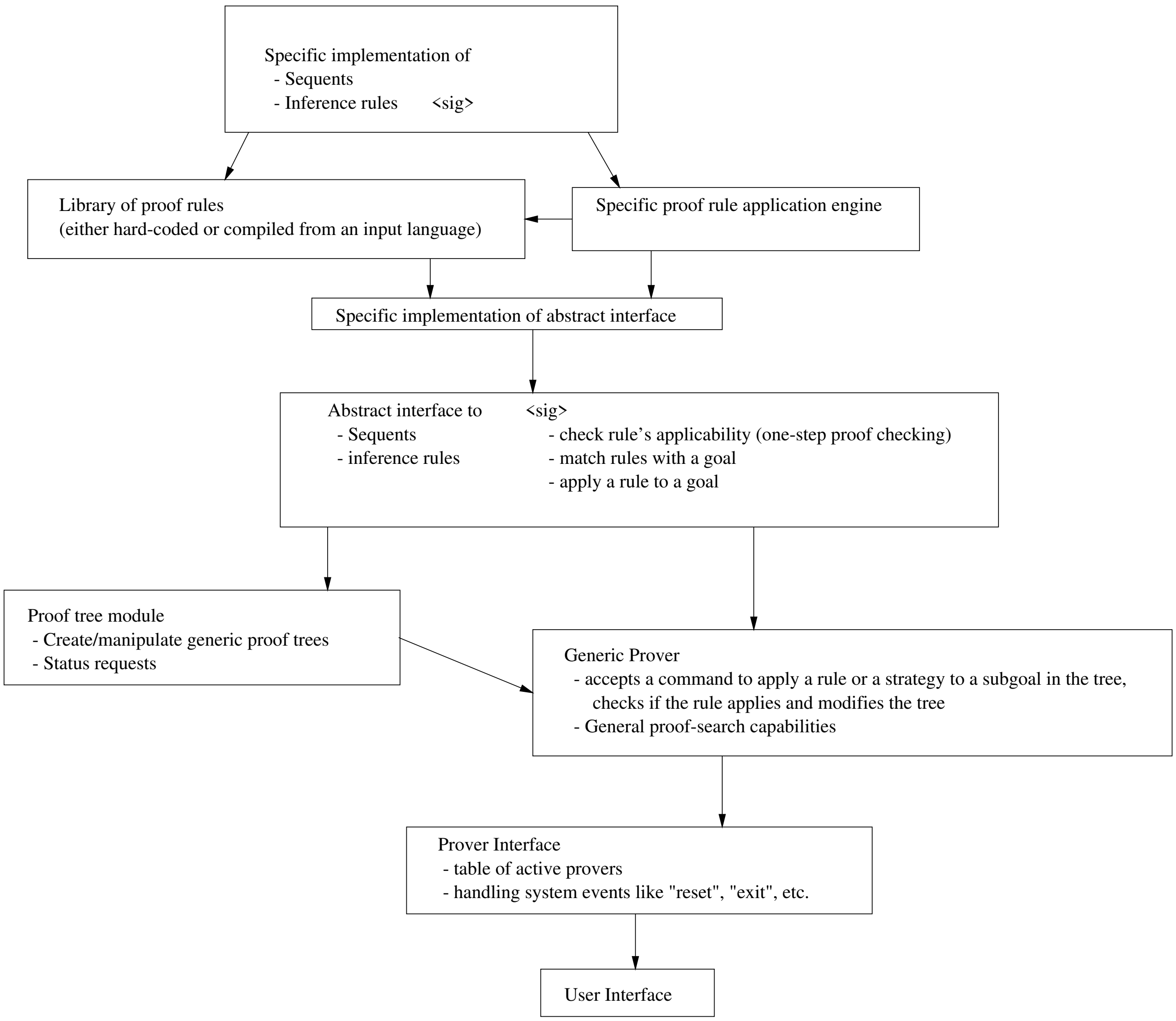

Figure 2: The prover modules.

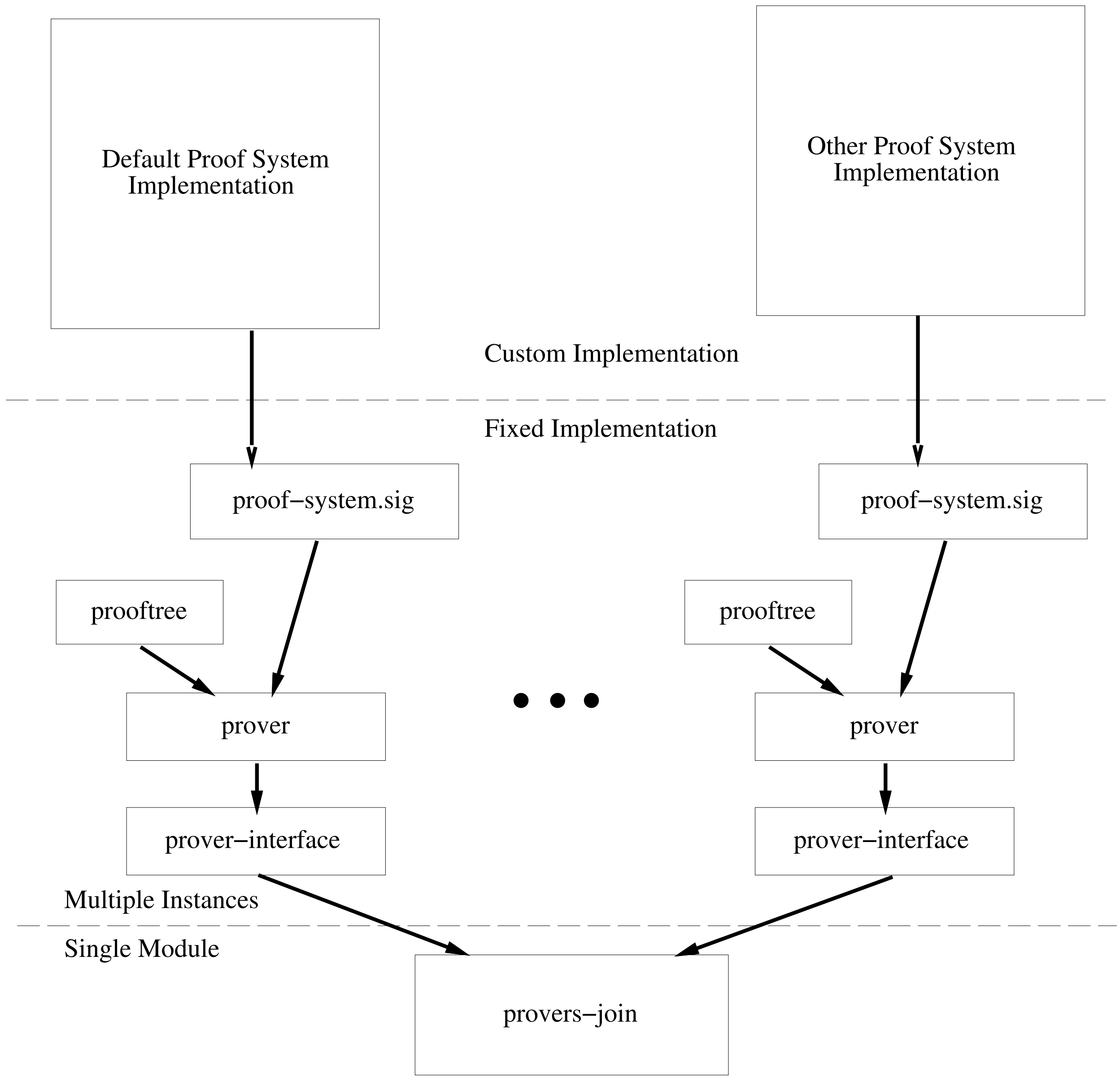

Figure 3: The (simplified) structure of the instances of the prover modules.

2.1 The Prover Modules Hierarchy

2.1.1 Common modules with fixed implementation

The user interface for the prover is implemented in the `interact` module. It initializes the rest of the prover and starts a read-evaluate loop for reading user commands and passing them to the `provers-join` module.

The `provers-join` module creates all the different types of the provers (instances of `prover-interface` that depend on different implementations of the `proof-system` module) and unifies their interfaces. It basically makes up for the lack of object-orientedness in SML/NJ.

The root module of each prover is the `prover-interface`. It's a single parameterized implementation of a few high-level functions that create a context (or *typecheck a program*), start up a new prover for a theorem, check the status of currently running provers, etc. This is the module that holds the state of all active provers.

Each prover is implemented by the `prover` module parameterized by the `proof-system`. It provides the interface to the proof tree manipulation, rule matching and application, and other related activities for a specific prover. The proof tree is an abstract data type implemented in the `prooftree` module and parameterized by the `sequent` module (which, naturally, implements the representation of the sequent and inference rules).

2.1.2 Abstract interfaces to custom modules

All the modules above have a single implementation, and are only instantiated multiple times with different module parameters. The rest of the modules are specific to a particular proof system, and only some of them have to conform to a particular signature. We describe these interfaces (signatures) in this subsection.

The `proof-system` signature provides the interface to the set of inference rules with an option to add a new rule and search the existing rules by name. It also encapsulates the `sequent` module.

The `sequent` interface provides the `input-language` module and comprises the abstract types for the sequent and the inference rule format with functions to manipulate them (construct a new sequent, match or apply a rule, etc.).

The `input-language` signature provides the abstract interface to parsing and typechecking routines for an input language together with the abstract program object representation.

These three signatures are the only means to communicate to the custom input language and proof system modules, so any language or proof system that can be meaningfully implemented within these interfaces can be (in theory) plugged into the SyMP prover and be ready for use. SyMP has it's "native," or *default* proof system and input language, described in Section 4.

2.2 Files: Saving Theorems and Other Data

At this point, we assume that each verification problem is specified as a single program, usually in a single file. The file extension gives a hint which proof system to use for that program (`.symp` is for the default proof system), but any other extensions are allowed. In the default input language, one can split the program in several files and join them together with `include` operator; the result, however, is still a single monolithic program, where all `include` operators are textually expanded into the main file before typechecking. "Separate compilation" could, however, be possible in other proof systems and languages.

2.2.1 Status file

Each program in SyMP has a single status file whose name is constructed by appending `.status` extension to the main program file. For example, if the main file is `mutex.symp`, its status file is `mutex.symp.status`. The status file keeps the global information about which theorems have what state (proven, unchecked, unfinished, etc.), and which file holds the proof for each theorem. If a theorem is not mentioned, it does not have any proof at all.

The format of this file, in general, is:

```
[<SECTION>]
  <field> = <value>
  <field> = <value>
```

```
   ...
[<SECTION>]
   ...
```

Note: section names must be in square brackets. Currently, `<SECTION>` can only be `THEOREM`, and the fields are `name`, `status`, and `file`, with the corresponding values. The `<value>` is a string starting at the first non-white-space character after the = and extending to the end of line sans the white space at the end. The values can be enclosed in double quotes, and the double quotes in the value itself must be escaped with a backslash (`"string \"Doo-bee-doo\""`). However, such enclosure is only needed if the value must contain spaces at the beginning or end, or have special characters like new line (`"\n"`).

Any line that starts with # is ignored (it is a comment) to the end of line. However, this file is supposed to be read and written automatically, so the comments are pretty useless, as they are erased every time the status file is updated.

2.2.2 Proof file

This file contains a proof of a theorem, and usually has an extension `.prf`, however, any extension is allowed. The file consists of entries of the following form:

```
(<subgoal> rule(<arguments>) (<subgoal>["hints"] ... <subgoal>["hints"]) )
```

This means, the subgoal labeled `<subgoal>` has an optional inference rule applied to it, which results in (again optional) list of new subgoals with optional hints (in double quotes). The entire entry must be enclosed in parentheses. The first entry in the file is the root subgoal (the original theorem).

The subgoals in the proof are identified only by their labels (type SubgoalID in ML), and do not carry any information about the actual sequents or even the theorem for which this proof is constructed; so any other theorem can reuse the proof.

Note, that this representation allows the proof to be an arbitrary graph; the tool, however, insures that the graph is acyclic after reading the proof.

3 Interactive User Interface

3.1 Abstract User Interface

SyMP by itself does not have any user interface (except for the primitive non-interactive command line one for batch mode operation which currently doesn't work). Normally, it runs in a *server mode* and communicates with an external user interface using a special communication protocol. The actual mechanism of this communication and the low-level representation of messages depends on a particular interface (e.g. emacs uses UNIX pipes and passes the commands in LISP format). These formats are defined in the `interface` module together with the functions for passing messages from SyMP to the external interface. The other direction is implemented separately for each type of the interface. E.g. emacs interface is implemented in the `emacs-server-interface` module located in the `emacs-interface` directory.

However, the message exchange protocol for the user interface is uniform across all the interfaces. Messages are classified as *input* (from UI to SyMP, Figure 4) and *output* (from SyMP to UI, Figure 5) *commands*. An input command is generated by the interface in response to the user's actions. An ouput command is a response or a state change report from the tool, which the UI must somehow convey to the user.

The user interface should be as stateless as possible. That is, it should not assume anything about the state of the tool. Each message must be a complete transaction on its own, at least from the UI implementation prospective. This way one avoids problems with "unexpected" messages from the tool, or with the absence of "expected" messages.

4 Default Input Language and Proof System

The default proof system is located in the `default` directory. The modules implementing the required interfaces reside in files `xxx-default.sml`, where `xxx` is `input-language`, `sequent` and `proof-system`.

Command	Arguments	Action
`typecheck`	*file* [, *prover_type*]	Typecheck the given file containing *prover_type*'s input language. If *prover_type* is omitted, the default prover type is used.
`provetheorem`	*theorem* | (*theorem*, *module*) [, *name*]	Start up a new prover for the *theorem* located in the *module* or on the top-level, and optionally give this theorem a *name*.
`prover`	*proverID*, *proverCommand*, *other arguments*	Send the command (with arguments) to the prover *proverID*
`proverstatus`	*proverID*	Request the prover's status (the `proverstatus` response)
`options`, `displayoptions`	[*field* [= *new value*] [, *field* [= *new value*] . . .]]	Without args, ask the server to return all current options and their values (with the corresponding `options` or `displayoptions` response). If arguments are present, return the value of those fields only, or set them to the *new value*s.
`debugnames`	(no arguments)	Request all the available names that can be given to the `debug` option.
`version`		Ask to print the current version
`userchoice`	*id*, *choice*	Reply to the `userchoice` message from the server.
`cancelchoice`	*id*	The user wants to cancel the choice and its action.
`showstatus`	(no arguments)	Request the global SyMP status (as written in the status file)

Figure 4: External User Interface $\longrightarrow$ SyMP protocol

The input language module is a wrapper for the completely custom-made parser (`parser-default`, which depends on `symp-parsetree` and `symp.{yacc,lex}`), typechecker (`typecheck` module) and the static evaluator (`evaluate`).

The sequent module, as expected, implements the representation of a sequent and the inference rules, and the basic rule matching mechanism.

The `proof-rules` module is the place where all the proof rules are actually defined. Besides the sequent module, it also depends on the abstraction and transition relation engines (`abstract` and `trans` modules) and the back-end translator to SMV (`trans_SMV`).

Finally, the proof system module puts these components together and provides the global storage for the proof rules (implemented as a hash table).

4.1 Proof System

The sequent in the default proof system is

$$\mathcal{M}, \mathcal{I};\ \Sigma;\ \Gamma \vdash \Delta,$$

where $\mathcal{M}$ and $\mathcal{I}$ are the model and the initial states predicate ($\mathcal{I}$ is optional), and Σ, Γ, and Δ are lists of formulas that represent invariants, assumptions, and conclusions respectively. The actual rules for the proof system are described in a separate document [1]. We only give a representation and the matching mechanism for the rules.

Each rule is represented as a list of *templates* for the premisses and the conclusion. The conclusion's template is matched against the actual sequent, which generates a *substitution* for variables in the template, and applying it to the templates for premisses generates the new proof subgoals.

Not every rule can be defined as a set of templates (for instance, the *modelcheck* rule requires a good deal more of transformations), and general ML functions can be used to match the conclusion and generate new subgoals. Below we only discuss the template format.

Command	Arguments	Action
`newprover`	*proverID*, *name*	New prover *proverID* for the theorem named *name* has been created. In emacs, this should create a new buffer for interacting with this prover.
`oldprover`	*proverID*, *name*	An existing prover is asking for attention. This usually happens when the user wants to prove a theorem that already has an active prover. In emacs, this should display the buffer for that prover, or create a new buffer if it does not exist.
`proverstatus`	*proverID*, $status_1$, $status_2$, $\ldots$, where $status_i \in \{$init, active, busy, proven, saved$\}$.	Report the status of the prover. An absence of a status entry means its negation. No status info means the prover doesn't exist. saved means the curent proof is saved.
`prover`	*proverID*, *output*	Print output of a prover *proverID*.
`showproof`	*proverID*, *proof_tree*	Display the proof relevant to the *proverID*. The proof is represented in the abstract user interface syntax (`UserInterfaceArgument` type in ML), not just a string; so different interfaces can display it differently.
`verbose`	*string*	Usually a short one-line status message without trailing end-of-line. In emacs this will be displayed in the minibuffer.
`error`	*type*, *string*	Error message of a given *type* $\in$ $\{\mathrm{general}, \mathrm{bug}, \mathrm{parse}, \mathrm{type}, \mathrm{prover}, \ldots\}$. The message *string* can be several lines long. In emacs this will be displayed in a pop-up buffer.
`debug`	*string*	Debugging output. It can be quite long and scrolling really fast. Ideally, it should be directed to a specially designated place, like a separate buffer in emacs.
`userchoice`	*id*, *type*, *message*, $choice_1$, $choice_2$, $\ldots$, where *type*$\in\{$any, strict, file, yesno, confirm$\}$	Print *message* (a question) and read the user's response. *type* advises the UI on the type of request (e.g. for autocompletion). $choice_i$ are advisable choices that the user is required to choose from when *type* = strict. Only the first choice is used as a default for file type. Reply with a `userchoice` message with the same *id*.
`bell`	(no parameters)	Ring a bell to the user
`options`, `displayoptions`	*field* = *value*, ...	Return the current values of the options' *field*s and, in the second form, request the interface to display it to the user.
`debugnames`	$name_1$, $name_2$, $\ldots$	Send all the available names that can be given to the `debug` option.
`showstatus`	*string*	Return the global SyMP status as a string (as written in the status file)

Figure 5: SyMP $\longrightarrow$ External User Interface protocol

4.1.1 Templates, Types, and Values

A *template* T is defined as follows:

$T ::=$	P	Propositional template (formula)
$\mid$	$Pl(T)$	List of formulas matching an optional template T
$\mid$	e	object constant (expression or type matched verbatim)
$\mid$	x	object variable (any expression)
$\mid$	$xl(T)$	list of expressions with an optional template
$\mid$	_	wildcard, same as x, but without name
$\mid$	$(T_1, \ldots, T_n)$	tuple
$\mid$	$T_1\ T_2$	application
$\mid$	$\forall T_1, \ldots, T_n.\, T$	
$\mid$	$\exists T_1, \ldots, T_n.\, T$	quantifiers
$\mid$	$C[T]$	context template: matches any expression containing subexpressions matching T
$\mid$	$T_1 : T_2$	object matching T_1 is of type matching T_2. Expression types are just same objects as expressions as far as templates go.
$\mid$	$T :: \pi$	Typed template; this restricts what T can match, as opposed to matching an object's type
$\mid$	Sk	New Skolem constant (only for instantiation, doesn't match anything directly); can be restricted to a certain type by another template: Sk : T.

Templates and values can have the following *constraints*:

$\pi ::=$	$*$	any value
$\mid$	$\mathsf{formula}[\alpha]$	match a formula in the sequent; α is a formula spec that the user may give as a rule's argument (part of the sequent and/or formula number, which maybe a glob). For example, $\alpha = i$ means any formula from the invariant part of the sequent, $*$ is any formula, -3 is a specific assumption, $+$ and 1 are all and the first conclusion, resp., etc.
$\mid$	$\mathsf{bound}[T]$	a variable bound by a quantifier matching an optional template (variable type, name, etc.)
$\mid$	$\mathsf{obj}[e]$	object; that is, any expression or type from the input language, with an optional parameter - the object it should match verbatim
$\mid$	π list	list of elements constrained to π
$\mid$	$\mathsf{context}$	constraint for $C[T]$ template value: context expression containing a certain subexpression

A sequent template is defined as follows:

$$\mathrm{ST} ::= X_{\mathcal{M}} :: \mathsf{obj},\ X_{\mathcal{I}} :: \mathsf{obj};\ T_1^I, \ldots, T_n^I :: \mathsf{formula}[i];\ T_1^A, \ldots, T_k^A :: \mathsf{formula}[-] \vdash T_1^C, \ldots, T_k^C :: \mathsf{formula}[+]$$

Note, that some templates may match lists of formulas, so the number of matched formulas can be different from the number of templates in the list.

A substitution $\sigma : T \to v$ maps templates into *values*:

$v ::=$	$\mathcal{M}$	model
$\mid$	$\mathcal{I}$	initial state predicate for the model
$\mid$	e	an expression or type
$\mid$	$name = v$	named value
$\mid$	$[v_1, \ldots, v_n]$	list of values
$\mid$	S	subgoal ID
$\mid$	$\lambda x.\, v$	value of a context template $C[\cdot]$; β-reduced application of this term to another value v' gives a value of $C[v']$
$\mid$	$v :: \pi$	constrained value

4.1.2 Template substitution and matching

Matching a sequent S with a template ST means finding a substitution σ such that $\sigma(\mathrm{ST}) = S$. Template variables may have multiple occurrences in a template. If a variable is scalar (non-list), then the corresponding subexpression must be exactly the same for all instances. List variables add all matched objects into one list from all the occurrences, pairing each value with a template constraint if applicable (e.g. $\mathsf{formula}[\alpha]$ with the appropriate value of α when the same variable *Pl* occurs in several parts of the sequent). Instantiating such list variables also depends on the expected result type (constraint); again, for a *Pl* variable, only formulas of the right type are extracted into a particular part of a sequent.

Substitution of templates. We formalize matching and substitution by the set of proof rules. The sequents for applying a substitution to a template and a list of templates are

$$\sigma(T :: \pi) = v \text{ and } \sigma(\mathcal{T} :: \pi) = \ell,$$

and the sequent for matching a list of templates with a list of objects is

$$\mathcal{T} :: \pi \vdash \ell \rightsquigarrow \sigma,$$

where $\mathcal{T}$ is a list of templates, ℓ is a list of *annotated objects*, and σ is the substitution that $\sigma(\mathcal{T}) = \ell$. Objects are annotated by theil formula specs in the sequent α. We write $\sigma(T)$ (without the constraint specification) to mean $\sigma(T : *)$. The constraint π in the template matching sequent is a context restriction; for example, when we are matching formulas from the assumption part of the logic sequent, the context constraint is $\mathsf{formula}[-]$.

$$\frac{v :: \pi}{\sigma[P/v](P : \pi) = v} \qquad \frac{v :: \pi}{\sigma[X/v](X : \pi) = v} \qquad \frac{}{\sigma(C :: \mathsf{context}) = C} \qquad \frac{}{\sigma(\_) = v}$$

$$\frac{\ell' \subseteq \ell \ \text{ s.t. } \ \forall v \in \ell'.\, v :: \pi}{\sigma[xl/\ell](xl :: \pi\ \mathsf{list}) = \ell'} \qquad \frac{\ell' \subseteq \ell \ \text{ s.t. } \ \forall v \in \ell'.\, v :: \pi}{\sigma[Pl/\ell](Pl :: \pi\ \mathsf{list}) = \ell'}$$

$$\frac{\sigma(C :: \mathsf{context}) = v_C \quad \sigma(T) = v_T \quad v_T :: \pi}{\sigma(C[T] :: \pi) = (v_C[v_T])} \qquad \frac{\sigma(T) = \tau \quad v_i \text{ is a new Skolem constant}}{\sigma(\mathrm{Sk} : T) = v_i : \tau}$$

$$\frac{\pi \preceq \pi' \quad \sigma[P/v](P :: \pi') = v'}{\sigma[P/(v :: \pi')](P :: \pi) = v' :: \pi'} \qquad \frac{\pi = \min(\pi_1, \pi_2) \quad \sigma(T :: \pi) = v :: \pi}{\sigma((T :: \pi_1) :: \pi_2) = v}$$

Substitution of the compound templates:

$$\frac{\sigma(T_1, \ldots, T_n :: \pi\ \mathsf{list}) = v_1, \ldots, v_m}{\sigma((T_1, \ldots, T_n) :: \pi) = (v_1, \ldots, v_m)}\ \text{tuple} \qquad \frac{\sigma(T_1 :: \pi) = v_1 \quad \sigma(T_2 :: \pi) = v_2}{\sigma(T_1\ T_2 :: \pi) = v_1 v_2}\ \text{appl}$$

$$\frac{\sigma(\mathcal{T} :: \mathsf{bound\ list}) = \ell \quad \sigma(T :: \pi) = v}{\sigma(\forall \mathcal{T}.\, T :: \pi) = \forall \ell.\, v}\ \forall \qquad \frac{\sigma(\mathcal{T} :: \mathsf{bound\ list}) = \ell \quad \sigma(T :: \pi) = v}{\sigma(\exists \mathcal{T}.\, T :: \pi) = \forall \ell.\, v}\ \exists$$

Substitution of lists of templates: all lists are flattened (no direct nesting) and concatenated. Scalar values are considered as one-element lists. A constraint leaves only those elements in the list that satisfy it, the others are omitted.

$$\frac{\sigma(T_{i_j} :: \pi\ [\mathsf{list}]) = \ell_{i_j} \quad j = 1..k \text{ for all } i_j \in 1..n \text{ for which } \sigma(T_{i_j} :: \pi\ [\mathsf{list}]) \text{ is defined}}{\sigma(T_1, \ldots, T_n :: \pi\ \mathsf{list}) = \ell_{i_1}, \ldots, \ell_{i_k}}$$

Here $\pi\ [\mathsf{list}]$ means we choose $\pi\ \mathsf{list}$ for $T_{i_j} \in \{Pl, xl\}$, and just π otherwise. Finally, substitution for the entire sequent. We use a bit slopy notation here that does not distinguish between values and actual formulas in the sequent, but the implicit translation is obvious.

$$\frac{\sigma(X_{\mathcal{M}}) = \mathcal{M} \quad \sigma(X_{\mathcal{I}}) = \mathcal{I} \quad \sigma(\mathcal{T}^I :: \mathsf{formula}[i]\ \mathsf{list}) = \Sigma \quad \sigma(\mathcal{T}^A :: \mathsf{formula}[-]\ \mathsf{list}) = \Gamma \quad \sigma(\mathcal{T}^C :: \mathsf{formula}[+]\ \mathsf{list}) = \Delta}{\sigma(X_{\mathcal{M}}, X_{\mathcal{I}};\ \mathcal{T}^I;\ \mathcal{T}^A \vdash \mathcal{T}^C) = (\mathcal{M}, \mathcal{I});\ \Sigma;\ \Gamma \vdash \Delta}$$

The "subtyping" $\preceq$ is partial order relation defined as a set inclusion of the corresponding constraint values: the more objects may fall under a certain constraint, the "greater" it is in this preorder.

$$\frac{}{\pi \preceq *} \qquad \frac{}{\mathsf{formula}[\alpha] \preceq \mathsf{formula}} \qquad \frac{}{\mathsf{bound}[T] \preceq \mathsf{bound}} \qquad \frac{}{\mathsf{obj}[e] \preceq \mathsf{obj}}$$

$$\frac{\pi_1 \preceq \pi_2}{\pi_1\ \mathsf{list} \preceq \pi_2\ \mathsf{list}} \qquad \frac{\alpha_1 \preceq \alpha_2}{\mathsf{formula}[\alpha_1] \preceq \mathsf{formula}[\alpha_2]}$$

where $\alpha_1 \preceq \alpha_2$ is defined in the obvious way; the more concrete the specification is, the "less" it is in the order. Some specifications are incomparable, for instance, i and $+$. The meaning of $\min(\pi_1, \pi_2)$ is obvious, and it is a partial function. The rules that have it as a premiss are not applicable when it's undefined.

Template matching. When matching a list of templates with a list of objects, we'd ideally like to treat both lists as sets, and match first the strictest templates, and then the weakest. Since this problem might be too hard in general (it might be already NP-hard in our setting), we use a simpler and, perhaps, in some sense, a more user-friendly solution: *Consider templates in the list from left to right. Match the first template with everything it matches, and match the rest of the list with the rest of the objects.* In particular, if a scalar template occurs several times in different places, it should come before other templates in the list to be matched successfully.

When we merge two substitutions $\sigma_1 \circ \sigma_2$, the result must be consistent with the original substitutions; that is, if both σ_1 and σ_2 are defined on a template T, then $\sigma_1(T) = \sigma_2(T)$. If substitutions have conflicting values, their merge is undefined.

Matching scalar templates with a single expression:

$$\frac{e \text{ is boolean}}{P \vdash e :: \alpha \rightsquigarrow [P/e]} \qquad \frac{}{x \vdash e :: \alpha \rightsquigarrow [x/e]} \qquad \frac{}{\_ :: \pi \vdash e :: \alpha \rightsquigarrow []} \qquad \frac{e' = e \quad T \vdash e :: \alpha \rightsquigarrow \sigma}{T :: \mathsf{obj}[e'] \vdash e :: \alpha \rightsquigarrow \sigma}$$

$$\frac{}{P :: \mathsf{formula} \vdash e :: \alpha \rightsquigarrow [P/(e :: \mathsf{formula}[\alpha])]} \qquad \frac{e \text{ is bound var} \quad T \vdash e :: \alpha \rightsquigarrow \sigma}{x :: \mathsf{bound}[T] \vdash e :: \alpha \rightsquigarrow \sigma \circ [x/(e :: \mathsf{bound})]}$$

$$\frac{\pi = \min(\pi_1, \pi_2) \quad T :: \pi \vdash e :: \alpha \rightsquigarrow \sigma}{(T :: \pi_1) :: \pi_2 \vdash e :: \alpha \rightsquigarrow \sigma} \qquad \frac{T \notin \{Pl(T), xl\} \quad \alpha' \preceq \alpha \quad T :: \mathsf{formula} \vdash e :: \alpha \rightsquigarrow \sigma}{T :: \mathsf{formula}[\alpha'] \vdash e :: \alpha \rightsquigarrow \sigma}$$

$$\frac{\mathcal{T} :: \pi\ \mathsf{list} \vdash (e_1 :: \alpha), \ldots, (e_n :: \alpha) \rightsquigarrow \sigma}{(\mathcal{T}) :: \pi \vdash (e_1, \ldots, e_n) :: \alpha \rightsquigarrow \sigma}\ \text{tuple} \qquad \frac{T_1 :: \pi \vdash e_1 :: \alpha \rightsquigarrow \sigma_1 \quad T_2 :: \pi \vdash e_2 :: \alpha \rightsquigarrow \sigma_2}{T_1\, T_2 :: \pi \vdash (e_1\, e_2) :: \alpha \rightsquigarrow \sigma_1 \circ \sigma_2}\ \text{appl}$$

$$\frac{T_1 :: \pi \vdash e :: \alpha \rightsquigarrow \sigma_1 \quad T_2 :: \pi \vdash \tau :: \alpha \rightsquigarrow \sigma_2}{T_1 : T_2 :: \pi \vdash e : \tau :: \alpha \rightsquigarrow \sigma_1 \circ \sigma_2} \qquad \frac{T :: \mathsf{obj} \vdash e_T :: \alpha \rightsquigarrow \sigma \quad e'[x/e_T] = e \text{ for some } e', e_T}{C[T] :: \pi \vdash e :: \alpha \rightsquigarrow \sigma \circ [C/\lambda x.\, e']}$$

$$\frac{\mathcal{T} :: \mathsf{bound\ list} \vdash (e_1 :: \alpha), \ldots, (e_n :: \alpha) \rightsquigarrow \sigma_1 \quad T :: \pi \vdash e :: \alpha \rightsquigarrow \sigma_2}{\forall \mathcal{T}.\, T :: \pi \vdash \forall e_1, \ldots, e_n.\, e :: \alpha \rightsquigarrow \sigma_1 \circ \sigma_2}$$

$$\frac{\mathcal{T} :: \mathsf{bound\ list} \vdash (e_1 :: \alpha), \ldots, (e_n :: \alpha) \rightsquigarrow \sigma_1 \quad T :: \pi \vdash e :: \alpha \rightsquigarrow \sigma_2}{\exists \mathcal{T}.\, T :: \pi \vdash \exists e_1, \ldots, e_n.\, e :: \alpha \rightsquigarrow \sigma_1 \circ \sigma_2}$$

Matching the list of templates:

$$\frac{}{\cdot :: \pi \vdash \cdot \rightsquigarrow []}\ \text{init} \qquad \frac{T \notin \{Pl(T'), xl(T')\} \quad (e :: \alpha) :: \pi \quad T :: \pi \vdash e :: \alpha \rightsquigarrow \sigma_1 \quad \mathcal{T} :: \pi\ \mathsf{list} \vdash \ell \rightsquigarrow \sigma_2}{T, \mathcal{T} :: \pi\ \mathsf{list} \vdash (e :: \alpha), \ell \rightsquigarrow \sigma_1 \circ \sigma_2}$$

$$\frac{T \in \{Pl(T'), xl(T')\} \quad \ell_1 :: \pi\ \mathsf{list} \quad \forall e \in \ell_1.\, T :: \pi \vdash e \rightsquigarrow \sigma_0 \quad T :: \pi\ \mathsf{list} \vdash \ell_1 \rightsquigarrow \sigma_1 \quad \mathcal{T} :: \pi\ \mathsf{list} \vdash \ell_2 \rightsquigarrow \sigma_2}{T, \mathcal{T} :: \pi\ \mathsf{list} \vdash \ell_1, \ell_2 \rightsquigarrow \sigma_0 \circ \sigma_1 \circ \sigma_2}$$

Notice, that σ_0 is the same for all $e \in \ell_1$.

Matching Sequent Template We call scalar template variables that occur two or more times in a sequent template *complex variables* (they are "hard to match"), and all the others *simple variables*.

To match complex variables efficiently, we need a good way of finding identical subterms. We use an idea similar to the *indexing technique* from the automatic theorem proving community, which also closely resembles BED structure. Each (sub)term in any expression has a `TermID`; any term refers to its subterms by their `TermID`s, and any time we construct a new term, we compare it with existing terms, and if there is one already, we assign it the same `TermID`. It is very similar to BDDs, only in a more general setting. In addition to `TermID`, each term keeps the list of all its parents. So, the actual term is identified by its `TermID` and the *context* — the list of parents all the way up to the top-most term. The `TermID` and the context uniquely identify any subterm is the sequent, and at the same time equivalent subterms always have the same `TermID`.

The matching algorithm is now the following.

1. Convert the original sequent into the `TermID` representation;
2. For each complex template variable create a list of terms that appear in the same parts of the sequent as this variable, together with the corresponding contexts, and filter those that match the template;
3. Create a (possibly lazy) list of all combinations of $(\mathtt{TermID} * \mathrm{context}\ \mathsf{list})$ pairs from all the lists of step 2. No two variables can receive the same value;
4. Initialize all the complex variables with the first combination of values, remove it from the list, and match the rest of the template by the straightforward matching algorithm.
5. Repeat step 4 until the match is successful, or the list of value combinations is empty; in the latter case the match fails.

The straightforward matching algorithm matches the first template in the list with some object in the corresponding list, and removes this object from the list (so the other templates wouldn't match it).

4.2 Typechecker

4.3 Evaluator

The sequent for the evaluator is

$$e \Rightarrow e',$$

and stands for "expression e evaluates to e'.

DONE (I think...) *TODO: propagate $e.f$ and tuple and type constructor application extractors down into e as far as possible (push inside* **if***,* **case***, and other statements like that), so that we never have to deal with non-elementary expressions in these cases.*

4.4 Common Rules

Here we define some generally useful functions: extracting variables from an expression (sequent $V \vdash e$), a "loose" equality between two state variable expressions (upto non-value expressions in array indices, sequent $\gamma_1 \approx \gamma_2$), and a partial order on state variable expressions reflecting one variable being a part of the other (sequent $\gamma_1 \preceq \gamma_2$).

The sequent $V \vdash e$ extracts the set of state variables V that appear in the expression e. Most cases are obvious recursion into e, and then recursion into the types of state variables (δ). The expression is recursed all the way down to atomic state variables (i.e. declared with **StateVar**, but not necessarily of primitive types, sequent $\Sigma;\ V \vdash_1 e$), and these state variables are split into primitive ones if possible (sequent $V \vdash_2 \gamma$). No delayed COI expressions are generated here.

We denote an abstraction of a type τ by $\omega(\tau)$, and $\mathcal{D}_\tau$ is the set of all values of the type τ. In particular, $\mathcal{D}_{\omega(\tau)}$ is the set of values of the abstraction of τ.

$$\frac{\cdot;\, V \vdash_1 e}{V \vdash e} \qquad \frac{V \vdash_2 \mathcal{E}_1(\ldots\mathcal{E}_n(\delta)\ldots)}{\mathcal{E}_1,\,\ldots,\,\mathcal{E}_n;\, V \vdash_1 \delta}\ \text{StateVar} \qquad \frac{x \text{ is a constant}}{\Sigma;\, \cdot \vdash_1 x}\ \text{const}$$

$$\frac{\Sigma;\, V_i \vdash_1 e_i \quad i = 1..n}{\Sigma;\, \bigcup_i V_i \vdash_1 (e_1 \mid \ \ldots \mid e_n)}\ \text{nondet} \qquad \frac{\Sigma,\, \mathcal{ER}_f;\, V \vdash_1 e}{\Sigma;\, V \vdash_1 e.f}\ \text{Rec}E$$

$$\frac{\Sigma;\, V_i \vdash_1 e_i}{\Sigma,\, \mathcal{ER}_{f_i};\, V_i \vdash_1 \{\ldots,\, f_i = e_i,\, \ldots\}}\ \text{Rec}I \quad \frac{\cdot;\, V_i \vdash_1 e_i \quad i = 1..n}{\cdot;\, \bigcup_i V_i \vdash_1 \{f_1 = e_1,\, \ldots,\, f_n = e_n\}}\ \text{Rec}$$

$$\frac{\Sigma;\, V \vdash_1 e_i}{\Sigma,\, \mathcal{ER}_{f_i};\, V \vdash_1 e\, \mathbf{with}\, [\ldots,\, f_i = e_i,\, \ldots]}\ \mathbf{with}I_1 \qquad \frac{\Sigma;\, V \vdash_1 e \quad g \neq f_i \text{ for all } i}{\Sigma,\, \mathcal{ER}_g;\, V \vdash_1 e\, \mathbf{with}\, [f_1 = e_i,\, \ldots,\, f_n = e_n]}\ \mathbf{with}I_2$$

$$\frac{\cdot;\, V \vdash_1 e \quad \cdot;\, V_i \vdash_1 e_i \quad i = 1..n}{\cdot;\, V \cup \bigcup_i V_i \vdash_1 e\, \mathbf{with}\, [f_1 = e_i,\, \ldots,\, f_n = e_n]}\ \mathbf{with}$$

$$\frac{\Sigma,\, \mathcal{ET}_i;\, V \vdash_1 e}{\Sigma;\, V \vdash_1 \mathcal{ET}_i(e)}\ \text{Tuple}E \quad \frac{\Sigma;\, V_i \vdash_1 e_i}{\Sigma,\, \mathcal{ET}_i;\, V_i \vdash_1 (\ldots,\, e_i,\, \ldots)}\ \text{Tuple}I \quad \frac{\cdot;\, V_i \vdash_1 e_i \quad i = 0..n-1}{\cdot;\, \bigcup_i V_i \vdash_1 (e_0,\, \ldots,\, e_{n-1})}\ \text{Tuple}$$

$$\frac{\cdot;\, V_i^c \vdash_1 c_i \quad \Sigma;\, V_j \vdash_1 e_j \quad i = 1..n,\, j = 1..n+1}{\Sigma;\, \bigcup_i V_i^c \cup \bigcup_j V_j \vdash_1 \mathbf{if}\ c_1\ \mathbf{then}\ e_1\ \ldots\ \mathbf{else}\ e_{n+1}\ \mathbf{endif}}\ \mathbf{if}$$

$$\frac{\cdot;\, V \vdash_1 e \quad \Sigma;\, V_i \vdash_1 e_i \quad i = 1..n}{\Sigma;\, V \cup \bigcup_i V_i \vdash_1 \mathbf{case}\ e\ \mathbf{of}\ P_1 \Rightarrow e_1 \mid \ \ldots \mid P_n \Rightarrow e_n\ \mathbf{endcase}}\ \mathbf{case}$$

$$\frac{\Sigma;\, V \vdash_1 e}{\Sigma;\, V \vdash_1 \mathbf{let}\ D_1\ \ldots\ D_n\ \mathbf{in}\ e\ \mathbf{end}}\ \mathbf{let} \qquad \text{no rule for } \mathbf{pick}$$

$$\frac{\Sigma;\, V \vdash_1 e}{\Sigma,\, \mathcal{EC}_c;\, V \vdash_1 (c\, e)}\ \text{TypeConstr}I_1 \qquad \frac{c \neq c'}{\Sigma,\, \mathcal{EC}_c;\, \cdot \vdash_1 (c'\, e)}\ \text{TypeConstr}I_2$$

$$\frac{\text{isValue}(d) \quad \Sigma,\, \mathcal{EA}_d;\, V \vdash_1 e}{\Sigma;\, V \vdash_1 (e\, d)}\ \text{Appl}I \qquad \frac{\text{isValue}(d) \quad ((\mathbf{fn}\ x \Rightarrow e)\ d) \Rightarrow e' \quad \Sigma;\, V \vdash_1 e'}{\Sigma,\, \mathcal{EA}_d;\, V \vdash_1 \mathbf{fn}\ x \Rightarrow e}\ \text{Appl}E$$

where $\mathcal{EA}_d(e) = (e\, d)$.

$$\frac{\neg\text{isValue}(d) \quad \cdot;\, V_1 \vdash_1 e_1 \quad \cdot;\, V_2 \vdash_1 e_2}{\Sigma;\, V_1 \cup V_2 \vdash_1 (e_1\, e_2)}\ \text{Appl} \qquad \frac{\cdot;\, V \vdash_1 e}{\Sigma;\, V \vdash_1 \mathbf{fn}\ x \Rightarrow e}\ \mathbf{fn}$$

$$\frac{\cdot;\, V \vdash_1 e}{\Sigma;\, V \vdash_1 \mathbf{forall}\ x_1,\, \ldots,\, x_n :\ e}\ \mathbf{forall} \qquad \frac{\cdot;\, V \vdash_1 e}{\Sigma;\, V \vdash_1 \mathbf{exists}\ x_1,\, \ldots,\, x_n :\ e}\ \mathbf{exists}$$

Splitting on the type of a state variable:

$$\frac{\delta : (\tau_0 \times \ldots \times \tau_{n-1}) \quad V_i \vdash_2 \mathcal{ET}_i(\delta) \quad \text{for } i = 0..n-1}{\bigcup_{i=0}^{n-1} V_i \vdash_2 \delta}$$

where $\mathcal{ET}_i(x_1,\, \ldots,\, x_i,\, \ldots,\, x_n) = x_i$.

$$\frac{\delta : \{f_1 :\ \tau_1,\, \ldots,\, f_n :\ \tau_n\} \quad V_i \vdash_2 \mathcal{ER}_{f_i}(\delta) \quad \text{for } i = 1..n}{\bigcup_{i=1}^{n} V_i \vdash_2 \delta}$$

where $\mathcal{ER}_{f_i}(\{f_1 = x_1,\, \ldots,\, f_i = x_i,\, \ldots,\, f_n = x_n\}) = x_i$.

For a datatype, we introduce an extra index variable x_δ that tells which constructor is used for each value. If this variable is $\bot$, then the value of the entire δ is $\bot$.

$$\frac{\delta : (C_1 \textbf{ of } \tau_1 \mid \ldots \mid C_n \textbf{ of } \tau_n) \quad V_i \vdash_2 \mathcal{EC}_{C_i}(\delta) \quad \text{for } i = 1..n}{\{x_\delta\} \cup \bigcup_{i=1}^{n} V_i \vdash_2 \delta}$$

where $\mathcal{EC}_C(C\ x) = x$, and C is a type constructor of a datatype.

We split an array into individual elements only when the index type is not too large. Otherwise we treat an array as a single variable.

$$\frac{\delta :\ \tau_1 \to \tau_2 \quad V_d \vdash_2 (\delta\ d) \quad |\mathcal{D}_{\omega(\tau_1)}| \leq \text{limit}}{\bigcup_{d \in \mathcal{D}_{\omega(\tau_1)} - \{*\}} V_d \vdash_2 \delta} \qquad \frac{\text{no other rule applies}}{\delta \vdash_2 \delta}$$

An equality relation for state variable expressions with similar "structure" upto expressions in the array indices: $\gamma_1 \approx \gamma_2$.

$$\frac{x \text{ is atomic}}{x \approx x} \qquad \frac{\gamma_1 \approx \gamma_2}{\gamma_1.f \approx \gamma_2.f}$$

$$\frac{\gamma_1 \approx \gamma_2}{\mathcal{ET}_n(\gamma_1) \approx \mathcal{ET}_n(\gamma_2)} \qquad \frac{\gamma_1 \approx \gamma_2}{\mathcal{EC}_c(\gamma_1) \approx \mathcal{EC}_c(\gamma_2)}$$

$$\frac{\text{isValue(d)} \quad \gamma_1 \approx \gamma_2}{(\gamma_1\ d) \approx (\gamma_2\ d)} \qquad \frac{\neg\text{isValue}(\text{e}_1) \text{ or } \neg\text{isValue}(\text{e}_2) \quad \gamma_1 \approx \gamma_2}{(\gamma_1\ e_1) \approx (\gamma_2\ e_2)}$$

The difference between the last two rules for application is that when d is a value, it must be the same for both γ_1 and γ_2.

The natural partial order on state variable expressions: $\gamma_1 \preceq \gamma_2$:

$$\frac{\gamma_1 \approx \gamma_2}{\gamma_1 \preceq \gamma_2} \qquad \frac{\gamma_1 \preceq \gamma_2}{\gamma_1.f \preceq \gamma_2} \qquad \frac{\gamma_1 \preceq \gamma_2}{(\gamma_1\ e) \preceq \gamma_2}$$

$$\frac{\gamma_1 \preceq \gamma_2}{\mathcal{ET}_n(\gamma_1) \preceq \gamma_2} \qquad \frac{\gamma_1 \preceq \gamma_2}{\mathcal{EC}_c(\gamma_1) \preceq \gamma_2}$$

4.5 Abstraction and Cone of Influence

4.6 Abstraction Mechanism.

The abstraction is defined on types: $\omega(\tau) = \tau'$. If a compound type contains at abstracted type but is not explicitly abstracted by itself, then its abstraction is the type with all the components replaced by their abstractions. If a type τ is not abstracted, then we define $\omega(\tau) = \tau$.

An abstraction of a type is a list of values of that type, with an obligatory "$*$" value which means "all the other values." The "$*$" value is treated specially: no array elements are included in the state space indexed by $*$, and a state variable $(\gamma\ *)$ is translated into "**anyvalue**."

4.7 Cone of Influence.

Cone of influence is a global data structure that stores information on variable dependencies. With each ground state variable it associates a set of state variables it depends on *in one step*. To get the full cone for a variable, one has to take the transitive closure of the cone for that variable.

If the transition system is defined over infinite or uninterpreted types, the cone cannot always be computed. For instance, when a state variable s of an array type $\tau_1 \to \tau_2$ is assigned an expression e (which must be a function, of course), and the type τ_1 is not (reasonably) finite, it is impossible to assign dependencies to the elements of the array. In this case, we do not split s into its components, but associate the expression e verbatim with s in the cone. In other words, we *defer* the computation of the cone for elements of s till later when we know more about which elements of s we actually need.

If later we want to test which variables, say, $s(0)$ depends on, we first look up $s(0)$ in the cone. If it is not present, we look up s, and if that gives us an expression e, we evaluate $e(0)$ and resume the cone computation for the result. For example, if we have an assignment

$$\mathbf{next}(s) := \mathbf{fn}\ (n : \mathbf{int}) \Rightarrow\ t(n)$$

then the cone for $s(0)$ will correctly return $t(0)$ — a single (and perhaps, finite-type variable), even though the original transition system was infinite and we couldn't have computed the cone beforehand.

Of course, we also need to be careful not to fall into an infinite loop, like in the case of

$$\mathbf{next}(s) := \mathbf{fn}\, n \Rightarrow\ s(n+1)$$

One of the ways to guarantee termination is to stop when we are about to look up the cone for s the second time and simply return the entire array s as a conservative approximation for the cone at that step. In the example, this would give just s as the cone for $s(0)$. In practice, we may have more complex structures like

$$\mathbf{next}((s.g\ d_1).f(d_2)) := (s.g(d_1+1)).f(d_2+2),$$

and we save a *wild card* of a state variable expression $(s.g\ *).f(*)$ which is the same as the original expression but with all infinite array indices replaced by a wild card. If we encounter the same wild card later in the cone expansion, we stop and return the topmost array with a wild card ($s.g$ in our example) as a conservative approximation. We only replace an index with an infinite or too large type with a wild card, and leave a "reasonably finite" index unchanged.

When we look up v_1 in the cone of v_2, and v_1 is not explicitly there, then we check $\gamma \in C(v_2)$ if $v_1 = (\gamma\ d)$ or $v_1 = \gamma.f$. If $C(v_2)$ is incomplete (has an expression, a.k.a. "delayed COI"), then either return "don't know" (interpret as "maybe yes"), or examine the expression, and if it clearly doesn't contain variables that depend on v_1, return "false."

More formally, let $C(\sigma(\gamma))$ be the list of variables or expressions directly associated with $\sigma(\gamma)$ in the database. We write $C(\sigma(\gamma)) := V$ for a (destructive) update to the cone of $\sigma(\gamma)$; the set of variables V is *added* to the existing set, it does not replace or remove variables from the cone. The sequent

$$\mathrm{addCone}(\sigma(\gamma) := e)$$

instructs a cone update for $\sigma(\gamma)$ with variables from e. In the rules for that sequent we use the destructive add operation on the cone, so there is no explicit result in the sequent. The rules for looking up the cone use the main sequent

$$C(\sigma(\gamma)) \Rightarrow V$$

that takes $\sigma(\gamma)$ and returns the set of variables (and only variables, no expressions for the delayed cone) that it depends on *in one step*. The full cone is its transitive closure. An auxiliary sequent computes variables and expressions for $\sigma(\gamma)$:

$$C(\sigma(\gamma)) \rightarrow V$$

$$\frac{\gamma : \tau_0 * \cdots * \tau_{n-1} \quad \mathcal{ET}_i(e) \Rightarrow e_i \quad \mathrm{addCone}(\sigma(\mathcal{ET}_i(\gamma)) := e_i) \quad \text{for } i = 0..n-1}{\mathrm{addCone}(\sigma(\gamma) := e)}\ \text{Tuple}$$

$$\frac{\gamma : \{f_1 : \tau_1, \ldots\ f_n : \tau_n\} \quad \mathcal{ER}_{f_i}(e) \Rightarrow e_i \quad \mathrm{addCone}(\sigma(\mathcal{ER}_{f_i}(\gamma)) := e_i) \quad \text{for } i = 1..n}{\mathrm{addCone}(\sigma(\gamma) := e)}\ \text{Record}$$

In the next rule the expression $\mathcal{EC}_{c_i}(e)$ may raise an exception when evaluated if e evaluates to $(c\ e')$ where $c \neq c_i$. In this case we know that the c_i's part of γ is not assigned, and thus, does not depend on e, so we abort the execution of that branch and move to the next constructor. Note that the internal selector variable x_γ depends on the entire expression e.

$$\frac{\begin{array}{c}\gamma : (c_1\ \mathbf{of}\ \tau_1 \mid \ldots \mid c_n\ \mathbf{of}\ \tau_n) \quad V \vdash e \quad C[\sigma(x_\gamma) \leftarrow V] \\ \mathcal{EC}_{c_i}(e) \Rightarrow e_i \quad \mathrm{addCone}(\sigma(\mathcal{EC}_{c_i}(\gamma)) := e_i) \quad \text{for } i = 1..n\end{array}}{\mathrm{addCone}(\sigma(\gamma) := e)}\ \text{Enum}$$

$$\frac{\gamma : \tau_1 \to \tau_2 \quad |\mathcal{D}_{\tau_1}| \leq \text{limit} \quad (e\ d) \Rightarrow e_d \quad \text{addCone}(\sigma(\gamma\ d) := e_d) \quad \text{for } d \in \mathcal{D}_{\tau_1}}{\text{addCone}(\sigma(\gamma) := e)}\ \text{Appl}_1$$

$$\frac{\gamma : \tau_1 \to \tau_2 \quad |\mathcal{D}_{\tau_1}| > \text{limit} \quad C[\sigma(\gamma) \leftarrow e]}{\text{addCone}(\sigma(\gamma) := e)}\ \text{Appl}_2$$

If no other rule applies, then γ must be of a primitive type, and we apply the last rule:

$$\frac{V \vdash e \quad C[\sigma(\gamma) \leftarrow V]}{\text{addCone}(\sigma(\gamma) := e)}\ \text{primitive type}$$

Now the rules for looking up variables in the cone, sequent $C(\sigma(\gamma)) \Rightarrow V$. We assume that γ is initially primitive (or of an unsplittable) type, and the cone either contains γ as a whole, or doesn't have it at all. The lookup rules implement only "one step" lookup. That is, it returns the variables γ depends on directly. The entire cone of γ is the transitive closure of this procedure.

The sequent $C(\sigma(\gamma)) \Rightarrow V$ is a wrapper for the actual lookup algorithm that forces all delayed cone expressions to finish:

$$\frac{C(\sigma(\gamma)) \to V, e_1, \ldots, e_n \quad V_i \vdash e_i \quad i = 1..n}{C(\sigma(\gamma)) \Rightarrow V \cup \bigcup_{i=1}^{m} V_i}$$

The core of the lookup algorithm is a sequent $C(\sigma(\gamma)) \to V$. The set V may contain delayed cone expressions. The invariant is that all such expressions are always of the same type as γ. If C contains the cone of $\sigma(\gamma)$ directly, then just return that cone:

$$\frac{C(\sigma(\gamma)) = [v_1, \ldots, v_n]}{C(\sigma(\gamma)) \to v_1, \ldots, v_n}\ \text{direct}$$

Otherwise we assume that $C(\sigma(\gamma)) = \emptyset$, and apply the other rules. In the Appl rule, if all variables in V_i' are of primitive types, then $\rho_i' = V_i'$, otherwise $\rho_i' = e_i'$, a delayed cone expression.

$$\frac{C(\sigma(\gamma)) \to V,\, e_1, \ldots, e_m \quad (e_i\ d) \Rightarrow e_i' \quad V_i' \vdash e_i' \quad i = 1..m}{C(\sigma(\gamma\ d)) \to V,\, \rho_1', \ldots, \rho_m'}\ \text{Appl}$$

$$\frac{C(\sigma(\gamma)) \to V,\, e_1, \ldots, e_m \quad e_i.f \Rightarrow e_i' \quad i = 1..m}{C(\sigma(\gamma.f)) \to V,\, e_1', \ldots, e_m'}\ \text{Record}$$

$$\frac{C(\sigma(\gamma)) \to V,\, e_1, \ldots, e_m \quad \mathcal{ET}_i(e_j) \Rightarrow e_j' \quad j = 1..m}{C(\sigma(\mathcal{ET}_i(\gamma))) \to V,\, e_1', \ldots, e_m'}\ \text{Tuple}$$

$$\frac{C(\sigma(\gamma)) \to V,\, e_1, \ldots, e_m \quad \mathcal{EC}_c(e_i) \Rightarrow e_i' \quad i = 1..m}{C(\sigma(\mathcal{EC}_c(\gamma))) \to V,\, e_1', \ldots, e_m'}\ \text{Enum}$$

$$\frac{\text{No other rule applies}}{C(\sigma(\gamma)) \to \cdot}\ \text{default}$$

When checking if a state variable γ is in the cone $C(\sigma(\gamma')) = V$, since γ may not be exactly as any of the variables listed in V, we must check whether it has any common part with any $v \in V$:

$$\frac{\gamma \preceq \delta \text{ or } \delta \preceq \gamma}{\gamma \in V, \delta}$$

4.8 Transition Relation Generator

Generating the transition relation consists of two phases: (1) Preparation phase: collect information about which variables are assigned by which assignments; and (2) Generation phase: translate the assignments into a boolean formula representing the transition relation.

4.8.1 Preparation Phase

The sequent for preparation is

$$N;\ L;\ I \vdash A[\text{or } AL]$$

where N and L are lists of combinational and latch variables, and I is the initial states. Inside the sequent, the input is the assignment on the RHS and an abstraction ω, the output is a tuple (C, N, L, I) on the LHS. The final result in this phase is the entire derivation of the sequent $N;\ L;\ I \vdash AL]$ and C — the cone of influence (association list mapping each variable to the set of variables it directly depends on).

$$\frac{N \vdash \gamma \ll e}{N;\ \cdot;\ \cdot \vdash \gamma := e}\ \text{Normal} \qquad \frac{L \vdash \mathbf{next}(\gamma) \ll e}{\cdot;\ L;\ \cdot \vdash \mathbf{next}(\gamma) := e}\ \mathbf{next} \qquad \frac{I \vdash \mathbf{init}(\gamma) \ll e}{\cdot;\ \cdot;\ I \vdash \mathbf{init}(\gamma) := e}\ \mathbf{init}$$

$$\frac{\begin{array}{l} N_1;\ L_1;\ I_1 \vdash A_1 \quad N_2;\ L_2;\ I_2 \vdash AL \\ \quad N_1 \cap N_2 = \emptyset \quad L_1 \cap L_2 = \emptyset \quad I_1 \cap I_2 = \emptyset \quad (N_1 \cup N_2) \cap (L_1 \cup L_2 \cup I_1 \cup I_2) = \emptyset \end{array}}{N_1 \cup N_2;\ L_1 \cup L_2;\ I_1 \cup I_2 \vdash A_1;\ AL}\ \text{AsstList}$$

$$\frac{N;\ L;\ I \vdash AL}{N;\ L;\ I \vdash \mathbf{let}\ D_1 \ldots D_n\ \mathbf{in}\ AL\ \mathbf{end}}\ \mathbf{let}$$

In **case**, **if**, and **choose** statements all the assigned variables depend on the conditions and guards, and we add them to the cone of influence.

$$\frac{\begin{array}{l} N_i;\ L_i;\ I_i \vdash AL_i \quad (\cup N_i) \cap (\cup L_i \cup I_i) = \emptyset \\ \quad V \vdash e \quad C := \text{merge}(C,\ [\bigcup (N_i \cup L_i \cup I_i) \leftarrow V]) \quad \forall v \in \cup N_i \cup I_i.\ v \notin C(v) \end{array}}{\cup N_i;\ \cup L_i;\ \cup I_i \vdash \mathbf{case}\ e\ \mathbf{of}\ P_1 \Rightarrow AL_1 | \ldots | P_n \Rightarrow AL_n\ \mathbf{endcase}}\ \mathbf{case}$$

$$\frac{\begin{array}{l} N_i;\ L_i;\ I_i \vdash AL_i \quad (\cup N_i) \cap (\cup L_i \cup I_i) = \emptyset \\ \quad V_i \vdash e_i \quad C := \text{merge}(C,\ [\bigcup (N_i \cup L_i \cup I_i) \leftarrow \cup V_i]); \quad \forall v \in \cup N_i \cup I_i.\ v \notin C(v) \end{array}}{\cup N_i;\ \cup L_i;\ \cup I_i \vdash \mathbf{if}\ e_1\ \mathbf{then}\ AL_1\ \ldots\ \mathbf{elsif}\ e_n\ \mathbf{then}\ AL_n\ \mathbf{else}\ AL_{n+1}\ \mathbf{endif}}\ \mathbf{if}$$

$$\frac{\begin{array}{l} N_i;\ L_i;\ I_i \vdash AL_i \quad (\cup N_i) \cap (\cup L_i \cup I_i) = \emptyset \\ \quad V_i \vdash e_i \quad C := \text{merge}(C,\ [\bigcup (N_i \cup L_i \cup I_i) \leftarrow \cup V_i]); \quad \forall v \in \cup N_i \cup I_i.\ v \notin C(v) \end{array}}{\cup N_i;\ \cup L_i;\ \cup I_i \vdash \mathbf{choose}\ e_1\ \Rightarrow AL_1 \mid \ldots \mid e_n \Rightarrow AL_n\ \mathbf{endchoose}}\ \mathbf{choose}$$

The rules for $N \vdash \gamma \ll e$ (extracting all the ground variables that γ stands for, and adding to their cones the variables from e) consists of two phases: splitting γ itself ($\Delta \vdash_1 \gamma$) and splitting on its type ($N \vdash_2 \delta \ll e$). The first phase computes the list of atomic state variables (ones that cannot be broken further into more primitive SyMP state variables), and the second phase splits the atomic state variables of complex types into vectors of variables of primitive types. The set of primitive type variables is the final result. The expression e is also being split into primitive type components along with δ to provide a refined cone of influence information.

Another side effect of the second phase is building the correspondence between SyMP state variables δ and the primitive variables from N. This information must be preserved so that we could later translate a counterexample from an external model checker back to SyMP state variables.

We write $\sigma(\gamma)$ for one of γ, $\mathbf{next}(\gamma)$, or $\mathbf{init}(\gamma)$; that is, $\sigma \in \{\cdot,\ \mathbf{next},\ \mathbf{init}\}$.

$$\begin{array}{rcl} \gamma & ::= & x \mid \gamma\, e \mid \gamma.f \\ \delta & ::= & x \mid \delta\, d \mid \delta.f, \text{ where isValue}(d) \end{array}$$

$$\frac{\delta_1,\ \ldots,\ \delta_n \vdash_1 \sigma(\gamma) \quad N_i \vdash_2 \delta_i \ll e \quad \text{for } i = 1..n}{\bigcup_{i=1}^{n} N_i \vdash \sigma(\gamma) \ll e}$$

Splitting on state variable expression γ:

Assignment to an array element with a static index (known at compile time):

$$\frac{\Delta \vdash_1 \sigma(\gamma) \quad \text{isValue}(d)}{\Delta_d \vdash_1 \sigma(\gamma\, d)}\ \text{Appl}_1$$

where $\Delta_d = \{\sigma(\delta\ d) \mid \sigma(\delta) \in \Delta\}$. Assignment to an array element with dynamic index:

$$\frac{\Delta' \vdash_1 \sigma(\gamma) \quad \neg\text{isValue}(e) \quad e : \tau}{\Delta \vdash_1 \sigma(\gamma\ e)}\ \text{Appl}_2$$

where $\Delta = \bigcup_{d \in \mathcal{D}_{\omega(\tau)} - \{*\}} \Delta'_d$ if $|\mathcal{D}_{\omega(\tau)}| \leq \text{limit}$, and $\Delta = \Delta'$ otherwise.

The difference between the two "Appl" rules is that if we know which index is assigned, we consider only that variable to be assigned; otherwise it is an assignment to the entire array, where one of the elements gets the RHS value, and the others either retain the old value (for $\mathbf{next}$ assignments) or get assigned $\perp$.

$$\frac{\Delta \vdash_1 \sigma(\gamma)}{\Delta.f \vdash_1 \sigma(\gamma.f)}\ \text{Record}$$

where $\Delta.f = \{\sigma(\delta.f) \mid \sigma(\delta) \in \Delta\}$.

$$\frac{\text{no other rule applies}}{\sigma(\gamma) \vdash_1 \sigma(\gamma)}\ \text{atomic}$$

Splitting on type:

$$\frac{\delta : (\tau_0 \times \ldots \times \tau_{n-1}) \quad \mathcal{ET}_i(e) \Rightarrow e'_i \quad N_i \vdash_2 \sigma(\mathcal{ET}_i(\delta)) \ll e'_i \quad \text{for } i = 0..n-1}{\bigcup_{i=0}^{n-1} N_i \vdash_2 \sigma(\delta) \ll e}$$

where $\mathcal{ET}_i(x_1, \ldots, x_i, \ldots, x_n) = x_i$.

$$\frac{\delta : \{f_1 : \tau_1, \ldots, f_n : \tau_n\} \quad \mathcal{ER}_{f_i}(e) \Rightarrow e'_i \quad N_i \vdash_2 \sigma(\mathcal{ER}_{f_i}(\delta)) \ll e'_i \quad \text{for } i = 1..n}{\bigcup_{i=1}^{n} N_i \vdash_2 \sigma(\delta) \ll e}$$

where $\mathcal{ER}_{f_i}(\{f_1 = x_1, \ldots, f_i = x_i, \ldots, f_n = x_n\}) = x_i$.

For a datatype, we introduce an extra index variable x_δ that tells which constructor is used for each value. If this variable is $\perp$, then the value of the entire δ is $\perp$.

$$\frac{\begin{array}{l}\delta : (C_1\ \mathbf{of}\ \tau_1 \mid \ldots \mid C_n\ \mathbf{of}\ \tau_n) \quad \mathcal{EC}_{C_i}(e) \Rightarrow e'_i \quad N_i \vdash_2 \sigma(\mathcal{EC}_{C_i}(\delta)) \ll e'_i \quad \text{for } i = 1..n \\ V \vdash e \quad C = \text{merge}(C, [\sigma(x_\delta) \leftarrow V])\end{array}}{\{\sigma(x_\delta)\} \cup \bigcup_{i=1}^{n} N_i \vdash_2 \sigma(\delta) \ll e}$$

where $\mathcal{EC}_C(C\ x) = x$, and C is a type constructor of a datatype.

We split an array into individual elements only when the index type is not too large. Otherwise we treat an array as a single variable.

$$\frac{\delta : \tau_1 \to \tau_2 \quad |\mathcal{D}_{\omega(\tau_1)}| \leq \text{limit} \quad (e\ d) \Rightarrow e' \quad N_d \vdash_2 \sigma(\delta\ d) \ll e' \quad d \in \mathcal{D}_{\omega(\tau_1)} - \{*\}}{\bigcup_{d \in \mathcal{D}_{\omega(\tau_1)} - \{*\}} N_d \vdash_2 \sigma(\delta) \ll e}$$

If none of the above rules applies, we treat δ as a primitive variable:

$$\frac{\text{no other rule applies,} \quad V \vdash e \quad C = \text{merge}(C, [\sigma(\delta) \leftarrow V])}{\sigma(\delta) \vdash_2 \sigma(\delta) \ll e}$$

4.8.2 Customization Phase

The previous preparation phase builds a derivation tree for

$$N;\ L;\ I \vdash AL$$

that serves as an internal representation of the assignments. In this phase we specialize this tree to a particular specification F: compute the cone of influence $V_F = C(F)$, update the abstraction information (the prover may give a more detailed abstraction at this point), and rebuild the derivation tree leaving only relevant variables and assignments. Also, we balance sets of variables for the assignments in the branches of **if** and **case** statements to find out which variables go into implicit **nop** at the end of each assignment list.

Also, since we use lazy cone computation (expressions of infinite types in the cone that cannot be expanded), we cannot always detect circular assignments. Therefore, we have to recheck for them every time we refine our abstraction or current set of state variables in the cone.

TODO: find out if we can get a circular assignment into the transition relation in a way that will not be detectable under some reasonable abstraction, but would allow us to prove a false property (soundness, absence of false positives).

If this phase still returns an infinite or too large transition system (the cone still contains unexpanded expressions), then these expressions can be given back to the prover as hints for which types require an abstraction to make the system finite. The prover then can decide on the abstraction and rerun this phase until the system is finite. In fact, we do not even have to rebuild the tree for it, we just need to compute the cone of influence to get this information.

Building the Cone of the Spec:

For each variable $a \in V.V \vdash spec$:

$$\frac{\tau \text{ is primitive}}{a : \tau \rightsquigarrow C(a)}$$

$$\frac{\omega(\tau_1) = \mathcal{D} \text{ or } \mathcal{D} = \mathcal{D}_{\tau_1} \quad (a\ d) : \tau_2}{a : \tau_1 \rightarrow \tau_2 \rightsquigarrow \bigcup_{d \in \mathcal{D}} V_d}$$

$$\frac{\Sigma \mathcal{T}_i(a) \rightsquigarrow V_i}{a : \tau_1 * \tau_2 \ldots \tau_n \rightsquigarrow \bigcup V_i}$$

$$\frac{\Sigma \mathcal{R}_{f_i}(a) \rightsquigarrow V_i}{a : \{f_1 : \tau_1, f_2 : \tau_2 \ldots f_n : \tau_n\} \rightsquigarrow \bigcup V_i}$$

$$\frac{\Sigma \mathcal{C}_{c_i}(a) \rightsquigarrow V_i}{a : C_1 \textbf{ of } \tau_1, C_2 \textbf{ of } \tau_2 \ldots C_n \textbf{ of } \tau_n\} \rightsquigarrow \{x_C\} \cup \bigcup V_i}$$

$$\frac{\omega(\tau) \ \ C(a) = V, e_1, \ldots e_n \ \ v_i^1, \ldots v_j^m \vdash e_i \ \ v_i^j : \tau_i^j \rightsquigarrow V_i^j}{a : \tau \rightsquigarrow V \cup \bigcup_{i,j} V_i^j}$$

4.8.3 Generation Phase

In this phase we assume that the derivation tree for the sequent

$$N;\ L;\ I \vdash AL$$

is fully customized, the cone is fully computed (all variables in N, L, and I are of primitive or abstracted types, and there are no unexpanded expressions from lazy cone), and all variables are balanced in conditional branches. Also, there are no cyclic or multiple assignments.

We generate the transition relation in several (mutually recursive) steps.

1. Convert the complex assignment expressions (like **case**, **if**, etc.) into a boolean formula that still contains elementary assignments ($\gamma := e$) as subformulas;

2. Lift all nondeterminism to the top level in the conditions of **if** and **case**, and in the indices of state variable arrays;

3. Replace $\gamma := (e_1 \mid \ldots \mid e_n)$ by $\bigvee_{i=1}^{n} \gamma := e_i$ (the semantics of nondeterministic choice);

4. Expand state variable expression γ in every elementary assigment $\gamma := e$ and split γ into the atomic state variables; i.e. do η-conversion for arrays and split assignments to records into assignments to individual fields.

In the implementation, we do all the steps in one pass. The corresponding sequents for these stages are:

$N;\ L;\ I \vdash A \rightsquigarrow \Phi$	Convert an assignment expression A into a boolean "formula" Φ. The other inputs are lists of combinational (N) and latched variables assigned next (L) and initial (I) values. The output "formula" is of the form $$\textbf{def}\ \mathit{defList}\ \textbf{in}\ \mathit{condList}\ \textbf{end},$$ where the **def**-structure is described below.
$e \uparrow\!\rightsquigarrow (e_1 \mid \ldots \mid e_n)$	Lifting explicit nondeterminism in e to the top level yields $(e_1 \mid \ldots \mid e_n)$. Note, that not all nondeterminism is eliminated in e_i's, since abstract interpretation may introduce it at the translation time.
$V \vdash \gamma := e \rightsquigarrow F$	γ is a state variable expression that stands for the set of ground variables V, and the assignment translates into F.
$e_1 = e_2 \rightsquigarrow F$	2-valued boolean equality between expressions translates into formula F.
$\mathrm{matches}(e,\ P) \rightsquigarrow F$	"e matches pattern P" is translated into F.
$op(F_1, \ldots, F_n) \downarrow$	Propagates the operator op down to the basic expressions of the **def**-structures and normalizes them, returning a single valid **def**-structure.

The structure "$\textbf{def}\ \mathit{defList}\ \textbf{in}\ \mathit{condList}\ \textbf{end}$" is a way to lift the nondeterminism from the assignments and expressions without blowing up the size of the formulas. The $\mathit{defList}$ is the list of (local) macro definitions for nondeterministic expressions, and $\mathit{condList}$ is a list of pairs

$$((c_1,\ A_1),\ \ldots,\ (c_n,\ A_n))$$

or

$$((c_1,\ A_1) \mid \ldots \mid (c_n,\ A_n))$$

representing a conditional assignment. The comma-separated conditionals are evaluated sequentially until a true one is found, and its assignment is executed. The bar-separated conditionals behave as a guarded nondeterministic choice (that is, they are simply and'ed together):

$$(c_1 \Rightarrow A_1) \wedge (c_2 \Rightarrow A_2) \wedge \cdots \wedge (c_n \Rightarrow A_n).$$

It is important that at any time at least one of the conditions must be satisfiable in both versions of the conditionals.

The second element of each pair A_i can either be another nested **def**-structure, or a triple $[W, F, T]$, where the three boolean formulas W, F and T define the invariant, next, and initial state predicates respectively. These formulas are in turn represented by a **def**-structure, only the $\mathit{condList}$ is given by

$$((c_1,\ e_1),\ \ldots,\ (c_n,\ e_n))$$

representing a conditional expression, or by

$$(\langle d_1,\ e_1\rangle,\ \ldots,\ \langle d_n,\ e_n\rangle)$$

representing an array of expressions indexed by the constants d_i (that is, the entire expression has a functional type). The second element of each pair e_i can either be a deterministic expression, possibly depending on the defined macros, or it can be a similar list of conditionals. As for the assignments, the semantics of the conditionals is sequential (like for a multi-choice **if-then-else** operator). There are no bar-separated conditionals for expressions. The conditionals and expressions may depend on the macros from $\mathit{defList}$, but they should not have any nondeterministic expressions themselves. The definitions of the macros are lists of nondet. choices, and each choice may in turn be a **def**-structure. Names of the macros must be disjoint in all the **def**-structures (use unique name generators for the macro names).

4.8.4 Normalization of def-structures

First, we define the "$\downarrow$" operator that computes the new "normalized" **def**-structure $F = op(F_1, \ldots, F_n)\downarrow$, where

$$F_i = \mathbf{def}\ D_1^i, \ldots, D_n^i\ \mathbf{in}\ ((c_1, e_1), \ldots, (c_{m_i}, e_{m_i}))\ \mathbf{end}.$$

Another variant of the normalization operator takes a body of a **def**-structure, which may contain some nested **def**-structures, and converts it into a legal **def**-structure:

$$F = ((C_1, F_1), \ldots, (C_n, F_n))\downarrow.$$

The later is done by merging all the macro definitions in the subexpressions, and building a new **def**-structure like this:

$$F = \mathbf{def}\ \mathcal{D}_{C_1}, \mathcal{D}_{F_1}, \ldots, \mathcal{D}_{C_n}, \mathcal{D}_{F_n}, c_1 = B_{C_1}, c_n = B_{C_n}\ \mathbf{in}\ ((c_1, B_{F_1}), \ldots, (c_n, B_{F_n}))\ \mathbf{end},$$

where $\mathcal{D}_F$ and B_F are declarations and the body of a **def**-structure F respectively. We assume that the **def**-structures in the subexpressions are normalized.

The normalization for assignments is much less complicated, and it only exists in the first form, that is, applying an operator to **def**-structures and making it a single **def**-structure. Otherwise **def**-structures for assignments can be nested arbitrarily without any normalization. However, it makes sense to move all the local definitions in assignments to the top level at the end, to simplify the translation to a back-end model checker, but it is rather simple a step and we will not discuss it here.

Translating built-in operators. Roughly, the rule for a built-in operator op is

$$\frac{e_i \rightsquigarrow F_i \quad i = 1..n}{op(e_1, \ldots, e_n) \rightsquigarrow op(F_1, \ldots, F_n)\downarrow},$$

and the expression $op(F_1, \ldots, F_n)\downarrow$ is then translated into

$$\frac{op(\chi_1, \ldots, \chi_n) \rightsquigarrow_{op} \mathbf{def}\ \mathcal{D}\ \mathbf{in}\ E\ \mathbf{end}}{op(F_1, \ldots, F_n)\downarrow \rightsquigarrow \mathbf{def}\ \chi_1 = F_1, \ldots, \chi_n = F_n, \mathcal{D}\ \mathbf{in}\ E\ \mathbf{end}\downarrow}.$$

More precisely, the normalization applies the operator to some atomic expressions (whether they are defined macros or actual atomic expressions), and then calls a special procedure that translates the application of op to deterministic arguments (sequent $op(e_1, \ldots, e_n) \rightsquigarrow_{op} F$). This procedure should also take care of any abstract interpretations defined for op. It may also call the normalization function recursively, as is the case with the equality and application.

In reality, some of the operators (notably, the equality, function application, and the extractors for tuples, records, and enumeration types) can be handled more efficiently if their arguments' **def**-structures are translated directly, and this is what we do.

4.8.5 Translating the extractors

Basically, the extractors are propagated down through the **def**-structures until they either annihilate with a compound "atomic" expression, or hit a true atomic expression and just land on it.

$$\frac{\mathcal{E}x(E) \rightsquigarrow_{\mathcal{E}x} F}{\mathcal{E}x(\mathbf{def}\ \mathcal{D}\ \mathbf{in}\ E\ \mathbf{end}) \rightsquigarrow_{\mathcal{E}x} F} \qquad \frac{\mathcal{E}x(E_j) \rightsquigarrow_{\mathcal{E}x} F_j \quad j = 1..n}{\mathcal{E}x((c_1, E_1), \ldots, (c_n, E_n)) \rightsquigarrow_{\mathcal{E}x} ((c_1, F_1), \ldots, (c_n, F_n))\downarrow}$$

$$\frac{e \text{ is atomic} \quad \mathcal{E}x(e) \Rightarrow e'}{\mathcal{E}x(e) \rightsquigarrow_{\mathcal{E}x} e'}$$

where $\mathcal{E}x$ stands for any one of $\mathcal{ET}_i$, $\mathcal{ER}_f$, $\mathcal{EC}_c$, or $\mathcal{EI}$. It is an error for the arguments for these extractors to be an array, because the types wouldn't match. The typechecker should guarantee that this doesn't happen though.

4.8.6 Translating 2-valued equality

In this section, we define the procedure for applying the 2-valued equality to **def**-structures of an arbitrary type. The idea is to split the structures by the type and apply the equality to the primitive-type components by recursing into the structures.

$$\frac{E_1 : \tau_1 * \cdots \tau_n \quad (\mathcal{ET}_i(E_1)) \downarrow =_2 (\mathcal{ET}_i(E_2)) \downarrow \rightsquigarrow_{=_2} F_i \quad i = 1..n}{E_1 =_2 E_2 \rightsquigarrow_{=_2} (\bigwedge_{i=1}^{n} F_i) \downarrow} \text{tuple}$$

$$\frac{E_1 : \{f_1 : \tau_1, \ldots, f_n : \tau_n\} \quad (\mathcal{ER}_{f_i}(E_1)) \downarrow =_2 (\mathcal{ER}_{f_i}(E_2)) \downarrow \rightsquigarrow_{=_2} F_i \quad i = 1..n}{E_1 =_2 E_2 \rightsquigarrow_{=_2} (\bigwedge_{i=1}^{n} F_i) \downarrow} \text{record}$$

$$\frac{E_1 : (c_1 \textbf{ of } \tau_1 \mid \cdots \mid c_n \textbf{ of } \tau_n) \quad (\mathcal{EI}(E_1)) \downarrow =_2 (\mathcal{EI}(E_2)) \downarrow \rightsquigarrow_{=_2} C \quad (\mathcal{EC}_{c_i}(E_1)) \downarrow =_2 (\mathcal{EC}_{c_i}(E_2)) \downarrow \rightsquigarrow_{=} F_i}{E_1 =_2 E_2 \rightsquigarrow_{=_2} (C \wedge \bigwedge_{i=1}^{n} F_i) \downarrow} \text{enum}$$

$$\frac{E_1 : \tau_1 \to \tau_2 \quad (E_1\ d) \downarrow =_2 (E_2\ d) \downarrow \rightsquigarrow_{=_2} F_d \quad d \in \mathcal{D}_{\omega(\tau_1)}}{E_1 =_2 E_2 \rightsquigarrow_{=_2} (\bigwedge_{d \in \mathcal{D}_{\omega(\tau_1)}} F_d) \downarrow} \text{apply}$$

$$\frac{E_1 : \tau, \quad \tau \text{ is primitive and not abstracted}}{E_1 =_2 E_2 \rightsquigarrow_{=_2} \textbf{def } \chi_1 = E_1, \chi_2 = E_2 \textbf{ in } \chi_1 = \chi_2 \textbf{ end}} \text{prim}$$

$$\frac{E_1 : \tau, \quad \tau \text{ is primitive and abstracted} \quad E_1 =_2 E_2 \rightsquigarrow_{\omega(=_2)} F}{E_1 = E_2 \rightsquigarrow_{=_2} F} \text{abstr}$$

In the implementation, we define an abstract interpretation of an operator as another SyMP function that is only allowed to compare values of its arguments (currently, of primitive types only) for equality and return either a constant value or one of its arguments, or a nondeterministic choice of constant values or its arguments. Application of such a function to our expressions is then translated into a **def**-structure by a special procedure which is relatively simple, as we only manipulate primitive types and data. For instance, a typical abstract interpretation of equality over $\tau = \{a, *\}$ would be

$$\begin{array}{rl} \omega_\tau(x = y) = & \textbf{if } x = a \wedge y = a \textbf{ then true} \\ & \textbf{elsif } x = a \vee y = a \textbf{ then false} \\ & \textbf{else } (\textbf{true} \mid \textbf{false}) \textbf{ endif} \quad \textit{(* here } x = * \wedge y = * \textit{ *)} \end{array}$$

which is translated into

$$\textbf{def } \chi = \{\textbf{true}, \textbf{false}\} \textbf{ in } ((x = a \wedge y = a, \textbf{true}), (x = a \vee y = a, \textbf{false}), (\textbf{true}, \chi)) \textbf{ end},$$

and this is what we denote by the sequent $e_1 = e_2 \rightsquigarrow_{\omega(=)} F$.

4.8.7 Translating application

The general idea: descend into the structure of the function and apply the atomic expressions to the argument.

$$\frac{(E\ G) \rightsquigarrow_{\text{app}} F}{(\textbf{def } D \textbf{ in } E \textbf{ end } G) \rightsquigarrow_{\text{app}} (\textbf{def } D \textbf{ in } F \textbf{ end}) \downarrow} \textbf{def}$$

$$\frac{(E_i\ G) \rightsquigarrow_{\text{app}} F_i}{((c_1, E_1), \ldots, (c_n, E_n)) \rightsquigarrow_{\text{app}} ((c_1, F_1), \ldots, (c_n, F_n)) \downarrow} \text{cond}$$

$$\frac{F_i = (G = d_i) \downarrow}{(\langle c_1, E_1 \rangle, \ldots, \langle c_n, E_n \rangle) \rightsquigarrow_{\text{app}} ((c_1, F_1), \ldots, (c_n, F_n)) \downarrow} \text{array}$$

$$\frac{E \text{ is atomic}}{(E\ G) \rightsquigarrow_{\text{app}} \textbf{def } \chi = G \textbf{ in } (E\ \chi) \textbf{ end}} \text{atom}$$

In the last rule (atom), E can only be a variable or a constant, possibly with some extractors. In the (array) rule, we should check for the false conditions and optimize them away.

4.8.8 Pattern matching

$$\frac{}{\text{matches}(e,\ \_) \rightsquigarrow \mathbf{true}} \qquad \frac{x \text{ is a variable}}{\text{matches}(e,\ x) \rightsquigarrow \mathbf{true}}$$

$$\frac{\text{matches}(e,\ P) \rightsquigarrow F}{\text{matches}(e,\ x\ \mathbf{as}\ P) \rightsquigarrow F} \qquad \frac{\text{matches}(e,\ P) \rightsquigarrow F}{\text{matches}(e,\ P : \tau) \rightsquigarrow F}$$

$$\frac{\mathcal{ET}_i(e) \Rightarrow e_i \quad \text{matches}(e_i,\ P_i) \rightsquigarrow F_i \quad i = 1..n}{\text{matches}(e,\ (P_1,\ \ldots,\ P_n)) \rightsquigarrow (\bigwedge_{i=1}^{n} F) \downarrow} \qquad \frac{\mathcal{ER}_{f_i}(e) \Rightarrow e_i \quad \text{matches}(e_i,\ P_i) \rightsquigarrow F_i \quad i = 1..n}{\text{matches}(e,\ \{f_1 = P_1,\ \ldots,\ f_n = P_n\}) \rightsquigarrow (\bigwedge_{i=1}^{n} F) \downarrow}$$

$$\frac{\mathcal{EI}(e) \Rightarrow e_I \quad \mathcal{EC}_c(e) \Rightarrow e_c \quad e_I = c \rightsquigarrow_= C \quad \text{matches}(e_c,\ P) \rightsquigarrow F}{\text{matches}(e,\ (c\,P)) \rightsquigarrow (C \wedge F) \downarrow}$$

$$\frac{d \text{ is constant} \quad e = d \rightsquigarrow_= F}{\text{matches}(e,\ d) \rightsquigarrow F}$$

4.8.9 Translating arbitrary expressions into def-structures

A **def**-structure for an expression can have an arbitrary type, but the internal structure of it is always nearly primitive: the body is a deterministic expression containing only atomic constants and variables, component extractors (for complex types), and application of built-in functions to primitive-type expressions (this includes equality). All the nondeterminism is lifted into macro definitions which, as you remember, are lists of nested **def**-structures. So, the idea is that once an expression is translated into such a **def**-structure, it is cheap and easy to apply functions to them, even when they are abstractly interpreted and yield nondeterministic results. Basically, the normalization operator $(\cdot)\downarrow$ computes a normalized **def**-structure for any operation done on other **def**-structures.

The translation consists of 3 (mutually recursive) steps:

$e \rightsquigarrow F$	The top-level sequent; split the expression into primitive components by its type;
$\Omega;\ e \rightsquigarrow_\Omega F$	Same as above, except translate only those components of the expression that are mentioned in Ω, where $\Omega = \{\Sigma_1,\ \ldots,\ \Sigma_n\}$, and each Σ_i is a list of extractors that can be applied to the expression e. In effect, every Σ_i defines a primitive-type part of e that is important. All the other parts are defined as $\perp$.
$\Sigma;\ e \rightsquigarrow F$	Descend into all the compound statements in e, rip the extractors off e, put them into Σ, and possibly re-apply them on-the-fly to simplify the expression until e becomes atomic; then proceed to wrapping them back. Note: the order in Σ is important.
$\Sigma;\ \mathcal{D};\ e \rightsquigarrow^{\text{wrap}} F$	Wrap the extractors from Σ back on e. This stage basically translates the application of expressions into element-wise application split by cases, and adds the translation of the argument to $\mathcal{D}$ (the order here, in principle, is not important, but the definitions should be acyclic). When Σ becomes empty, build a final **def**-structure with $\mathcal{D}$ and e in the body.

Now, the rules. Splitting the expression into primitive type components:

$$\frac{e : \tau_1 * \cdots * \tau_n \quad \mathcal{ET}_i(e) \Rightarrow e_i \quad e_i \rightsquigarrow F_i, \quad i = 1..n}{e \rightsquigarrow (F_1,\ \ldots,\ F_n) \downarrow}\ \text{tuple}$$

$$\frac{e : \{f_1 : \tau_1,\ \ldots,\ f_n : \tau_n\} \quad \mathcal{ER}_{f_i}(e) \Rightarrow e_i \quad e_i \rightsquigarrow F_i, \quad i = 1..n}{e \rightsquigarrow \{f_1 = F_1,\ \ldots,\ f_n = F_n\} \downarrow}\ \text{record}$$

$$\frac{e : (c_1\ \mathbf{of}\ \tau_1 \mid \ldots \mid c_n\ \mathbf{of}\ \tau_n) \quad \mathcal{EI}(e) \Rightarrow e^I \quad \mathcal{EC}_{c_i}(e) \rightsquigarrow F_i \quad e^I = c_i \rightsquigarrow C_i \text{ for } i = 1..n}{e \rightsquigarrow (\ldots,\ (C_i,\ F_i),\ \ldots) \downarrow}\ \text{enum}$$

$$\frac{e : \tau_1 \to \tau_2 \quad (e\ d_i) \Rightarrow F_i \quad D_{\omega(\tau_1)} = \{d_1,\ \ldots,\ d_n\}, \quad n \leq \text{limit}}{e \rightsquigarrow (\ldots,\ \langle C_i,\ F_i \rangle,\ \ldots) \downarrow}\ \to$$

The rules for the restricted version of this sequent are similar, except that we need to check with Ω before applying any extractor. If Ω happens to become empty, just return the undefined value:

$$\frac{}{\emptyset;\, e \leadsto_\Omega \bot}\ \mathbf{empty}$$

Otherwise shave off the top-level extractors from all Σ's in Ω and proceed. Here $\mathcal{E}(\Omega)$ stands for:

$$\mathcal{E}(\Omega) = \{\Sigma \mid (\Sigma,\, \mathcal{E}) \in \Omega\},$$

where $\mathcal{E}$ is any extractor.

$$\frac{e : \tau_1 * \cdots * \tau_n \quad \mathcal{ET}_i(e) \Rightarrow e_i \quad \mathcal{ET}_i(\Omega);\, e_i \leadsto_\Omega F_i, \quad i = 1..n}{\Omega;\, e \leadsto_\Omega (F_1,\, \ldots,\, F_n) \downarrow}\ \text{tuple}$$

$$\frac{e : \{f_1 : \tau_1,\, \ldots,\, f_n : \tau_n\} \quad \mathcal{ER}_{f_i}(e) \Rightarrow e_i \quad \mathcal{ER}_{f_i}(\Omega);\, e_i \leadsto_\Omega F_i, \quad i = 1..n}{\Omega;\, e \leadsto_\Omega \{f_1 = F_1,\, \ldots,\, f_n = F_n\} \downarrow}\ \text{record}$$

$$\frac{\begin{array}{c} e : (c_1 \ \mathbf{of}\ \tau_1 \mid \ldots \mid c_n \ \mathbf{of}\ \tau_n) \quad \mathcal{EI}(e) \Rightarrow e^I \quad \mathcal{EI}(\Omega);\, e^I \leadsto_\Omega E^I \\ \mathcal{EC}_{c_i}(\Omega);\, \mathcal{EC}_{c_i}(e) \leadsto_\Omega F_i \quad E^I = c_i \leadsto C_i \text{ for } i = 1..n \end{array}}{\Omega;\, e \leadsto_\Omega \text{Enum}[E^I](\ldots,\, (C_i,\, F_i),\, \ldots) \downarrow}\ \text{enum}$$

$$\frac{e : \tau_1 \to \tau_2 \quad \text{app}_*(\Omega);\, (e\, d_i) \leadsto_\Omega F_i \quad \{d \mid (\Sigma,\, d) \in \Omega\} = \{d_1,\, \ldots,\, d_n\}, \quad n \leq \text{limit}}{\Omega;\, e \leadsto_\Omega (\ldots,\, \langle C_i,\, F_i \rangle,\, \ldots) \downarrow}\ \to$$

If the type of e is primitive, proceed to translating the expression's structure. At this point, Ω must be a singleton consisting of an empty Σ:

$$\frac{e : \tau, \quad \tau \text{ is primitive} \quad \cdot;\, e \leadsto F}{e \leadsto F}\ \text{prim} \qquad \frac{e : \tau, \quad \tau \text{ is primitive} \quad \cdot;\, e \leadsto F}{\{\cdot\};\, e \leadsto_\Omega F}\ \text{prim}$$

Note, that in the rule for a functional type ($\to$) the type of the argument must be finite and rather small. Therefore, we must be careful about translating expressions into **def**-structures. (There might be a case when we want to deal with infinite-domain functions applied to only a few fixed elements; we don't want to translate the entire function in that case.)

Rules for descending into the expression structure. First, we rip the extractors from e:

$$\frac{\Sigma,\, \mathcal{ET}_n;\, e \leadsto F}{\Sigma;\, \mathcal{ET}_n(e) \leadsto F}\ \text{tuple}_I \qquad \frac{\Sigma,\, \mathcal{ER}_f;\, e \leadsto F}{\Sigma;\, \mathcal{ER}_f(e) \leadsto F}\ \text{record}_I$$

$$\frac{\Sigma,\, \mathcal{EC}_c;\, e \leadsto F}{\Sigma;\, \mathcal{EC}_c(e) \leadsto F}\ \text{enum}_I \qquad \frac{\Sigma,\, \mathcal{EI};\, e \leadsto F}{\Sigma;\, \mathcal{EI}(e) \leadsto F}\ \text{index}_I$$

$$\frac{\Sigma,\, e';\, e \leadsto F}{\Sigma;\, (e\, e') \leadsto F}\ \text{apply}_I$$

Then, we try to put them back if it immediately simplifies the expression. In fact, if the expression is of one of these forms, it is an error for Σ to be empty or to contain a wrong extractor. (Of course, the only "legal" error that could happen here is the wrong datatype constructor; the rest must be ruled out by the typechecker.)

$$\frac{\Sigma;\, e_i \leadsto F_i}{\Sigma,\, \mathcal{ET}_i;\, (\ldots,\, e_i,\, \ldots) \leadsto F_i}\ \text{tuple}_E \qquad \frac{\Sigma;\, e \leadsto F}{\Sigma,\, \mathcal{ER}_f;\, \{\ldots,\, f = e,\, \ldots\} \leadsto F}\ \text{record}_E$$

$$\frac{\Sigma;\, e_f \leadsto F}{\Sigma,\, \mathcal{ER}_f;\, e \ \mathbf{with}\ [\ldots,\, f = e_f,\, \ldots] \leadsto F}\ \mathbf{with} \qquad \frac{f \notin \mathcal{L} \quad \Sigma,\, \mathcal{ER}_f;\, e \leadsto F}{\Sigma,\, \mathcal{ER}_f;\, e \ \mathbf{with}\ [\mathcal{L}] \leadsto F}\ \mathbf{with}_\emptyset$$

$$\frac{\Sigma;\, e \leadsto F}{\Sigma,\, \mathcal{EC}_c;\, (c\, e) \leadsto F}\ \text{enum}_E \qquad \frac{}{\Sigma,\, \mathcal{EI};\, (c\, e) \leadsto c}\ \text{index}_E$$

$$\frac{((\mathbf{fn}\ x \Rightarrow e)\ e') \Rightarrow e_1 \quad \Sigma;\ e_1 \rightsquigarrow F}{\Sigma,\ e';\ (\mathbf{fn}\ x \Rightarrow e) \rightsquigarrow F}\ \text{apply}_E$$

Next, we handle the compound expressions:

$$\frac{c_i = \mathbf{true} \rightsquigarrow C_i \quad c_i = \bot \rightsquigarrow C_i^{\bot} \quad \Sigma;\ e_i \rightsquigarrow F_i}{\begin{array}{l}\Sigma;\ \mathbf{if}\ c_1\ \mathbf{then}\ e_1\ \cdots\ \mathbf{elsif}\ c_n\ \mathbf{then}\ e_n\ \mathbf{else}\ e_{n+1}\ \mathbf{endif} \\ \quad \rightsquigarrow ((C_1,\ F_1),\ (C_1^{\bot},\ \bot),\ \ldots,\ (C_n,\ F_n),\ (C_n^{\bot},\ \bot),\ (\mathbf{true},\ F_{n+1}))\downarrow\end{array}}\ \mathbf{if}$$

$$\frac{e \rightsquigarrow E \quad \chi = \bot \rightsquigarrow C_{\bot} \quad \text{matches}(\chi,\ P_i) \rightsquigarrow C_i \quad \Sigma;\ e_i[\chi/e] \rightsquigarrow F_i}{\begin{array}{l}\Sigma;\ \mathbf{case}\ e\ \mathbf{of}\ P_1 \Rightarrow e_1 \mid \cdots \mid P_n \Rightarrow e_n\ \mathbf{endcase} \\ \quad \rightsquigarrow \mathbf{def}\ \chi = E\ \mathbf{in}\ ((C_{\bot},\ \bot),\ (C_1,\ F_1),\ \ldots,\ (C_n,\ F_n),\ (\mathbf{true},\ \bot))\ \mathbf{end}\downarrow\end{array}}\ \mathbf{case}$$

$$\frac{\Sigma;\ e \rightsquigarrow F}{\Sigma;\ \mathbf{let}\ \vec{\mathcal{D}}\ \mathbf{in}\ e\ \mathbf{end} \rightsquigarrow F}\ \mathbf{let}$$

The **let** rule should really translate the **let**-closure into a corresponding **def**-structure, but this calls for a great deal of modifications of the parser, the typechecker, and a host of other modules; in addition, it would change the semantics of SyMP as it is defined right now (to a more sane version, but still we want to avoid it for the time being).

Constants, boolean connectives, quantifiers, temporal and other built-in operators are handled specially. When a built-in operator is encountered, Σ must be empty.

`[ finish this ]`

Finally, when the expression becomes atomic, we start applying the extractors back to it, paying a special attention to the application:

$$\frac{e \text{ is atomic} \quad \Sigma;\ \cdot;\ e \rightsquigarrow^{\text{wrap}} F}{\Sigma;\ e \rightsquigarrow F}$$

$$\frac{\Sigma;\ \mathcal{D};\ \mathcal{ET}_n(e) \rightsquigarrow^{\text{wrap}} F}{\Sigma,\ \mathcal{ET}_n;\ \mathcal{D};\ e \rightsquigarrow^{\text{wrap}} F}\ \text{tuple} \qquad \frac{\Sigma;\ \mathcal{D};\ \mathcal{ER}_f(e) \rightsquigarrow^{\text{wrap}} F}{\Sigma,\ \mathcal{ER}_f;\ \mathcal{D};\ e \rightsquigarrow^{\text{wrap}} F}\ \text{record}$$

$$\frac{\Sigma;\ \mathcal{D};\ \mathcal{EC}_c(e) \rightsquigarrow^{\text{wrap}} F}{\Sigma,\ \mathcal{EC}_c;\ \mathcal{D};\ e \rightsquigarrow^{\text{wrap}} F}\ \text{enum} \qquad \frac{\Sigma;\ \mathcal{D};\ \mathcal{EI}(e) \rightsquigarrow^{\text{wrap}} F}{\Sigma,\ \mathcal{EI};\ \mathcal{D};\ e \rightsquigarrow^{\text{wrap}} F}\ \text{index}$$

$$\frac{d \text{ is const} \quad \Sigma;\ \mathcal{D};\ (e\ d) \rightsquigarrow^{\text{wrap}} F}{\Sigma,\ d;\ \mathcal{D};\ e \rightsquigarrow^{\text{wrap}} F}\ \text{apply}_{\text{const}} \qquad \frac{e' \text{ is not const} \quad e' \rightsquigarrow F' \quad \Sigma;\ \mathcal{D},\ \chi = F';\ (e\ \chi) \rightsquigarrow^{\text{wrap}} F}{\Sigma,\ e';\ \mathcal{D};\ e \rightsquigarrow^{\text{wrap}} F}\ \text{apply}$$

When Σ becomes empty, build the final **def**-structure:

$$\frac{}{\cdot;\ \mathcal{D};\ e \rightsquigarrow^{\text{wrap}} \mathbf{def}\ \mathcal{D}\ \mathbf{in}\ e\ \mathbf{end}}\ \text{final}$$

4.8.10 Translating other (unabstracted) built-in functions

Most of the built-in functions preserve the $\bot$ value (that is, $f(v_1,\ \ldots,\ \bot,\ \ldots,\ v_n) = \bot$), and therefore, can be handled uniformly:

$$\frac{e_1^i = \bot \rightsquigarrow C_{\bot}^i\ i = 1..n \quad \bot = e_2 \rightsquigarrow F_{\bot} \quad f(e_1^1,\ \ldots,\ e_1^n) = e_2 \rightsquigarrow_f F}{f(e_1^1,\ \ldots,\ e_1^n) = e_2 \rightsquigarrow (((\bigvee_{i=1}^n C_{\bot}^i) \to F_{\bot}) \wedge (\neg(\bigvee_{i=1}^n C_{\bot}^i) \to F))\downarrow}$$

where $f(e_1^1,\ \ldots,\ e_1^n) = e_2 \rightsquigarrow_f F$ is an implementation of the classical interpretation of f, where the arguments cannot be $\bot$. Note, that the result of f itself may still be $\bot$.

There are, however, a few built-in functions, namely, binary boolean connectives: **and**, **or**, **implies**, etc., and quantifiers, that do not respect $\bot$, and must be handled specially. Their semantics is the obvious "shortcut" semantics. For instance, if one of the arguments of **and** is **false**, then the entire expression is false, no matter whether there are arguments with the $\bot$ value or not.

The rule for built-in 3-valued equality. Note, that the built-in equality does not preserve the $\perp$ value, e.g., $\perp = \perp$ yields $\mathtt{true}$, and it always returns $\mathtt{true}$ or $\mathbf{false}$. Thus, we can use the 2-valued equality sequent for it directly, however, translate the result to 3-valued boolean type to compare with the other expression.

$$\frac{e_1 =_2 e_2 \rightsquigarrow F}{e_1 = e_2 \rightsquigarrow ((F, \mathbf{true}_3),\ (\mathbf{true}_2,\ \mathbf{false}_3)) \downarrow}\ =_3$$

The rules for boolean connectives.

$$\frac{\begin{array}{c}(\cdot;\ \mathbf{true}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^t \quad (\cdot;\ \mathbf{false}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^f \quad (\cdot;\ \perp) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^{\perp} \\ (\cdot;\ e_{1i}) = \mathbf{true} \rightsquigarrow^{\text{prim}}_{=} C_i^t \quad (\cdot;\ e_{1i}) = \mathbf{false} \rightsquigarrow^{\text{prim}}_{=} C_i^f \quad (\cdot;\ e_{1i}) = \perp \rightsquigarrow^{\text{prim}}_{=} C_i^{\perp}\end{array}}{(\cdot;\ e_{11}\ \mathbf{and}\ e_{12}) = e_2 \rightsquigarrow^{\text{prim}}_{=} (C_1^t \wedge C_2^t \to F^t) \wedge ((C_1^t \wedge C_2^{\perp}) \vee (C_1^{\perp} \wedge C_2^t) \to F^{\perp}) \wedge (C_1^f \vee C_2^f \to F^f) \downarrow}\ \mathbf{and}$$

$$\frac{\begin{array}{c}(\cdot;\ \mathbf{true}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^t \quad (\cdot;\ \mathbf{false}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^f \quad (\cdot;\ \perp) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^{\perp} \\ (\cdot;\ e_{1i}) = \mathbf{true} \rightsquigarrow^{\text{prim}}_{=} C_i^t \quad (\cdot;\ e_{1i}) = \mathbf{false} \rightsquigarrow^{\text{prim}}_{=} C_i^f \quad (\cdot;\ e_{1i}) = \perp \rightsquigarrow^{\text{prim}}_{=} C_i^{\perp}\end{array}}{(\cdot;\ e_{11}\ \mathbf{or}\ e_{12}) = e_2 \rightsquigarrow^{\text{prim}}_{=} (C_1^f \wedge C_2^f \to F^f) \wedge ((C_1^f \wedge C_2^{\perp}) \vee (C_f^{\perp} \wedge C_2^t) \to F^{\perp}) \wedge (C_1^t \vee C_2^t \to F^t) \downarrow}\ \mathbf{or}$$

$$\frac{\begin{array}{c}(\cdot;\ \mathbf{true}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^t \quad (\cdot;\ \mathbf{false}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^f \quad (\cdot;\ \perp) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^{\perp} \\ (\cdot;\ e_{1i}) = \mathbf{true} \rightsquigarrow^{\text{prim}}_{=} C_i^t \quad (\cdot;\ e_{1i}) = \mathbf{false} \rightsquigarrow^{\text{prim}}_{=} C_i^f \quad (\cdot;\ e_{1i}) = \perp \rightsquigarrow^{\text{prim}}_{=} C_i^{\perp}\end{array}}{(\cdot;\ e_{11}\ \mathbf{implies}\ e_{12}) = e_2 \rightsquigarrow^{\text{prim}}_{=} \left(\begin{array}{l}(C_1^t \wedge C_2^f \to F^f) \wedge ((C_1^t \wedge C_2^{\perp}) \vee (C_t^{\perp} \wedge C_2^t) \to F^{\perp}) \\ \wedge(C_1^f \vee C_2^t \to F^t)\end{array}\right) \downarrow}\ \mathbf{implies}$$

$$\frac{\begin{array}{c}(\cdot;\ \mathbf{true}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^t \quad (\cdot;\ \mathbf{false}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^f \quad (\cdot;\ \perp) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^{\perp} \\ (\cdot;\ e_1) = \mathbf{true} \rightsquigarrow^{\text{prim}}_{=} C^t \quad (\cdot;\ e_1) = \mathbf{false} \rightsquigarrow^{\text{prim}}_{=} C^f \quad (\cdot;\ e_1) = \perp \rightsquigarrow^{\text{prim}}_{=} C^{\perp}\end{array}}{(\cdot;\ \mathbf{not}\ e_1) = e_2 \rightsquigarrow^{\text{prim}}_{=} (C^t \to F^f) \wedge (C^{\perp} \to F^{\perp}) \wedge (C_1^f \to F^t) \downarrow}\ \mathbf{not}$$

And, finally, the quantifiers:

$$\frac{\begin{array}{l}\vec{x} : \vec{\tau} \\ (\cdot;\ \mathbf{true}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^t \quad (\cdot;\ \mathbf{false}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^f \quad (\cdot;\ \perp) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^{\perp} \\ (\cdot;\ e[\vec{d}/\vec{x}]) = \mathbf{true} \rightsquigarrow^{\text{prim}}_{=} C_{\vec{d}}^t \quad (\cdot;\ e[\vec{d}/\vec{x}]) = \mathbf{false} \rightsquigarrow^{\text{prim}}_{=} C_{\vec{d}}^f \quad (\cdot;\ e[\vec{d}/\vec{x}]) = \perp \rightsquigarrow^{\text{prim}}_{=} C_{\vec{d}}^{\perp}\end{array}}{(\cdot;\ \mathbf{forall}\ \vec{x} :\ e_1) = e_2 \rightsquigarrow^{\text{prim}}_{=} \begin{array}{l}(\bigwedge_{\vec{d}\in\mathcal{D}_{\omega(\vec{\tau})}} C_{\vec{d}}^t \to F^t) \\ \wedge((\bigvee_{\vec{d}\in\mathcal{D}_{\omega(\vec{\tau})}} C_{\vec{d}}^{\perp}) \bigwedge_{\vec{d}\in\mathcal{D}_{\omega(\vec{\tau})}} (C_{\vec{d}}^t \vee C_{\vec{d}}^{\perp}) \to F^{\perp}) \\ \wedge((\bigvee_{\vec{d}\in\mathcal{D}_{\omega(\vec{\tau})}} C_{\vec{d}}^f) \to F^f) \downarrow\end{array}}\ \mathbf{forall}$$

$$\frac{\begin{array}{l}\vec{x} : \vec{\tau} \\ (\cdot;\ \mathbf{true}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^t \quad (\cdot;\ \mathbf{false}) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^f \quad (\cdot;\ \perp) = e_2 \rightsquigarrow^{\text{prim}}_{=} F^{\perp} \\ (\cdot;\ e[\vec{d}/\vec{x}]) = \mathbf{true} \rightsquigarrow^{\text{prim}}_{=} C_{\vec{d}}^t \quad (\cdot;\ e[\vec{d}/\vec{x}]) = \mathbf{false} \rightsquigarrow^{\text{prim}}_{=} C_{\vec{d}}^f \quad (\cdot;\ e[\vec{d}/\vec{x}]) = \perp \rightsquigarrow^{\text{prim}}_{=} C_{\vec{d}}^{\perp}\end{array}}{(\cdot;\ \mathbf{exists}\ \vec{x} :\ e_1) = e_2 \rightsquigarrow^{\text{prim}}_{=} \begin{array}{l}(\bigwedge_{\vec{d}\in\mathcal{D}_{\omega(\vec{\tau})}} C_{\vec{d}}^f \to F^f) \\ \wedge((\bigvee_{\vec{d}\in\mathcal{D}_{\omega(\vec{\tau})}} C_{\vec{d}}^{\perp}) \bigwedge_{\vec{d}\in\mathcal{D}_{\omega(\vec{\tau})}} (C_{\vec{d}}^f \vee C_{\vec{d}}^{\perp}) \to F^{\perp}) \\ \wedge((\bigvee_{\vec{d}\in\mathcal{D}_{\omega(\vec{\tau})}} C_{\vec{d}}^t) \to F^t) \downarrow\end{array}}\ \mathbf{exists}$$

4.8.11 Top-level compound assignments

The main sequent for generating the transitions relation from the list of (possibly compound) assignments is

$$N;\ L;\ I \vdash AL \rightsquigarrow D,$$

where N, L, and I are the sets of primitive variables in the normal (combinational), next, and initial assignments respectively That is, it's the entire cone of influence, and only these variables need to be mentioned in the final assignment **def**-structure D.

In this subsection, we show how to split the compound assignments and assignment lists into atomic assignments, which are then handled by the sequent

$$V \vdash \sigma(\gamma) := e \rightsquigarrow F,$$

where V is the set of variables in the cone for this particular assignment, $\sigma \in \{\cdot,\ \mathbf{next},\ \mathbf{init}\}$ defines what type of assignment it is, γ is a state variable expression, and F is the result of the translation: an expression **def**-structure of a 2-valued boolean type.

$$\frac{\begin{array}{l} y_i \vdash y_i := \perp \rightsquigarrow W_i \quad x_j \vdash \mathbf{next}(x_j) := x_j \rightsquigarrow F_j \quad z_k \vdash \mathbf{init}(z_k) := z_k \rightsquigarrow T_k \\ \quad \text{for } i = 1..m,\ \ j = 1..n,\ \ k = 1..l \end{array}}{y_1, \ldots, y_m;\ x_1, \ldots, x_n;\ z_1, \ldots, z_l \vdash \mathbf{nop} \rightsquigarrow [(\bigwedge_{i=1}^m W_i)\downarrow;\ (\bigwedge_{j=1}^n F_j)\downarrow;\ (\bigwedge_{k=1}^l T_k)\downarrow]}\ \mathbf{nop}$$

$$\frac{e \Rightarrow e' \quad N \vdash \gamma := e' \rightsquigarrow F}{N;\ \cdot;\ \cdot \vdash \gamma := e \rightsquigarrow [F;\ \mathbf{true};\ \mathbf{true}]}\ \text{combAsst}$$

$$\frac{e \Rightarrow e' \quad L \vdash \mathbf{next}(\gamma) := e' \rightsquigarrow F}{\cdot;\ L;\ \cdot \vdash \mathbf{next}(\gamma) := e \rightsquigarrow [\mathbf{true};\ F;\ \mathbf{true}]}\ \mathbf{next}$$

$$\frac{e \Rightarrow e' \quad I \vdash \gamma := e' \rightsquigarrow F}{\cdot;\ \cdot;\ I \vdash \mathbf{init}(\gamma) := e \rightsquigarrow [\mathbf{true};\ \mathbf{true};\ F]}\ \mathbf{init}$$

$$\frac{N_1;\ L_1;\ I_1 \vdash A \rightsquigarrow D_1 \quad N_2;\ L_2;\ I_2 \vdash AL \rightsquigarrow D_2}{N_1 \cup N_2;\ L_1 \cup L_2;\ I_1 \cup I_2 \vdash A;\ AL \rightsquigarrow (D_1 \wedge D_2)\downarrow}\ \text{AsstList}$$

$$\frac{N;\ L;\ I \vdash AL \rightsquigarrow F}{N;\ L;\ I \vdash \mathbf{let}\ D_1\ \ldots\ D_n\ \mathbf{in}\ AL\ \mathbf{end} \rightsquigarrow F}\ \mathbf{let}$$

In an **if** statement we check the conditions sequentially until we hit one that is $\mathbf{true}$ or $\perp$. In the latter case, we assign all the variables the value $\perp$. The sequence of conditionals will be:

$$((c_1 = \mathbf{true},\ A_1),\ (c_1 = \perp,\ A_\perp),\ (c_2 = \mathbf{true},\ A_2) \cdots)$$

$$\frac{\begin{array}{l} N;\ L;\ I \vdash AL_i \rightsquigarrow F_i \quad c_i = \mathbf{true} \rightsquigarrow C_i \quad c_i = \perp \rightsquigarrow C_i^\perp \\ \quad N;\ L;\ I \vdash N := \perp;\ \mathbf{next}(L) := \perp;\ \mathbf{init}(I) := \perp \rightsquigarrow F_\perp \end{array}}{\begin{array}{l} N;\ L;\ I \vdash \mathbf{if}\ c_1\ \mathbf{then}\ AL_1\ \mathbf{elsif}\ c_2\ \mathbf{then}\ AL_2\ \ldots\ \mathbf{else}\ AL_{n+1}\ \mathbf{endif} \\ \quad \rightsquigarrow \mathbf{def}\ \ w_\perp = W_\perp,\ f_\perp = F_\perp,\ t_\perp = T_\perp,\ (\chi_1,\ \chi_1^\perp) = (C_1,\ C_1^\perp),\ \ldots,\ (\chi_n,\ \chi_n^\perp) = (C_n,\ C_n^\perp) \\ \quad\quad \mathbf{in}\ ((\chi_1,\ F_1),\ (\chi_1^\perp,\ F_\perp),\ (\chi_2,\ F_2),\ \ldots,\ (\mathbf{true},\ F_{n+1})) \\ \quad\quad \mathbf{end} \end{array}}\ \mathbf{if}$$

In a **case** statement, if the selector is $\perp$, then all the variables get assigned $\perp$. We add an extra implicit condition to check for that in the translation. Variables are assigned $\perp$ also when no pattern matches the selector, except for the next state variables, which keep their values unchanged. (Well, eventually the typechecker should make sure it does not happen, but it does not do it yet). The selector is assigned to a macro variable to guarantee the same nondeterministic choice for all the branches.

$$\frac{\begin{array}{l} e \rightsquigarrow E \quad \chi = \perp \rightsquigarrow C_\perp \quad \text{matches}(\chi,\ P_i) \rightsquigarrow C_i \quad \text{for } i = 1..n \\ \quad N;\ L;\ I \vdash \mathbf{nop} \rightsquigarrow D_0 \quad N;\ L;\ I \vdash N := \perp;\ \mathbf{next}(L) := \perp;\ \mathbf{init}(I) := \perp \rightsquigarrow D_\perp \\ \quad\quad N;\ L;\ I \vdash AL_i[\chi/e] \rightsquigarrow D_i \end{array}}{\begin{array}{l} N;\ L;\ I \vdash \mathbf{case}\ e\ \mathbf{of}\ P_1 \Rightarrow AL_1 \mid \ldots \mid P_n \Rightarrow AL_n\ \mathbf{endcase} \\ \quad \rightsquigarrow \mathbf{def}\ \chi = E\ \mathbf{in}\ ((C_\perp,\ D_\perp),\ \ldots,\ (C_i,\ D_i),\ \ldots,\ (\mathbf{true},\ D_0))\ \mathbf{end}\downarrow \end{array}}\ \mathbf{case}$$

A **choose** statement picks one of the branches where $e_i = \mathbf{true}$ and executes the corresponding assignments. If no branch has $e_i = \mathbf{true}$ (all guards are either **false** or $\perp$), then execute an implicit **nop** assignment. In case a **choose** statement has parameters, these are being instantiated with all their possible values, and the guards and assignments are replicated for each instantiation. So, in this rule we assume that all the parameters are already expanded.

$$\frac{\begin{array}{c} N;\ L;\ I \vdash AL_i \rightsquigarrow D_i \quad e \Rightarrow e' \quad e'_i = \mathbf{true} \rightsquigarrow C_i \text{ for } i = 1..n \\ N;\ L;\ I \vdash \mathbf{nop} \rightsquigarrow D_\perp \end{array}}{\begin{array}{c} N;\ L;\ I \vdash \mathbf{choose}\ e_1 \Rightarrow AL_1 \mid \ldots \mid e_n \Rightarrow AL_n\ \mathbf{chooseend} \\ \rightsquigarrow ((C_1,\, D_1) \mid \cdots \mid (C_n,\, D_n) \mid (\bigwedge_{i=1}^n \neg C_i,\, D_\perp)) \downarrow \end{array}}\ \mathbf{choose}$$

$$\frac{(x_1,\, \ldots,\, x_n) : \tau \quad N;\ L;\ I \vdash AL[\vec{d}/\vec{x}] \rightsquigarrow F_{\vec{d}} \quad \vec{d} \in \mathcal{D}_{\omega(\tau)}}{N;\ L;\ I \vdash \mathbf{foreach}\ x_1,\, \ldots,\, x_n : AL\ \mathbf{endforeach} \rightsquigarrow (\bigwedge_{\vec{d} \in \mathcal{D}_{\omega(\tau)}} F_{\vec{d}}) \downarrow}\ \mathbf{foreach}$$

Another side note: we only allow completely deterministic expressions as actual static parameters of modules. This way we can safely assume that a formal static parameter is a constant, and it is equal to itself at all times.

4.8.12 Atomic assignments

The translation of the atomic assignment:

$$V \vdash \sigma(\gamma) := e \rightsquigarrow F$$

is done in two big steps: translation of the expression e (w.r.t. V), and then splitting the assignment into primitive-type assignments and wrapping up the final **def**-structure:

$$\frac{V \vdash_\Omega \gamma \rightsquigarrow \Omega \quad \Omega;\, e \rightsquigarrow_\Omega E \quad V;\, \cdot \vdash \sigma(\gamma) \ll E \rightsquigarrow F}{V \vdash \sigma(\gamma) := e \rightsquigarrow F}$$

The first step involves translating the expression, which could still be of infinite type, and therefore we need to compute Ω from γ and V to restrict it to finite-size parts:

$$\frac{\cdot;\ \delta_i \vdash_\Omega \gamma \rightsquigarrow \Sigma_i}{\{\delta_1,\, \ldots,\, \delta_n\} \vdash_\Omega \gamma \rightsquigarrow \{\Sigma_1,\, \ldots,\, \Sigma_n\}}$$

$$\frac{\delta \approx \gamma}{\Sigma;\ \delta \vdash_\Omega \gamma \rightsquigarrow \Sigma} \qquad \frac{\Sigma, \mathcal{E};\ \delta \vdash_\Omega \gamma \rightsquigarrow \Sigma'}{\Sigma;\ \mathcal{E}(\delta) \vdash_\Omega \gamma \rightsquigarrow \Sigma'}$$

The second part of the translation descends into the γ's structure and takes care of the expressions used as array indices. Σ on the left hand side is the list of extractors ripped of γ, the order in it is important.

$$\frac{V;\, \Sigma,\, \mathcal{E} \vdash \sigma(\gamma) \ll E \rightsquigarrow F}{V;\, \Sigma \vdash \sigma(\mathcal{E}(\gamma)) \ll E \rightsquigarrow F}$$

$$\frac{\begin{array}{c} V \vdash \gamma \Rightarrow D \quad e' \rightsquigarrow E' \quad V;\, \Sigma,\, d \vdash \sigma(\gamma) \ll E \rightsquigarrow F_d \\ C_d = (d = E') \downarrow \quad U_d = (\sigma(\Sigma(\gamma\ d)) = E_\perp(\sigma, (\gamma\ d))) \downarrow \quad \text{for } d \in D \end{array}}{V;\, \Sigma \vdash \sigma(\gamma\ e') \ll E \rightsquigarrow (\ldots,\, (C_d,\, (F_d \wedge \bigwedge_{d' \neq d} U_{d'}) \downarrow),\, \ldots,\, ((\bigwedge \neg C_d) \downarrow,\, (\bigwedge U_d) \downarrow)) \downarrow}$$

where

$$E_\perp(\sigma,\, \gamma) = \begin{cases} \gamma & \textbf{if } \sigma = \mathbf{next} \\ \perp & \text{otherwise} \end{cases}$$

The final rule wraps back all the extractors on the state variable and translates its equality with the expression's **def**-structure. It is important to turn off all the abstractions in the last translation; even if the type of the assignment is abstracted, the equality should be treated in its standard interpretation, not an abstract one.

$$\frac{x \text{ is atomic}}{V;\, \Sigma \vdash \sigma(x) \ll E \rightsquigarrow (\sigma(\Sigma(x)) = E) \downarrow}$$

The rules for generating D are the following:

$$\frac{V \vdash \gamma \Rightarrow D \quad (\delta \preceq \gamma) \Rightarrow d}{V, \delta \vdash \gamma \Rightarrow D \cup \{d\}} \qquad \frac{V \vdash \gamma \Rightarrow D \quad (\delta \preceq \gamma) \Rightarrow \emptyset}{V, \delta \vdash \gamma \Rightarrow D}$$

$$\frac{\delta \approx \gamma}{((\delta\ d) \preceq \gamma) \Rightarrow d} \qquad \frac{\delta \not\approx \gamma \quad (\delta \preceq \gamma) \Rightarrow d'}{((\delta\ d) \preceq \gamma) \Rightarrow d'} \qquad \frac{(\delta \preceq \gamma) \Rightarrow d}{(\delta.f \preceq \gamma) \Rightarrow d}$$

$$\frac{(\delta \preceq \gamma) \Rightarrow d}{(\mathcal{ET}_n(\delta) \preceq \gamma) \Rightarrow d} \qquad \frac{(\delta \preceq \gamma) \Rightarrow d}{(\mathcal{EC}_c(\delta) \preceq \gamma) \Rightarrow d} \qquad \frac{x \text{ is atomic}}{(x \preceq \gamma) \Rightarrow \emptyset}$$

4.9 Translating for a Back-End Model Checker

The purpose of this module is to translate the transition relation represented as a **def**-structure into an input language of a particular model checker. Of course, for each model checker there must be a separate translator, however, all of them will have some features in common. We describe a few of such features to keep in mind on the example of SMV (the only back-end implemented in this version).

The transition relation is represented as a list of top-level **def**-structures. One easy way to convert a **def**-structure into SMV is to declare the definitions as either "**DEFINE**" macros, or new "normal" state variables, depending on whether we want to preserve the nondeterministic choices. The safe trade-off is to check whether a definition is deterministic, and if it is, use "**DEFINE**", and otherwise declare a new state variable and assign the definition to it combinationally (as a "normal" assignment).

The problem with defining new state variables is that the resulting system fails to remain bisimilar to the original SyMP input, however it still remains trace-equivalent. The reason is that, according to the SyMP semantics, every nondeterministic choice in each definition is available in the current state before the transition, whereas after the translation such a choice would be made at the time of previous transition and then is predetermined by the current state.

If one needs to preserve the bisimilarity in the translation, then about the only option, at least for SMV, is to instantiate the body of each **def**-structure with each nondeterministic choice in the definitions and "or" them together. This may result in an exponential blow-up in the size of the formula. A more practical alternative to that is to modify the model checker to have *constrained input variables* for the definitions that would not be a part of the state. The defined names in the **def**-structure would become such inputs, and their definitions would become their constraints. Then the model checker would use these inputs with constraints to restrict the next state computation, but will quantify the inputs out at the end.

More formally, given the state variables $\vec{s}$, input variables $\vec{i}$ with a boolean formula $C(\vec{i}, \vec{s})$ representing their constraints, and a transition relation T, the image of a set S will be computed by the formula:

$$\mathrm{Img}(S)(\vec{s'}) = \exists \vec{s}. \exists \vec{i}. S(\vec{s}) \wedge C(\vec{i}, \vec{s}) \wedge T(\vec{s}, \vec{s'})$$

References

[1] Sergey Berezin. Proof system for the first order μ-calculus: Combining model checking with theorem proving. Unpublished manuscript, November 1999.