

Scunak 1.0 Users Manual

Chad E. Brown



First and foremost, Scunak would not exist if I had not spent a year of my life writing it, so I thank myself. Also, first, I thank my partner Elena Tregubova for emotionally supporting me during this era of my life. Now, for the professional acknowledgements: Fulya Horozal was the first Scunak user (other than myself) and her experience directly led to a number of changes (such as the inclusion of coercions in the PAM syntax). Numerous conversations with my colleagues influenced the Scunak tutor. Most of these conversations were with Magdalena Wolska, but others included Marvin Schiller, Dominik Dietrich, and Dimitra Tsovaltzi. George Gogvadze gave me some helpful advice regarding the Scunak tutor over email just before the release of Scunak 1.0. I also had several conversations with Alberto González regarding, among other topics, parsing. Finally, I want to thank the Ω mega group in general (and Jörg Siekmann, Christoph Benzmüller, and Serge Autexier, in particular) for supporting me during the time I wrote Scunak.

July 11, 2006

Contents

Chapter 1. Introduction	5
1. Installing Scunak	5
2. Starting Scunak	6
3. A First Scunak Session	7
4. How People Can Use Scunak	8
5. How Programs Can Use Scunak	9
6. How Scunak Can Use Programs	9
7. A Quick Tutoring Example	10
Chapter 2. Representing Mathematics in Scunak	17
1. Sets	17
2. Propositions	17
3. Proof Types	18
4. Constructing Sets Using Properties	18
5. Quantifiers	18
6. Declaring Definitions and Claims	18
7. Working with Pairs	19
8. Working with Functions	19
Chapter 3. Scip: The Scunak Interactive Prover	21
1. Scip commands for printing information	22
2. Telling Scip you think you're done	22
3. Asserting a new fact	22
4. Making Temporary Claims in Scip	22
5. The Scip intro command	23
6. Unfolding Definitions	23
7. Reducing Goals	23
8. Example: Distributivity of Union over Intersection	23
Chapter 4. The Scunak Type Theory	33
1. Example of Type Checking	34
Chapter 5. The PAM Syntax	37
Chapter 6. The Scunak Tutor	41
1. WOZ1: Sets	41
2. WOZ2: Binary Relations	43
Chapter 7. Proofreading With Scunak	47
1. Proofreading The First Bartle Sherbert Proof	47
2. Extending the Proofreader	53

Appendix A. API	55
1. Specifying Data	55
1.1. Names	55
1.2. Generic Application	55
1.3. Binary Operators	55
1.4. Binders	56
1.5. Extraction and Normal Terms	56
1.6. Dependent Types	57
1.7. PAM vs. S-PAM	57
2. Main Sockets	59
2.1. Top Level IO	59
2.2. Scip Top Level IO	61
2.3. Tutor Top Level IO	63
2.4. Prompting for Booleans	65
2.5. Prompting for Numbers	65
2.6. Prompting for Names	65
2.7. Prompting for Lines	65
2.8. Prompting for Filenames	65
2.9. Prompting for Extraction Terms	65
3. Input Agents	65
4. Output Agents	65
5. Fill Gap Agents	66
Appendix B. Socket Interaction Examples	67
1. WOZ1 Examples	67
1.1. First WOZ1 Example	68
1.2. Second WOZ1 Example	70
1.3. Third WOZ1 Example	71
1.4. Fourth WOZ1 Example	73
1.5. Fifth WOZ1 Example	74
2. WOZ2 Examples	77
2.1. WOZ2 Preparation Example	78
2.2. WOZ2 Warmup Example	83
2.3. WOZ2 Exercise A	85
2.4. WOZ2 Exercise B	88
Appendix. Bibliography	91

CHAPTER 1

Introduction

Scunak 1.0¹ is a mathematical assistant based on set theory formalized in a dependent type theory with proof terms. The purpose of Scunak is to formalize mathematics from an axiomatic foundation (a form of set theory). There are many good reasons to formalize mathematics. One reason is good enough for me: I want to do it, or at least see it done. Scunak can also be used as an interactive prover, as a tutor for mathematical proofs, and (to some degree) as a proofreader for proofs in L^AT_EX.

WARNING: This documentation is, at best, incomplete.

1. Installing Scunak

We assume a Unix-style operating system.

Scunak has been tested using two versions of Lisp: Allegro (version 6.0, but other recent versions should be fine) and CLISP. Allegro is available at <http://www.franz.com/> and CLISP is available (for free) at <http://clisp.cons.org/>.

We assume lisp invokes Allegro Common Lisp. This is likely to correspond to the executable `alisp8` in recent distributions of Allegro.

We assume clisp invokes CLISP.

The script `install` distributed with Scunak can create both an Allegro and a CLISP version of Scunak. In both cases, the corresponding lisp loads the file `make.lisp`. The file `make.lisp` depends on three parameters which users may want to reset:

- `*dxldir*` This string indicates the directory where the image files should be created.
- `*execdir*` This string indicates the directory where the script files for invoking Scunak should be created.
- `*datadir*` This string indicates the directory where the global Scunak data is stored. This should correspond to the distributed `data` directory.

Loading `make.lisp` compiles and loads the source files and creates image files `scunak-allegro.dxl` and `scunak-clisp.mem` in the `*dxldir*` directory. If there do not already exist script files `scunak-acl` and `scunak-clisp` in the `*execdir*` directory, then such script files will be created. (Note: You may need to modify the script files. In particular, some versions of `scunak-acl` need the line `setenv LD_ASSUME_KERNEL 2.4.0` before lisp is invoked, but others probably do not.)

If you only want to install Scunak using Allegro Lisp, then start Allegro lisp and load `make.lisp`.

¹Copyright Chad E. Brown 2006

If you only want to install Scunak using CLISP, then start CLISP and load `make.lisp`.

2. Starting Scunak

A user starts Scunak by starting lisp with the appropriate Scunak image file and giving command line options. Usually one does this by calling a script. For instance, the script `scunak-clisp` created by `make.lisp` will contain a line

```
clisp -M ~/scunak/scunak-clisp.mem -- -d ~/scunak/data/ $*
```

Executing the script `scunak-clisp` with command line options will start `clisp` with the `scunak-clisp.mem` image with data directory `~/scunak/data/`. The command line options sent to the script will be sent to `clisp`.

We will refer to the main executable for Scunak as `scunak`. This may actually be `scunak-acl` or `scunak-clisp`. Of course, one can copy or link the preferred version of Scunak to have the name `scunak`.

Some command line options are:

- `-k kernelname` Give the name of a “kernel” signature. A “kernel” signature usually contains basic constructions (e.g., set operations, pairs, relations, functions) and facts about these constructions. The options at the moment are `mu`, `macu`, `f5`, `ijcar2006`, or `none`. (`none` starts Scunak with no kernel.) If no kernel is specified, then a default kernel is chosen (at the moment the default is `macu`).
- `-p lispfilenames` Give several lisp files to load before checking the main file.
- `-f mainlispfilenames` Give several PAM files containing declarations to typecheck and add to the signature.
- `-v` Be verbose.
- `-V` Be very verbose.
- `-C` Use colors. Colors should work in ordinary xterms, or in emacs shells if you either:
 - (1) Add `(autoload 'ansi-color-for-comint-mode-on "ansi-color" nil t)` and `(add-hook 'shell-mode-hook 'ansi-color-for-comint-mode-on)` to your `.emacs` file. *OR*
 - (2) Within emacs, do `M-x load-library RET ansi-color RET` and then `M-x ansi-color-for-comint-mode-on`.The colors look better against a black background. WARNING: Colors can slow things down.
- `-d directory` Data directory for Scunak. Scunak looks in this directory for `pam` files and `lisp` files if the file is not in the current working directory already.
- `-n name` Give your name to personalize Scunak’s output a bit.
- `-m machinename` Give a remote machine name for connecting to a remote passive socket for remote communication.
- `-s socketnum` Give a remote socket number for connecting to a remote passive socket for remote communication.

3. A First Scunak Session

Here we give a first Scunak Session. Start Scunak using one of the following commands:

- `scunak -k f`
- `scunak-acl -k f`
- `scunak-clisp -k f`

This starts Scunak with the basic set theory kernel named `f` (`f` stands for “finite” since there is no axiom of infinity in `f-kernel`, but enough axioms to construct all Hereditarily finite sets).

At the top level command line interface, declare three local variables `A`, `B`, and `C` corresponding to sets A , B and C :

```
>[A:set]
>[B:set]
>[C:set]
```

Suppose we want to prove $A \subseteq (A \cup B)$. In Scunak’s PAM syntax (see Chapter 5), we write this proposition as $(A \leq (A \cup B))$. If Scunak already knows this fact, we can find the proof term using the top level command `justify`.

```
>justify (A <= (A \cup B))
```

Proof Term:

```
(binunionLsub A B)
```

The command justify does not end in a period. Most commands do end in a period. If Scunak says it “does not understand” then look for such syntactic problems first.

That is, Scunak (with the `f` kernel) already knows this fact. The proof term is given by `(binunionLsub A B)` where `binunionLsub` is an abbreviation (or, lemma) in the `f-kernel` signature. The proof term `(binunionLsub A B)` has type $\vdash (A \leq (A \cup B))$ (i.e., “proof of $A \subseteq (A \cup B)$ ”).

```
>typeof (binunionLsub A B)
```

```
Type: |- (A<=(A \cup B))
```

We can similarly justify $(B \subseteq (A \cup B))$:

```
>justify (B <= (A \cup B))
```

Proof Term:

```
(binunionRsub A B)
```

However, Scunak (with the `f` kernel) cannot justify $B \subseteq (A \cup (B \cup C))$:

```
>justify (B <= (A \cup (B \cup C)))
```

```
Could not justify prop
```

We instead make a new claim for this proposition:

```
(myclaim A B C):|- (B <= (A \cup (B \cup C)))?
```

Now we can use the interactive prover to prove `myclaim` (see Chapter 3 for more information):

```
Give name for obj>A
```

```
Give name for obj>B
```

```
Give name for obj>C
```

```
>pplan
```

```
Support (Objects, Assumptions and Derived Facts in Context):
```

```
A:obj
```

```

B:obj
C:obj
Goal (What you need to show): |- (B<=(A \cup (B \cup C)))
>intro
OK
Give name for (in B)>x
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
A:obj
B:obj
C:obj
x:(in B)
fact0: |- (x::B)
Goal (What you need to show): |- (x::(A \cup (B \cup C)))
>clearly
Enter Proposition>(x::(B \cup C))
Correct.
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
A:obj
B:obj
C:obj
x:(in B)
fact0: |- (x::B)
fact1: |- (x::(B \cup C))
Goal (What you need to show): |- (x::(A \cup (B \cup C)))
>d
Done with subgoal!
Output PAM Proof Term to File [y/n, Default y]>
Filename [Default File myproofs.pam]>
Output Lisp Proof Term to File [y/n, Default y]>
Filename [Default File myproofs.lisp]>
Scip Out!

```

Upon completing the proof, a PAM version and Lisp version of the proof is (optionally) saved in a file and the user returns to the top level. A call to `help` reveals that `myclaim` is now an abbreviation (i.e., has a proof):

```

>help myclaim
myclaim is an abbreviation of type:
{x0:obj}{x1:obj}{x2:obj}|- (x1<=(x0 \cup (x1 \cup x2)))
Defn: (\x0 x1 x2.subsetI1 x1 (x0 \cup (x1 \cup x2)) (\x3.binunionIR x0 (x1 \cup x2) x3 (bi
myclaim is essentially a derived proof rule.
Due to proof irrelevance, the abbreviation never needs to be expanded.

```

Finally, we quit:

```
>q
```

4. How People Can Use Scunak

The usual way to use Scunak is as follows:

- (1) Create a file (e.g., `all.pam`) which contains a list of load commands. For example,


```
load "file1.pam"!
load "file2.pam"!
load "file3.pam"!
```
- (2) Create PAM files `file1.pam` which contain the definitions of sets and propositions, as well as claims to be proven.
- (3) Start Scunak using `scunak -f all.pam` so that your PAM files are loaded. If your PAM files are correct, Scunak begins in an interactive top level and waits for your command.
- (4) Use `all-claims` to obtain a list of open claims to prove.
- (5) Use `prove <claimname>` to enter the top level for Scunak interactive prover Scip and begin proving the claim.
- (6) Use Scip to prove the claim. Upon success, the proof term will be printed to the screen and you will be returned to the Scunak top level. **Make sure to save the proof term** in one of your PAM files (after the declaration of the claim) as follows


```
<claimname>:=<proofterm>.
```
- (7) Prove another claim, or leave Scunak using the `q` command.

5. How Programs Can Use Scunak

Other programs can start Scunak and communicate with Scunak via sockets. The `-m` and `-s` command line options are used for this purpose. For example,

- `scunak -m hostname -s 32800`

would start Scunak and Scunak would immediately connect to a passive (server) socket using port number 32800 at the machine `hostname`. Scunak will connect to this socket twice, resulting in two sockets: socket **A** (the first connection) and socket **B** (the second connection). Scunak uses socket **A** to receive input from the server and to send S-expression output. Socket **B** sends strings representing the literal output of Scunak. For more information about the input and output of Scunak using sockets, see Appendix A.

6. How Scunak Can Use Programs

Scunak can also use sockets to connect to other programs which can provide certain services. In particular,

A user can ask Scunak to request input to add elements to the signature as follows:

- `input-sig-agent "hostname" "32800"`

For example, the socket may connect to a program at port number 32800 on host `hostname` which can parse PAM files (or some future syntax) much faster than the Earley parser implemented in Scunak. One can also send extra information as a string:

- `input-sig-agent "hostname" "32800" "info"`

Upon request, Scunak can output the signature (or parts of a signature) through a socket using `output-sig-agent`. This command has the format

```
output-sig-agent "MACHINE" "PORTNUM" [ "INFO" [ "BEGINTIMESTAMP" [ "ENDTIMESTAMP" ] ] ] .
```

This could be used, for example, to send the signature to a program that can translate the information into Automath, Twelf, \LaTeX , HTML, or OMDOC [14].

The other main service for which Scunak can use an external agent is “filling gaps”. When one uses `justify` at the top level, or any number of commands in the Scip or tutor top level, Scunak must try to determine if there is an “easy” way to create a term of a certain proof type `pf A` in a given context Γ . Using the command `add-fill-gap-agent` or the command `add-fill-gap-agent-usable` one can instruct Scunak to connect to a socket and send this socket any “fill gap” requests. If the external agent returns NIL, then the “fill gap” request is sent to the next external agent (or Scunak tries to fill the gap itself).

The difference between `add-fill-gap-agent` and `add-fill-gap-agent-usable` is that a “usable” set of signature elements is explicitly sent through the socket if `add-fill-gap-agent-usable` is used. Otherwise, only the context Γ and the proof type `pf A` is sent through the socket.

A fill gap agent can be removed (by name) using `remove-fill-gap-agent`. The format for using these commands is as follows:

```
add-fill-gap-agent NAME "MACHINE" "PORTNUM"
add-fill-gap-agent-usable NAME "MACHINE" "PORTNUM"
remove-fill-gap-agent NAME
```

7. A Quick Tutoring Example

We show how Scunak can be used to give information useful for tutoring on the example problem $A \cup B = B \cup A$. We start Scunak with the following command line options:

```
scunak -k macu -p bool-props-sets-sm -f setrules.pam bool-props-sets.pam
bool-props-sets-sm loads the lisp file data/bool-props-sets-sm.lisp giving the “student model” (i.e., sets the value of some global variables). setrules.pam contains a rule for arguing by cases when we know  $x \in A \cup B$ . bool-props-sets.pam contains a series of claims about binary intersection and binary union.
```

On the top level, we can set some parameters for the tutor. (These may be already set in `data/bool-props-sets-sm.lisp`, but we make the values explicit here.) `tutor-auto-back` lists the rules that the student is expected to eagerly apply backwards. In our case, we assume the student immediately reduces proving two sets are equal to proving the two subset directions (without explicitly noting this step). `tutor-student-usable` sets the rules that the tutor and student are both expected to know. One can also use `tutor-only-usable` to indicate rules that the tutor can use, but the student cannot.

```
>tutor-auto-back settextsub.
>tutor-student-usable notE contradiction subsetI1 subsetI2
binintersectEL binintersectER binintersectI binunionIL binunionIR
binunionE binunionCases emptysetsubset subsetemptysetimpeq.
```

(The line breaks are only for readability, do not hit return in Scunak until the list is complete.)

Still on the top level, we declare two sets A and B :

```
>[A:set]
>[B:set]
```

Now we invoke the tutor top level on the conjecture:

```
>tutor (unionComm A B)
```

Hello, Dave. Please try to prove the following:

```
|- ((A \cup B)==(B \cup A))
```

Now the student can enter a series of `let`, `assume`, `clearly`, and `qed` commands to successfully solve the problem. We also include possible mistakes a student may make.

We begin by assuming x is an object in $A \cup B$.

```
Dave> let x:obj.
```

OK

```
Dave> assume (x::(A \cup B)).
```

OK

We argue by cases (technically, this corresponds to the rule `binunionCases` which is defined in the file `data/setrules.pam`).

```
Dave> assume (x::A).
```

OK

Suppose the student is confused and thinks this implies $x \in B$ holds. Scunak rejects this input.

```
Dave> clearly (x::B).
```

Not OK.

I'm afraid that doesn't follow, Dave.

Now the student realizes his mistake and gives the correct conclusion.

```
Dave> clearly (x::(B \cup A)).
```

OK

```
Dave> qed.
```

OK

Good Dave, you're done with this part of the proof, but there is more to do.

Next the student should consider the case where x is in B . Suppose the student is confused and tries to assume x is in A again. Scunak rejects this possibility.

```
Dave> assume (x::A).
```

Not OK.

I could not find any reason for you to make such an assumption, Dave.

The student makes the correct assumption of $x \in B$. Since the subgoal in this case is to show $x \in (B \cup A)$ holds, the student can simply indicate the subgoal is complete using `qed`.

```
Dave> assume (x::B).
```

OK

```
Dave> qed.
```

OK

Good Dave, you're done with this part of the proof, but there is more to do.

Now the student should show the other direction of subset. Again, Scunak knows the student should do the other direction, so the student cannot make an inappropriate assumption.

```
Dave> let x:obj.
```

OK

```
Dave> assume (x::(A \cup B)).
```

Not OK.

I could not find any reason for you to make such an assumption, Dave.

The student makes the correct assumption for this direction and then argues by cases as before.

```
Dave> assume (x::(B \cup A)).
OK
Dave> assume (x::A).
OK
Dave> qed.
OK
Good Dave, you're done with this part of the proof, but there is more to do.
Dave> assume (x::B).
OK
Dave> qed.
Congratulations, you're done with the proof, Dave!
Successful Term:
(\x0 x1.settextsub (x0 \cup x1) (x1 \cup x0)
 (subsetI2 (x0 \cup x1) (x1 \cup x0)
  (\x2 x3.binunionCases x0 x1 x2 (x2::(x1 \cup x0))
   x3 (binunionIR x1 x0 x2) (binunionIL x1 x0 x2)))
 (subsetI2 (x1 \cup x0) (x0 \cup x1)
  (\x2 x3.binunionCases x1 x0 x2 (x2::(x0 \cup x1)) x3
   (binunionIR x0 x1 x2) (binunionIL x0 x1 x2))))
```

One can also use Scunak as an external component in a larger tutoring system. Suppose there is a server waiting for a connection on host `hostname` with port number 33800. We start Scunak with these options.

```
scunak -k macu -p bool-props-sets-sm -f setrules.pam bool-props-sets.pam
      -m hostname -s 33800
```

Scunak connects to this server twice, giving two sockets **A** and **B**. We repeat the tutoring example above indicating what information Scunak should receive via socket **A**, what Scunak sends through sockets **A** and **B**. Upon initialization, socket **A** sends

```
SCUNAK
1.0
READY
```

Socket **B** sends a sequence of lines including the line

```
Scunak Text Output Socket
```

```

A gets (TUTOR-AUTO-BACK "settextsub")
A sends READY
A gets (TUTOR-STUDENT-USABLE "notE" "contradiction"
      "subsetI1" "subsetI2" "binintersectEL"
      "binintersectER" "binintersectI" "binunionIL"
      "binunionIR" "binunionE" "binunionCases"
      "emptysetsubset" "subsetemptysetimpeq")

A sends OK
A sends READY
A gets (LET "A" OBJ)
A sends READY
A gets (LET "B" OBJ)
A sends READY
A gets (TUTOR ("unionComm" "A" "B"))
B sends Hello, Chad. Please try to prove the following:
B sends |- ((A \cup B)==(B \cup A))
A sends READY-TUTOR
A gets (LET "x" OBJ)
A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME ("x" IN ("A" CUP "B")))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME ("x" IN "A"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY ("x" IN "B"))
A sends NOT-OK
B sends Not OK.
B sends I'm afraid that doesn't follow, Chad.
A sends TUTOR-STATUS-UNCHANGED
A sends READY-TUTOR
A gets (CLEARLY ("x" IN ("B" CUP "A")))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends OK
B sends OK
B sends Good Chad, you're done with this part of the proof, but there is more to do.
A sends READY-TUTOR

```

A gets (ASSUME ("x" IN "A"))
A sends NOT-OK
B sends Not OK.
B sends I could not find any reason for you to make such an assumption, Chad.
A sends TUTOR-STATUS-UNCHANGED
A sends READY-TUTOR
A gets (ASSUME ("x" IN "B"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends OK
B sends OK
B sends Good Chad, you're done with this part of the proof, but there is more to do.
A sends READY-TUTOR

A gets (LET "x" OBJ)
A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME ("x" IN ("A" CUP "B")))
A sends NOT-OK
B sends Not OK.
B sends I could not find any reason for you to make such an assumption, Chad.
A sends TUTOR-STATUS-UNCHANGED
A sends READY-TUTOR
A gets (ASSUME ("x" IN ("B" CUP "A")))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME ("x" IN "A"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends OK
B sends OK
B sends Good Chad, you're done with this part of the proof, but there is more to do.
A sends READY-TUTOR
A gets (ASSUME ("x" IN "B"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends STUDENT-SUCCESSFUL
B sends Congratulations, you're done with the proof, Chad!
B sends Successful Term:
B sends ($x_0 \cup x_1$.setextsub ($x_0 \cup x_1$) ($x_1 \cup x_0$) (subsetI2 ($x_0 \cup x_1$) ($x_1 \cup x_0$)))
A sends EXIT-TUTOR
A sends READY

CHAPTER 2

Representing Mathematics in Scunak

Scunak supports a variety of set theories in which Mathematics can be encoded. The particular set theory depends on which kernel you use when you start Scunak (see the `-k` option). The points we describe here are common to all the current kernels. We describe information that can either be put into PAM files or given interactively as commands to the Scunak top level. Here we describe elements of the PAM syntax as needed. More information can be found in Chapter 5.

1. Sets

First, we assume all objects are sets, so we use the terms “set” (or `set`) and “object” (or `obj`) interchangeably. You can add sets/objects temporarily to the context as follows:

```
[A:set]
[B:set]
[x:obj]
[y:obj]
```

In the “context” above, one can construct several new sets using set constructors. Examples:

- (1) `(powerset A)` - the powerset of A
- (2) `{x}` - the set containing x
- (3) `{x,A}` - the set containing x and A
- (4) `{x,A,B}` - the set containing x , A , and B
- (5) `{}` - the empty set
- (6) `(A \cup B)` - the union of A and B
- (7) `(A \cap B)` - the intersection of A and B
- (8) `⟨x,y⟩` - the ordered pair of x and y
- (9) `(A \times B)` - the set of pairs of elements of A with elements of B
- (10) `(funcSet A B)` - the set of functions from A to B

2. Propositions

The type `prop` represents propositions. Assuming the context above, one can form propositions as follows:

- (1) `(x:A)` - $x \in A$ (x is a member of A)
- (2) `(x==y)` - $x = y$ (x equals y)
- (3) `(A <= B)` - $A \subseteq B$ (A is a subset of B)

Since `obj` and `set` are really the same, the propositions `(A::x)` (representing $A \in x$), `(A==B)` and `(x==A)` are all of type `prop`.

Just as one can add sets to a context, one can add propositions (temporarily) to the context.

[M:prop]

[N:prop]

Given M and N in context, one can form new propositions using the usual logical constructors:

- (1) (not M) - the negation of M
- (2) (M & N) - the conjunction of M and N
- (3) (M | N) - the disjunction of M and N
- (4) (M => N) - M implies N
- (5) (M <=> N) - M and N are equivalent

3. Proof Types

Given any proposition, one can form the type of “proofs” of this proposition. This type will be nonempty iff there is a proof of the proposition. The concrete syntax for such a type is $| - M$ where M is a proposition.

4. Constructing Sets Using Properties

Another common set constructor in Mathematics is $\{x \in A | \varphi(x)\}$ where A is a given set and $\varphi(x)$ is a “property” depending on x . In PAM syntax, one can write $\{x:A | \text{phi}\}$ where **phi** is a proposition that may depend on x (and use the fact that x is assumed to be in A). For example, one could write $\{x:A | (x : B)\}$. (This is how binary intersection is defined.)

5. Quantifiers

Many propositions are formed using the quantifiers \forall and \exists . In the current Scunak kernels, these quantifiers are always relativized to domain sets. That is, one can represent propositions of the form $(\forall x \in A \varphi(x))$ or $(\exists x \in A \varphi(x))$, but not $(\forall x \varphi(x))$ or $(\exists x \varphi(x))$. The syntax for these propositions are (**forall** $x:A$. **phi**) and (**exists** $x:A$. **phi**) where **phi** is a proposition depending on x (as a member of A).

Note: There must be a space before and after the . in the syntax above! Parenthesis and dots are vital for parsing in Scunak!

In addition to sets and propositions, one can put generic members of a given set in context. For example:

[A:set]

[x:A]

[X:(powerset A)]

Here A is a temporary set we are considering, x is a temporary member of A , and X is a temporary member of the powerset of A . We can then freely use the variables in the rest of the PAM file (unless they are masked by new declarations).

6. Declaring Definitions and Claims

Putting sets or propositions (or other local variables) in context is temporary. The information in PAM files which is in memory after reading the file are the *declarations*. There are two kinds of declarations that are important: definitions and claims.

A definition is declared as follows:

(<name> <localvars>):<type>=<defn>.

A claim is declared as follows:

```
(<name> <localvars>):<type>?
```

Usually a claim is of proof type (but need not be).

Consider the following potential content of a PAM file `file1.pam`

```
[A:set]
[B:set]
[x:obj]
(myclaim A B x):|- ((x::(A \cup B)) => (x::(B \cup A)))?

(myprop A B):prop=(forall x:A . (not (x::B))).

(myclaim2 A B x):|- (((myprop A B) & (x::B)) => (not (x::A)))?
```

In this file, `myclaim` and `myclaim2` are claims depending on three objects A , B and x and have proof types. The type of `(myclaim A B x)` is the type of proofs of the proposition $((x \in (A \cup B)) \Rightarrow (x \in B \cup A))$. Meanwhile, `myprop` is defined to be a proposition depending on two objects A and B . The proposition `(myprop A B)` represents $(\forall x \in A. x \notin B)$ (intuitively, this means A and B are disjoint).

In principle, one could use the Scunak interactive prover to prove `myclaim` and `myclaim2` and then enter the resulting proof terms into the file. This would make `myclaim` and `myclaim2` definitions instead of claims. (Typically, the goal is to turn claims into definitions by giving proofs.)

7. Working with Pairs

An ordered pair $\langle x, y \rangle$ is represented by `<<x,y>>` where x and y are objects. Sometimes one wants to consider pairs $\langle x, y \rangle$ where $x \in A$ and $y \in B$. `(A \times B)` represents the set of such pairs. One can represent a set of pairs $\{\langle x, y \rangle \in A \times B \mid \varphi(x, y)\}$ using `{<<x,y>>:(A \times B) | phi}`.

8. Working with Functions

The recommended way to work with functions in Scunak is to use the set `(funcSet A B)` of functions from A to B . For instance, one can put such functions into context as follows:

```
[A:set]
[B:set]
[f:(funcSet A B)]
[g:(funcSet A B)]
```

In order to make use of functions, one must be able to apply a function to an argument. The “official” syntax for doing this in Scunak is a bit long. One uses a constructor called `ap2`, giving the domain and codomain sets, the function and the argument. Extend the context above to include

```
[x:A]
```

We can represent $f(x)$ in PAM syntax using `(ap2 A B f x)`. The idea is that `ap2 A B` is the application operator for functions from A to B . The type of the term `(ap2 A B f x)` is B (i.e., element of the set B).

Note: Often one can simply write `(f x)` and Scunak will reconstruct the full term `(ap2 A B f x)`.

One can also declare the following “notation”:

notation $@$ (`ap2 A B`).

Once you have done this, you can write $(f @ x)$ instead of $(\text{ap2 } A \ B \ f \ x)$. This is only for input.

However, Scunak still outputs the term as $(\text{ap2 } A \ B \ f \ x)$.

A common way to define a function from A to B is to give a term `body` which uses an object $x \in A$ to determine an element of B . The `lam2` constructor can be used to give such functions. For example, the identity function from A to A can be represented by $(\text{lam2 } A \ A \ (\lambda x. x))$. (The backslash represents a λ .) Any well typed term of the form $(\text{lam2 } A \ B \ \dots)$ will be of type $(\text{funcSet } A \ B)$.

One could combine `ap2` and `lam2` in an interesting way. Consider the term $(\text{lam2 } A \ B \ (\lambda x. (\text{ap2 } A \ B \ f \ x)))$. The type of this term is $(\text{funcSet } A \ B)$. Note this is the same type as f . In fact, the two functions are in some sense the same. f represents a function f which takes an element $x \in A$ to $f(x)$. $(\text{lam2 } A \ B \ (\lambda x. (\text{ap2 } A \ B \ f \ x)))$ represents a function taking an element x to the value represented by $(\text{ap2 } A \ B \ f \ x)$, i.e., the result of applying f to x , i.e. $f(x)$. So, we have the equation

$$((\text{lam2 } A \ B \ (\lambda x. (\text{ap2 } A \ B \ f \ x)))) = f$$

This equation is known as η -equality in the λ -calculus and is a lemma proven in the Scunak kernels. Similarly, an internal version of β -equality

$$((\text{ap2 } A \ B \ (\text{lam2 } A \ B \ f) \ x)) = (f \ x)$$

is a lemma proven in the various kernels.

Scip: The Scunak Interactive Prover

The Scunak interactive prover (Scip) can be used to interactively construct the proof term of a claim. The interactive style of Scip is similar to the provers TPS [1, 2] and Coq [9, 11, 10]. A user enters the Scip top level using the prove command in the Scunak top level as follows:

```
prove <claimname>
```

We start by giving the commands for constructing a proof.

The help command lists commands which can be used to construct proofs.

You are currently in the Scip level (Scunak Interactive Prover).

Scip Commands:

```
d % assert that the goal (or current subgoal) is done.
willshow % reduce current goal to given subgoal
hence % infer given proposition from last assertion in context
clearly % infer given proposition
apply % apply a given extraction proof term to infer a proposition
claim % assert an arbitrary proposition to be used and proved later
claimtype % assert a first-order type returning a proof type to be
              used and proved later
lemma % add a new claim to the global signature and use it
              in this proof
b % choose a backwards reasoning step introducing at most
              one new subgoal
b2 % choose a backwards reasoning step introducing at most
              two new subgoals
f % choose a forwards reasoning step introducing at most
              one new subgoal
contradiction % proof by contradiction
xmcases % use proof by cases based on P or (not P),
              where P is given by the user
cases % use proof by cases if a disjunction is in the context.
exists <name> % claim and use the existence of a member of set
              satisfying a property
intro % apply an (generic) introduction rule to prove a particular
              kind of goal (depends on the kind of goal)
unfoldgoalhead % If head of goal is an abbreviation,
              prove the unfolded version
unfoldhead <factname> % If factname in context has an abbreviation
              at the head, unfold it
unfoldhead <abbrevname> % Find a fact in context with <abbrevname>
```

```

                                at head, and unfold <abbrevname>
unfoldgoal <abbrevname> % Unfold outermost occurrences of
                                <abbrevname> in the goal
unfold <abbrevname> % Unfold outermost occurrences of <abbrevname>
                                in the most recent fact containing <abbrevname>
reducegoal % reduce the goal (eg, using beta and eta rules)
args= % reduce a goal ((f A1 ... An)==(f B1 ... Bn)) to showing
                                each Ai is the same as Bi

undo % undo
pplan % print the current plan
pplan* % print the current plan omitting proof term parts of pairs
pstatus % print all gaps with supports
choose-task % Choose which gap is current
pterm % print the current (open) proof term
help % print this message
help <name> % print info about name
q % quit

```

We describe some of these commands in more detail below.
 To demonstrate the prover, we will consider examples.

1. Scip commands for printing information

The Scip `pplan`, `pplan*` and `pstatus` commands print information about the current proof state.

`pplan` and `pplan*` display the current assumptions and facts and current goal. (`pplan*` omits some proof terms to make a more human-readable output.)

`pstatus` prints the number of subgoals still open with some basic information.

2. Telling Scip you think you're done

The Scip `d` command checks if the subgoal can be filled in one step. If so, it justifies the subgoal. If this is the last subgoal, the final proof term is printed and Scunak exits Scip. Scunak offers to save the proof term in a PAM file and in a lisp file before returning to the top level. Also, the claim Scip was proving becomes an abbreviation.

3. Asserting a new fact

The Scip `clearly` command (equivalently, `fact` command) allows a user to assert a new fact which will be justified by Scunak and added to the current support. If Scunak cannot justify the fact using the current support context, then the fact will not be added to the support context. Several examples of using `clearly` are in the example below. `clearly` is a very commonly used command.

4. Making Temporary Claims in Scip

The `claim` and `claimtype` commands can be used to make a claim which can be used to help justify the current goal. Eventually, you will need to prove the claim.

5. The Scip intro command

The Scip `intro` command can be used to reduce a variety of goals in a variety of ways, including:

- If the goal is to prove $(A \Rightarrow B)$, assume A and prove B .
- If the goal is to prove $(A \ \& \ B)$, prove A and B as two subgoals.
- If the goal is to prove $(f == g)$ where f and g are in a function set $(\text{funcSet } A \ B)$, then let x be in A and prove $(\text{ap2 } A \ B \ f \ x) == (\text{ap2 } A \ B \ g \ x)$ (i.e., prove $f(x) = g(x)$ for all $x \in A$).
- If the goal is to prove $(A == B)$, prove $(A \leq B)$ and $(B \leq A)$.
- If the goal is to prove $(\text{forall } x:A \ . \ \text{phi})$, let x be in A and prove phi .
- If the goal is to prove $(A \leq B)$, let x be in A and prove $x :: B$.
- If the goal is to prove $(A \Leftrightarrow B)$, then prove the conjunction of $(A \Rightarrow B)$ and $(B \Rightarrow A)$.

6. Unfolding Definitions

The Scip commands `unfold`, `unfoldgoal`, `unfoldgoalhead`, and `unfoldhead` allow a user to unfold abbreviations in the support or in the goal.

7. Reducing Goals

The Scip command `reducegoal` tries to apply β or η reduction to functions. For example, $(\text{ap2 } A \ B \ (\text{lam2 } A \ B \ f) \ x)$ may simplify to $(f \ x)$.

This only works in simple cases. In general, you may want to assert the particular equation you believe holds using `claim`, use this equation, and later justify the equation. However, when `reducegoal` applies, it may significantly simplify your task.

8. Example: Distributivity of Union over Intersection

Consider the problem of proving

$$X \cup (Y \cap Z) == (X \cup Y) \cap (X \cup Z).$$

This problem is SET171+3 in the TPTP library [17]. A higher-order version of the problem was considered in [4].

In PAM syntax, we have the following version of the problem:

```
[X:set]
[Y:set]
[Z:set]
```

```
(unionintersectDist2 X Y Z):|- ((X \cup (Y \cap Z))==(X \cup Y) \cap (X \cup Z))?
```

This claim is distributed with Scunak in the file `bool-props-sets.pam`. One can begin Scip with the problem as follows:

```
scunak -f bool-props-sets.pam
...
>prove unionintersectDist2
```

The following is the output of such a Scip session with some commentary.

Give name for obj>X
 Give name for obj>Y
 Give name for obj>Z

In order to see the current plan, we can use pplan:

```
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
Goal (What you need to show): |- ((X \cup (Y \cap Z)) == ((X \cup Y) \cap (X \cup Z)))
```

The goal is to construct a term of type of a proof of

$$X \cup (Y \cap Z) == (X \cup Y) \cap (X \cup Z)$$

using the objects X , Y and Z .

The `intro` command notes that the goal is showing equality of two sets, which can be reduced to showing each side is a subset of the other.

```
>intro
OK
```

Using `pplan` we see the current subgoal is now to show

$$X \cup (Y \cap Z) \subseteq (X \cup Y) \cap (X \cup Z).$$

The `pstatus` command shows there are two open goals. The second (delayed) goal is the other direction of subset. (The command `choose-task` can be used to change which goal is current.)

```
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
Goal (What you need to show): |- ((X \cup (Y \cap Z)) <= ((X \cup Y) \cap (X \cup Z)))
>pstatus
0) (Z Y X) |- ((X \cup (Y \cap Z)) <= ((X \cup Y) \cap (X \cup Z)))
1) (Z Y X) |- (((X \cup Y) \cap (X \cup Z)) <= (X \cup (Y \cap Z)))
```

We use `choose-task` to switch to the other goal, then use `pstatus` and `pplan` to show the resulting status.

```
>choose-task
Enter a number between 0 and 1
1
>pstatus
0) (Z Y X) |- (((X \cup Y) \cap (X \cup Z)) <= (X \cup (Y \cap Z)))
1) (Z Y X) |- ((X \cup (Y \cap Z)) <= ((X \cup Y) \cap (X \cup Z)))
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
Goal (What you need to show): |- (((X \cup Y) \cap (X \cup Z)) <= (X \cup (Y \cap Z)))
```

Using `intro` again applies the subset introduction rule. That is, we introduce a new object (whose name we choose as “a”) and the new hypothesis that a is in $(X \cup Y) \cap (X \cup Z)$.

```
>intro
OK
Give name for (in ((X \cup Y) \cap (X \cup Z)))>a
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in ((X \cup Y) \cap (X \cup Z)))
fact0: |- (a::((X \cup Y) \cap (X \cup Z)))
Goal (What you need to show): |- (a::(X \cup (Y \cap Z)))
```

Since we have just assumed $a \in ((X \cup Y) \cap (X \cup Z))$, we can conclude that $a \in (X \cup Y)$ and so either $a \in X$ or $a \in Y$. This is modelled in Scip by two applications of the “clearly” command.

```
>clearly
Enter Proposition>(a::(X \cup Y))
Correct.
>clearly
Enter Proposition>((a::X) | (a::Y))
Correct.
```

Now we have a disjunction on the context:

```
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in ((X \cup Y) \cap (X \cup Z)))
fact0: |- (a::((X \cup Y) \cap (X \cup Z)))
fact1: |- (a::(X \cup Y))
fact2: |- ((a::X) | (a::Y))
Goal (What you need to show): |- (a::(X \cup (Y \cap Z)))
```

Applying the `cases` command, we split the goal into two cases based on the disjunction in context. In one case, we have the new assumption that $a \in X$. The second case in which $a \in Y$ is delayed.

```
>cases
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in ((X \cup Y) \cap (X \cup Z)))
fact0: |- (a::((X \cup Y) \cap (X \cup Z)))
fact1: |- (a::(X \cup Y))
fact2: |- ((a::X) | (a::Y))
```

```

ass0: |- (a::X)
Goal (What you need to show): |- (a::(X \cup (Y \cap Z)))
>pstatus
0) (ass0 fact2 fact1 fact0 a Z Y X) |- (a::(X \cup (Y \cap Z)))
1) (fact2 fact1 fact0 a Z Y X) {x7:|- (a::Y)}|- (a::(X \cup (Y \cap Z)))
2) (Z Y X) |- ((X \cup (Y \cap Z))<=((X \cup Y) \cap (X \cup Z)))
    This goal is trivial, since we know  $a \in X$  and we want to show  $a \in (X \cup (Y \cap Z))$ .
>d
Done with subgoal!

```

We now consider the other case, in which $a \in Y$.

```

>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in ((X \cup Y) \cap (X \cup Z)))
fact0: |- (a::((X \cup Y) \cap (X \cup Z)))
fact1: |- (a::(X \cup Y))
fact2: |- ((a::X) | (a::Y))
ass1: |- (a::Y)
Goal (What you need to show): |- (a::(X \cup (Y \cap Z)))

```

We cannot directly show the goal in this case. We must use another case analysis starting from $a \in (X \cup Z)$. We can conclude $a \in (X \cup Z)$ from `ass0` with the “clearly” command.

```

>clearly
Enter Proposition>(a::(X \cup Z))
Correct.
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in ((X \cup Y) \cap (X \cup Z)))
fact0: |- (a::((X \cup Y) \cap (X \cup Z)))
fact1: |- (a::(X \cup Y))
fact2: |- ((a::X) | (a::Y))
ass1: |- (a::Y)
fact3: |- (a::(X \cup Z))
Goal (What you need to show): |- (a::(X \cup (Y \cap Z)))

```

Now we can use `clearly` to conclude that either $a \in X$ or $a \in Z$.

```

>clearly
Enter Proposition>((a::X) | (a::Z))
Correct.
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj

```

```

Z:obj
a:(in ((X \cup Y) \cap (X \cup Z)))
fact0: |- (a::((X \cup Y) \cap (X \cup Z)))
fact1: |- (a::(X \cup Y))
fact2: |- ((a::X) | (a::Y))
ass1: |- (a::Y)
fact3: |- (a::(X \cup Z))
fact4: |- ((a::X) | (a::Z))
Goal (What you need to show): |- (a::(X \cup (Y \cap Z)))

```

Again, we use the `cases` command on this disjunction. (There are two disjunctions in context. The `cases` command chooses the most recently added.)

```

>cases
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in ((X \cup Y) \cap (X \cup Z)))
fact0: |- (a::((X \cup Y) \cap (X \cup Z)))
fact1: |- (a::(X \cup Y))
fact2: |- ((a::X) | (a::Y))
ass1: |- (a::Y)
fact3: |- (a::(X \cup Z))
fact4: |- ((a::X) | (a::Z))
ass2: |- (a::X)
Goal (What you need to show): |- (a::(X \cup (Y \cap Z)))

```

As previously, this case is trivial enough for the `d` command, leaving only the case in which $a \in Y$ and $a \in Z$.

```

>d
Done with subgoal!
>pstatus
0) (ass3 fact4 fact3 ass1 fact2 fact1 fact0 a Z Y
   X) |- (a::(X \cup (Y \cap Z)))
1) (Z Y X) |- ((X \cup (Y \cap Z)) <=& ((X \cup Y) \cap (X \cup Z)))
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in ((X \cup Y) \cap (X \cup Z)))
fact0: |- (a::((X \cup Y) \cap (X \cup Z)))
fact1: |- (a::(X \cup Y))
fact2: |- ((a::X) | (a::Y))
ass1: |- (a::Y)
fact3: |- (a::(X \cup Z))
fact4: |- ((a::X) | (a::Z))
ass3: |- (a::Z)
Goal (What you need to show): |- (a::(X \cup (Y \cap Z)))

```

Due to the new assumption $a \in Z$ and the older assumption $a \in Y$, we can use clearly command to conclude $a \in Y \cap Z$.

```
>clearly
Enter Proposition>(a::(Y \cap Z))
Correct.
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in ((X \cup Y) \cap (X \cup Z)))
fact0: |- (a::((X \cup Y) \cap (X \cup Z)))
fact1: |- (a::(X \cup Y))
fact2: |- ((a::X) | (a::Y))
ass1: |- (a::Y)
fact3: |- (a::(X \cup Z))
fact4: |- ((a::X) | (a::Z))
ass3: |- (a::Z)
fact5: |- (a::(Y \cap Z))
Goal (What you need to show): |- (a::(X \cup (Y \cap Z)))
```

Since we know $a \in Y \cap Z$, we can use the d command to finish the case by concluding a is in the union.

```
>d
Done with subgoal!
```

Now all that remains is to prove the other direction.

```
>pstatus
0) (Z Y X) |- ((X \cup (Y \cap Z)) <= ((X \cup Y) \cap (X \cup Z)))
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
Goal (What you need to show): |- ((X \cup (Y \cap Z)) <= ((X \cup Y) \cap (X \cup Z)))
```

Again we use intro to prove the subset by introducing a new object a assumed to be in $X \cup (Y \cap Z)$.

```
>intro
OK
Give name for (in (X \cup (Y \cap Z)))>a
```

Since a is in the union, it is in either X or $Y \cap Z$.

```
>clearly
Enter Proposition>((a::X) | (a::(Y \cap Z)))
Correct.
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
```

```

a:(in (X \cup (Y \cap Z)))
fact6: |- (a::(X \cup (Y \cap Z)))
fact7: |- ((a::X) | (a::(Y \cap Z)))
Goal (What you need to show): |- (a::((X \cup Y) \cap (X \cup Z)))

```

We split on cases.

```

>cases
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in (X \cup (Y \cap Z)))
fact6: |- (a::(X \cup (Y \cap Z)))
fact7: |- ((a::X) | (a::(Y \cap Z)))
ass4: |- (a::X)
Goal (What you need to show): |- (a::((X \cup Y) \cap (X \cup Z)))

```

Hence $a \in (X \cup Y)$ and $a \in (X \cup Z)$.

```

>clearly
Enter Proposition>(a::(X \cup Y))
Correct.
>clearly
Enter Proposition>(a::(X \cup Z))
Correct.
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in (X \cup (Y \cap Z)))
fact6: |- (a::(X \cup (Y \cap Z)))
fact7: |- ((a::X) | (a::(Y \cap Z)))
ass4: |- (a::X)
fact8: |- (a::(X \cup Y))
fact9: |- (a::(X \cup Z))
Goal (What you need to show): |- (a::((X \cup Y) \cap (X \cup Z)))

```

We use `d` to finish this case.

```

>d
Done with subgoal!
>pstatus
0) (ass5 fact7 fact6 a Z Y X) |- (a::((X \cup Y) \cap (X \cup Z)))
>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in (X \cup (Y \cap Z)))
fact6: |- (a::(X \cup (Y \cap Z)))

```

```

fact7: |- ((a::X) | (a::(Y \cap Z)))
ass5: |- (a::(Y \cap Z))
Goal (What you need to show): |- (a::((X \cup Y) \cap (X \cup Z)))

```

In the second case, $a \in Y$ and $a \in Z$.

```

>clearly
Enter Proposition>(a::Y)
Correct.
>clearly
Enter Proposition>(a::Z)
Correct.

```

We still need to show $a \in (X \cup Y) \cap (X \cup Z)$.

```

>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in (X \cup (Y \cap Z)))
fact6: |- (a::(X \cup (Y \cap Z)))
fact7: |- ((a::X) | (a::(Y \cap Z)))
ass5: |- (a::(Y \cap Z))
fact10: |- (a::Y)
fact11: |- (a::Z)
Goal (What you need to show): |- (a::((X \cup Y) \cap (X \cup Z)))

```

Using clearly twice, we obtain $a \in (X \cup Y)$ and $a \in (X \cup Z)$.

```

>clearly
Enter Proposition>(a::(X \cup Y))
Correct.
>clearly
Enter Proposition>(a::(X \cup Z))
Correct.

```

Now the gap is very small:

```

>pplan
Support (Objects, Assumptions and Derived Facts in Context):
X:obj
Y:obj
Z:obj
a:(in (X \cup (Y \cap Z)))
fact6: |- (a::(X \cup (Y \cap Z)))
fact7: |- ((a::X) | (a::(Y \cap Z)))
ass5: |- (a::(Y \cap Z))
fact10: |- (a::Y)
fact11: |- (a::Z)
fact12: |- (a::(X \cup Y))
fact13: |- (a::(X \cup Z))
Goal (What you need to show): |- (a::((X \cup Y) \cap (X \cup Z)))

```

Using `d`, we close this final gap. Scip notes the success and offers to save the proof term in PAM and lisp syntax in files. Note that when Scip exits, the claim is converted into an abbreviation using the proof term.

```
>d
Done with subgoal!
Output PAM Proof Term to File [y/n, Default y]>
Filename [Default File myproofs.pam]>
Output Lisp Proof Term to File [y/n, Default y]>
Filename [Default File myproofs.lisp]>
Scip Out!
>help unionintersectDist2
unionintersectDist2 is an abbreviation of type:
{x0:obj}{x1:obj}{x2:obj}|- ((x0 \cup (x1 \cap x2))==(x0 \cup x1) \cap (x0 \cup x2))
Defn: (\x0 x1 x2.settextsub ...)
unionintersectDist2 is essentially a derived proof rule.
Due to proof irrelevance, the abbreviation never needs to be expanded.
```


The Scunak Type Theory

The type theory implemented in Scunak is second-order dependent type theory with a base type of objects, a base type of propositions, a basic family of proof types, and sigma types between objects and proof types. The sigma types are used in order to coerce predicates on objects into types. More discussion of the type theory can be found in [7].

Let \mathcal{V} be an infinite set of variables and \mathcal{C} be a set of constants. We use x, y, z, x^1, \dots to denote variables and c, d, c^1, \dots to denote constants. We define terms and types as follows:

Terms $M, N, P, Q, R, \varphi, \dots$ $:= x|c|(\lambda x.M)|(M N)|\langle M, N \rangle|\pi_1(M)|\pi_2(M)$
Types A, B, C, \dots $:= \text{obj}|\text{prop}|\text{(pf } P)|\text{(class } \varphi)|(\Pi x : A.B)$

Intuitively, P should have type `prop` in the type `(pf P)`, and φ should have type `($\Pi x : \text{obj}.\text{prop}$)` in the type `(class φ)`. These conditions are checked in the typing rules. We restrict types of constants to third-order and types of bound-variables to second-order. This provides a second-order type theory.

The internal representation uses de Bruijn indices instead of named variables for the Π and λ binders. Scunak lisp functions can be used to directly construct the internal representations of terms and types. Using this direct lisp format has certain advantages: the parsing is trivial, the terms are completely specified, and the syntax does not depend on any particular signature. Since lisp can parse the expressions, type-checking is much faster when one declares a signature using this direct encoding. However, the main disadvantage is that the expressions are difficult for humans to read and write. For this reason, it often make more sense to use the PAM syntax described in Chapter 5. The PAM syntax does assume a certain amount of set theory is included in the kernel signature.

We now give the lisp functions for constructing the internal lisp representation of terms and types. For terms M use:

- Number (for Debruijn Indices)
- Symbol (naming a variable, constant, abbreviation or claim)
- (APP $M M$)
- (PAIR $M M$)
- (LAM M)
- (FST M)
- (SND M)

For types A use:

- (OBJ)
- (PROP)
- (PF M)

- (DCLASS M)
- (DPI $A A$)

Lisp Functions can also be used to declare constants, abbreviations and claims (along with the names of the authors of the declaration).

- (newconst '*name* A '("typeauthor1name" ... "typeauthorname"))
- (newclaim '*name* A '("typeauthor1name" ... "typeauthorname"))
- (newabbrev '*name* $A M$ '("typeauthor1name" ... "typeauthorname")
'("termauthor1name" ... "termauthorname"))
- (claim2abbrev '*name* M '("termauthor1name" ... "termauthorname"))

When the expression is evaluated, Scunak ensures the type A is a well-defined second-order type. (We do not give the technical rules for type checking here. See [7].)

1. Example of Type Checking

As an example, we show how one can begin to encode propositional logic starting from the empty signature (no kernel). First, start scunak with no kernel:

```
scunak -k none
```

We can use the Scunak `lisp` command to evaluate arbitrary lisp expressions, including those for adding Scunak signature elements. We make use of this command below.

First, we declare a constant `false` of propositional type.

Note that since the signature is empty, there is no preexisting signature element named `false`.

```
>help false
```

```
Sorry, I have no information about false.
```

We directly declare a constant `false` of type `prop`.

```
>lisp
```

```
(newconst '|false| (prop) '("Chad Edward Brown"))
```

```
>help false
```

```
false is a constant of type:
```

```
prop
```

Next, we declare an implication operator as a constant.

```
>lisp
```

```
(newconst '|imp| (dpi (prop) (dpi (prop) (prop)))
```

```
'("Chad Edward Brown"))
```

We declare a negation operator as an abbreviation for "implies false".

```
>lisp
```

```
(newabbrev '|not| (dpi (prop) (prop))
```

```
(lam (app (app '|imp| '0) '|false|))
```

```
'("Chad Edward Brown") '("Chad Edward Brown"))
```

One can in principle specify different authors for the type of an abbreviation and the definition of an abbreviations.

When an abbreviation is created, Scunak also automatically creates constants for folding and unfolding definitions. This is used instead of using δ -reduction in type checking. In this particular case, Scunak creates two symbols (where the types are shown using named bound variables instead of deBruijn indices for readability):

`not#F : $\Pi x0 : o. \Pi x1 : \text{imp } x0 \text{ false. } \vdash \text{not } x0$`

`not#U : $\Pi x0 : o. \Pi x1 : \text{not } x0. \vdash \text{imp } x0 \text{ false}$`

All this information is displayed when one requests help for `not`.

```
>help not
```

```
not is an abbreviation of type:
```

```
{x0:prop}prop
```

```
Defn: ( $\lambda x0.(x0 \Rightarrow \text{false})$ )
```

There are two related constants for folding and unfolding this abbreviation:

```
not#F : {x0:prop}{x1:|- (x0 => false)}|- (not x0)
```

```
not#U : {x0:prop}{x1:|- (not x0)}|- (x0 => false)
```

As is common in logical frameworks, we represent natural deduction rules as constants returning a proof type. For instance, the rules for implication introduction and elimination are given by:

```
>lisp
```

```
(newconst '|impI|
  (dpi (prop)
    (dpi (prop)
      (dpi (dpi (pf '1) (pf '1))
        (pf (app (app '|imp| '2) '1))))))
  '("Chad Edward Brown"))
```

```
>lisp
```

```
(newconst '|impE|
  (dpi (prop)
    (dpi (prop)
      (dpi (pf (app (app '|imp| '1) '0))
        (dpi (pf '2) (pf '2))))))
  '("Chad Edward Brown"))
```

Using these constants and abbreviations, we can define introduction and elimination rules for negation. We first define the rule for negation introduction:

```
>lisp
```

```
(newabbrev '|notI|
  (dpi (prop)
    (dpi (dpi (pf '0) (pf '|false|))
      (pf (app '|not| '1))))
  (lam (lam (app (app '|not#F| '1)
    (app (app (app '|impI| '1) '|false|) '0))))))
  '("Chad Edward Brown") '("Chad Edward Brown"))
```

Scunak does not introduce new constants for folding and unfolding `|notI|` since there is no need. The underlying type system builds in proof irrelevance, so any term of the type of `|notI|` is already considered equal to `|notI|`.

We could similarly define `|notE|`. Alternatively, we can declare `|notE|` as a claim we intend to fill in as an abbreviation later.

```
lisp
```

```
(newclaim '|notE|
  (dpi (prop)
    (dpi (pf (app '|not| '0))
      (dpi (pf '1) (pf '|false|))))
  '("Chad Edward Brown"))
```

The help command shows this is a claim without a proof.

```

>help notE
notE is a claim of type:
{x0:prop}{x1:|- (not x0)}{x2:|- x0}|- false
notE is essentially an open conjecture.
  We convert this to an abbreviation later using claim2abbrev:
>lisp
(claim2abbrev '|notE| (lam (lam (app (app (app '|impE| '1) '|false|)
                                   (app (app '|not#U| '1) '0))))
                  '("Chad Edward Brown"))
  A final call to help shows this is now an abbreviation (derived proof rule) instead
of a claim (conjecture).
>help notE
notE is an abbreviation of type:
{x0:prop}{x1:|- (not x0)}{x2:|- x0}|- false
Defn: (\x0 x1.impE x0 false (not#U x0 x1))
notE is essentially a derived proof rule.
Due to proof irrelevance, the abbreviation never needs to be expanded.

```

The PAM Syntax

PAM stands for “Pseudo-Automath”. This name is chosen since the PAM syntax is very close to the syntax traditionally used by Automath [12, 13, 18], with some modifications derived from the syntax of Twelf [15] (for Π -types).

First, a name (*NAME*) is a string of alphanumeric characters (plus `-`, `+`, `^`, `_`, `~`, and `#`) which begins with an alphabetic character. A namelist (*NAMELIST*) is a nonempty list of names separated by whitespace.

Syntax for normal terms (*NORMAL*): A normal term is one of four forms: an extraction *EXTRACT* (see below), a pair $\langle EXTRACT, EXTRACT \rangle$, or an n -ary lambda-prefix (with $n > 0$) of the form $(\backslash NAMELIST . EXTRACT)$ or $(\backslash NAMELIST . \langle EXTRACT, EXTRACT \rangle)$.

Syntax for extraction terms (*EXTRACT*): One should always put parentheses around an extraction, unless there is no parsing ambiguity. There are two basic forms for extractions: *NAME* (for constants, abbreviations or variables) and *NAME NORMALL* (for n -ary applications) where *NORMALL* is a nonempty list of *NORMAL*’s separated by whitespace.

We never explicitly represent “first” and “second” operators for pairing since these can be automatically added during type checking.

Additionally, there are many other ways to write extractions which are included for readability purposes. Many of these extra forms are *EXTRACT BINOP EXTRACT* where *BINOP* is any of the following:

- `=>` (for implication, `imp`)
- `&` (for conjunction, `and`)
- `|` (for disjunction, `or`)
- `<=>` (for equivalence, `equiv`)
- `==` (for equality, `eq`)
- `::` (for set membership, `in`)
- `<=` (for subset, `subset`)
- `;` (for adjoining an element to a set, `setadjoin`)
- `\cup` (for binary union, `binunion`, defined below)
- `\cap` (for binary intersection, `binintersect`, defined below)

The other extra forms are listed below:

- `~EXTRACT` (for negation, (`not EXTRACT`))
- `forall NAMELIST:EXTRACT . EXTRACT` (for bounded forall quantifier, `all`)
- `exists NAMELIST:EXTRACT . EXTRACT` (for bounded existence quantifier, `ex`)
- `{}` (for the emptyset, `emptyset`)

- $\{ EXTRACT, \dots, EXTRACT \}$ (for an enumerated set, expanded using `setadjoin` and `unitset`, where `unitset` is defined below)
- $\{ NAME : EXTRACT \mid EXTRACT \}$ (for a set defined by separation, `setconstr`)

Syntax for atomic types (*DTYPE0*):

- `obj` (or, equivalently, `set`).
- `prop`
- `|- EXTRACT` (for the type of proofs of *EXTRACT*)
- *EXTRACT* (for a class type). There are really two different cases here. If the *EXTRACT* has type Π `obj prop`, then this is `dclass EXTRACT`. If the *EXTRACT* has type `obj`, then this is seen as a set and coerced to be a predicate which is used in the Σ -type `dclass` construction.

Syntax for first order types (*DTYPE1*):

- *DTYPE0*
- $\{ NAME : DTYPE0 \} DTYPE1$ (for Π types)
- $DTYPE0 \rightarrow DTYPE1$ (for Π -types which do not use the bound variable)

Syntax for second order types (*DTYPE2*):

- *DTYPE1*
- $\{ NAME : DTYPE0 \} DTYPE2$ (for Π types)
- $\{ NAME : DTYPE1 \} DTYPE2$ (for Π types)
- $DTYPE0 \rightarrow DTYPE2$ (for Π -types which do not use the bound variable)
- $DTYPE1 \rightarrow DTYPE2$ (for Π -types which do not use the bound variable)
- $(DTYPE1) \rightarrow DTYPE2$ (for Π -types which do not use the bound variable)
- $DTYPE1 \rightarrow (DTYPE2)$ (for Π -types which do not use the bound variable)

Syntax for declarations:

One can add a new (named) variable to the context using the syntax $[NAME : DTYPE1]$

For example,

```
[A:prop]
```

```
[u:|- A]
```

puts two new variables in the context: a variable *A* of propositional type and a variable *u* of the type of a proof of *A*. (The declaration of *u* corresponds to assuming *A* has a proof.) Both of these are of atomic type. Here is an example of a declaration of *DTYPE1*:

```
[A:prop]
```

```
[B:obj->prop]
```

```
[u:{x:obj}|- A -> |- (B x)]
```

The syntax for declaring constants is

- *NEWAPP*: *DTYPE2*.

NEWAPP is either a *NAME* or of the form $(NAME \dots NAME)$. The idea is that one simply writes *NAME* if no declared bound variables occur in the type. Otherwise, one types $(NAME \dots NAME)$ to list the dependencies on the bound variables (in order). All bound variables must be listed, and in a valid order (eg, $\varphi : o$ must come before $u : \vdash \varphi$), or Scunak will signal an error.

Examples:

```

false:prop.
[phi:prop]
[dpsi:|- phi -> prop]
(dimp phi dpsi):prop.
[dpsipf:{v:|- phi}|- (dpsi v)]
(dimpI phi dpsi dpsipf):|- (dimp phi dpsi).

```

Generally users should not declare constants. All the constants are declared in the kernel. If a user wishes to delay giving the definition of an abbreviation (or, as a special case, the proof of some “claim”), then the user should declare a “claim” as follows:

- *NEWAPP:DTYPE2?*

The syntax for declaring an abbreviation is

- *NEWAPP:DTYPE2=NORMAL.*

If the definition is an extraction, then the type can be omitted:

- *NEWAPP=EXTRACT.*

If the *NEWAPP* above is already a claim, then this simply gives the definition. Hence we can omit the type even if the definition is normal. The syntax for filling claims without repeating the type is

- *NEWAPP:=NORMAL.*

Examples:

```

[phi:prop]
[psi:prop]
(imp phi psi):prop=(dimp phi (\x.psi)).
[phi:prop]
(imp phi):prop->prop=(\psi.dimp phi (\x.psi)).
[phi:prop]
[psi:prop]
(imp phi psi):prop?
imp:=(\u v.dimp u (\x.v)).

```

There should never be a need to write *DTYPE2* (second order types) since one can always put the first order arguments in context as bound variables and then list these dependencies in the declaration.

The Scunak Tutor

Scunak can also be used as a tutor. We have already seen a small example of the Scunak tutor in the introduction (see Section 7). Here we consider a few more examples. These examples come from two Wizard of Oz experiments performed as part of the DiaLog project (see [5, 6]).¹

Note that in Scip our purpose was to help a user construct a proof term quickly and conveniently. With the Scunak tutor, the purpose is to provide feedback to a student's attempt to prove a theorem. In this case, even if Scunak realizes a subgoal has been completed, the student is required to explicitly note that he or she recognizes this fact.

To use the tutor, one begins by giving information intended to model the tutoring situation. In particular, we can indicate rules which the tutor and student should know using the top level command `tutor-student-usable`. We can indicate rules which the tutor alone knows (or is allowed to use) using the command `tutor-only-usable`. A further command is `tutor-auto-back`. This sets rules which the tutor and student can use in order to reduce a goal without explicitly saying the reduction has been done.

The help command in the tutor top level lists all possible commands on this top level.

Commands:

```
qed. % assert that the goal (or current subgoal) is done.
let x. % introduce a variable x for an object
let x y ... z. % introduce variables x, y, ... z for objects
let x y ... z:A. % introduce variables x, y, ... z for objects of type A
assume <Proposition>. % introduce an assumption
willshow <Proposition>. % reduce current goal to given subgoal
clearly <Proposition>. % infer given proposition
hence <Proposition>. % infer given proposition from last assertion in context
i give up. % quit
```

1. WOZ1: Sets

The first DiaLog Wizard of Oz experiment [5] used basic examples about sets. We here consider the third example from the experiment, in which the student is asked to prove

$$(A \cap B) \in \mathcal{P}((A \cup C) \cap (B \cup C))$$

The other WOZ1 examples are considered in the appendix.

¹Some aspects of the Scunak proof tutor resulted from several fruitful discussions with my colleague Magdalena Wolska.

We start Scunak giving the appropriate command line arguments for the WOZ1 examples. We assume the student is named *Dave*.

```
scunak -k macu -p woz1-sm.lisp -f woz1-lemmas.pam woz1-claims.pam -n Dave
```

We begin by initializing the tutoring situation:

```
>[U:set]
>[A:(powerset U)]
>[B:(powerset U)]
>[C:(powerset U)]
```

There are three sets A , B , and C which are all subsets of a common set U .

The tutor and student are allowed to eager apply set extensionality `setextsub` (reducing equality of sets to showing both directions of subset), proof by cases `orE`, and reducing showing something is in a powerset of X by showing it is a subset of X .

```
>tutor-auto-back setextsub orE powersetTI.
```

We explicitly give the rules the tutor knows and expects the student to already know.

```
>tutor-student-usable notE xmcases eqI notI contradiction symeq
transeq symtrans1eq symtrans2eq binunionT binintersectT powersetT
setminusT complementT setextT subsetTI powersetTI powersetTE
powersetTI1 powersetTE1 complementTI complementTE complementTI1
complementTE1 binintersectTEL binintersectTER binintersectTI
binunionTE1 binunionTE binunionTIL binunionTIR binintersectTELcontra
binintersectTERcontra binunionTILcontra binunionTIRcontra
binunionTEcontra demorgan1a demorgan2a demorgan2a1 demorgan2a2
demorgan1b demorgan2b2 demorgan2b demorgan1 demorgan2 demorgan1Eq2
demorgan2Eq2 demorgan3Eq2 demorgan4Eq2 contrasubsetT contrasubsetT1
binintersectTSub1 binintersectTSub2 binunionTSub1 binunionTSub2 impI
impE woz13rule1 woz13rule2 woz13rule3 doubleComplementSub1
doubleComplementSub2 doubleComplementEq subsetTrans.
```

We give one rule `woz13rule4` that the tutor knows, but does not believe the student knows (yet). The rule corresponds to concluding $(X \cap Y) \subseteq (Z \cap W)$ from $X \subseteq Z$ and $Y \subseteq W$. In our example, the student will attempt to make an inference that could be justified by this rule. However, since the student should not yet know this rule, the tutor considers the step to be too coarse with respect to granularity [16] and rejects the step.

```
>tutor-only-usable woz13rule4.
```

We now invoke the tutor top level with the appropriate example.

```
>tutor (woz1_3 U A B C)
Hello, Dave. Please try to prove the following:
|- ((binintersectT U A B)::(powersetT U
                                     (binintersectT U (binunionT U A C) (binunionT U B C))))
```

Dave begins by inferring two true facts using `clearly`.

```
Dave> clearly (A <= (binunionT U A C)).
OK
Dave> clearly (B <= (binunionT U B C)).
OK
```

These steps are accepted because we listed `binunionTSub1` as a rule the student knows when we initialized `tutor-student-usable`.

Suppose next Dave asserts he is done. Scunak rejects this.

```
Dave> qed.
Not OK
You don't seem to actually be done with anything, Dave.

Suppose Dave is certain he has shown the appropriate subset proposition and explicitly gives this proposition with clearly
Dave> clearly ((binintersectT U A B)
               <=
               (binintersectT U (binunionT U A C) (binunionT U B C))).
```

Not OK.
How does that follow, Dave? Try using smaller steps.

Scunak knows (given `tutor-only-usable`) that this can be justified using a rule the tutor knows but the student does not. So the tutor asks for more steps. This is how the issue of *granularity* is handled in the Scunak tutor. [16]

Dave now gives the needed steps and completes the proof.

```
Dave> clearly ((binintersectT U A B) <= (binunionT U A C)).
OK
Dave> clearly ((binintersectT U A B) <= (binunionT U B C)).
OK
Dave> clearly ((binintersectT U A B) <=
               (binintersectT U (binunionT U A C) (binunionT U B C))).
OK
Dave> qed.
Congratulations, you're done with the proof, Dave!
```

2. WOZ2: Binary Relations

The second DiaLog Wizard of Oz experiment [6] used binary relations. For simplicity, we consider the preparation example (where a proof was explicitly given to the students). In this example, one must show $R = (R^{-1})^{-1}$. The other WOZ2 examples are considered in the appendix.

The Scunak tutor can be started on the `woz2Ex` example as follows (where we indicate the student's name is *Dave*).

```
scunak -k macu -p woz2-sm.lisp -f woz2-lemmas.pam woz2-claims.pam -n Dave
...
[M:set]
[R:(breln1 M)]
tutor-auto-back setextsub orE.
tutor-student-usable breln1invprop breln1inv breln1invI breln1invE
                    breln1compprop breln1comp breln1compI breln1compE breln1unionprop
                    breln1union breln1unionI breln1unionIL breln1unionIR breln1unionE
                    breln1all subbreln1 contradiction.
tutor (woz2Ex M R)
Hello, Dave. Please try to prove the following:
|- (R==(breln1inv M (breln1inv M R)))
```

Suppose Dave knows he wants to show R and $(R^{-1})^{-1}$ have the same elements. Then he should start by declaring these elements and assuming the pair to be in R .

```
Dave> let x y:M.  
OK  
Dave> assume (<<x,y>>::R).  
OK
```

The response of OK means that Dave has declared a variable or made an assumption which contributes to the proof state. If Dave tried now to declare a variable or make an assumption which did not contribute in a clear way to the proof, the response would be Not OK.

Suppose Dave says he is done. Scunak rejects this.

```
Dave> qed.  
Not OK  
You don't seem to actually be done with anything, Dave.
```

Suppose Dave now decides to infer $(y, x) \in R^{-1}$. Dave uses the "clearly" command. (The command "hence" is similar.) Suppose first Dave makes a mistake and forgets to interchange the x and y . This is Not OK:

```
Dave> clearly (<<x,y>>::(breln1inv M R)).  
Not OK.  
The statement is not so obvious to me, Dave.
```

Dave sees his error and interchanges x and y .

```
Dave> clearly (<<y,x>>::(breln1inv M R)).  
OK
```

This is OK, meaning the Scunak tutor has verified that the fact follows.

Dave can now assert that this case is finished using the "qed" command:

```
Dave> qed.  
OK  
Good Dave, you're done with this part of the proof, but there is more to do.
```

Dave now goes on to the second case. This case is handled similarly. Dave begins by declaring the eigenvariables (as members of M).

```
Dave> let x y:M.  
OK
```

Suppose Dave is confused and tries to make the assumption from the first case again. While Dave is free to do the two cases in either order, the tutor does not allow him to do the same case twice.

```
Dave> assume (<<x,y>>::R).  
Not OK.  
I could not find any reason for you to make such an assumption, Dave.
```

Dave fixes this and makes the correct assumption.

```
Dave> assume (<<x,y>>::(breln1inv M (breln1inv M R))).  
OK
```

For a variation, Dave uses "willshow" to work backwards instead of "clearly" to work forwards.

Dave> willshow (<<y,x>>::(brelnlinv M R)).

OK

Scutor responds OK since the goal of showing $(x, y) \in R$ follows (in one step) from $(y, x) \in R^{-1}$.

Dave now asserts that he is done and Scutor accepts this. Scutor outputs the proof term and exits to the top level.

Dave> qed.

Congratulations, you're done with the proof, Dave!

Successful Term:

(\x0 x1....)

Proofreading With Scunak

We are using the word “proofreading” with a double meaning: Scunak reads proofs and Scunak proofreads. “Proof proofreader” would be more accurate terminology. The purpose of the Scunak proofreader is to read a latex file purportedly containing a proof of a claim. Scunak either builds the proof term for the claim (thus “verifying” the latex proof) or signals a failure with some diagnosis of the problem.

1. Proofreading The First Bartle Sherbert Proof

At the moment, the proofreader only works on the examples derived from the first proof in [3] as reported on in [8]. The latex source proof described in [8] is distributed with Scunak in the file `data/bs-justpf.tex`. There also many other latex files in `data` which contain variations of this proof. Some of the variations are correct and complete, some are incorrect, one is correct but incomplete, and one is incorrect and incomplete. To try the proofreader on these examples, start and initialize Scunak as follows:

```
scunak-acl -k mu -p bs-1-1-4d-mode.lisp -f bs-1-1-4d.pam
>[A:set]
>[B:set]
>[C:set]
```

We next give a series of “Proofs” corresponding to the content of latex files distributed in the `data` directory. We follow each of these proofs by the portion of the Scunak session verifying or refuting the “proof.”

Proof: `[bs-justpf]` (This is quoted directly from [3].) In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$.

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (A \cap C)$ are equal. QED

```
>proofread "bs-justpf.tex" (bs114d A B C).
Proof successfully checked!
```

Proof Term:

(\x0 x1 x2.setextsub ...)

Proof: [bs-shorterpf] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$.

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (A \cap C)$ are equal. QED

>proofread "bs-shorterpf.tex" (bs114d A B C).

Proof successfully checked!

Proof Term:

(\x0 x1 x2.setextsub ...)

Proof: [bs-shorterpf2] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. QED

>proofread "bs-shorterpf2.tex" (bs114d A B C).

Proof successfully checked!

Proof Term:

(\x0 x1 x2.setextsub ...)

Proof: [bs-badpf1] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in B$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$.

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (A \cap C)$ are equal. QED

>proofread "bs-badpf1.tex" (bs114d A B C).

Not OK.

Verification Problem:
then $x \in B$ and $x \in B \cup C$
 $B \cup C$
I am not sure that statement follows, Dave.

Proof: [bs-badpf2] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$.

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (A \cap C)$ are equal. QED

>proofread "bs-badpf2.tex" (bs114d A B C).
Not OK.

Verification Problem:
so $x \in (A \cap B) \cup (A \cap C)$
 $(A \cap B) \cup (A \cap C)$
I am not sure that statement follows, Dave.

Proof: [bs-badpf3] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$.

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (A \cap C)$ are equal. QED

>proofread "bs-badpf3.tex" (bs114d A B C).
Not OK.

Verification Problem:
This shows that $(A \cap B) \cup (A \cap C)$ is a
subset of $A \cap (B \cup C)$
 $(A \cap B) \cup (A \cap C)$
We have not shown that.

Proof: [bs-badpf4] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $x \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$.

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (A \cap C)$ are equal. QED

>proofread "bs-badpf4.tex" (bs114d A B C).
Not OK.

Verification Problem:

Then,

either (iii) $x \in A \cap B$, or (iv) $y \in A \cap C$

The statement is not so obvious to me, Dave.

Proof: [bs-badpf5] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$.

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (C \cap A)$ are equal. QED

>proofread "bs-badpf5.tex" (bs114d A B C).
Not OK.

Verification Problem:

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (C \cap A)$ are equal

I'm afraid that doesn't follow, Dave.

Proof: [bs-badpf6] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cup (B \cap C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$.

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (A \cap C)$ are equal. QED

>proofread "bs-badpf6.tex" (bs114d A B C).
Not OK.

Verification Problem:

Let x be an element of $A \cup (B \cap C)$
The type you gave for x does not seem to be correct.

Proof: [bs-badpf7] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap C) \cup (B \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$.

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (A \cap C)$ are equal. QED

>proofread "bs-badpf7.tex" (bs114d A B C).
Not OK.

Verification Problem:

let y be an element of $(A \cap C) \cup (B \cap C)$
The type you gave for y does not seem to be correct.

Proof: [bs-badpf8] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$.

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (A \cap C)$ are equal. QED

>proofread "bs-badpf8.tex" (bs114d A B C).
Not OK.

Verification Problem:

Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$

The statement is not so obvious to me, Dave.

Proof: [bs-badpf9] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

Conversely, let y be an element of $(A \cap B) \cup (A \cap C)$. Then, either (iii) $y \in A \cap B$, or (iv) $y \in A \cap C$. It follows that $y \in A$, and either $y \in B$ or $y \in C$. Therefore, $y \in A$ and $y \in B \cup C$ so that $y \in A \cap (B \cup C)$. Hence $(A \cap B) \cup (A \cap C)$ is a subset of $A \cap (B \cup C)$.

In view of Definition 1.1.1, we conclude that the sets $A \cap (B \cup C)$ and $(A \cap B) \cup (A \cap C)$ are equal. QED

>proofread "bs-badpf9.tex" (bs114d A B C).
Not OK.

Verification Problem:

Therefore, either $x \in A \cap B$ or $x \in A \cap C$
I am not sure that statement follows, Dave.

Proof: [bs-incompletepf] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

QED

>proofread "bs-incompletepf.tex" (bs114d A B C).
File ended without completing proof.

Proof: [bs-badincompletepf] In order to give a sample proof, we shall prove the first equation in (d). Let x be an element of $A \cap (B \cup C)$, then $x \in A$ and $x \in B \cup C$. This means that $x \in A$, and either $x \in B$ or $x \in C$. Hence we either have (i) $x \in A$ and $x \in B$, or we have (ii) $x \in A$ and $x \in C$. Therefore, either $x \in A \cap B$ or $x \in A \cap C$, so $x \in (A \cap B) \cup (A \cap C)$. This shows that $A \cap (B \cup C)$ is a subset of $(A \cap B) \cup (A \cap C)$.

QED

>proofread "bs-badincompletepf.tex" (bs114d A B C).
Not OK.

Verification Problem:

```
so $x\in
(A\cap B) \cup (A\cap C)$
I am not sure that statement follows, Dave.
```

2. Extending the Proofreader

To extend the proofreader to cover more latex proofs, the following work would need to be done.

First, one must make sure there is enough mathematics encoded into Scunak that it can understand the proof (with steps in the text corresponding to steps in Scunak). One must also set some global variables so that Scunak knows which rules are valid to use during proofreading (see, e.g., `bs-1-1-4d-mode.lisp`).

Second, one must extend the natural language rules for converting the latex source into Scunak commands. This is fairly easy to do. One simply adds *Scunakation methods* (corresponding to parsing rules) by evaluating lisp S-expressions of the form:

```
(scunakator~defmethod <NAME>
  (precondition (text-fits <RULE>))
  (postcondition (text-is <CATEGORY>))
  (priority <NUMBER>)
  (defn <S-EXPRESSION>)
  (help-msg <STRING>))
```

For concrete examples of such methods, see `src/scunakation-methods.lisp`. The expressions defining Scunakation methods can be placed in a lisp file which is loaded when the user starts Scunak by using the `-p` option.

APPENDIX A

API

There are two interaction modes for Scunak: with a command line interface and with a socket interface. There are three top levels: the main top level, the Scip top level, and the tutor top level. In each top level, the available commands differ. In each interaction mode, the syntax for issuing commands differs.

1. Specifying Data

Data is sent as S-expressions. This means we cannot directly use the PAM syntax. Instead, we use S-expression approximations of PAM syntax. We sometimes refer to this as S-PAM or “pamela” syntax.

1.1. Names. Sending names *S-NAME* through sockets. Since LISP is (usually) case-insensitive, with an upper case default, names are usually sent enclosed either in double quotes (giving lisp strings) or in vertical bars (giving lisp symbols). If the name happens to be all upper case, the double quotes or vertical bars can be omitted. For example, then name *Ab* can be sent as "*Ab*" or *|Ab|*. The name *AB* can be sent as "*AB*" or *|AB|* or *AB*.

1.2. Generic Application. A generic application (*S-GENAP*) is either a name *S-NAME* or a nonempty list of *S-NAME*'s. In the second case, all the names on the list except the first should refer to local parameters.

1.3. Binary Operators. The user definable binary operators *S-UBINOP* are

- @ (user definable)
- @@ (user definable)
- + (user definable)
- ++ (user definable)
- * (user definable)
- ** (user definable)
- - (user definable)
- -- (user definable)
- / (user definable)
- // (user definable)
- < (user definable)
- > (user definable)

The binary operators *S-BINOP* which can be used in infix in extraction terms are given below:

- != (not equal)
- == (equal)
- <= (subset)

- => (implies)
- <=> (equivalence)
- & (and)
- OR (disjunction)
- IN (set membership)
- |;| (adjoining an element to a set)
- CUP (binary union)
- CAP (binary intersection)
- TIMES (Cartesian product of sets)
- *S-UBINOP* (user definable, see above)

1.4. Binders. The binders which can be used to form extraction terms are given below.

- FORALL
- EXISTS
- EXISTS1

1.5. Extraction and Normal Terms. We define the pamel syntax for sending extraction terms *S-EXTR* and normal terms *S-NORM* through sockets by mutual recursion. First, we define *S-EXTR*:

- *S-NAME*
- (*S-EXTR S-BINOP S-EXTR*)
- (*S-BINDER S-NAME S-EXTR S-EXTR*) For example, (FORALL "x" *A P*) means $\forall x \in A.P$.
- (*S-BINDER (S-NAME⁺) S-EXTR S-EXTR*) For example, (FORALL ("x" "y" "z") *A P*) means $\forall x, y, z \in A.P$.
- (*SETOF S-NAME S-EXTR S-EXTR*) For example, (SETOF "x" *A P*) means $\{x \in A | P\}$.
- (*SETOFPAIRS S-NAME S-NAME S-EXTR S-EXTR S-EXTR*) For example, (SETOFPAIRS "x" "y" *A B P*) means $\{(x, y) \in A \times B | P\}$.
- (*CLASSOF S-NAME S-EXTR*)
- (*SETENUM S-EXTR⁺*) For example, (SETENUM *A B*) denotes $\{A, B\}$. (You must give at least one *S-EXTR*, i.e., this is not a way to specify the empty set. To specify the empty set, give "emptyset".)
- (*S-NAME S-NORM**) This refers to applying the name at the head to the arguments given by normal terms.

Next, we define *S-NORM*:

- (*PAIR S-EXTR S-EXTR*)
- (*LAM S-NAME S-EXTR*)
- (*LAM (S-NAME⁺) S-EXTR*)
- (*LAM S-NAME (PAIR S-EXTR S-EXTR)*)
- (*LAM (S-NAME⁺) (PAIR S-EXTR S-EXTR)*)
- *S-EXTR*

Sometimes Scunak sends a term out of a socket in the form *S-EXTR-OUT* or *S-NORM-OUT*. An *S-EXTR-OUT* is

- (*APP S-EXTR-OUT . S-NORM-OUT*)
- (*FST . S-EXTR-OUT*)
- (*SND . S-EXTR-OUT*)

- *NUMBER* (a deBruijn index)
- *SYMBOL* (a local parameter or a member of the signature)

An *S-NORM-OUT* is

- (LAM . *S-NORM-OUT*)
- (PAIR *S-EXTR-OUT* . *S-EXTR-OUT*)
- *S-EXTR-OUT*

1.6. Dependent Types. Below is the syntax for sending dependent types *S-DTYPE* as S-expressions to Scunak through sockets.

- (*S-DTYPE* -> *S-DTYPE*)
- (PI *S-NAME S-DTYPE S-DTYPE*)
- (PF *S-EXTR*)
- PROP
- OBJ
- *S-EXTR*

Sometimes Scunak sends a dependent type out of a socket in the form *S-DTYPE-OUT*:

- (DPI *S-DTYPE-OUT* . *S-DTYPE-OUT*)
- (DCLASS . *S-NORM-OUT*)
- (PF . *S-EXTR-OUT*)
- (OBJ)
- (PROP)

1.7. PAM vs. S-PAM. Here we give the correspondence between the PAM and S-PAM syntax for giving data.

Since sockets accept S-expressions instead of strings (for the most part), there is an S-expression version of the PAM syntax. We refer to this syntax as S-PAM or *pamela* syntax. This is used to send terms and types through the socket interface to Scunak. Here we give the correspondence between PAM and S-PAM syntax. In Appendix A, we give a complete specification of S-PAM syntax.

Some binary operators must be slightly modified from PAM to S-PAM to obtain S-expressions. We list this correspondence first.

$BINOP_{pam}$ PAM	$BINOP_{S-pam}$ S-PAM
@	@
@@	@@
+	+
++	++
*	*
**	**
-	-
--	--
/	/
//	//
<	<
>	>
!=	!=
==	==
<=	<=
=>	=>
<=>	<=>
&	&
	OR
::	IN
;	;
\cup	CUP
\cap	CAP
\times	TIMES

Next, extraction terms correspond as follows:

E_{pam} PAM	E_{S-pam} S-PAM
$NAME$	"NAME"
$(NAME N_{pam}^1 \dots N_{pam}^n)$	("NAME" $N_{S-pam}^1 \dots N_{S-pam}^n$)
$\{NAME: E_{pam} \mid E'_{pam}\}$	(SETOF "NAME" E_{S-pam} E'_{S-pam})
$\{\langle NAME, NAME' \rangle: E_{pam} \setminus E'_{pam} \mid E''_{pam}\}$	(SETOFFPAIRS "NAME" "NAME'" E_{S-pam} E'_{S-pam} E''_{S-pam})
$(E_{pam} BINOP_{pam} E'_{pam})$	$(E_{S-pam} BINOP_{S-pam} E'_{S-pam})$
{}	"emptyset"

Normal terms correspond as follows:

N_{pam} PAM	N_{S-pam} S-PAM
E_{pam}	E_{S-pam}
$\langle E_{pam}, E'_{pam} \rangle$	(PAIR E_{S-pam} E'_{S-pam})
$(\setminus x_1 \dots x_n \cdot E_{pam})$	(LAM ("x1" ... "xn") E_{S-pam})
$(\setminus x_1 \dots x_n \cdot \langle E_{pam}, E'_{pam} \rangle)$	(LAM ("x1" ... "xn") (PAIR E_{S-pam} E'_{S-pam}))

Types (we do not distinguish between *DTYPE0*, *DTYPE1* and *DTYPE2* since there is no need to do so) correspond as follows:

A_{pam}	PAM	A_{S-pam}	S-PAM
obj		OBJ	
set		OBJ	
prop		PROP	
(pf E_{pam})		(PF E_{S-pam})	
E_{pam}		E_{S-pam}	
($\{x : A_{pam}\} A'_{pam}$)		(PI $x A_{S-pam} A'_{S-pam}$)	
($A_{pam} \rightarrow A'_{pam}$)		($A_{S-pam} \rightarrow A'_{S-pam}$)	

2. Main Sockets

We describe the valid input and output of the main socket **A** (see Section 5). The main socket **B** receives no input, but can output any sequence of ASCII characters.

The valid input and output of socket **A** depends on the current state of Scunak (of course). Each message sent to Scunak via socket **A** should be a lisp S-expression followed by a newline character (ASCII 10).

Socket **A** is in a certain state and waiting for input when one of the following forms is output via socket **A**:

- **READY** This means Scunak is in the main top level waiting for a command.
- **READY-SCIP** This means Scunak is in the Scip top level waiting for a command.
- **READY-TUTOR** This means Scunak is in the tutor top level waiting for a command.
- **PROMPT-BOOL** This means Scunak is prompting for a boolean.
- **(PROMPT-NUM *NUM*)** This means Scunak is prompting for a number.
- **PROMPT-NAME** This means Scunak is prompting for a name.
- **PROMPT-LINE** This means Scunak is prompting for a line.
- **PROMPT-FILENAME** This means Scunak is prompting for a filename.
- **PROMPT-EXTR** This means Scunak is prompting for an extraction term.

Upon initialization, Scunak always sends the following output via socket **A**:

```
SCUNAK
1.0
READY
```

The final **READY** indicates Scunak is waiting for a top level command.

2.1. Top Level IO. If the last message from **A** was **READY**, then **A** expects to receive one of the following:

- **(Q)** Quit Scunak, closing all connections (equivalent forms are, **(QUIT)**, **(X)**, **(EXIT)**)
- **(LISP *S-EXPR*)** Evaluate the given S-expression in LISP and return the result (an arbitrary S-expression).
- **(PROVE *S-NAME*)** If the name is the name of an open claim of proof type, enter the Scip interactive prover top level. If the name is not the name of a claim, Scunak sends **PROVE-NOT-PROOF-TYPE** or **PROVE-NOT-CLAIM** via socket **A**.

- (TUTOR *S-GENAP*) If the generic application is an extraction with a type returning a proof type, then Scunak enters the tutor top level. Otherwise, Scunak sends BAD-TUTOR-CLAIM via socket **A**.
- (LET *S-NAME S-DTYPE*) Add a parameter (locally bound variable) to context.
- (CONST *S-GENAP S-DTYPE*) Add a new constant of the given type.
- (CLAIM *S-GENAP S-DTYPE*) Add a new claim of the given type.
- (ABBREV *S-GENAP S-DTYPE S-NORM*) Add a new abbreviation of the given type defined by the normal term.
- (ABBREVE *S-GENAP S-EXTR*) Add a new abbreviation defined by the extraction term, with Scunak extracting the type.
- (CLAIM2ABBREV *S-GENAP S-NORM*) Change a claim to be an abbreviation using the normal term as definition.
- (COERCION *S-EXTR*) If the extraction term has the type of a proof that a set **A** is a subset of type **B**, then add a coercion so that any term of class type (**in A**) can be used as a term of class type (**in B**).
- (NOTATION-INFIX *S-UBINOP S-EXTR*) Define the user definable binary operator using the extraction. The extracted type of the given extraction term should be a function type expected two arguments.
- (NOTATION *S-NAME S-EXTR*) Use the name as shorthand for the extraction. The notation will always immediately be expanded.
- (LOAD *STRING*) Load the PAM file named by the given string.
- (LOADL *STRING*) Load the lisp file named by the given string.
- (AUTHORS *STRING⁺*) Set the names of the authors for the following signature elements.
- (HELP) Print the general help information. (The help is intended for Scunak users without sockets.)
- (HELP *S-NAME*) Print help about the name. If the name is a signature element, then information about the signature element is output through socket **A**.
- (VAMPIRE *S-NAME*) Call Vampire on the given claim. This requires the global variable `*vampire-executable*` to be set to the executable file for Vampire.
- (INPUT-SIG-AGENT *STRING NUMBER STRING*) The first argument names a machine, the second argument gives a port number, and the last argument is a string with some arbitrary information. Scunak connects to a passive socket using the port number and machine name in order to obtain new signature information (see Section 3).
- (OUTPUT-SIG-AGENT *STRING NUMBER STRING NUMBERORNIL NUMBERORNIL*) The first argument names a machine, the second argument gives a port number, the third argument is a string with some arbitrary information. The fourth and fifth optional arguments are numbers indicating a beginning and ending “timestamp” on the signature. The default beginning time stamp is 0 (the time when Scunak started) and the default ending time stamp is the current time stamp. Scunak connects to a passive socket using the port number and machine name in order to send the part of the signature between the beginning and ending time stamps (see Section 4).

- (ADD-FILL-GAP-AGENT *S-NAME STRING NUMBER*) Add an external fill gap agent named by the name by connecting to the socket at the machine given by the string at the port given by the number (see Section 5).
- (ADD-FILL-GAP-AGENT-USABLE *S-NAME STRING NUMBER*) This command also adds a fill gap agent, but these fill gap agents will be sent an explicit “usable” set in addition to the actual gap (see Section 5).
- (REMOVE-FILL-GAP-AGENT *S-NAME*) Remove the fill gap agent named by the given name.
- (ALL-CLAIMS) Output a list of all claims in the signature.
- (SHOW-CTX) Output the current bound variables (parameters) in context.
- (CLEAR-CTX) Remove all bound variables (parameters) from the context.
- (PUSH-CTX) Push the current context onto a stack, so that one can return the context later.
- (POP-CTX) Pop to the previous context on the stack.
- (TYPEOF *S-EXTR*) Compute the type of the extraction. **A** sends ILL-TYPED if the term is ill-typed, and returns (TYPE *S-DTYPE-OUT*) if the term is well-typed.
- (JUSTIFY *S-EXTR*) If the extraction is a proposition, then find a proof term for the proof of the proposition. If the justify is not successful, **A** may send ILL-TYPED, NOT-A-PROP, or COULD-NOT-JUSTIFY. If successful, **A** sends (JUSTIFICATION *S-NORM-OUT*) giving the normal proof term.
- (UNIF+) Increase unification bounds.
- (UNIF-) Decrease unification bounds.
- (PROOFREAD *STRING S-GENAP*) Proofread the latex proof in the file named by string (see [8]). The *S-GENAP* should be an extraction with type proof of some proposition *M*, and the file should contain a \LaTeX proof of *M*.
- (USE *S-NAME**) lists names to be included on usable. Some special names used in this command are all (add all of the signature to usable), none (make usable empty), fo (add all first-order rules to the signature), defns (add all rules for folding and unfolding definitions), and so (add all second-order rules to usable). Any other name must refer to a specific signature element. Note that any call to USE resets usable, forgetting anything that was in the usable set before.
- (TUTOR-AUTO-BACK *S-NAME**) Use the named rules eagerly in the tutor and in the proofreader. (Note: This simply sets the global variable `*tutor-auto-back*`.)
- (TUTOR-STUDENT-USABLE *S-NAME**) Indicate the rules that both tutors and students are allowed to use. (Note: This simply sets the global variable `*sig-granularity-perfect*`.)
- (TUTOR-ONLY-USABLE *S-NAME**) Indicate the rules that only tutors but not students are allowed to use. (Note: This simply sets the global variable `*sig-granularity-toohigh*`.)

If the given input is not a command, NOT-A-COMMAND will be sent through socket **A**. Whenever the input fails to evaluate properly, FAILED should be sent through socket **A**.

2.2. Scip Top Level IO. If the last message from **A** was READY-SCIP, then **A** expects to receive one of the following:

- (Q) Quit Scip, returning to the main top level.
- (D) Current gap is done (i.e., can be trivially filled).
- (WILLSHOW *S-EXTR*) If the extraction is a proposition and the current goal can be easily reduced to this proposition, then reduce the goal.
- (CLEARLY *S-EXTR*) If the extraction is a proposition which easily follows from the current support, then add the proposition to the support as a new fact.
- (FACT *S-EXTR*) This is exactly the same as CLEARLY
- (HENCE *S-EXTR*) This is the same as CLEARLY but may force the last proposition on the support to be used.
- (CLAIM *S-EXTR*) If the extraction is a proposition, then add the proposition to the support and add a new gap to prove the proposition later.
- (CLAIMTYPE *S-DTYPE*) If the type corresponds to a first-order proof rule, then add it to the context and add a new gap to fill this type later.
- (LEMMA *S-NAME (S-NAME*) S-DTYPE*) Add a new claim with the given name, parameters and type to the global signature. Use this lemma to add a new fact to the context.
- (TYPEOF *S-EXTR*) Compute the type of the extraction. **A** sends ILL-TYPED if the term is ill-typed, and returns (TYPE *S-DTYPE-OUT*) if the term is well-typed.
- (APPLY *S-EXTR*) If the extraction is well-typed and has a type corresponding to a first-order proof rule, then add this type to the current supporting context as a new fact.
- (XMCASES) Scip prompts for a proposition and uses this to split the gap into two cases, one where the proposition holds and one where its negation holds.
- (CASES) Scip finds the most recent disjunction in the support and uses this to split the gap into two cases.
- (ADJCASES) Scip finds the most recent fact in the support of the form $x \in (y; A)$ and uses this to split the gap into a case where $x = y$ and a case where $x \in A$. Note that a special case of this is when there is a fact $x \in \{y, a_1, \dots, a_n\}$.
- (EXISTS *S-NAME*) Scip finds the most recent existential statement in the support and applies existential elimination with the eigenvariable named by the given name.
- (SETUNIONEXISTS *S-NAME*) This is the same as EXISTS, but uses $x \in \bigcup A$ instead of an existential proposition.
- (CONTRADICTION) If proving a negation of M , assume M and prove false (this is simply negation introduction). If proving false, then take no action. Otherwise, assume the negation of the goal and prove false.
- (INTRO) Perform a “simple” introduction: negation introduction, implication introduction, conjunction introduction, forall introduction, subset introduction, reducing equivalence to a conjunction of implications, functional extensionality for equality of functions, and set extensionality for equality of sets.
- (ELIM) Recursively apply all “simple” eliminations. These include conjunction elimination, implication elimination and forall elimination. Though note that implication and forall eliminations simply change the “reified”

proposition to be a non-atomic type. This does not, for example, choose the relevant instantiation. (In other words, the proposition becomes a “rule”.)

- (UNFOLDGOAL) Unfold an abbreviation in the goal.
- (UNFOLDGOALHEAD) Unfold the abbreviation at the head of the goal.
- (UNFOLD *S-NAME*) Unfold the abbreviation named by name in the support.
- (UNFOLDHEAD *S-NAME*) Unfold a definition at the head of a support. The name can either name the support element or the abbreviation.
- (ARGS=) Reduce a goal of the form $(f a_1 \dots a_n) = (f b_1 \dots b_n)$ to showing n gaps of the form $a_i = b_i$.
- (REDUCEGOAL) Reduce the goal using (internal) β or η reduction, or reduction of $a \in \{x \in A \mid \varphi(x)\}$ to $\varphi(a)$ (if $a \in A$).
- (B) Find all backwards steps reducing the current gap to a new gap.
- (B2) Find all backwards steps reducing the current gap to two new gaps.
- (F) Find some forward steps for the current support.
- (F2) Find more forward steps for the current support.
- (HELP) Print general help for Scip.
- (HELP *S-NAME*) Print help for the given name.
- (PSTATUS) Print the current status (number and nature of remaining gaps).
- (PTERM) Print the current (open) proof term.
- (CHKTERM) Type check the current proof term (for sanity).
- (PPLAN) Print the supporting context and goal of the current gap.
- (PPLAN*) Same as PPLAN, except proof parts of pairs are omitted.
- (CHOOSE-TASK) Scip prompts for a number and uses this number to focus on a different gap. PSTATUS lists the gaps with the appropriate numbers.
- (UNDO) Undo the last step. Alternative: (U).
- (USE *S-NAME**) lists names to be included on usable. Some special names used in this command are all (add all of the signature to usable), none (make usable empty), fo (add all first-order rules to the signature), defns (add all rules for folding and unfolding definitions), and so (add all second-order rules to usable). Any other name must refer to a specific signature element. Note that any call to USE resets usable, forgetting anything that was in the usable set before.
- (UNIF+) Increase unification bounds.
- (UNIF-) Decrease unification bounds.
- (LISP *S-EXPR*) Evaluate the given S-expression and return the result.

If the given input is not a scip command, NOT-SCIP-COMMAND is sent through **A**.

When Scip exits **A** may output PFTERM-HAS-FREES (if there are remaining free variables in the final proof term), PFTERM-WRONG-TYPE (if the final proof term has the wrong type), or (SCIP-PFTERM *S-NORM-OUT*). In this final case, Scip tries to save the proof in a file. This means Scip prompts for a boolean (NIL or T). If T, then Scip prompts for a string (giving the PAM file name to save a PAM version of the proof). Scip next prompts for a boolean (NIL or T). If T, then Scip prompts for a string (giving the lisp file name to save a lisp version of the proof). Finally, Scip sends SCIP-OUT before returning to the main top level.

2.3. Tutor Top Level IO. If the last message from **A** was READY-TUTOR, then **A** expects to receive one of the following:

- (STATUS) This prints the current status of the tutor, including all alternatives.
- (HELP) This prints generic help for the tutor top level.
- (QED) This indicates a current task is completed. Alternative: (DONE)
- (UNDO)
- (Q) (or (X) or (QUIT) or (EXIT) or (I-GIVE-UP)) Exit the tutor top level and return to the main top level.
- (LET *S-NAME*) Assume a variable of type obj (i.e., set) is in context, if there is a reason to do so.
- (LET (*S-NAME*⁺)) Assume variables of type obj (i.e., set) is in context, if there is a reason to do so.
- (LET *S-NAME S-DTYPE*) Assume a variable of the given type is in context, if there is a reason to do so.
- (LET (*S-NAME*⁺) *S-DTYPE*) Assume a variable of the given type is in context, if there is a reason to do so.
- (ASSUME *S-EXTR*) If the extraction is a proposition and there is a reason to assume it, then do so.
- (WILLSHOW *S-EXTR*) If the extraction is a proposition and the current goal can be reduced to this proposition, then do so.
- (CLEARLY *S-EXTR*) If the extraction is a proposition and can be justified by the current supports, then add it to the support.
- (HENCE *S-EXTR*) This is the same as CLEARLY, but may force the use of the last proposition on the support.
- (EXISTS *S-NAME S-EXTR S-EXTR*) If the existential statement $\exists x \in A\varphi(x)$ can be justified from the current context (where x is the name, A is the first extraction and $\varphi(x)$ is the second extraction), then justify the proposition and immediately apply existential elimination with the eigenvariable x .

After most commands, the tutor will send either OK or NOT-OK through **A** to indicate if the “step” was in some sense valid. Sometimes more information containing potential “diagnoses” of the problem will also be sent. A diagnosis is of the form (DIAGNOSIS *STEP-DIAGNOSIS*) where *STEP-DIAGNOSIS* is of one of the following forms:

- STEP-TOO-BIG
- STEP-TOO-SMALL
- STEP-IRRELEVANT
- STEP-UNJUSTIFIED
- INSUFFICIENT-TO-JUSTIFY-GOAL
- UNTRUE-CONCLUSION
- (EXPECTED-CONCLUSION *S-EXTR-OUT*)
- (ILL-FORMED-TYPE *S-EXPR*)
- (ILL-FORMED-PROP *S-EXPR*)
- (ILL-FORMED-ASSUMPTION *S-EXPR*)
- (BAD-TYPE-FOR *S-NAME*)
- NO-REASON-TO-MAKE-AN-ASSUMPTION
- NO-REASON-TO-LET
- (SHOULD-ASSUME *S-EXTR-OUT*)
- (SHOULD-LET *S-DTYPE-OUT*)

NOTE: Several diagnoses may be sent, followed by OK indicating that the step is okay. A step is only NOT-OK if *every* possible alternative situation has some diagnosis of a problem.

Whenever a command did not change the current state of the tutor, Scunak sends TUTOR-STATUS-UNCHANGED via **A**. If the tutor is sent something which is not a command, then NOT-TUTOR-COMMAND is sent through **A**.

Upon completion, the tutor sends via **A** either STUDENT-SUCCESSFUL or STUDENT-FAILED. Finally, the tutor sends EXIT-TUTOR via **A** and Scunak returns to the main top level.

2.4. Prompting for Booleans. If the last message from **A** was PROMPT-BOOL, then **A** expects to receive NIL or T.

2.5. Prompting for Numbers. If the last message from **A** was a list starting with PROMPT-NUM, then the second element of the list should be a number n and **A** expects to receive a number (i.e., a sequence of digits) between 0 and n (inclusive).

2.6. Prompting for Names. If the last message from **A** was PROMPT-NAME, then **A** expects to receive an (LINE *STRING*).

2.7. Prompting for Lines. If the last message from **A** was PROMPT-LINE, then **A** expects to receive an (LINE *STRING*).

2.8. Prompting for Filenames. If the last message from **A** was PROMPT-FILENAME, then **A** expects to receive (LINE *STRING*).

2.9. Prompting for Extraction Terms. If the last message from **A** was PROMPT-EXTR, then **A** expects to receive an *S-EXTR*.

3. Input Agents

When Scunak connects to an input signature agent with a socket **I**, Scunak first sends a line SCUNAK-INPUT to **I**. Next Scunak sends to **I** the *info STRING* the Scunak user asked to send. Next, the external agent sends either the S-expression PAMELA or LISP. If Scunak receives PAMELA, then it reads the remaining S-expressions from **I** and executes them as commands at the main top level (see Section 2.1). If Scunak receives LISP, then the remaining S-expressions from **I** are evaluated in lisp. The input agent is finished when the socket **I** is closed by the remote agent.

4. Output Agents

When Scunak connects to an output agent via a socket **O**, Scunak sends a line SCUNAK-OUTPUT and then the *info STRING* the user asked to send. Scunak then reads either PAM, PAM*, LISP, or LISP-PROPS from **O**. In each case, Scunak then sends the signature elements between the indicated beginning and ending time stamps to **O** in the indicated format. These formats are:

- PAM PAM syntax
- PAM* PAM syntax except the proof half of pairs are omitted
- LISP Lisp S-expressions which can be evaluated by Scunak to define the signature element.
- LISP-PROPS The set of S-expression properties of each signature elements. In this format, we include automatically generated signature elements such as those for folding and unfolding definitions.

After sending the subset of the signature, Scunak closes the socket **O**.

5. Fill Gap Agents

When a fill gap agent is added, a socket **F** to the fill gap agent is opened. Scunak sends SCUNAK-FILL through **F**. Then in the future when a gap needs to be filled, Scunak will send requests to **F** in one of the following forms:

- (SCUNAK-FILL-REQUEST *CTX S-DTYPE-OUT*) where *CTX* is (*S-DTYPE-OUT**)
- (SCUNAK-FILL-REQUEST *CTX S-DTYPE-OUT USABLE*) where *CTX* is (*S-DTYPE-OUT**) and *USABLE* is (*NAME**).

Scunak then waits to receive an S-expression from **F**. Ultimately, when **F** sends an expression of the form (ANSWER *S-EXPR*), then the *S-EXPR* (which should be a proof term for the gap or NIL) is returned.

Before **F** returns an ANSWER, it may request information from Scunak in the following forms:

- (SIGELT *S-NAME*) If the name names a constant, Scunak sends via **F** (CONST *S-NAME S-DTYPE-OUT*). If the name names an abbreviation, Scunak sends via **F** (ABBREV *S-NAME S-DTYPE-OUT S-NORM-OUT*). If the name names a claim, Scunak sends via **F** (CLAIM *S-NAME S-DTYPE-OUT*). Otherwise, Scunak sends via **F** (UNKNOWN *S-NAME*).
- (LISP *S-EXPR*) Scunak evaluates the S-expression.

APPENDIX B

Socket Interaction Examples

In this appendix we give a trace of the input and output of sockets **A** and **B** when using the Scunak tutor via sockets. We do not include every diagnosis of the form (DIAGNOSIS *STEP-DIAGNOSIS*) output by Scunak via socket **A** (since there are far too many repetitions), but do include a representative sample. These are the only elements left out of the trace.

1. WOZ1 Examples

Scunak Initialization.

```
A sends SCUNAK
A sends 1.0
B sends Scunak Text Output Socket
B sends Loading main patch file.
B sends ; Loading /home/cebrown/scunak/data/patch.lisp
B sends ; Loading /home/cebrown/scunak/data/macui-kernel.lisp
B sends ; Loading /home/cebrown/scunak/data/woz1-sm.lisp
A sends READY
```

Declaration of local parameters:

```
A gets (LET "U" OBJ)
A sends READY
A gets (LET "A" ("powerset" "U"))
A sends READY
A gets (LET "B" ("powerset" "U"))
A sends READY
A gets (LET "C" ("powerset" "U"))
A sends READY
A gets (LET "D" ("powerset" "U"))
A sends READY
```

Declaration of usable sets:

```
A gets (TUTOR-AUTO-BACK "setextsub" "orE"
                        "powersetII")
A sends READY
```

A gets (TUTOR-STUDENT-USABLE "notE" "xmcases" "eqI"
 "notI" "contradiction" "syseq" "transeq"
 "symtrans1eq" "symtrans2eq" "binunionT"
 "binintersectT" "powersetT" "setminusT"
 "complementT" "setextT" "subsetTI" "powersetTI"
 "powersetTE" "powersetTI1" "powersetTE1"
 "complementTI" "complementTE" "complementTI1"
 "complementTE1" "binintersectTEL"
 "binintersectTER" "binintersectTI" "binunionTE1"
 "binunionTE" "binunionTIL" "binunionTIR"
 "binintersectTELcontra" "binintersectTERcontra"
 "binunionTILcontra" "binunionTIRcontra"
 "binunionTEcontra" "demorgan1a" "demorgan2a"
 "demorgan2a1" "demorgan2a2" "demorgan1b"
 "demorgan2b2" "demorgan2b" "demorgan1"
 "demorgan2" "demorgan1Eq2" "demorgan2Eq2"
 "demorgan3Eq2" "demorgan4Eq2" "contrasubsetT"
 "contrasubsetT1" "binintersectTSub1"
 "binintersectTSub2" "binunionTSub1"
 "binunionTSub2" "impI" "impE" "woz13rule1"
 "woz13rule2" "woz13rule3" "doubleComplementSub1"
 "doubleComplementSub2" "doubleComplementEq"
 "subsetTrans")

A sends OK

A sends READY

A gets (TUTOR-ONLY-USABLE "woz13rule4")

A sends OK

A sends READY

1.1. First WOZ1 Example. $K(A) \in \mathcal{P}(K(A \cap B))$

A gets (TUTOR ("woz1_1" "U" "A" "B"))

B sends Hello, Chad. Please try to prove the following:

B sends $\vdash ((\text{complementT } U \text{ } A) :: (\text{powersetT } U (\text{complementT } U (\text{binintersectT } U \text{ } A \text{ } B))))$

A sends READY-TUTOR

A gets (LET ("x") "U")

A sends (DIAGNOSIS

(SHOULD-ASSUME

(APP |not| APP

(APP |subset| FST APP (APP |complementT| . U

. A)

FST APP (APP |complementT| . U) APP

(APP (APP |binintersectT| . U) . A) . B)))

A sends OK

B sends OK

A sends READY-TUTOR

A gets (ASSUME ("x" IN ("complementT" "U" "A")))

A sends OK

B sends OK

A sends READY-TUTOR

```

A gets (CLEARLY ("not" ("x" IN "A")))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY
      ("not"
       ("x" IN ("binintersectT" "U" "A" "B"))))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends STUDENT-SUCCESSFUL
B sends Congratulations, you're done with the proof, Chad!
B sends Successful Term:
B sends (\x0 x1 x2....)
A sends EXIT-TUTOR
A sends READY
A gets (TUTOR ("woz1_1" "U" "A" "B"))
B sends Hello, Chad. Please try to prove the following:
B sends |- ((complementT U A)::(powersetT U (complementT U (binintersectT U A B))))
A sends READY-TUTOR
A gets (LET ("x") "U")
A sends (DIAGNOSIS
      (SHOULD-ASSUME
       (APP |not| APP
        (APP |in| FST APP (APP |powersetT| . U) APP
         (APP |complementT| . U) APP
         (APP (APP |binintersectT| . U) . A) . B)
         FST APP (APP |complementT| . U) . A)))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME ("x" IN ("complementT" "U" "A")))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY ("not" ("x" IN "A")))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY
      ("not"
       ("x" IN ("binintersectT" "U" "A" "B"))))
A sends OK
B sends OK
A sends READY-TUTOR

```

```

A gets    (CLEARLY
           ("complementT" "U" "A") <=
           ("complementT" "U"
            ("binintersectT" "U" "A" "B"))))
A sends   (DIAGNOSIS STEP-UNJUSTIFIED)
A sends   NOT-OK
A sends   TUTOR-STATUS-UNCHANGED
B sends   Not OK.
B sends   The statement is not so obvious to me, Chad.
A sends   READY-TUTOR
A gets    (CLEARLY
           ("x" IN
           ("complementT" "U"
            ("binintersectT" "U" "A" "B"))))
A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
           ("complementT" "U" "A") <=
           ("complementT" "U"
            ("binintersectT" "U" "A" "B"))))
A sends   (DIAGNOSIS STEP-UNJUSTIFIED)
A sends   (DIAGNOSIS
           (EXPECTED-CONCLUSION
            (APP
             (APP |in| FST APP (APP |complementT| . U) APP
              (APP (APP |binintersectT| . U) . A) . B)
              FST . |x|)))
A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (QED)
A sends   STUDENT-SUCCESSFUL
B sends   Congratulations, you're done with the proof, Chad!
B sends   Successful Term:
B sends   (\x0 x1 x2....)
A sends   EXIT-TUTOR
A sends   READY

```

1.2. Second WOZ1 Example. $K((A \cup B) \cap (C \cup D)) = (K(A) \cap K(B)) \cup (K(C) \cap K(D))$

```

A gets    (TUTOR ("woz1_2" "U" "A" "B" "C" "D"))
B sends   Hello, Chad. Please try to prove the following:
B sends   |- ((complementT U (binintersectT U (binunionT U A B) (binunionT U C D)))=(binu
A sends   READY-TUTOR

```

```

A gets    (CLEARLY
           (("complementT" "U"
            ("binintersectT" "U" ("binunionT" "U" "A" "B")
             ("binunionT" "U" "C" "D"))))
           ==
           ("binunionT" "U"
            ("complementT" "U" ("binunionT" "U" "A" "B"))
            ("complementT" "U"
             ("binunionT" "U" "C" "D")))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
           (("complementT" "U" ("binunionT" "U" "A" "B")) ==
            ("binintersectT" "U" ("complementT" "U" "A")
             ("complementT" "U" "B"))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
           (("complementT" "U" ("binunionT" "U" "C" "D")) ==
            ("binintersectT" "U" ("complementT" "U" "C")
             ("complementT" "U" "D"))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (QED)
A sends   (DIAGNOSIS NOT-YET-DONE)
A sends   STUDENT-SUCCESSFUL
A sends   EXIT-TUTOR
B sends   Congratulations, you're done with the proof, Chad!
B sends   Successful Term:
B sends   (\x0 x1 x2 x3 x4...)
A sends   READY

```

1.3. Third WOZ1 Example. $(A \cap B) \in \mathcal{P}((A \cup C) \cap (B \cup C))$

```

A gets    (TUTOR ("woz1_3" "U" "A" "B" "C"))
B sends   Hello, Chad. Please try to prove the following:
B sends   |- ((binintersectT U A B)::(powersetT U (binintersectT U (binunionT U A C) (binu
A sends   READY-TUTOR
A gets    (CLEARLY ("A" <= ("binunionT" "U" "A" "C"))
A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY ("B" <= ("binunionT" "U" "B" "C"))
A sends   OK
B sends   OK
A sends   READY-TUTOR

```

```

A gets (QED)
A sends (DIAGNOSIS NOT-YET-DONE)
A sends NOT-OK
A sends TUTOR-STATUS-UNCHANGED
B sends Not OK
B sends You don't seem to actually be done with anything, Chad.
A sends READY-TUTOR
A gets (CLEARLY
                                         (("binintersectT" "U" "A" "B") <=
                                         ("binintersectT" "U" ("binunionT" "U" "A" "C")
                                         ("binunionT" "U" "B" "C"))))
A sends (DIAGNOSIS STEP-TOO-BIG)
A sends NOT-OK
A sends TUTOR-STATUS-UNCHANGED
B sends Not OK.
B sends How does that follow, Chad? Try using smaller steps.
A sends READY-TUTOR
A gets (CLEARLY
                                         (("binintersectT" "U" "A" "B") <=
                                         ("binunionT" "U" "A" "C"))))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY
                                         (("binintersectT" "U" "A" "B") <=
                                         ("binunionT" "U" "B" "C"))))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY
                                         (("binintersectT" "U" "A" "B") <=
                                         ("binintersectT" "U" ("binunionT" "U" "A" "C")
                                         ("binunionT" "U" "B" "C"))))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends STUDENT-SUCCESSFUL
B sends Congratulations, you're done with the proof, Chad!
B sends Successful Term:
B sends (\x0 x1 x2 x3....)
A sends EXIT-TUTOR
A sends READY

```

1.4. Fourth WOZ1 Example. $A \subseteq K(B)$ implies $B \subseteq K(A)$

```

A gets (TUTOR ("woz1_4" "U" "A" "B"))
B sends Hello, Chad. Please try to prove the following:
B sends {x0:|- (A<=(complementT U B))}|- (B<=(complementT U A))
A sends READY-TUTOR
A gets (ASSUME ("A" <= ("complementT" "U" "B")))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends (DIAGNOSIS NOT-YET-DONE)
B sends Not OK
B sends You don't seem to actually be done with anything, Chad.
A sends NOT-OK
A sends TUTOR-STATUS-UNCHANGED
A sends READY-TUTOR
A gets (LET ("x") "U")
A sends (DIAGNOSIS
                                (SHOULD-ASSUME
                                (APP |not| APP (APP |subset| FST . B) FST APP
                                (APP |complementT| . U) . A)))

A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME ("x" IN "B"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY ("not" ("x" IN "A")))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends STUDENT-SUCCESSFUL
B sends Congratulations, you're done with the proof, Chad!
B sends Successful Term:
B sends (\x0 x1 x2 x3....)
A sends EXIT-TUTOR
A sends READY
A gets (TUTOR ("woz1_4" "U" "A" "B"))
B sends Hello, Chad. Please try to prove the following:
B sends {x0:|- (A<=(complementT U B))}|- (B<=(complementT U A))
A sends READY-TUTOR
A gets (ASSUME ("A" <= ("complementT" "U" "B")))
A sends OK
B sends OK
A sends READY-TUTOR

```

```

A gets (LET ("x") "U")
A sends (DIAGNOSIS
        (SHOULD-ASSUME
         (APP |not| APP (APP |subset| FST . B) FST APP
          (APP |complementT| . U) . A)))

A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME ("x" IN "B"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY ("not" ("x" IN "A")))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY ("x" IN ("complementT" "U" "A")))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends STUDENT-SUCCESSFUL
B sends Congratulations, you're done with the proof, Chad!
B sends Successful Term:
B sends (\x0 x1 x2 x3....)
A sends EXIT-TUTOR
A sends READY

```

1.5. Fifth WOZ1 Example. $K(A \cup B) \in \mathcal{P}(K(A))$

```

A gets (TUTOR ("woz1_5" "U" "A" "B"))
B sends Hello, Chad. Please try to prove the following:
B sends |- ((complementT U (binunionT U A B))::(powersetT U (complementT U A)))
A sends READY-TUTOR
A gets (LET ("x") "U")
A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME
        ("x" IN
         ("complementT" "U"
          ("binunionT" "U" "A" "B"))))

A sends OK
B sends OK
A sends READY-TUTOR

```

```

A gets      (CLEARLY
                                                    ("not" ("x" IN ("binunionT" "U" "A" "B"))))
A sends     OK
B sends     OK
A sends     READY-TUTOR
A gets      (CLEARLY ("not" ("x" IN "A")))
A sends     OK
B sends     OK
A sends     READY-TUTOR
A gets      (CLEARLY ("x" IN ("complementT" "U" "A")))
A sends     OK
B sends     OK
A sends     READY-TUTOR
A gets      (QED)
A sends     STUDENT-SUCCESSFUL
B sends     Congratulations, you're done with the proof, Chad!
B sends     Successful Term:
B sends     (\x0 x1 x2....)
A sends     EXIT-TUTOR
A sends     READY
A gets      (TUTOR ("woz1_5" "U" "A" "B"))
B sends     Hello, Chad. Please try to prove the following:
B sends     |- ((complementT U (binunionT U A B))::(powersetT U (complementT U A)))
A sends     READY-TUTOR
A gets      (LET ("x") "U")
A sends     OK
B sends     OK
A sends     READY-TUTOR
A gets      (ASSUME
                                                    ("x" IN
                                                    ("complementT" "U"
                                                    ("binunionT" "U" "A" "B"))))
A sends     OK
B sends     OK
A sends     READY-TUTOR
A gets      (CLEARLY
                                                    ("not" ("x" IN ("binunionT" "U" "A" "B"))))
A sends     OK
B sends     OK
A sends     READY-TUTOR
A gets      (CLEARLY ("not" ("x" IN "A")))
A sends     OK
B sends     OK
A sends     READY-TUTOR
A gets      (CLEARLY ("x" IN ("complementT" "U" "A")))
A sends     OK
B sends     OK
A sends     READY-TUTOR

```

```

A gets    (CLEARLY
          ("complementT" "U" ("binunionT" "U" "A" "B")) <=
          ("complementT" "U" "A"))))
A sends   (DIAGNOSIS
          (EXPECTED-CONCLUSION
          (APP
          (APP |in| FST APP (APP |powersetT| . U) APP
          (APP |complementT| . U) . A)
          FST APP (APP |complementT| . U) APP
          (APP (APP |binunionT| . U) . A) . B)))
A sends   (DIAGNOSIS STEP-UNJUSTIFIED)
A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (QED)
A sends   STUDENT-SUCCESSFUL
B sends   Congratulations, you're done with the proof, Chad!
B sends   Successful Term:
B sends   (\x0 x1 x2....)
A sends   EXIT-TUTOR
A sends   READY
A gets    (TUTOR-STUDENT-USABLE "woz1_4" "notE" "xmcases"
          "eqI" "notI" "contradiction" "symeq" "transeq"
          "symtrans1eq" "symtrans2eq" "binunionT"
          "binintersectT" "powersetT" "setminusT"
          "complementT" "setextT" "subsetTI" "powersetTI"
          "powersetTE" "powersetTI1" "powersetTE1"
          "complementTI" "complementTE" "complementTI1"
          "complementTE1" "binintersectTEL"
          "binintersectTER" "binintersectTI" "binunionTE1"
          "binunionTE" "binunionTIL" "binunionTIR"
          "binintersectTELcontra" "binintersectTERcontra"
          "binunionTILcontra" "binunionTIRcontra"
          "binunionTEcontra" "demorgan1a" "demorgan2a"
          "demorgan2a1" "demorgan2a2" "demorgan1b"
          "demorgan2b2" "demorgan2b" "demorgan1"
          "demorgan2" "demorgan1Eq2" "demorgan2Eq2"
          "demorgan3Eq2" "demorgan4Eq2" "contrasubsetT"
          "contrasubsetT1" "binintersectTSub1"
          "binintersectTSub2" "binunionTSub1"
          "binunionTSub2" "impI" "impE" "woz13rule1"
          "woz13rule2" "woz13rule3" "doubleComplementSub1"
          "doubleComplementSub2" "doubleComplementEq"
          "subsetTrans")
A sends   OK
A sends   READY

```

```

A gets (TUTOR ("woz1_5" "U" "A" "B"))
B sends Hello, Chad. Please try to prove the following:
B sends |- ((complementT U (binunionT U A B)):(powersetT U (complementT U A)))
A sends READY-TUTOR
A gets (CLEARLY ("A" <= ("binunionT" "U" "A" "B")))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY
                                         (("binunionT" "U" "A" "B") <=
                                          ("complementT" "U"
                                           ("complementT" "U"
                                            ("binunionT" "U" "A" "B")))))

A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY
                                         ("A" <=
                                          ("complementT" "U"
                                           ("complementT" "U"
                                            ("binunionT" "U" "A" "B")))))

A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY
                                         (("complementT" "U" ("binunionT" "U" "A" "B")) <=
                                          ("complementT" "U" "A")))

A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends STUDENT-SUCCESSFUL
B sends Congratulations, you're done with the proof, Chad!
B sends Successful Term:
B sends (\x0 x1 x2....)
A sends EXIT-TUTOR
A sends READY

```

2. WOZ2 Examples

WOZ2 Relation Examples. Initialization.

```

A sends SCUNAK
A sends 1.0
B sends Scunak Text Output Socket
B sends Loading main patch file.
B sends ; Loading /home/cebrown/scunak/data/patch.lisp
B sends ; Loading /home/cebrown/scunak/data/macu-kernel.lisp
B sends ; Loading /home/cebrown/scunak/data/woz2-sm.lisp
A sends READY
A gets (LET "M" OBJ)
A sends READY
A gets (LET "R" ("breln1" "M"))
A sends READY
A gets (LET "S" ("breln1" "M"))
A sends READY
A gets (LET "T" ("breln1" "M"))
A sends READY
A gets (TYPEOF ("woz2W" "M" "R"))
A sends (TYPE
(DPI (DCLASS APP |breln1| . M) PF APP
(APP |eq| FST APP (APP |breln1inv| . M) APP
(APP (APP |breln1comp| . M) . R) . 0)
FST APP
(APP (APP |breln1comp| . M) APP
(APP |breln1inv| . M) . 0)
APP (APP |breln1inv| . M) . R))
B sends Type: {x4:(breln1 M)}|- ((breln1inv M (breln1comp M R x4))==(breln1comp M (breln1
A sends READY
A gets (TUTOR-AUTO-BACK "settextsub" "orE")
A sends READY
A gets (TUTOR-STUDENT-USABLE "breln1invprop" "breln1inv"
"breln1invI" "breln1invE" "breln1compprop"
"breln1comp" "breln1compI" "breln1compE"
"breln1unionprop" "breln1union" "breln1unionI"
"breln1unionIL" "breln1unionIR" "breln1unionE"
"breln1all" "subbreln1" "contradiction")
A sends OK
A sends READY

```

2.1. WOZ2 Preparation Example. $R = (R^{-1})^{-1}$

```

A gets (TUTOR ("woz2Ex" "M" "R"))
B sends Hello, Chad. Please try to prove the following:
B sends |- (R==(breln1inv M (breln1inv M R)))
A sends READY-TUTOR

```

```

A gets (LET ("x" "y"))
A sends (DIAGNOSIS (BAD-TYPE-FOR |x|))
B sends Not OK.
B sends The type you gave for x does not seem to be correct.
A sends (DIAGNOSIS (SHOULD-LET (DCLASS APP |in| . M)))
A sends (DIAGNOSIS
                                (SHOULD-ASSUME
                                  (APP |not| APP (APP |subset| FST . R) FST APP
                                    (APP |breln1inv| . M) APP
                                    (APP |breln1inv| . M) . R)))

A sends NOT-OK
A sends TUTOR-STATUS-UNCHANGED
A sends READY-TUTOR
A gets (LET ("x" "y") "M")
A sends (DIAGNOSIS
                                (SHOULD-ASSUME
                                  (APP |not| APP (APP |eq| FST . R) FST APP
                                    (APP |breln1inv| . M) APP
                                    (APP |breln1inv| . M) . R)))

A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME (("kpair" "x" "y") IN "R"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends (DIAGNOSIS NOT-YET-DONE)
A sends NOT-OK
A sends TUTOR-STATUS-UNCHANGED
B sends Not OK
B sends You don't seem to actually be done with anything, Chad.
A sends READY-TUTOR
A gets (CLEARLY
                                (("kpair" "x" "y") IN ("breln1inv" "M" "R")))

A sends (DIAGNOSIS STEP-UNJUSTIFIED)
A sends NOT-OK
A sends TUTOR-STATUS-UNCHANGED
B sends Not OK.
B sends The statement is not so obvious to me, Chad.
A sends READY-TUTOR
A gets (CLEARLY
                                (("kpair" "y" "x") IN ("breln1inv" "M" "R")))

A sends OK
B sends OK
A sends READY-TUTOR

```

A gets (QED)
A sends OK
B sends OK
B sends Good Chad, you're done with this part of the proof, but there is more to do.
A sends READY-TUTOR
A gets (LET ("x" "y") "M")
A sends (DIAGNOSIS
(SHOULD-ASSUME
(APP |not| APP
(APP |subset| FST APP (APP |breln1inv| . M)
APP (APP |breln1inv| . M) . R)
FST . R)))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME (("kpair" "x" "y") IN "R"))
A sends (DIAGNOSIS
(SHOULD-ASSUME
(APP
(APP |in| FST APP (APP |breln1inv| . M) APP
(APP |breln1inv| . M) . R)
FST APP (APP |kpair| FST . |x|) FST
. |y|)))
B sends Not OK.
B sends I could not find any reason for you to make such an assumption, Chad.
A sends (DIAGNOSIS
(SHOULD-ASSUME
(APP
(APP |in| FST APP (APP |breln1inv| . M) APP
(APP |breln1inv| . M) . R)
FST APP (APP |kpair| FST . |x|) FST
. |y|)))
A sends NOT-OK
A sends TUTOR-STATUS-UNCHANGED
A sends READY-TUTOR
A gets (ASSUME
(("kpair" "x" "y") IN
("breln1inv" "M" ("breln1inv" "M" "R"))))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (WILLSHOW
(("kpair" "y" "x") IN ("breln1inv" "M" "R")))
A sends OK
B sends OK
A sends READY-TUTOR

A gets (QED)
A sends STUDENT-SUCCESSFUL
B sends Congratulations, you're done with the proof, Chad!
B sends Successful Term:
B sends (\x0 x1....)
A sends EXIT-TUTOR
A sends READY
A gets (TUTOR ("woz2Ex" "M" "R"))
B sends Hello, Chad. Please try to prove the following:
B sends |- (R==(breln1inv M (breln1inv M R)))
A sends READY-TUTOR
A gets (LET ("x" "y") "M")
A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME

("kpair" "x" "y") IN
("breln1inv" "M" ("breln1inv" "M" "R"))))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (HENCE

("kpair" "y" "x") IN ("breln1inv" "M" "R"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends OK
B sends OK
B sends Good Chad, you're done with this part of the proof, but there is more to do.
A sends READY-TUTOR
A gets (LET ("u" "v") "M")
A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME (("kpair" "u" "v") IN "R"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends (DIAGNOSIS NOT-YET-DONE)
A sends NOT-OK
A sends TUTOR-STATUS-UNCHANGED
B sends Not OK
B sends You don't seem to actually be done with anything, Chad.
A sends READY-TUTOR

A gets (HENCE
("kpair" "y" "x") IN ("breln1inv" "M" "R"))
A sends (DIAGNOSIS STEP-UNJUSTIFIED)
A sends NOT-OK
A sends TUTOR-STATUS-UNCHANGED
B sends Not OK.
B sends I am not sure that statement follows, Chad.
A sends READY-TUTOR
A gets (HENCE
("kpair" "v" "u") IN ("breln1inv" "M" "R"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends STUDENT-SUCCESSFUL
B sends Congratulations, you're done with the proof, Chad!
B sends Successful Term:
B sends (\x0 x1...)
A sends EXIT-TUTOR
A sends READY
A gets (TUTOR ("woz2Ex" "M" "R"))
B sends Hello, Chad. Please try to prove the following:
B sends |- (R==(breln1inv M (breln1inv M R)))
A sends READY-TUTOR
A gets (LET ("x" "y") "M")
A sends (DIAGNOSIS
(SHOULD-ASSUME
(APP |not| APP (APP |eq| FST . R) FST APP
(APP |breln1inv| . M) APP
(APP |breln1inv| . M) . R)))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME
("kpair" "x" "y") IN
("breln1inv" "M" ("breln1inv" "M" "R"))))
A sends (DIAGNOSIS
(SHOULD-ASSUME
(APP (APP |in| FST . R) FST APP
(APP |kpair| FST . |x|) FST . |y|)))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (WILLSHOW
("kpair" "y" "x") IN ("breln1inv" "M" "R"))
A sends OK
B sends OK
A sends READY-TUTOR

```

A gets    (CLEARLY
                                                (("kpair" "y" "x") IN ("breln1inv" "M" "R")))
A sends  OK
B sends  OK
A sends  READY-TUTOR
A gets   (LET ("x" "y") "M")
A sends  (DIAGNOSIS
                                                (SHOULD-ASSUME
                                                (APP |not| APP (APP |subset| FST . R) FST APP
                                                (APP |breln1inv| . M) APP
                                                (APP |breln1inv| . M) . R)))
A sends  OK
B sends  OK
A sends  READY-TUTOR
A gets   (ASSUME (("kpair" "x" "y") IN "R"))
A sends  OK
B sends  OK
A sends  READY-TUTOR
A gets   (WILLSHOW
                                                (("kpair" "y" "x") IN ("breln1inv" "M" "R")))
A sends  OK
B sends  OK
A sends  READY-TUTOR
A gets   (CLEARLY
                                                (("kpair" "y" "x") IN ("breln1inv" "M" "R")))
A sends  OK
B sends  OK
A sends  READY-TUTOR
A gets   (QED)
A sends  STUDENT-SUCCESSFUL
B sends  Congratulations, you're done with the proof, Chad!
B sends  Successful Term:
B sends  (\x0 x1...)
A sends  EXIT-TUTOR
A sends  READY

```

2.2. WOZ2 Warmup Example. $(R \circ S)^{-1} = (S^{-1} \circ R^{-1})$

```

A gets   (TUTOR ("woz2W" "M" "R" "S"))
B sends  Hello, Chad. Please try to prove the following:
B sends  |- ((breln1inv M (breln1comp M R S))== (breln1comp M (breln1inv M S) (breln1inv M
A sends  READY-TUTOR
A gets   (LET ("x" "y") "M")
A sends  OK
B sends  OK
A sends  READY-TUTOR

```

```

A gets    (ASSUME
                                         (("kpair" "x" "y") IN
                                         ("breln1inv" "M"
                                         ("breln1comp" "M" "R" "S"))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
                                         (("kpair" "y" "x") IN
                                         ("breln1comp" "M" "R" "S"))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (EXISTS "z" "M"
                                         (((("kpair" "y" "z") IN "R") &
                                         ("kpair" "z" "x") IN "S"))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
                                         (("kpair" "z" "y") IN ("breln1inv" "M" "R")))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
                                         (("kpair" "x" "z") IN ("breln1inv" "M" "S")))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (QED)
A sends   OK
B sends   OK
B sends   Good Chad, you're done with this part of the proof, but there is more to do.
A sends   READY-TUTOR
A gets    (LET ("x" "y") "M")
A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (ASSUME
                                         (("kpair" "x" "y") IN
                                         ("breln1comp" "M" ("breln1inv" "M" "S")
                                         ("breln1inv" "M" "R"))))

A sends   OK
B sends   OK
A sends   READY-TUTOR

```

```

A gets (EXISTS "z" "M"
      (((("kpair" "x" "z") IN ("breln1inv" "M" "S")) &
        ("kpair" "z" "y") IN
          ("breln1inv" "M" "R")))))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY ((("kpair" "z" "x") IN "S"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY ((("kpair" "y" "z") IN "R"))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY
      ((("kpair" "y" "x") IN
        ("breln1comp" "M" "R" "S"))))
A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends STUDENT-SUCCESSFUL
B sends Congratulations, you're done with the proof, Chad!
B sends Successful Term:
B sends (\x0 x1 x2....)
A sends EXIT-TUTOR
A sends READY
A gets (TUTOR-STUDENT-USABLE "woz2W" "woz2Ex"
      "breln1invprop" "breln1inv" "breln1invI"
      "breln1invE" "breln1compprop" "breln1comp"
      "breln1compI" "breln1compE" "breln1unionprop"
      "breln1union" "breln1unionI" "breln1unionIL"
      "breln1unionIR" "breln1unionE" "breln1all"
      "subbreln1" "contradiction")
A sends OK
A sends READY

```

2.3. WOZ2 Exercise A. $((R \cup S) \circ T) = ((R \circ T) \cup (S \circ T))$

```

A gets (TUTOR ("woz2A" "M" "R" "S" "T"))
B sends Hello, Chad. Please try to prove the following:
B sends |- ((breln1comp M (breln1union M R S) T)==(breln1union M (breln1comp M R T) (bre
A sends READY-TUTOR
A gets (LET ("x" "y") "M")
A sends OK
B sends OK
A sends READY-TUTOR

```

```

A gets    (ASSUME
           (("kpair" "x" "y") IN
            ("breln1comp" "M" ("breln1union" "M" "R" "S")
             "T")))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (EXISTS "z" "M"
           (((("kpair" "x" "z") IN
              ("breln1union" "M" "R" "S"))
             & (("kpair" "z" "y") IN "T"))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
           (((("kpair" "x" "z") IN "R") OR
              ("kpair" "x" "z") IN "S"))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (ASSUME ((("kpair" "x" "z") IN "R"))
           ("kpair" "x" "z") IN "R"))
A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
           (("kpair" "x" "y") IN
            ("breln1comp" "M" "R" "T")))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (QED)
A sends   OK
B sends   OK
B sends   Good Chad, you're done with this part of the proof, but there is more to do.
A sends   READY-TUTOR
A gets    (ASSUME ((("kpair" "x" "z") IN "S"))
           ("kpair" "x" "z") IN "S"))
A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
           (("kpair" "x" "y") IN
            ("breln1comp" "M" "S" "T")))

A sends   OK
B sends   OK
A sends   READY-TUTOR

```

```

A gets (QED)
A sends OK
B sends OK
B sends Good Chad, you're done with this part of the proof, but there is more to do.
A sends READY-TUTOR
A gets (LET ("x" "y") "M")
A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME
                                         ( ("kpair" "x" "y") IN
                                           ("breln1union" "M" ("breln1comp" "M" "R" "T")
                                             ("breln1comp" "M" "S" "T"))))

A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY
                                         ((( "kpair" "x" "y") IN
                                           ("breln1comp" "M" "R" "T")
                                           OR
                                           ( ("kpair" "x" "y") IN
                                             ("breln1comp" "M" "S" "T"))))

A sends OK
B sends OK
A sends READY-TUTOR
A gets (ASSUME
                                         ( ("kpair" "x" "y") IN
                                           ("breln1comp" "M" "R" "T"))

A sends OK
B sends OK
A sends READY-TUTOR
A gets (EXISTS "z" "M"
                                         ((( "kpair" "x" "z") IN "R") &
                                           ( ("kpair" "z" "y") IN "T")))

A sends OK
B sends OK
A sends READY-TUTOR
A gets (CLEARLY
                                         ( ("kpair" "x" "z") IN
                                           ("breln1union" "M" "R" "S")))

A sends OK
B sends OK
A sends READY-TUTOR
A gets (QED)
A sends OK
B sends OK
B sends Good Chad, you're done with this part of the proof, but there is more to do.
A sends READY-TUTOR

```

```

A gets    (ASSUME
                                                    (("kpair" "x" "y") IN
                                                    ("breln1comp" "M" "S" "T")))
A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (EXISTS "z" "M"
                                                    (((("kpair" "x" "z") IN "S") &
                                                    ("kpair" "z" "y") IN "T"))))
A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
                                                    (("kpair" "x" "z") IN
                                                    ("breln1union" "M" "R" "S")))
A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (QED)
A sends   STUDENT-SUCCESSFUL
B sends   Congratulations, you're done with the proof, Chad!
B sends   Successful Term:
B sends   (\x0 x1 x2 x3....)
A sends   EXIT-TUTOR
A sends   READY
A gets    (TUTOR-STUDENT-USABLE "breln1unionCommutes"
                                                    "woz2A" "woz2W" "woz2Ex" "breln1invprop"
                                                    "breln1inv" "breln1invI" "breln1invE"
                                                    "breln1compprop" "breln1comp" "breln1compI"
                                                    "breln1compE" "breln1unionprop" "breln1union"
                                                    "breln1unionI" "breln1unionIL" "breln1unionIR"
                                                    "breln1unionE" "breln1all" "subbreln1"
                                                    "contradiction")
A sends   OK
A sends   READY

```

2.4. WOZ2 Exercise B. $(R \cup S) \circ T = (T^{-1} \circ S^{-1})^{-1} \cup (T^{-1} \circ R^{-1})^{-1}$

```

A gets    (TUTOR ("woz2B" "M" "R" "S" "T"))
B sends   Hello, Chad. Please try to prove the following:
B sends   |- ((breln1comp M (breln1union M R S) T)==(breln1union M (breln1inv M (breln1com
A sends   READY-TUTOR

```

```

A gets    (CLEARLY
           ("breln1comp" "M" ("breln1union" "M" "R" "S")
            "T")
           ==
           ("breln1union" "M" ("breln1comp" "M" "R" "T")
            ("breln1comp" "M" "S" "T"))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
           ("breln1comp" "M" "R" "T") ==
           ("breln1inv" "M"
            ("breln1inv" "M"
             ("breln1comp" "M" "R" "T"))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
           ("breln1comp" "M" "S" "T") ==
           ("breln1inv" "M"
            ("breln1inv" "M"
             ("breln1comp" "M" "S" "T"))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
           ("breln1comp" "M" ("breln1union" "M" "R" "S")
            "T")
           ==
           ("breln1union" "M"
            ("breln1inv" "M"
             ("breln1inv" "M" ("breln1comp" "M" "R" "T"))
            ("breln1inv" "M"
             ("breln1inv" "M"
              ("breln1comp" "M" "S" "T"))))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
           ("breln1inv" "M" ("breln1comp" "M" "R" "T")) ==
           ("breln1comp" "M" ("breln1inv" "M" "T")
            ("breln1inv" "M" "R"))))

A sends   OK
B sends   OK
A sends   READY-TUTOR

```

```

A gets    (CLEARLY
          ((("breln1inv" "M" ("breln1comp" "M" "S" "T")) ==
            ("breln1comp" "M" ("breln1inv" "M" "T")
              ("breln1inv" "M" "S")))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
          ((("breln1comp" "M" ("breln1union" "M" "R" "S")
            "T")
            ==
            ("breln1union" "M"
              ("breln1inv" "M"
                ("breln1comp" "M" ("breln1inv" "M" "T")
                  ("breln1inv" "M" "R"))))
              ("breln1inv" "M"
                ("breln1comp" "M" ("breln1inv" "M" "T")
                  ("breln1inv" "M" "S"))))))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (CLEARLY
          ((("breln1union" "M"
            ("breln1inv" "M"
              ("breln1comp" "M" ("breln1inv" "M" "T")
                ("breln1inv" "M" "R"))))
            ("breln1inv" "M"
              ("breln1comp" "M" ("breln1inv" "M" "T")
                ("breln1inv" "M" "S")))))
            ==
            ("breln1union" "M"
              ("breln1inv" "M"
                ("breln1comp" "M" ("breln1inv" "M" "T")
                  ("breln1inv" "M" "S"))))
              ("breln1inv" "M"
                ("breln1comp" "M" ("breln1inv" "M" "T")
                  ("breln1inv" "M" "R"))))))))

A sends   OK
B sends   OK
A sends   READY-TUTOR
A gets    (QED)
A sends   (DIAGNOSIS NOT-YET-DONE)
A sends   STUDENT-SUCCESSFUL
A sends   EXIT-TUTOR
B sends   Congratulations, you're done with the proof, Chad!
B sends   Successful Term:
B sends   (\x0 x1 x2 x3....)
A sends   READY

```

Bibliography

- [1] Peter B. Andrews, Matthew Bishop, and Chad E. Brown. System description: TPS: A theorem proving system for type theory. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 164–169, Pittsburgh, PA, USA, 2000. Springer-Verlag.
- [2] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning*, 16:321–353, 1996.
- [3] R.G. Bartle and D.R. Sherbert. *Introduction to Real Analysis*. John Wiley & Sons, New York, 1982.
- [4] C. Benzmüller, V. Sorge, M. Jamnik, and M. Kerber. Can a higher-order and a first-order theorem prover cooperate? In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 3452 of *LNAI*, pages 415–431. Springer, 2005.
- [5] Christoph Benzmüller, Armin Fiedler, Malte Gabsdil, Helmut Horacek, Ivana Kruijff-Korbayová, Manfred Pinkal, Jörg Siekmann, Dimitra Tsovaltzi, Bao Quoc Vo, and Magdalena Wolska. A wizard of oz experiment for tutorial dialogues in mathematics. In *Proceedings of AI in Education (AIED 2003) Workshop on Advanced Technologies for Mathematics Education*, Sydney, Australia, 2003.
- [6] Christoph Benzmüller, Helmut Horacek, Henri Lesourd, Ivana Kruijff-Korbayova, Marvin Schiller, and Magdalena Wolska. A corpus of tutorial dialogs on theorem proving; the influence of the presentation of the study-material. In *Proceedings of International Conference on Language Resources and Evaluation (LREC 2006)*, Genova, Italy, 2006. ELDA. To appear.
- [7] Chad E. Brown. Combining Type Theory and Untyped Set Theory. In *Automated Reasoning, Third International Joint Conference (IJCAR 2006)*, Seattle, Washington, 2006. To appear.
- [8] Chad E. Brown. Verifying and Invalidating Textbook Proofs using Scunak. In *Proceedings of the 5th International Conference on Mathematical Knowledge Management (MKM 2006)*, Wokingham, England, 2006. To appear.
- [9] The Coq proof assistant. <http://coq.inria.fr/>.
- [10] Thierry Coquand and Gérard Huet. Constructions: A Higher Order Proof System for Mechanizing Mathematics. In Bruno Buchberger, editor, *EUROCAL '85: European Conference on Computer Algebra*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184, Linz, Austria, April 1985. Springer-Verlag.
- [11] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
- [12] N .G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 579–606. Academic Press, 1980.
- [13] Fairouz Kamareddine. Special issue mechanizing and automating mathematics: In honour of N.G. de Bruijn - preface. *J. Autom. Reasoning*, 29(3-4):183–188, 2002.
- [14] Michael Kohlhase. Omdoc: Towards an internet standard for the administration, distribution, and teaching of mathematical knowledge. In John A. Campbell and Eugenio Roanes-Lozano, editors, *Artificial Intelligence and Symbolic Computation: International Conference AISC 2000*, volume 1930 of *Lecture Notes in Artificial Intelligence*, pages 32–52. Springer-Verlag, 2001.

- [15] Frank Pfenning and Carsten Schürmann. System Description: Twelf—A Meta-Logical Framework for Deductive Systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, Trento, Italy, 1999. Springer-Verlag.
- [16] Marvin Schiller. Mechanizing Proof Step Evaluation for Mathematics Tutoring - the Case of Granularity. Master's thesis, Universität des Saarlandes, 2005.
- [17] Geoff Sutcliffe, Christian Suttner, and Theodor Yemenis. The TPTP problem library. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 252–266, Nancy, France, 1994. Springer-Verlag.
- [18] Freek Wiedijk. A new implementation of Automath. *J. Autom. Reasoning*, 29(3-4):365–387, 2002.