The IMPS User's Manual

First Edition, Version 2

William M. Farmer
Joshua D. Guttman
F. Javier Thayer

The MITRE Corporation
Bedford, MA 01730 USA
617-271-2907

{farmer,guttman,jt}@mitre.org

11 April 1995

Contents

List of Tables

List of Figures

Part I

Introductory Material

Chapter 1

Introduction

IMPS is an Interactive Mathematical Proof System developed at The MITRE Corporation by W. Farmer, J. Guttman, and J. Thayer. For a technical overview of the system, see [11].

1.1 Overview of the Manual

The manual consists of three parts:

(1) *Introductory Material.* This part should definitely be read first. Chapter 3, "A Brief Tutorial," provides a quick way to start using IMPS.

(2) *User's Guide.* This part describes how to use IMPS, particularly, how to build theories.

(3) *Reference Manual.*

1.2 Goals

IMPS aims at computational support for traditional techniques of mathematics. It is based on three observations about rigorous mathematics:

- Mathematics emphasizes the axiomatic method. Characteristics of mathematical structures are summarized in axioms. Theorems are derived for all structures satisfying the axioms. Frequently, a clever change of perspective shows that a structure is an instance of another theory, thus also bringing its theorems to bear.

- Many branches of mathematics emphasize functions, including partial functions. Moreover, the classes of objects studied may be nested, as are the integers and the reals; or overlapping, as are the bounded functions and the continuous functions.

- Proof proceeds by a blend of computation and formal inference.

1.2.1 Support for the Axiomatic Method

IMPS supports the "little theories" version of the axiomatic method, as opposed to the "big theory" version. In the big theory approach, all reasoning is carried out within one theory—usually some highly expressive theory, such as Zermelo-Fraenkel set theory. In the little theories approach, reasoning is distributed over a network of theories. Results are typically proved in compact, abstract theories, and then transported as needed to more concrete theories, or indeed to other equally abstract theories. Theory interpretations provide the mechanism for transporting theorems. The little theories style of the axiomatic method is employed extensively in mathematical practice; in [10], we discuss its benefits for mechanical theorem provers, and how the approach is used in IMPS.

1.2.2 Logic

Standard mathematical reasoning in many areas focuses on functions and their properties, together with operations on functions. For this reason, IMPS is based on a version of simple type theory.[1] However, we have adopted a version, called LUTINS,[2] containing partial functions, because they are ubiquitous in both mathematics and computer science. Although terms, such as $2/0$, may be nondenoting, the logic is bivalent and formulas always have a truth value. In particular, an atomic formula is false if any of its constituents is nondenoting. This convention follows an approach common in traditional rigorous mathematics, and it entails only small changes in the axioms and rules of classical simple type theory [4].

[1] This version is *many sorted*, in that there may be several types of basic individuals. Moreover, it is *multivariate*, in that a function may take more than one argument. Currying is not required. However, taking (possibly n-ary) functions is the only type-forming operation.

[2] Pronounced as in French. See [4, 5, 7] for studies of logical issues associated with LUTINS; see [16] for a detailed description of its syntax and semantics.

Moreover, LUTINS allows subtypes to be included within types. Thus, for instance, the natural numbers form a subtype of the reals, and the continuous (real) functions a subtype of the functions from reals to reals. The subtyping mechanism facilitates machine deduction, because the subtype of an expression gives some immediate information about the expression's value, if it is defined at all. Moreover, many theorems have restrictions that can be stated in terms of the subtype of a value, and the theorem prover can be programmed to handle these assertions using special algorithms.

This subtyping mechanism interacts well with the type theory only because functions may be partial. If σ_0 is a subtype of τ_0, while σ_1 is a subtype of τ_1, then $\sigma_0 \to \sigma_1$ is a subtype of $\tau_0 \to \tau_1$. In particular, it contains just those partial functions that are never defined for arguments outside σ_0, and which never yield values outside σ_1.

1.2.3 Computation and Proof

One problem in understanding and controlling the behavior of theorem provers is that a derivation in predicate logic contains too much information.

The mathematician devotes considerable effort to proving lemmas that justify computational procedures. Although these are frequently equations or conditional equations, they are sometimes more complicated quantified expressions, and sometimes they involve other relations, such as ordering relations. Once the lemmas are available, they are used repeatedly to transform expressions of interest. Algorithms justified by the lemmas may also be used; the algorithm for differentiating polynomials, for example. The mathematician has no interest in those parts of a formal derivation that "implement" these processes within predicate logic.

On the other hand, to understand the structure of a proof (or especially, a partial proof attempt), the mathematician wants to see the premises and conclusions of the informative formal inferences.

Thus, the right sort of proof is broader than the logician's notion of a formal derivation in, say, a Gentzen-style formal system. In addition to purely formal inferences, IMPS also allows inferences based on sound computations. They are treated as atomic inferences, even though a pure formalization might require hundreds of Gentzen-style formal inference steps. We believe that this more inclusive conception makes IMPS proofs more informative to its users.

1.3 Components of the System

The IMPS system consists of four components.

1.3.1 Core

All the basic logical and deductive machinery of IMPS on which the soundness of the system depends is included in the *core* of IMPS. The core is the specification and inference engine of IMPS. The other components of the system provide the means for harnessing the power of the engine.

The organizing unit of the core is the IMPS *theory*. Mathematically, a theory consists of a LUTINS language plus a set of axioms. A theory is implemented, however, as a data structure to which a variety of information is associated. Some of this information procedurally encodes logical consequences of the theory that are especially relevant to low-level reasoning within the theory. For example, the great majority of questions about the definedness of expressions are answered automatically by IMPS using tabulated information about the domain and range of the functions in a theory. Theories may be enriched by the definition of new sorts and constants and by the installation of theorems.

Proofs in a theory are constructed interactively using a natural style of inference based on sequent calculus. IMPS builds a data structure which records all the actions and results of a proof attempt. This object, called a *deduction graph*, allows the user to survey the proof, and to choose the order in which he works on different subgoals. Alternative approaches may be tried on the same subgoal. Deduction graphs also are suitable for analysis by software.

The user is only allowed to modify a deduction through the application of *primitive inferences*, which are akin to rules of inference. Most primitive inferences formalize basic laws of predicate logic and higher-order functions. Others implement computational steps in proofs. For example, one class of primitive inferences performs expression simplification, which uses the logical structure of the expression [20], together with algebraic simplification, deciding linear inequalities, and applying rewrite rules.

Another special class of primitive inferences "compute with theorems" by applying extremely simple procedures called macetes.[3] An *elementary macete*, which is built by IMPS whenever a theorem is installed in a theory,

[3] *Macete*, in Portuguese, means a chisel, or in informal usage, a clever trick.

applies the theorem in a manner determined by its syntax (e.g., as a conditional rewrite rule). Compound macetes are constructed from elementary and other kinds of atomic macetes (including macetes that beta-reduce and simplify expressions). They apply a collection of theorems in an organized way.

In addition to the machinery for building theories and reasoning within them, the core contains machinery for relating one theory to another via theory interpretations. A theory interpretation can be used to "transport" a theorem from the theory it is proved in to any number of other theories. Theory interpretations are also used in IMPS to show relative consistency of theorems, to formalize symmetry and duality arguments, and to prove universal facts about polymorphic operators. The great majority of the theory interpretations needed by the IMPS user are built by software without user assistance. For example, when a theorem is applied outside of its home theory via a *transportable macete*, IMPS automatically builds the required theory interpretation if needed.

1.3.2 Supporting Machinery

We have built an extension of the core machinery with the following goals in mind:

- To facilitate building and printing of expressions by providing various parsing and printing procedures.
- To make the inference mechanism more flexible and more autonomous. In particular, the set of commands available to users includes an extensible set of inference procedures called *strategies*. Strategies affect the deduction graph only by using the primitive inference procedures of the core machinery, but are otherwise unrestricted.
- To facilitate construction of theories and interpretations between them.

1.3.3 User Interface

The user interface, which is a completely detachable component of IMPS, is mainly designed to provide users with facilities to easily direct and monitor proofs. Specifically, it controls three critical elements of an interactive system:

- The display of the state of the proof. This includes graphical displays of the deduction graph as a tree, TEX typesetting of the proof history, and proof scripts composed of the commands used to build the proof. The graphical display of the deduction graph permits the user to visually determine the set of unproven subgoals and to select a suitable continuation point for the proof. The TEX typesetting facilities allow the user to examine the proof in a mathematically more appealing notation than is possible by raw textual presentation alone. And, with proof scripts, the user can replay the proof, in whole or in part.

- The presentation of options for new proof steps. For any particular subgoal, the interface presents the user with a well-pruned list of commands and macetes to apply. This list is obtained by using syntactic and semantic information which is made available to the interface by the IMPS supporting machinery. For example, in situations where over 400 theorems are available to the user, there are rarely more than ten macetes presented to the user as options.

- The processing of user commands. The commands are then submitted to the inference software, while any additional arguments required to execute the command are requested from the user.

1.3.4 Theory Library

IMPS is equipped with a library of basic theories, which can be augmented as desired by the user. The theory of the reals, the central theory in the library specifies the real numbers as a complete ordered field. (The completeness principle is formalized as a second-order axiom, and the rationals and integers are specified as the usual substructures of the reals.) The library also contains various "generic" theories that contain no nonlogical axioms (except possibly the axioms of the theory of the reals). These theories are used for reasoning about objects such as sets, unary functions, and sequences.

1.4 Acknowledgments

IMPS was designed and implemented at The MITRE Corporation under the MITRE-Sponsored Research program. Ronald D. Haggarty, Vice President of Research and Technology, deserves special thanks for his strong, unwavering support of the IMPS project.

Several of the key ideas behind IMPS were originally developed by Dr. Leonard Monk on the Heuristics Research Project, also funded by MITRE-Sponsored Research, during 1984–1987. Some of these ideas are described in [20].

The IMPS core and support machinery are written in the T programming language [18, 23], developed at Yale by N. Adams, R. Kelsey, D. Kranz, J. Philbin, and J. Rees. The IMPS user interface is written in the GNU Emacs programming language [25], developed by R. Stallman.

We are grateful to the Harvard Mathematics Department for providing an ftp site for IMPS.

Chapter 2

Distribution

IMPS is available without fee via anonymous ftp (under the terms of a public license) at math.harvard.edu. IMPS runs under the X Window System or OpenWindows on Sun 4 SPARCstations that have at least 24 MB physical memory (preferably more) and at least a 60 MB swap partition.

The first section of this chapter describes how to get and install IMPS, report bugs, and join the IMPS mailing list. The second section contains a copy of IMPS copyright notice and public license.

2.1 How to Get and Install IMPS

2.1.1 How to Get IMPS

Ftp to **math.harvard.edu**, login as `anonymous`, and give your e-mail address as the password. Then type

```
cd imps
```

The directory you will be in contains a directory named **doc** and four files:

(1) the text in this section (**README**),

(2) the IMPS system (**imps.tar.gz**),

(3) the IMPS manual (**imps-manual.dvi.gz**), and

(4) the IMPS public license (**public-license**).

The IMPS manual and public license are also included in the IMPS system.

To get a copy of the IMPS system, type

```
binary
get README
get imps.tar.gz
```

The transfer will take a few minutes.

2.1.2 How to Install IMPS

(1) Create a directory called **imps** (or whatever you prefer) somewhere in your file system where you have 35-40 MB free. (About 20 MB of this space is for IMPS; the rest is needed only during installation.) Let us refer to this directory as **/.../imps**. Execute (the shell command)

```
cd /.../imps
```

(2) Copy the file **imps.tar.gz** to the **/.../imps** directory. Then execute the following commands

```
gunzip imps.tar.gz
tar -xvf imps.tar
```

Each of these operations will take several minutes. After they are done, you may delete the file **imps.tar** or recompress it and put it somewhere else.

(3) Choose the version of Emacs you want to use with IMPS. The preferred version is Free Software Foundation GNU Emacs 19, but we also support GNU Emacs 18 and Lucid GNU Emacs 19. If you will be using Free Software Foundation GNU Emacs 19 or Lucid GNU Emacs 19, put the line

```
(setq tea-use-comint t)
```

in your **.emacs** file. Let us refer to the location of the Emacs executable as **/.../emacs**.

Warning: Depending on the version of Emacs you choose, you may have to recompile the **.el** files in the directory

/.../imps/el

(4) Execute

```
cd /.../imps/src
```

and edit the file **start_imps.sh** to change the two lines

```
IMPS=<imps-path>/tea
IMPS_EMACS=<emacs-path>
```

to

```
IMPS=/.../imps/tea
IMPS_EMACS=/.../emacs
```

Finally, execute

```
make
```

(You may ignore the error messages that are printed.)

2.1.3 Instructions for IMPS Users

In your **.cshrc** (or wherever you define your shell search path) add

```
/.../imps/bin
```

to your path. You may find it convenient to set the shell variable **IMPS** in your **.cshrc** using the command

```
setenv IMPS /.../imps/tea
```

If you want a form of Emacs different than **/.../emacs**, the official IMPS Emacs, set the shell variable **EMACS_COMMAND** in your **.cshrc** to the form of Emacs you want; for example,

```
setenv EMACS\_COMMAND 'emacs18 -w 94x56+250+20'
```

If you will be using Free Software Foundation GNU Emacs 19 or Lucid GNU Emacs 19, put the line

```
(setq tea-use-comint t)
```

in your **.emacs** file.

If IMPS is being run with Free Software Foundation GNU Emacs 19, the frame parameters (position, background and foreground colors, etc.) of the various IMPS windows are specified in the file

/.../imps/el/frame-specs.el

(Background colors and the default font have been commented out.)

You will probably want to modify these specifications to suit your own taste. To do this, first copy the file **frame-specs.el** to

/.../home/imps/imps-emacs.el

(where **/.../home** is your home directory and **imps** is a subdirectory of your home directory that is created when you first run IMPS). Then change the frame specifications in this file as you like.

To subscribe to the IMPS mailing list, send your name and e-mail address to

imps-request@linus.mitre.org

We strongly urge that all users of IMPS *subscribe to the* IMPS *mailing list.*

2.1.4 How to Start IMPS

To run IMPS, start X or OpenWindows, and then execute

```
start_imps &
```

2.1.5 The IMPS User's Manual

The IMPS user's manual is written in LaTeX and is available on-line. The **.dvi** file of the manual is located in the directory

/.../imps/doc/manual

(Also, both **.dvi** and **.ps** files of the manual are located in the **imps** directory at **math.harvard.edu**.) The manual is approximately 300 pages long.

2.1.6 IMPS Papers

In the **imps/doc** directory at **math.harvard.edu**, there are several papers on IMPS in compressed LaTeX dvi and ps formats.

2.1.7 Bug Reports and Questions

Please mail information about bugs or problems with using IMPS to

imps-bugs@linus.mitre.org

Other questions can be mailed to

imps-request@linus.mitre.org

2.2 IMPS Copyright Notice

Authors: W. M. Farmer, J. D. Guttman, F. J. Thayer

Chapter 3

A Brief Tutorial

In this tutorial, we will assume that you have the X Window System running on your workstation. If your workstation can run X but it is not installed, enlist the aid of a system guru to get him or her to set things up for you. Although it is possible to run IMPS from a dumb terminal (for example, to work from home, where mere mortals only have terminal emulators), you will miss the functionality that X provides, such as fancy TEX previewing of formulas and proofs, and interaction with IMPS using the mouse and the menu facility. We will also assume that you have the shell variable **IMPS** correctly set.

To start IMPS enter this command at the shell prompt:

```
$IMPS/../bin/start_imps
```

Executing this command will do the following:

- Start an Emacs session.
- Start IMPS (which is actually a "Tea" process) subordinate to Emacs.
- Make two new directories in your home directory:

  $HOME/imps and **$HOME/imps/theories**

  needed for the IMPS exercises.
- Add some files to your **/tmp** directory.
- Start one or more iconified TEX preview windows.

To quit IMPS type `C-x C-c`. This will kill Tea and the Emacs process.

If you are trying to learn to use IMPS, it would be a good idea to work through the exercises described in Chapters 5 and 6, after reading this tutorial.

3.1 Interacting with IMPS

Most of your interaction with IMPS can be initiated by selecting of one of a number of menu options. The menu format depends on the version of Emacs you are using.

- *Free Software Foundation version 19*[1] menus are invoked by clicking on the menubar. The resulting menu has a number of *options*.

- *Version 18* menus are invoked by clicking the right mouse button on the mode line. Moreover, the menu consists of one or more *panes* which are stacked like cards; each individual pane has a *label* plus a number of *options*.

- *Lucid version 19* menus are invoked by clicking on the menubar. The resulting menu has a number of *options*. Some of the options open up to new submenus.

These three menu systems are quite similar to each other; it is easy to translate instructions for one menu system into instructions for the other.

You select an option within a menu or pane by pointing with the mouse (notice the selected option is enclosed in a box) and clicking the right mouse button. If you do not want to select anything after the menu appears, click right on the option **CANCEL MENU REQUEST** (or just click right when the mouse points to somewhere off the menu.) This will cause the menu to disappear.

Notice that some of the options are followed by a sequence of keystrokes enclosed in parentheses. This means that these options can also be invoked directly using these keystrokes. As you become more familiar with IMPS, you will probably want to use these more direct invocations. Note that you can use the menus to change buffers or modify the number of windows on the screen.

[1] This is the preferred version of Emacs.

You should also be aware that the panes which appear on the menu, and the options within each pane, usually depend on the window you clicked on to bring up the menu.

3.2 On-line Documentation

The following documentation is available on-line:

- *The User's Manual*. If you have a TEX previewer, the IMPS manual [2] is available on-line. You can locate specific items in the manual by clicking right on the option **IMPS manual entry** in the pane **Help**.
- *Def-forms*. A *def-form* is essentially a specification of an IMPS entity. In the course of your interaction with IMPS you may want to examine the definition of some object. You can locate the def-form which specifies it by clicking right on the option **Find def-form (C- .)** in the pane **Help**.

For more details on on-line documentation, see Chapter 4.

3.3 Languages and Theories

For now, you can think of a *theory* as a mathematical object consisting of a *language* and a set of closed formulas in that language called *axioms*. For example, one of the last things that was printed out in the startup procedure was the line

```
Current theory: #{imps-theory 3: H-O-REAL-ARITHMETIC}
```

h-o-real-arithmetic is a basic IMPS theory which contains an axiomatic theory of the real numbers, characterized as a complete ordered field with the integers and rationals as imbedded structures. This is a fairly extensive theory, so we only describe it informally here. The atomic sorts of the language are **N**, **Z**, **Q**, **R** denoting the natural numbers, integers, rationals, and reals. The language constants of this theory are of three kinds:

- The primitive function and relation constants $+$, $*$, $/$, $\hat{}$, sub , $-$, $<$, $\leq$ that denote the arithmetic operations of addition, multiplication, division, exponentiation, subtraction, negation, and the binary predicates less than, and less than or equal to, respectively.

[2] The IMPS manual is the document you are looking at.

Operator	Domain
`+`	$\mathbf{R} \times \mathbf{R}$
`*`	$\mathbf{R} \times \mathbf{R}$
`-`	$\mathbf{R}$
`sub`	$\mathbf{R} \times \mathbf{R}$
`^`	$\mathbf{E}$
`/`	$\mathbf{R} \times (\mathbf{R} \setminus \{0\})$
`sum`	$\mathbf{S}$
`prod`	$\mathbf{S}$
`!`	$\mathbf{N}$
`<`	$\mathbf{R} \times \mathbf{R}$
`<=`	$\mathbf{R} \times \mathbf{R}$
`>`	$\mathbf{R} \times \mathbf{R}$
`>=`	$\mathbf{R} \times \mathbf{R}$

where:

- $\mathbf{E} = ((\mathbf{R} \setminus \{0\}) \times \mathbf{Z}) \cup (\{0\} \times \{z \in \mathbf{Z} : 0 < z\})$
- $\mathbf{S} = \{(m, n, f) \in \mathbf{Z} \times \mathbf{Z} \times (\mathbf{Z} \to \mathbf{R}) : \forall k \in \mathbf{Z}, m \leq k \leq n \supset f(k) \downarrow\}$

Table 3.1: Domains of Arithmetic Operators.

- An infinite set of individual constants, one for each rational number.
- Various defined constants such as the operator $\sum$.

The domains of the primitive constants and some of the defined constants are given in Table 3.1.

The axioms of this theory are the usual field and order axioms as well as the completeness axiom which states that any predicate which is nonvacuous and bounded above has a least upper bound. This theory also contains the full second-order induction principle for the integers as an axiom.

At any given time there is a theory that IMPS regards as the *current theory.* For example, as we mentioned above, the current theory is **h-o-real-arithmetic** when IMPS finishes loading. The current theory affects the behavior of IMPS in two very significant ways:

- Any expression that is read in by IMPS belongs to the language of the current theory (unless another language is explicitly specified).
- In any proof that you begin, you may avail yourself of all the axioms of the current theory and all the theorems previously proved and entered in the current theory.

You can change the current theory by selecting the **Set current theory** option of the pane **General.** When you are finished playing around with changing theories, set the current theory back to **h-o-real-arithmetic**, which is needed for the tutorial.

3.4 Syntax

Mathematicians and logicians usually think of a formula as a syntactic object represented by some text such as $x+y = y+x$. For IMPS, on the other hand, a *formula* is a complex data structure, essentially a record, with numerous fields. To build a formula in a convenient way, you need to have a *parser*. A parser takes a formula represented as a string and produces a formula represented as a data structure that IMPS can deal with. You will probably want to use a parser which supports infix and prefix operators, and settable precedence relations between different operators. You can also use a Lisp-like syntax if you prefer. The syntax we will use by way of illustration is quite straightforward and you should have no difficulty in building expressions after seeing a few examples.

We now describe the *string syntax* for IMPS. First, some preliminary considerations. A language in IMPS has two kinds of objects: *sorts* and *expressions.* Sorts are intended to denote classes of elements; they are used to help specify the value of an expression and to restrict quantification. They are especially useful for reasoning with respect to overlapping domains. For example, suppose `zz` and `rr` are sorts denoting the integers and the real numbers, respectively. Then the Archimedean principle for the real numbers can be expressed quite naturally as

```
forall(a:rr,forsome(n:zz,a<n)).
```

Sorts are of two kinds: *atomic sorts* and *compound sorts.* An atomic sort is just a name. A compound sort is a list `[s0,...,sn]` where each `si` is itself a compound sort or an atomic sort. Compound sorts are intended to denote the set of all n-ary partial functions from the "domain" sorts `s0`, ..., `s(n-1)`

Sort	Type of Sort	Expressions of given sort
nn	ind	iota(k:nn,k^2=9)
zz	ind	[-1],0,1
qq	ind	[3/2]
rr	ind	iota(x:rr,x^2=2)
ind	ind	with(x:ind,x)
prop	prop	truth, falsehood

Table 3.2: Sorts for **h-o-real-arithmetic**

into the "range" sort `sn` (an exception to this rule about intended meanings of sorts is discussed in the Chapter 7). A language distinguishes certain sorts as *types* and assigns a type to every sort. Distinct types are intended to denote sets of elements which need not be related. `prop` is a "base" type which belongs to every language and which is intended to denote the set of truth values. The sorts for **h-o-real-arithmetic** are given in Table 3.2.

The expressions of a language are built from language constants and variables as specified by Table 3.3 below. A constant is a character string specified as such by the **def-language** form. A variable is also character string. Variables however, must begin with a text character (that is, an alphabetic character or one of the symbols `_ % $ &`) and contain only text characters or digits. Moreover, the sorting of a variable within an expression must be specified either by an enclosing binder or by an enclosing "with" declaration.

In the theory **h-o-real-arithmetic** there is additional syntax for arithmetic operators, described in Table 3.4. For example, `-` is used for both sign negation and the binary subtraction operator. The binary operators are all infix with different binding powers to reduce parentheses on input and output: In order of increasing binding power, the arithmetic operators are `+ * / ^ -`.

To check the syntax of an expression (enclosed in double quotes), select the entry **Check expression syntax** in the pane **IMPS help**. IMPS will request an expression, which you can supply by clicking the right mouse button on the formula. You can also type the formula directly in the minibuffer. As an exercise, build the formula which asserts that for every nonnegative integer n the sum of squares of the first n integers is $n(n+1)(2n+1)/6$.

Expression Category	Syntax	Sort
Truth	`truth`	`prop`
Falsehood	`falsehood`	`prop`
Negation	`not(p)`	`prop`
Conjunction	`p1 and ... and pn`	`prop`
Disjunction	`p1 or ... or pn`	`prop`
Implication	`p1 implies p2`	`prop`
Biconditional	`p1 iff p2`	`prop`
If-then-else formula	`if_form(p1,p2,p3)`	`prop`
Universal	`forall(v1:s1,...,vn:sn,p)`	`prop`
Existential	`forsome(v1:s1,...,vn:sn,p)`	`prop`
Equality	`t1=t2`	`prop`
Application	`f(t1,...,tn)`	`r`
Abstraction	`lambda(v1:s1,...,vn:sn,t)`	`[s1,...,sn,s]`
Definite description	`iota(v:s,p)`	`s`
If-then-else term	`if(p,t1,t2)`	`s1` $\sqcup$ `s2`
Definedness	`#(t)`	`prop`
Sort-definedness	`#(t,s)`	`prop`
Undefined expression	`?s`	`s`
With	`with(v1:s1,...,vn:sn,t)`	`s`

Notes:

(1) `p`, `p1` ... `pn` are syntactic variables which denote formulas, that is, expressions of sort `prop`. `t`, `t1` ... `tn` denote arbitrary expressions of sort `s`, `s1` ... `sn`.

(2) `s1` $\sqcup$ `s2` is the smallest sort which syntactically includes `s1` and `s2`.

(3) For *equality* and *if-then-else term*, `t1`, `t2` must be of the same type.

(4) For *application*, `f` is an arbitrary expression of sort `[r1,...,rn,r]`, and the sorts `ri` and `si` must have the same enclosing type.

(5) *With* is not a constructor; it is merely a device to specify the sorts of free variables.

Table 3.3: String Syntax for Constructors

Term Category	Syntax	Sort
Sum	`t1+ ... +tn`	`rr`
Product	`t1* ... *tn`	`rr`
Sign negation	`-t`	`rr`
Subtraction	`t1-t2`	`rr`
Exponentiation	`t1^t2`	`rr`
Division	`t1/t2`	`rr`
Sequential sum	`sum(t1,t2,t3)`	`rr`
Sequential product	`prod(t1,t2,t3)`	`rr`
Factorial	`t!`	`rr`
Less than	`t1<t2`	`prop`
Less than or equals	`t1<=t2`	`prop`
Greater than	`t1>t2`	`prop`
Greater than or equals	`t1>=t2`	`prop`

Notes:

(1) The terms `t1` ...`tn` must be of type `ind`. Do not confuse this syntactic requirement for well-formedness with semantic conditions for well-definedness.

(2) For sequential sum and product, the term `t3` must be of type `[ind,ind]`.

(3) All operators associate on the left except exponentiation which associates on the right.

Table 3.4: Syntax for **h-o-real-arithmetic**

3.5 Proofs

A formula in a theory is *valid* if it is true in every "legitimate" model of the theory; if the formula has free variables, we have to add: "with every legitimate assignment of values to its free variables." To prove a formula means to offer conclusive mathematical evidence of its validity. The rules for admissible evidence are given by something called a *proof system.* In this section we will describe the IMPS proof system.

To begin an interactive proof in IMPS, select the **start dg** option in the **Deduction Graphs** pane of the menu. For version 19, click on the menubar on **DG** and then select **Start dg**. IMPS will then prompt you in the minibuffer with the text

```
Formula or reference number:
```

This is an IMPS request for some formula to prove. You can supply this formula by typing its reference number in the minibuffer. Alternatively, you can click the *right* mouse button on the formula (provided that it is enclosed between double quotes), and then press `<RET>`. You are then ready to begin proving the formula.

After telling Emacs to start the deduction, two new buffers will be created. One buffer, labelled `*Deduction-graph-1*` displays a graphical representation of a data structure called a *deduction graph.* The other Emacs buffer labeled `*Sequent-nodes-1*` displays a textual representation of a data structure called a *sequent node.* Notice that the mode-line for the sequent node buffer also contains the text

```
h-o-real-arithmetic: Sqn 1 of 1.
```

This gives you the following information:

(1) **h-o-real-arithmetic** is the theory in which the proof is being carried out.

(2) You are currently looking at the first sequent node of the deduction graph.

(3) There is only one sequent node in the deduction graph.

To understand the significance of the new buffer, note that a deduction graph provides a snapshot of the current status of the proof. A deduction graph is a directed graph with nodes of two kinds, called *sequent nodes*

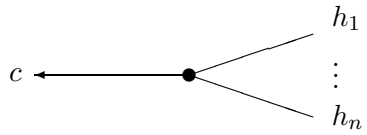

Figure 3.1: An Inference Node.

and *inference nodes.* An interactive proof in IMPS is carried out by adding inference nodes and sequent nodes to the deduction graph.

A sequent node consists of the following:

- A theory.
- A single formula B called the *sequent node assertion.*
- A set of formulas $A_1, \cdots, A_n$ called the *sequent node assumptions.*

Logically, a sequent node just represents the formula

$$A_1 \wedge \cdots \wedge A_n \supset B.$$

In the `*Sequent-nodes-1*` buffer, a sequent node is represented as a list

$$A_1 \cdots A_n \texttt{=>} B.$$

Inference nodes are to be understood in part by their relationship to sequent nodes: Every inference node i has a unique conclusion node c and a (possibly empty) set of hypothesis nodes $\{h_1, \cdots, h_n\}$. An inference node represents the following mathematical relationship between c and $h_1, \cdots, h_n$: if every one of the sequent nodes $h_1, \cdots, h_n$ is valid, then the sequent node c is valid. Alternatively, we can view this as asserting that the validity of the sequent node c can be reduced to the validity of $h_1, \cdots, h_n$. An inference node also contains other information, providing the mathematical justification for asserting the relationship between c and $h_1, \cdots, h_n$. This justification is called an *inference rule.*

Note that if the inference node i has no hypotheses, then c is obviously valid. When this is the case, c is said to be immediately grounded. More generally, c is valid if all the hypotheses $h_1, \cdots, h_n$ of i are valid. Thus validity propagates from hypotheses to conclusion. A sequent node which is recognized as valid in this way is said to be *grounded.* The system indicates that it has recognized validity of a sequent node when it marks it as grounded. A proof is complete when the original sequent node of the deduction graph (i.e., the deduction graph's *goal node*) is marked as grounded.

Sequent nodes and inference nodes are added to a deduction graph using deduction graph *commands.* Some commands correspond to the use of one inference rule; for example **extensionality** or **force-substitution** fall into this category. Applying such a command either does nothing or adds exactly one inference node to the deduction graph. Moreover, this one inference node corresponds quite closely to the application of one mathematical fact. Other commands such as **simplify** have similar behavior in that they either do nothing or add exactly one inference node. Yet they are fundamentally different because behind the scene they are often carrying out thousands of very low-level inferences, such as logical and algebraic simplification and rewrite-rule application. Finally a third class of commands are essentially *strategies* developed to systematically apply primitive commands in search of a proof.

3.6 Extending IMPS

IMPS can be extended in the following ways:

- By adding definitions to an existing theory.
- By building a new theory altogether, with new primitive constants, sorts, and axioms.

For example, in the theory of **h-o-real-arithmetic**, evaluation of the following s-expression adds the definition of convergence to ∞ of a real-valued sequence:

```
(def-constant CONVERGENT%TO%INFINITY
  "lambda(s:[zz,rr],
     forall(m:rr, forsome(x:zz, forall(j:zz,
       x<=j implies m<=s(j)))))"
  (theory h-o-real-arithmetic))
```

As an exercise, try defining the concept of limit of real-valued sequences. Next, we illustrate how new theories are built by building a theory of monoids. Building a theory is a two-step process. First, we build a language with the required sorts and constants:

```
(def-language MONOID-LANGUAGE
  (embedded-languages h-o-real-arithmetic)
  (base-types uu)
  (constants
   (e "uu")
   (** "[uu,uu,uu]")))
```

The next step builds the theory itself by listing the axioms:

```
(def-theory MONOID-THEORY
  (component-theories h-o-real-arithmetic)
  (language monoid-language)
  (axioms
   (associative-law-for-multiplication-for-monoids
    "forall(z,y,x:uu, x**(y**z)=(x**y)**z)" rewrite)
   (right-multiplicative-identity-for-monoids
    "forall(x:uu,x**e=x)" rewrite)
   (left-multiplicative-identity-for-monoids
    "forall(x:uu,e**x=x)" rewrite)
   ("total_q(**,[uu,uu,uu])" d-r-convergence)))
```

There are numerous other ways IMPS can be extended. These are discussed in detail in Chapter 17. In addition, there are a number of annotated files which show you how theories are built and theorems proved in them. These are described in the chapters of this manual (5 and 6) on the micro exercises and the exercises.

3.7 The Little Theories Style

The IMPS methodology for formalizing mathematics is based on a particular version of the axiomatic method. The axiomatic method is commonly used both for encoding existing mathematics and for creating new mathematics. A chunk of mathematics is represented as an *axiomatic theory* consisting of a formal language plus a set of sentences in the language called *axioms*. The axioms specify the mathematical objects to be studied, and facts about

the objects are obtained by reasoning logically from the axioms, that is, by proving theorems.

The axiomatic method comes in two styles, both well established in modern mathematical practice. We refer to them as the "big theory" version and the "little theories" version. In the "big theory" version all reasoning is performed within a single powerful and highly expressive axiomatic theory, such as Zermelo-Fraenkel set theory. The basic idea is that the theory we work in is expressive enough so that any model of it contains all the objects that will be of interest to us, and powerful enough so that theorems about these objects can be proved entirely within this single theory.

In the little theories version, a number of theories are used in the course of developing a portion of mathematics. Different theorems are proved in different theories, depending on the amount and kind of mathematics that is required. Theories are logically linked together by translations called theory interpretations which serve as conduits to pass results from one theory to another. We argue in [10] that this way of organizing mathematics across a network of linked theories is advantageous for managing complex mathematics by means of abstraction and reuse.

IMPS supports both the big theory and little theories versions of the axiomatic method. However, the IMPS little theories machinery of multiple theories and theory interpretations offers the user a rich collection of formalization techniques (described in Chapter 9) that are not easy to imitate in a strict big theory approach.

Chapter 4

On-Line Help

Most of the help and documentation facilities described in this chapter require that you have a workstation running Emacs under X windows. In particular, you need a TEX previewer and one of the Emacs X-menu facilities. On-line help falls into three categories:

- Retrieval of specific entries in the manual, such as documentation of an interactive proof command.
- Retrieval of def-forms. These are the forms which, when evaluated, define the various IMPS objects.
- Presentation and selection of options via pull-down or pop-up menus. This is especially important if you are new to IMPS, because most of your interaction with IMPS can be initiated using the menus.

4.1 The Manual

The manual can be viewed on a TEX preview window by selecting the **Manual** option in the **Help** menu or by entering the Emacs command `M-x imps-manual-entry`. You will be prompted for a specific entry in the manual. The entry can be in one the following categories:

- A command name.
- A def-form name.
- A glossary entry.

As usual, the Emacs input completion facility is available, so that you only need to type in the first few letters of the entry. When you complete your selection, the previewer window (which might be iconified) will display the IMPS manual on the page which contains the entry. However, other pages of the IMPS manual are accessible by using the previewer's page motion commands. This allows you to easily examine related entries in the manual.

4.2 Def-forms

All IMPS objects are built using a specification form called a *def-form.* The following are some examples of def-forms:

```
(def-constant CONVERGENT%TO%INFINITY
  "lambda(s:[zz,rr],
     forall(m:rr,
       forsome(x:zz,
         forall(j:zz, x<=j implies m<=s(j)))))"
  (theory h-o-real-arithmetic))

(def-theory PARTIAL-ORDER
  (language language-for-partial-order)
  (component-theories h-o-real-arithmetic)
  (axioms
   (prec-transitivity
    "forall(a,b,c:uu,
       a prec b and b prec c
        implies
       a prec c)")
   (prec-reflexivity
    "forall(a:uu, a prec a)")
   (prec-anti-symmetry
    "forall(a,b:uu,
       a prec b and b prec a
        implies
       a = b)")))
```

Def-forms have a *tag* which is the second s-expression of the form (that is, the name immediately following the string `def-`). Thus the tags of the two s-expressions above are **convergent%to%infinity** and **partial-order**.

The tag of the def-form is usually the name of the object being defined or modified.

There are two good reasons you may want to locate def-forms:

- You may need to know how an object is defined. For example, in the case of a theory, you may want to know what the primitive constants of the theory are.
- You may want to define a new object modeled on an existing one.

Def-forms can be viewed in an Emacs buffer by selecting the **Find def-form** option in the **Help** menu or by typing `C-c .` in an IMPS buffer. IMPS will then request a name. As usual, the Emacs input completion facility is available, so that you only need to type in the first few letters of the entry. When you complete your selection, IMPS will then search in the theory library for those forms which define the selected entry. Note that there may be more than one entry. There are several reasons for this:

- There may be several theories which have an identically named object.
- An object may be defined in one def-form and modified in another def-form. The definition finder will attempt to locate all such occurrences.
- A name may refer to different kinds of objects. The name **fuba** might refer to a translation, a theorem, and a constant.

Note also that the located form may have a different tag than the name selected. For example, the definition of the theory **partial-order** contains the axiom **prec-transitivity**.

4.3 Using Menus

Almost all user interaction with IMPS can be initiated using menus. The method of menu invocation as well as the menu format depends on the version of Emacs you are using:

- *Free Software Foundation version 19* and *Lucid version 19*menus are invoked by clicking on the menubar. The menus are of the more familiar pull-down variety available on most home computers. The resulting menu has a *label* plus a number of *options*. In the case of Lucid Emacs, some of the options open up to new submenus.

- *Version 18* menus are invoked by clicking the right mouse button on the mode line. Moreover, the menu consists of one or more *panes* which are stacked like cards; each individual pane has a *label* plus a number of *options*.

The options available to the user are determined by the buffer of the selected window. The following is a partial list of the options associated to the two IMPS-related buffer modes:

- *Scheme Mode.* This is the mode used for files of def-forms. The following are some of the options that can be selected from the menus in this mode:
  - Start a deduction graph.
  - Set the current theory.
  - Insert a def-form template.
  - Insert a proof script.
  - Run a proof script (by line or by region).
- *Sequent Mode.* This is the mode used for examining sequent nodes in Emacs buffers. The following are some options that can be selected from the menus in this mode:
  - List applicable commands in a menu. Each entry in this menu can in turn be selected buy clicking right with the mouse. This will apply the corresponding command to the sequent node displayed in the buffer.
  - List applicable macetes in a menu. Each entry in this menu can in turn be selected with the mouse. This will apply the corresponding macete to the sequent node displayed in the buffer.
  - List applicable macetes with a brief description in an Emacs buffer. Each macete can be selected by moving the cursor to the corresponding menu entry and typing `m`.

Note that the applicable commands and macetes are determined dynamically based on the form of the current sequent.

Chapter 5

Micro Exercises

The purpose of these micro exercises is to familiarize you with the basic techniques used in IMPS. Hence we will want to illustrate aspects of the interface as well as several of the basic deductive methods. To describe the micro-exercises, we will assume that you are using the Free Software Foundation Emacs version 19 menu system.

If you have not already done so, start IMPS by issuing the command `start_imps` to a Unix shell, from within the X Window System. When IMPS is ready and has informed you of the current theory (the theory of real arithmetic), pull down the **IMPS-Help** menu on the menubar by depressing and holding any mouse button while pointing to it. Point to the entry **Next micro exercise** and then release the mouse button. At any point in this sequence of exercises, you can always restart the exercise you are working on, advance to the next exercise, or return to the previous one by selecting the appropriate entry under **IMPS-Help.**

5.1 A Combinatorial Identity

The first four micro exercises use variants of a single formula, which IMPS prints (through TEX) in the form shown in Figure 5.1. The word "implication" here asserts that the implication that follows holds between the bulleted items. Similarly, the word "conjunction" on the next line refers to the conjunction of the following (subbulleted) items. The symbol $\Longleftrightarrow$, which we will encounter later, asserts that the the following items are equivalent.

This formula asserts that the **comb** function $\binom{m}{k}$ (often read m choose k) may be computed by Pascal's triangle, in which each entry is the sum of

the two entries above it. In our development, $\binom{m}{k}$ is defined to be equal to $m!/(k! \cdot (m-k)!)$ for any integers m and k. The proof of our combinatorial identity will involve unfolding the definition of **comb** and using a number of algebraic laws (including the recursive properties of factorial). Because of the role of division here, and the fact that we must avoid division by zero, we will also need to reason about the well-definedness of several expressions.

The exercises will involve carrying out portions of the proof several times. At first, for didactic reasons only, we will do the proofs in the slowest, most explicit way imaginable.

A Word about the IMPS Interface. In the course of these micro-exercises, several X windows will be used:

(1) *One or more Emacs windows.* This is the primary way of interacting with IMPS, which is implemented by a process running a Lisp-like language under Emacs's control.

(2) *A TEX previewer.* This window displays typeset output created by IMPS. When IMPS creates new output, raising the window will suffice to have it refresh its contents with the new output. If the window is already raised, a capital `R` (for "Redisplay") will cause it to do the same. The lower case letters `n` and `p` can be used to display the next and previous pages, when more than one page is available.[1]

To raise a partly obscured window, click with the left button on its header stripe.

[1] We also refer to this as the "xview" window, and the process of displaying something in this form as "xviewing" it.

for every $k, m : \mathbf{Z}$ implication
- conjunction
  - $1 \leq k$
  - $k \leq m$
- $\binom{1+m}{k} = \binom{m}{k-1} + \binom{m}{k}$.

Figure 5.1: The Combinatorial Identity

Sequent 1.

for every $k, m : \mathbf{Z}$ implication
- conjunction
  - $1 \leq k$
  - $k \leq m$
- $(1+m)!/(k! \cdot (1+m-k)!) = \binom{m}{k-1} + \binom{m}{k}$.

Figure 5.2: The Combinatorial Identity, with One Occurrence Unfolded

for every $x, y, z : \mathbf{R}$ $\iff$
- $y/z = x$
- conjunction
  - $z^{-1}\downarrow$
  - $x \cdot z = y$.

Figure 5.3: The **left-denominator-removal-for-equalities** Theorem

5.2 A First Step

If you have not started the first micro exercise by means of the **Next micro exercise** item on the **IMPS-Help** menu, do so now.

You will find that IMPS starts a derivation with the formula shown in Figure 5.2. This is simply the identity with the left-most occurrence of **comb** unfolded using its definition. To proceed, you would like to eliminate the denominator on the left side of the equality by multiplying both sides. This manipulation is justified by the theorem shown in Figure 5.3 (be sure to read the $\iff$ as "the following are equivalent"). The downward arrow in $z^{-1}\downarrow$ asserts that z^{-1} is well defined; in effect, it requires that the denominator was nonzero. In the ASCII display used in Emacs buffers, this is written in the form `#(z^[-1])`. This theorem has been proved from the axioms of real arithmetic, and it has been made available for a kind of rewriting in the form of a *macete*.[2] Its behavior as a rewrite is to replace the equation by a

[2] A *macete*, meaning a clever trick in Portuguese slang, is a lemma or group of lemmas

Replace:	with $x, z, y : \mathbf{R}$ $\quad y/z = x.$
By:	with $y, x, z : \mathbf{R}$ $\quad z^{-1}\downarrow \wedge\ x \cdot z = y.$

Figure 5.4: Rewriting Done by Left Denominator Removal

conjunction, as indicated in Figure 5.4.

Step 1. You want to extend the deduction graph (i.e., the partial proof) that you have just begun. Thus, hold down a mouse button while pointing to the **Extend-DG** menubar item and select the option **Macete description.** This will cause IMPS to compute all the macetes that can in fact alter the current goal. In the case at hand, there are 11 macetes that are potentially applicable to our goal, out of 362 currently loaded. A description of each applicable macete is sent to TEX, and will be available after 10–15 seconds on the "xdvi" window.

When you have inspected the possibilities, and have noted the number of left denominator removal on the list, return to the Emacs window. Type the number, followed by a carriage return, to cause IMPS to apply that macete.

Result. You will now see a new subgoal on the Emacs display. It is the result of the rewriting carried out by the selected macete. The letters **p** and **n** will carry you to the previous display and then back to the next one. To inspect both sequents using TEX, pull down the TeX menu and select the **Xview sqns** item. Emacs will prompt you (at the bottom of the screen) for the numbers. Enter 1 and 2 separated by a space and terminated by a carriage return. After 10–15 seconds you will be able to compare the two formulas on the "xdvi" screen.

Please also observe in the "xdg" window that Sequent 1 has been reduced to Sequent 2 by the inference **left-denominator-removal-for-equalities**. If one later carries out inferences that complete the proof of Sequent 2, then they will also suffice to establish (or "ground") Sequent 1.

---

made available to the user for various kinds of rewriting depending on syntactic form.

5.3 Taking More Steps

Under the **IMPS-Help** menu, please select the next micro exercise. You will now see the combinatorial identity displayed as the goal of a new deduction.

Step 1. Under the **Extend-DG** menu, please select the item **Commands** (this function may also be invoked without menus by typing a `?`). This will cause IMPS to examine which proof steps (other than macetes) are applicable to the current sequent. Of the 70 commands currently loaded, only 16 are possible, given the syntactic form of the goal. They will be displayed on a menu. Move your mouse to the item **unfold-single-defined-constant**, and click on it (this function may also be invoked without menus by typing a lower case `u`). IMPS will prompt at the bottom of the screen for the numerical index of the occurrence to unfold. This is a zero-based index. Please unfold the leftmost occurrence by typing 0, followed by a carriage return.

Result. IMPS will post a new sequent with the leftmost occurrence replaced by its definition. This is identical to the formula you started the first micro exercise with.

Step 2. Invoke the macete left denominator removal again. This may be done as before by using the macete description item. Alternatively, it is less cumbrous to find **Macetes (with minor premises)** entry under **Extend-DG.** Clicking on this will put up a menu of names of the currently applicable macetes. Click on **left-denominator-removal-for-equalities**.

Result. IMPS will post an additional sequent with the result of the rewriting. This is identical to the formula you obtained in the first micro exercise.

Step 3. Again unfold the leftmost occurrence of **comb.** As this formula has more than one defined constant in it (**factorial** is also introduced by definition), IMPS will prompt with a menu displaying the syntactically available constants which may be chosen. You will again need to type `0` to the prompt at the bottom of the Emacs screen to indicate the leftmost occurrence.

Result. IMPS will post another sequent with the subgoal shown in Figure 5.5. Since this has a subexpression of the form $(a/b) + c$, you will want to rewrite it to the form $(a + b \cdot c)/b$.

for every $k, m : \mathbf{Z}$ implication
- conjunction
  - $1 \leq k$
  - $k \leq m$
- conjunction
  - $(k! \cdot (1 + m - k)!)^{-1}$ ↓
  - $(m!/((k-1)! \cdot (m - (k-1))!) + \binom{m}{k}) \cdot k! \cdot (1 + m - k)! = (1 + m)!$.

Figure 5.5: Result of the Second Unfolding

Step 4. Use the macete description menu item under **Extend-DG** to find a macete to carry out this transformation.

Result. The resulting sequent has a subexpression of the form $a/b \cdot c$. You will want to bring this to the form $(a \cdot c)/b$.

Step 5. Use the macete description menu item under **Extend-DG** to find a macete to carry out this transformation.

Clearly, invoking individual macetes in this way can become unbearably tedious. You can now see the importance of being able to group together a collection of lemmas into a "compound macete." They can thus all be used in one step, wherever they apply syntactically. A compound macete may be constructed so as to apply the lemmas repeatedly until there are no more opportunities for applying any of them.

5.4 Taking Larger Steps

Under the **IMPS-Help** menu, please select the next micro exercise. You will again see the combinatorial identity displayed as the goal of a new deduction. This time we will start by eliminating all the occurrences of **comb** in favor of the defining expression.

Step 1. Under the **Extend-DG** menu, please again select the item **Commands.** Move your mouse to the item **unfold-single-defined-constant-globally**, and click on it (this function may also be invoked without menus

Sequent 3.

for every $k, m : \mathbf{Z}$ implication
- conjunction
  - $1 \leq k$
  - $k \leq m$
- $(1+m)!/(k! \cdot (1+m-k)!)$
  $= m!/((k-1)! \cdot (m-(k-1))!) + m!/(k! \cdot (m-k)!)$.

Figure 5.6: All Occurrences have been Eliminated

by typing `U`. IMPS does not need to prompt for a numerical index of the occurrence to unfold, as it will operate on all of the occurrences.

Result. The resulting sequent, shown in Figure 5.6, will need a good deal of algebraic manipulation. A large part of it can be carried out by a single compound macete, named **fractional-expression-manipulation**.

Step 2. Under the **TeX** menubar item, click on **Xview Macete.** IMPS will prompt you at the bottom of the screen for a name. Type the first few letters of fractional-expression-manipulation followed by a few spaces, and Emacs will complete the name by comparing against the possible macete names.

In 10–15 seconds you will see the description of this bulky compound macete. It first applies beta-reduction, to plug in the arguments of any functions expressed using the λ variable-binding constructor. After that, it repeatedly applies a sequence of elementary macetes until none of them can be applied any more. It then moves on to a second sequence of elementary macetes, and applies them repeatedly until no further progress is made.

It is possible to write nonterminating macetes using the "repeat" operator. IMPS is designed so that it can be interrupted easily and safely. To do so, type `ESC x interrupt-tea-process`. If you decide that you should not have interrupted it, you can cause it to return to the computation at which it was interrupted. To do so, put your cursor at the end of the ***tea*** buffer and type `(ret)` followed by a carriage return.

To execute the macete **fractional-expression-manipulation** on your current goal, use the item **Macetes (with minor premises)** under

```
(SERIES BETA-REDUCE
        (REPEAT INVERSE-REPLACEMENT
                SUBTRACTION-REPLACEMENT
                NEGATIVE-REPLACEMENT
                ADDITION-OF-FRACTIONS-LEFT
                ADDITION-OF-FRACTIONS-RIGHT
                MULTIPLICATION-OF-FRACTIONS-LEFT
                MULTIPLICATION-OF-FRACTIONS-RIGHT
                DIVISION-OF-FRACTIONS
                DIVISION-OF-FRACTIONS-2
                EXPONENTS-OF-FRACTIONS
                NEGATIVE-EXPONENT-REPLACEMENT)
        (REPEAT LEFT-DENOMINATOR-REMOVAL-FOR-EQUALITIES
                RIGHT-DENOMINATOR-REMOVAL-FOR-EQUALITIES
                RIGHT-DENOMINATOR-REMOVAL-FOR-STRICT-INEQUALITIES
                LEFT-DENOMINATOR-REMOVAL-FOR-STRICT-INEQUALITIES
                RIGHT-DENOMINATOR-REMOVAL-FOR-INEQUALITIES
                LEFT-DENOMINATOR-REMOVAL-FOR-INEQUALITIES
                MULTIPLICATION-OF-FRACTIONS-LEFT
                MULTIPLICATION-OF-FRACTIONS-RIGHT))
```

Figure 5.7: The Form of **fractional-expression-manipulation**

Sequent 4.

for every $k, m : \mathbf{Z}$ implication
- conjunction
  - $1 \leq k$
  - $k \leq m$
- conjunction
  - $(k! \cdot (1 + m + -1 \cdot k)!)^{-1}\downarrow$
  - conjunction
    - $(k! \cdot (m + -1 \cdot k)! \cdot (k + -1 \cdot 1)! \cdot (m + -1 \cdot (k + -1 \cdot 1))!)^{-1}\downarrow$
    - $(1 + m)! \cdot k! \cdot (m + -1 \cdot k)! \cdot (k + -1 \cdot 1)! \cdot (m + -1 \cdot (k + -1 \cdot 1))!$
      $= ((k + -1 \cdot 1)! \cdot (m + -1 \cdot (k + -1 \cdot 1))! \cdot m! +$
      $k! \cdot (m + -1 \cdot k)! \cdot m!) \cdot k! \cdot (1 + m + -1 \cdot k)!.$

Figure 5.8: The Result of **fractional expression manipulation**

the **Extend-DG** menubar item. You will find **fractional-expression-manipulation** among the list of potentially useful macetes. It will execute for about 20 seconds.

Result. The macete will cause IMPS to carry out all the algebraic operations involved in adding and multiplying fractions, and will cause it to cross-multiply (eliminate denominators) for equations such as $a/b = c$. Since this is permissible only when $b^{-1}\downarrow$, we obtain a conjunction of assertions. The first two conjuncts assert that two denominators were nonzero, and the third is a bulky algebraic equation which—assuming the denominators were nonzero—suffices for our previous subgoal to be true.

We want next to break up the logical structure of this subgoal. To dispense with the leading "for all," we will consider an arbitrary k and m. To break up the implication, we will pull the antecedent into the left hand side of the sequent. We refer to this as its *context*. That is, we shall try to prove the consequent of the conditional in a context where the antecedent is assumed true. Finally, we will break up the conjunction so that we can prove each conjunct separately.

Step 3. To break up the sequent in this way, look for **direct-and-antecedent-inference-strategy** under the command menu. The com-

Sequent 9.

Let $m, k : \mathbf{Z}$.

Assume:

(0) $k \leq m$.

(1) $1 \leq k$.

Then:

$(k! \cdot (1 + m + -1 \cdot k)!)^{-1} \downarrow$.

Figure 5.9: The Simpler Definedness Subgoal

mand menu may be invoked from **Extend-DG.**

Result. As a consequence nine new sequent nodes will be added to the deduction graph. Of these, three are "leaf nodes" and will need to be proved separately. They correspond to the three conjuncts of the previous subgoal. We will prove one of them in a slightly tedious step-by-step process, and the other two by applying compound macetes. The first subgoal (Sequent 9) is shown in Figure 5.9. To prove this in a step-by-step process, we will need to use the the following facts:

(1) the domain of inverse is $\mathbf{R}\backslash\{0\}$;

(2) a product $x \cdot y$ is nonzero just in case both x and y are;

(3) a factorial $j!$ is always nonzero.

We will then use the IMPS simplifier to complete the proof of this subgoal.

Step 4. Find the macete **domain-of-inverse** using either **Macete description** or the macete menu.

Step 5. Find the macete **non-zero-product** using either **Macete description** or the macete menu.

Step 5. Find the macete **factorial-non-zero** using either **Macete description** or the macete menu.

Step 6. Under the **Command** menu item, find **simplify** (usually about half-way down the list).

Result. You will see that IMPS has marked this subgoal as "Grounded" in the black stripe under the sequent node buffer. This means that IMPS has now ascertained that it is true. In the **xdg** window you will see that Sequent 7 is marked as grounded and the sequents supporting it are no longer displayed.

To move to the next subgoal, either click (left) on it in the **xdg** window, or else look under the **Nodes** menubar item for the **First unsupported relative.** The resulting sequent is a more complicated definedness assertion.

Step 7. Rather than using the individual lemmas, we will use the compound macete **definedness-manipulations**. To see how the macete is built, use the **Xview macete** item under the **TeX** menubar item. Invoke it via the macete menu or through the **Macete description** item. It will ground the subgoal after about 30 seconds of computation.

Step 8. To move to the last remaining subgoal, either click (left) on it in the **xdg** window, or else look under the **Nodes** menubar item for the **First unsupported relative.** Apply the compound macete **factorial-reduction** (Figure 5.10), which will complete the proof after another 30 seconds of computation.

You will note that it uses a macete constructor **without minor premises.** This (as well as the phrase "with minor premises" appearing on the **Extend-DG** menu) may be explained with reference to the macete **factorial-out**. The substitution of $(m-1)! \cdot m$ in place of $m!$ is permissible only when $1 \leq m$. (! is defined by a syntactic recursion, rather than by reference to the Γ function; thus in particular $0! = 1 \neq (-1)! \cdot 0 = 0$.)

There are two ways then to use the macete, both of them perfectly sound from a logical point of view. The first is conservative: do the substitution only when we can immediately see that the condition is satisfied for the instance at hand. The second approach is more speculative: do the substitution, but post an additional subgoal for the user to prove later. The

```
(REPEAT
 (WITHOUT-MINOR-PREMISES
  (REPEAT FACTORIAL-OUT FACTORIAL-OF-NONPOSITIVE))
 SIMPLIFY)
```

Where:

(1) **factorial-out:**

Replace: with $m : \mathbf{Z}$ $m!$.
By: with $m : \mathbf{Z}$ $(m-1)! \cdot m$.
Subject to: with $m : \mathbf{Z}$ $1 \leq m$.

(2) **factorial-of-nonpositive:**

Replace: with $j : \mathbf{Z}$ $j!$.
By: 1.
Subject to: with $j : \mathbf{Z}$ $j \leq 0$.

Figure 5.10: The **factorial-reduction** Macete

```
(apply-macete-with-minor-premises domain-of-inverse)
(apply-macete-with-minor-premises nonzero-product)
(apply-macete-with-minor-premises factorial-nonzero)
simplify
```

Figure 5.11: Commands to Delete

additional subgoal states that the condition is satisfied, and the user can use whatever ingenuity is needed in order to prove it.

We refer to such an additional subgoal as a minor premise. Experience shows that in the great majority of cases, it is more useful to make the substitution and to post any conditions that cannot be discharged immediately as future obligations.

However, in the case of a computational macete such as this one, the opposite is the case. We want to "compute out" as many of the occurrences of factorial as we can in the expression at hand. But we can reduce a particular instance of $t!$, only if we know which rule to use. In particular, if whether t is positive depends on the particular choice of values for free variables in t, then no expansion can be correct in general. The macete constructor **without minor premises** ensures that the macetes within its scope will be used without minor premises no matter how the macete is called by the user.

5.5 Saving a Proof

Do not start another micro exercise. Instead, open a new file. You may use the **File** menubar item, clicking on the **Open file** item, supplying an unused filename. With your cursor in the buffer for the new file, open the **Scripts** menubar item, and click on **Insert proof script.** You will see a textual summary of the interactive proof that you have constructed. This textual summary can be executed again, or edited as text and then re-executed.

Now click on the **Next micro exercise** item within **IMPS-Help.** IMPS will start yet another derivation of the combinatorial identity.

If you are unfamiliar with Emacs, drag the mouse (left), and use the **Edit** menubar entries to remove the text describing four steps shown in Figure 5.11. These were the detailed steps used to prove the first definedness

Sequent 1.

for every $m : \mathbf{Z}$ implication
- $0 \leq m$
- $\sum_{j=0}^{m} j = m \cdot (m+1)/2$.

Figure 5.12: The Sum of the First n Natural Numbers

assertion. In their place it would suffice to use definedness manipulations. Mark the following step by dragging the mouse (left), and then use the **Copy** and **Paste** entries in the **Edit** menu to duplicate the line. (Add a carriage return if appropriate.)

Starting with your cursor on the first line of the new sequence of commands, click on the menu item **Execute proof line** under **Scripts.** When IMPS updates the sequent nodes buffer with the new subgoal, repeat the process. At each step, wait until the previous step has completed. Six steps complete the proof.

5.6 Induction, Taking a Single Step

Axioms or theorems that we may regard as induction principles play a crucial role in many theories. IMPS provides good support for inductive reasoning in a form that is independent of particular theories. That is, when an axiom or theorem of the right form is available in any theory, then it can be applied with the full power of the IMPS induction command.

We will illustrate IMPS's treatment of induction first in a manual way that shows only the single most crucial step.

If you again click the menu item for the next micro exercise, you will be relieved to see a new goal, also shown in Figure 5.12.

Step 1. Invoke the macete **little-integer-induction** on the macete menu.

Result. The macete will identify the predicate of induction, and apply the induction principle on the integers, resulting in the subgoal shown in Figure fig:inductdone. The first conjunct here is the base case, and the second conjunct is the induction step. Within the induction step, we see the

Sequent 2.

conjunction
- $\sum_{j=0}^{0} j = 0 \cdot (0+1)/2$
- for every $t : \mathbf{Z}$ implication
  - $0 \leq t$
  - implication
    - $\sum_{j=0}^{t} j = t \cdot (t+1)/2$
    - $\sum_{j=0}^{t+1} j = (t+1) \cdot (t+1+1)/2$.

Figure 5.13: Result of Applying the **little-integer-induction** Macete

hypothesis $0 \leq t$ which is needed because we are working with $\mathbf{Z}$ rather than $\mathbf{N}$. Subject to this assumption, the induction step asserts that the predicate remains true of $t+1$, assuming that it held true of t.

5.7 The Induction Command

There are many heuristics that pay off with induction principles. For instance, recursively defined operators applied to the inductive term should be expanded in the induction step.

For this reason, IMPS contains a command for induction which will apply an induction principle as in the previous micro exercise, after which it separates the base case from the induction step and applies appropriate heuristics to each.

IMPS offers some flexibility to the developer of a theory to control what heuristics will be used in induction. This is encoded in an object called an inductor. An inductor contains the induction principle to be used, in the form of a macete similar to the little integer induction macete used above. In addition, it may contain additional macetes or commands to be used in the base case or induction step. It may also control whether definitions are expanded, and if so which ones.

Clicking on the next micro exercise will restart a derivation for the sum of the first n natural numbers.

Step 1. Within the **Extend-DG** menubar item, click on the **Commands** entry. Near the top of the resulting menu will be the **induction** command. Click on it. IMPS will respond with another menu prompting for the name of an inductor. **integer inductor** is the most frequent choice, which expands recursive and nonrecursive definitions in syntactically appropriate contexts. The **nonrecursive-integer-inductor** expands only nonrecursive definitions, and the **trivial-integer-inductor** expands none. In the case we are currently considering, select the option **integer-inductor**.

Result. After about 10–15 seconds IMPS will post 11 new subgoals, all of which are grounded, thus completing the proof.

To inspect the proof in detail, under the **DG** menubar item, click on **Verbosely update dg display.** This will show the structure of the derivation. To view all of the individual nodes, under **TeX,** click on **Xview sqns.** Enter all of the numbers from 1 to 12 separated by spaces.

You are now ready for the more interesting exercises which are intended to illustrate how interesting portions of mathematics and computer science can be developed using IMPS. This is the topic of the next chapter of this manual.

Chapter 6

Exercises

6.1 Introduction

This chapter describes about a dozen self-guided, content-based exercises to illustrate the use of IMPS. Each exercise uses a fairly self-contained piece of mathematics, logic, or computer science to show how to organize and develop a theory in IMPS, and how to carry out the proofs needed for the process. We will try to describe in each case the main content of the exercise, as well as the specific techniques that it is intended to illustrate.

We assume in this chapter that you are using the Lucid 19 menu system.

6.1.1 How to Use an Exercise

In order to start an exercise, press down any mouse button while pointing to the **IMPS-Help** menubar item. Release the button while pointing to the entry **Exercises.** IMPS will present a pop-up menu listing the file names of the available exercises. Clicking on a file name will cause the following actions:

- The system will create a new copy of the file under your own directory, in the subdirectory ∼**/imps/theories**. Since this is a new file, you can safely edit it or delete it. If a file of the expected name already exists, IMPS will prompt whether to overwrite the file, thus destroying the previous copy.

- IMPS will create a new Emacs buffer displaying the copy. You will use this buffer to view and edit the file.

- IMPS will search for the first item in the file meriting your attention. Each such item will be marked by the string `(!)`. These items may be candidate theorems to prove, theory declarations or definitions to read, or macetes or scripts to create.
- Anything above the first `(!)` will be sent as input to the underlying T process. Thus, IMPS will define any notions mentioned at the top of the file.

At any point in the exercise, to proceed to the next item of interest, select—from the **IMPS-Help** menubar item—the **Current Exercise: Next Entry** request.

- IMPS will search for the next item in the file marked by the string `(!)`.
- Anything beyond the first `(!)`, up to the current occurrence, will be sent as input to the underlying T process. Thus, IMPS will install the definitions, theorems, and so forth described in that portion of the file.

If you want to restart an exercise in a clean way, click on **Exit IMPS** under **General,** and then on **Run IMPS.** The system will prompt you to confirm the desired amount of virtual memory; the default is plenty for current purposes. Having started a fresh invocation of IMPS, put your cursor in the buffer that you want to work on. Click on the item **Restart Exercise in Current Buffer** under **IMPS-Help.**

6.1.2 Paths Through the Exercises

The exercises divide roughly into four tiers of difficulty, which are summarized in Table 6.1. We recommend that you do one or more exercises in a tier before moving on to those in the next tier.

6.2 Mathematical Exercises

One of the main goals of the IMPS system is to support the rigorous development of mathematics. We hope that it will come to serve as a mathematical data base, storing mathematical concepts, theories, theorems, and proof techniques in a form that makes them available when they are relevant to new work. We believe that as a consequence IMPS will develop into a system that can serve students as a general mathematics laboratory. In addition, we hope it will eventually serve as a useful adjunct in mathematics research.

Tier	Exercise file name	Section
1.	primes-exercise.t	6.2.1
	calculus-exercise.t	6.2.2
2.	indicators-exercise.t	6.3.1
	monoid-exercise.t	6.2.3
3.	compiler.t	6.4.1
	limits.t	6.2.4
	contraction.t	6.2.5
	groups-exercise.t	6.2.6
4.	flatten.t	6.4.2
	temporal.t	6.3.2

Table 6.1: Exercise Files by "Tier"

6.2.1 Primes

Contents. This exercise introduces two main definitions into the theory of real arithmetic, namely divisibility and primality. It then includes a few lemmas, which lead to a "computational" macete[1] to determine whether a positive integer is prime. After introducing the notion of twin primes, the exercise develops two proof scripts that can be used to find a twin prime pair in a given interval, if one exists.

A more extensive treatment of primes, including the material of this exercise, is included in the IMPS theory library in the file

$IMPS/theories/reals/primes.t.

That file also contains statements and proofs of the following:

- the infinity of primes;
- the main properties of gcd;
- the existence and uniqueness of prime factorizations (the Fundamental Theorem of Arithmetic).

[1] A *macete*, meaning a clever trick in Portuguese slang, is a lemma or group of lemmas made available to the user for various kinds of rewriting depending on syntactic form.

Purpose. This exercise illustrates basic definitions, and it illustrates how to carry out several simple proofs. Its main interest however is to illustrate the mix of computation and proof in IMPS. This balance is reflected in the macete mechanism, and in the script that searches for a pair of twin primes.

6.2.2 A Very Little Theory of Differentiation

Contents. This file develops a little axiomatic theory of the derivative operator on real functions. Its axioms include the rules for:

- sum;
- product;
- constant functions;
- the identity function;

as well as the principle that the derivative of f is defined at a value x only if f is. Naturally, these properties hold true of the derivative as explicitly defined in the usual way.

The exercise then develops a succession of consequences of these properties, such as the power rule shown in Figure 6.1. The properties and their consequences are then grouped together into a macete that allows expressions involving derivatives to be evaluated.

Purpose. This exercise is intended to illustrate, in addition to inductive reasoning, the manipulation of higher-order objects (functions represented using λ), and the power of macetes to encode symbolic computations.

for every $n : \mathbf{Z}, f : \mathbf{R} \rightharpoonup \mathbf{R}, x : \mathbf{R}$ implication
- $\mathcal{D}(f)(x) \downarrow \wedge\, 2 \leq n$
- $\mathcal{D}(\lambda\{x : \mathbf{R} \,|\, f(x)^n\})(x) = n \cdot f(x)^{n-1} \cdot \mathcal{D}(f)(x)$.

Figure 6.1: Generalized Power Rule

for every $f : \mathbf{Z} \rightharpoonup \mathbf{U}, m, n : \mathbf{Z}$ implication
- conjunction
  - $m \leq n$
  - $\forall j : \mathbf{Z} \quad (m \leq j \wedge j \leq n + 1) \supset f(j) \downarrow$
- $\prod_m^n \delta(f) = \text{minus}(f(n+1), f(m))$.

Figure 6.2: Telescoping Product Formula

6.2.3 Monoids

Contents. This exercises introduces a theory of monoids, a monoid being a domain $\mathbf{U}$ equipped with an associative operator $\cdot$ and an identity element e. A recursive definition introduces an iterated monoid product operator $\prod$, such that, when $i, j : \mathbf{Z}$ and $f : \mathbf{Z} \rightharpoonup \mathbf{U}$, then $\prod(i, j, f)$ satisfies:

- $\prod_i^j f = e$ when $j < i$;
- $\prod_i^j f = (\prod_i^{j-1} f) \cdot f(j)$ when $i \leq j$.

By adding an inverse operator and axioms on it, the monoid theory is extended to a theory of groups (defined independently of the theory in Exercise 6.2.6). The file then defines (slightly incongruously) minus to mean:

$$\lambda x, y : \mathbf{U} \,.\, x \cdot y^{-1},$$

and δ to mean:

$$\lambda f : \mathbf{Z} \rightharpoonup \mathbf{U} \,.\, \lambda j : \mathbf{Z} \,.\, \text{minus}(f(j+1), f(j)).$$

That is, if f is a function of sort $\mathbf{Z} \rightharpoonup \mathbf{U}$, then $\delta(f)$ is another function of the same sort, and its value for an integer j is the difference (in the sense of "minus") between $f(j+1)$ and $f(j)$.

We can then prove the telescoping product formula shown in Figure 6.2. This in turn can serve as the basis of an alternative proof for formulas such as the ones for the sum of the first n natural numbers or of their squares. To do so, we introduce an interpretation mapping $\mathbf{U}$ to $\mathbf{R}$ and the monoid operator $\cdot$ to real addition. As a consequence, the operator $\prod$ within the monoid is carried to the (real-valued) finite summation operator $\sum$.

Purpose. This exercise illustrates how to develop little theories and how to apply them via interpretation. It also illustrates a simple recursive definition.

6.2.4 Some Theorems on Limits

Contents. This exercise defines the limit of a sequence of real numbers. A sequence of reals is represented in IMPS as a partial function mapping integers to real numbers. The limit is defined using the **iota** constructor ι, where $\iota x{:}\sigma\ .\ \varphi(x)$ has as its value the unique value of x such that $\varphi(x)$ holds true. The iota-expression is undefined if φ is satisfied by more than one value or if it is not satisfied for any value. In this case, the instance of φ is the familiar property that, for every positive real ϵ, there is an integer n, such that for indices p greater than n, $|x - s(p)| \leq \epsilon$.

Theorems include a variant of the definition in which **iota** does not occur, as well as the familiar condition for the existence of the limit. The homogeneity of limit is also proved.

Purpose. The main purpose of this exercise is to illustrate how to reason with **iota**. In particular, it illustrates the importance of deriving an "iota-free" characterization for a notion defined in terms of **iota**.

6.2.5 Banach's Fixed Point Theorem

Contents. This extensive file, **contraction.t**, contains a complete proof of Banach's contractive mapping fixed point theorem. An alternative proof using a little more general machinery is contained in the theory library in file

$IMPS/theories/metric-spaces/fixed-point-theorem.t

The theorem states that in a complete metric space $\langle \mathbf{P}, d\rangle$, any contractive function has a unique fixed point. A function f is contractive if, there exists a real value $k < 1$ such that for all points x and y in $\mathbf{P}$,

$$d(f(x), f(y)) \leq k \cdot d(x, y).$$

The heart of the proof is to show that under this condition the sequence s of iterates of f starting from an arbitrary point x,

$$s = \langle x, f(x), f(f(x)), \ldots, f^n(x), \ldots\rangle$$

will converge. It is then easy to show that the limit of s is a fixed point. Finally, if any two distinct points x and y were both fixed, then we would have

$$d(x, y) = d(f(x), f(y)) \leq k \cdot d(x, y) < d(x, y).$$

This theorem serves as a very good illustration of IMPS's "little theories" approach. It combines:

- Concrete theorems about the reals, such as the geometric series formula, which are needed to bound the differences between points in s;
- A few abstract facts about the sequence of iterates of a function, which are perfectly generic and independent of the fact that we will eventually apply them to functions $f : \mathbf{P} \to \mathbf{P}$;
- A third group of theorems proved in the theory of a metric space, applying the previous theorems.

Purpose. The exercise has three purposes:

- To illustrate the little theories approach and how natural it is in mathematics;
- To illustrate inductive proof (in combination with some other techniques) in real arithmetic and also in the generic theory covering iteration;
- To provide a richer, more extended example of how to develop a substantial proof using IMPS.

6.2.6 Basics of Group Theory

Contents. This file develops the very rudiments of group theory, only up to a definition of subgroup and a proof that subgroups are groups. Most of the content is devoted to building up computational lemmas. This exercise covers only a very small part of the portion of group theory that is developed in the IMPS theory library, in the files contained in the directory:

$IMPS/theories/groups/

Purpose. The purpose of this exercise is to illustrate how to handcraft the simplification mechanism by proving rewrite rules. In addition, a compound macete is introduced to carry out computations not suited to the simplifier. The file also illustrates the use of a symmetry translation mapping the theory to itself. This translation maps the multiplication operator $\cdot$ to

$$\lambda x, y \; . \; y \cdot x.$$

Thus, for instance, right cancellation follows by symmetry from left cancellation.

6.3 Logical Exercises

6.3.1 Indicators: A Representation of Sets

Contents. An indicator is a function used to represent a set. It has a sort of the form $[\alpha, \text{unit\%sort}]$, where α is an arbitrary sort. (In the string syntax, the sort $[\alpha, \text{unit\%sort}]$ is printed as `sets[`α`]`.) **unit%sort** is a type in **the-kernel-theory** that contains exactly one element named **an%individual**. The type is of kind ι, so indicators may be partial functions. Since **the-kernel-theory** is a component theory of every IMPS theory, **unit%sort** is available in every theory.

Indicators are convenient for representing sets. The basic idea is that a set s containing objects of sort α can be represented by an indicator of sort $[\alpha, \text{unit\%sort}]$, namely the one whose domain is s itself. Simplifying expressions involving indicators is to a large extent a matter of checking definedness—for which the simplifier is equipped with special-purpose algorithms. The theory **indicators** consists of just a single base sort **U**, for the "universe" of some unspecified domain. Since the theory **indicators** contains no constants nor axioms, theory interpretations of **indicators** are trivial to construct, and thus theorems of **indicators** are easy to use as transportable macetes.

The logical operators in IMPS are fixed, but *quasi-constructors* can in some respects effectively serve as additional logical operators. Quasi-constructors are desirable for several reasons:

- Once a quasi-constructor is defined, it is available in every theory whose language contains the quasi-constructor's home language. That is, a quasi-constructor is a kind of *global constant* that can be freely

used across a large class of theories. (A constructor, which we also call a logical constant, is available in *every* theory.)

- Quasi-constructors are *polymorphic* in the sense that they can be applied to expressions of several different types. (Several of the constructors, such as = and **if**, are also polymorphic in this sense.)

- Quasi-constructors are preserved under translation. Hence, to reason about expressions involving a quasi-constructor, we may prove laws involving the quasi-constructor in a single generic theory. The translation mechanism can then be used to apply the theorems to expressions in any theory that involve the same quasi-constructor.

- Quasi-constructors can be used to represent operators in nonclassical logics, such as modal or temporal logics (see Section 6.3.2), and operators on generic objects such as sets (as represented by indicators) and sequences.

Quasi-constructors are implemented as "macro/abbreviations." The IMPS reader treats them as macros which cause the creation of a particular syntactic pattern in the expression, while the IMPS printer is responsible for printing them in abbreviated form (wherever those patterns may have arisen).

The exercise derives a few facts about sets, their unions, and the inclusion relation. It applies them to intervals of integers.

Purpose. The purpose of this exercise is to provide experience reasoning with quasi-constructors.

6.3.2 Temporal Logic

Contents. Temporal logic is a kind of modal logic for reasoning about dynamic phenomenon. In recent years temporal logic has won favor as a logic for reasoning about programs, particularly concurrent programs. The following is an exercise to develop a version of propositional temporal logic (PTL) in IMPS.

In this exercise we view time as discrete and linear; that is, we identify time with the integers. The goal of the exercise is to build machinery for reasoning about "time predicates," which are simply expressions of sort $\mathbf{Z} \rightarrow$ **prop**. In traditional PTL, the time argument is suppressed and objects which

are syntactically formulas are manipulated instead of syntactic predicates. In IMPS we will deal directly with the predicates.

This is a very open-ended exercise, which should be done after the user is fairly familiar with IMPS and its *modus operandi*.

6.4 Exercises related to Computer Science

IMPS also provides some facilities for reasoning about computing applications, although these are still less extensive than those for mathematics. The exercises illustrate a facility for defining recursive data types, as well as an application of a theory of state machines.

6.4.1 The World's Smallest Compiler

Contents. This exercise introduces two syntaxes as abstract data types. Each syntax is given a semantics as a simple programming language. A function defined by primitive recursion on one language (the "source language") maps its expressions to values in the other (the "target language"). This function is the compiler. We prove that the semantics of its output code matches the semantics of its input.

Purpose. The primary purpose of this exercise is to illustrate the BNF mechanism for introducing a recursive abstract data type. Connected with a BNF is a schema for recursion on the data type. This is a form of primitive recursion, as distinguished from IMPS's more general facility for monotone inductive definitions. The primitive recursion schema is preferable when it applies because it automatically creates rewrite rules for the separate syntactic cases.

A secondary purpose is to illustrate in a very simple case an approach to compiler verification.

6.4.2 A Linearization Algorithm for a Compiler

Contents. A linearizer is a compiler component responsible for transforming a tree-like intermediate code into a linear structure. Its purpose is typically to transform (potentially nested) conditional statements into the representation using branch and conditional branch instructions that take numerical arguments. The exercise is contained in the file **flatten.t**.

It also depends on the BNF and primitive recursion mechanisms. However, the proofs have more interesting induction hypotheses than in the world's smallest compiler, and the development calls for an auxiliary notion of "regular" instruction sequences.

Purpose. This exercise can be used as a more realistic and open-ended example of a computer science verification.

6.4.3 The Bell-LaPadula Exercise

Contents. This exercise consists of two files, namely:

- **$IMPS/theories/exercises/fun-thm-security.t**
- **$IMPS/theories/exercises/bell-lapadula.t**

The first file proves the Bell-LaPadula "Fundamental Theorem of Security" in the context of the theory of an arbitrary deterministic state machine with a start state, augmented by an unspecified notion of "secure state." (A similar theorem also holds for nondeterministic state machines.) The proof is by induction on the accessible states of the machine.

The second file instantiates this theory in the case of a simple access control device (reference monitor). The states are functions, which, given a subject and an object returns a pair; one component of the pair represents whether the subject has current read access to the object, while the other represents current write access. The operations of the machine correspond to four requests by a subject concerning an object; the request may be to add or delete and access, which may in turn be either read or write access. The subjects and objects have security levels, and the requests are adjudicated based on their levels.

The theory exploits the fact that read and write are duals, when the sense of the partial ordering on levels is inverted. This duality is expressed by means of a theory interpretation mapping the theory to itself. It is used to manufacture definitions; for instance, the $*$-property is *defined* to be the dual of the simple security property. It is also used to create new theorems. For instance, the theorem that the $*$-property is preserved under *get-write* is the dual of the theorem that the simple security property is preserved under *get-read*; hence, it suffices to prove the latter, and the former follows automatically.

Purpose. This exercise has three main purposes:

- To illustrate how to extend and to apply the state machine theory, and how to prove theorems using state machine induction;
- To illustrate the use of an interpretation from a theory to itself to encode a "symmetry" or "duality;"
- To provide a model for a familiar kind of security verification.

Part II

User's Guide

Chapter 7

Logic Overview

7.1 Introduction

The logic of IMPS is called LUTINS[1], a Logic of Undefined Terms for Inference in a Natural Style. LUTINS is a kind of predicate logic that is intended to support standard mathematical reasoning. It is classical in the sense that it allows nonconstructive reasoning, but it is nonclassical in the sense that terms may be nondenoting. The logic, however, is bivalent; formulas are always defined.

Unlike first-order predicate logic, LUTINS provides strong support for specifying and reasoning about partial functions (i.e., functions which are not necessarily defined on all arguments), including the following mechanisms:

- λ-notation for specifying functions;
- an infinite hierarchy of function types for organizing higher-order functions, i.e., functions which take other functions as arguments;
- full quantification (universal and existential) over all function types.

In addition to these mechanisms, LUTINS possesses a definite description operator and a system of subtypes, both of which are extremely useful for reasoning about functions as well as other objects.

The treatment of partial functions in LUTINS is studied in [4], while the treatment of subtypes is the subject of [5]. In this chapter we give a brief

[1] Pronounced as the word in French.

overview of LUTINS. For a detailed presentation of LUTINS, see [16][2].

7.2 Languages

A LUTINS language contains two kinds of objects: *sorts* and *expressions*. Sorts denote nonempty domains of elements (mathematical objects), while expressions denote members of these domains. Expressions are used for both referring to elements and making statements about them.

In this section we give a brief description of a hypothetical language $\mathcal{L}$. This language will be presented using the "mathematical syntax" which will be used throughout the rest of the manual. There are other syntaxes for LUTINS, including the "string syntax" introduced in Chapter 3 and the "s-expression syntax" presented in [16].

Expressions are formed by applying *constructors* to *variables*, *constants*, and compound expressions. The constructors are given in Table 7.1; they serve as "logical constants" that are available in every language. Variables and constants belong to specific languages.

In the mathematical syntax, the symbols for the constructors, the common punctuation symbols, and the symbol **with** are reserved. By a *name*, we mean any unreserved symbol.

7.2.1 Sorts

The sorts of $\mathcal{L}$ are generated from a finite set $\mathcal{A}$ of names called *atomic sorts*. More precisely, a *sort* of $\mathcal{L}$ is a sequence of symbols defined inductively by:

(1) Each $\alpha \in \mathcal{A}$ is a sort of $\mathcal{L}$.

(2) If $\alpha_1, \ldots, \alpha_{n+1}$ are sorts of $\mathcal{L}$ with $n \geq 1$, then $[\alpha_1, \ldots, \alpha_{n+1}]$ is a sort of $\mathcal{L}$.

Sorts of the form $[\alpha_1, \ldots, \alpha_{n+1}]$ are called *compound* sorts. In the next section we shall show that a compound sort denotes a domain of functions. Let $\mathcal{S}$ denote the set of sorts of $\mathcal{L}$.

$\mathcal{L}$ assigns each $\alpha \in \mathcal{A}$ a member of $\mathcal{S}$ called the *enclosing sort* of α. The atomic/enclosing sort relation determines a partial order $\preceq$ on $\mathcal{S}$ with the following properties:

(1) If $\alpha \in \mathcal{A}$ and β is the enclosing sort of α, then $\alpha \preceq \beta$.

[2] In [16], LUTINS is called **PF**.

Constructor	String syntax	Mathematical syntax
the-true	`truth`	T
the-false	`falsehood`	F
not	`not(p)`	$\neg\varphi$
and	`p1 and ... and pn`	$\varphi_1 \wedge \ldots \wedge \varphi_n$
or	`p1 or ... or pn`	$\varphi_1 \vee \ldots \vee \varphi_n$
implies	`p1 implies p2`	$\varphi \supset \psi$
iff	`p1 iff p2`	$\varphi \equiv \psi$
if-form	`if_form(p1,p2,p3)`	if-form$(\varphi_1, \varphi_2, \varphi_3)$
forall	`forall(v1:s1,...,vn:sn,p)`	$\forall v_1{:}\alpha_1, \ldots, v_n{:}\alpha_n, \varphi$
forsome	`forsome(v1:s1,...,vn:sn,p)`	$\exists v_1{:}\alpha_1, \ldots, v_n{:}\alpha_n, \varphi$
equals	`t1=t2`	$t_1 = t_2$
apply	`f(t1,...,tn)`	$f(t_1, \ldots, t_n)$
lambda	`lambda(v1:s1,...,vn:sn,t)`	$\lambda v_1{:}\alpha_1, \ldots, v_n{:}\alpha_n, t$
iota	`iota(v1:s1,p)`	$\iota v{:}\alpha, \varphi$
iota-p	`iota_p(v1:s1,p)`	$\iota_{\mathrm{p}} v{:}\alpha, \varphi$
if	`if(p,t1,t2)`	if(φ, t_1, t_2)
is-defined	`#(t)`	$t{\downarrow}$
defined-in	`#(t,s)`	$t \downarrow \alpha$
undefined	`?s`	$\perp_\alpha$

Table 7.1: Table of Constructors

(2) $\alpha_1 \preceq \beta_1,\ \ldots,\ \alpha_n \preceq \beta_n,\ \alpha_{n+1} \preceq \beta_{n+1}$ iff
$[\alpha_1, \ldots, \alpha_n, \alpha_{n+1}] \preceq [\beta_1, \ldots, \beta_n, \beta_{n+1}]$.

(3) If $\alpha, \beta \in \mathcal{S}$ with $\alpha \preceq \beta$ and α is compound, then β is compound and has the same length as α.

(4) For every $\alpha \in \mathcal{S}$, there is a unique $\beta \in \mathcal{S}$ such that $\alpha \preceq \beta$ and β is maximal with respect to $\preceq$.

$\preceq$ is intended to denote set inclusion.

The members of $\mathcal{S}$ which are maximal with respect to $\preceq$ are called *types*. A *base type* is a type which is atomic. Every language has the base type $*$ which denotes the set $\{\mathrm{T}, \mathrm{F}\}$ of standard truth values. The *type* of a sort α, written $\tau(\alpha)$, is the unique type β such that $\alpha \preceq \beta$. The *least upper bound* of α and β, written $\alpha \sqcup \beta$, is the least upper bound of α and β in the partial order $\preceq$. (The least upper bound of two sorts with the same type is always defined).

A *function sort* is a sort α such that $\tau(\alpha)$ has the form $[\alpha_1, \ldots, \alpha_n, \alpha_{n+1}]$. The *range sort* of a function sort α, written $\mathrm{ran}(\alpha)$, is defined as follows: if $\alpha = [\alpha_1, \ldots, \alpha_n, \alpha_{n+1}]$, then $\mathrm{ran}(\alpha) = \alpha_{n+1}$; otherwise, $\mathrm{ran}(\alpha) = \mathrm{ran}(\beta)$) where β is the enclosing sort of α.

Sorts are divided into two kinds as follows: A sort is of *kind* $*$ if either it is $*$ or it is a compound sort $[\alpha_1, \ldots, \alpha_{n+1}]$ where α_{n+1} is of kind $*$; otherwise it is of *kind* ι. We shall see later the purpose behind this division. All atomic sorts except $*$ are of kind ι.

7.2.2 Expressions

$\mathcal{L}$ contains a set $\mathcal{C}$ of names called *constants*. Each constant is assigned a sort in $\mathcal{S}$. An *expression of* $\mathcal{L}$ *of sort* α is a sequence of symbols defined inductively by:

(1) If x is a name and $\alpha \in \mathcal{S}$, then $x{:}\alpha$ is an expression of sort α.

(2) If $A \in \mathcal{C}$ is of sort α, then A is an expression of sort α.

(3) If $A, B, C, A_1, \ldots, A_n$ are expressions of sort $*$ with $n \geq 0$, then T, F, $\neg(A)$, $(A \supset B)$, $(A \equiv B)$, if-form(A, B, C), $(A_1 \wedge \cdots \wedge A_n)$, and $(A_1 \vee \cdots \vee A_n)$ are expressions of sort $*$

(4) If $x_1, \ldots, x_n$ are distinct names; $\alpha_1, \ldots, \alpha_n \in \mathcal{S}$; A is an expression of sort β; and $n \geq 1$, then $(\forall\ x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n,\ A)$ and
$(\exists\ x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n,\ A)$ are expressions of sort $*$.

(5) If A and B are expressions of sort α and β, respectively, with $\tau(\alpha) = \tau(\beta)$, then $(A = B)$ is an expression of sort $*$.

(6) If F is an expression of sort α and type $[\alpha_1, \ldots, \alpha_n, \alpha_{n+1}]$; $A_1, \ldots, A_n$ are expressions of sort $\alpha'_1, \ldots, \alpha'_n$; and $\alpha_1 = \tau(\alpha'_1),\ \ldots,\ \alpha_n = \tau(\alpha'_n)$, then $F(A_1, \ldots, A_n)$ is an expression of sort $\mathrm{ran}(\alpha)$.

(7) If $x_1, \ldots, x_n$ are distinct names; $\alpha_1, \ldots, \alpha_n \in \mathcal{S}$; A is an expression of sort $*$; and $n \geq 1$, then $(\lambda\ x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n,\ A)$ is an expression of sort $[\alpha_1, \ldots, \alpha_n, \beta]$.

(8) If x and y are names; α and β are sorts are of kind ι and $*$, respectively; and A is an expression of sort $*$, then $(\iota\ x{:}\alpha, A)$ and $(\iota_{\mathrm{p}}\ y{:}\beta, A)$ are expressions of sort α and β, respectively.

(9) If A, B, C are expressions of sort $*, \beta, \gamma$, respectively, with $\tau(\beta) = \tau(\gamma)$, then $\mathrm{if}(A, B, C)$ is an expression of sort $\beta \sqcup \gamma$.

(10) If A is an expression and $\alpha \in \mathcal{S}$, then $(A{\downarrow})$ and $(A \downarrow \alpha)$ are expressions of sort $*$.

(11) If α is a sort of kind ι, then $\bot_\alpha$ is an expression of sort α.

A *variable* is an expression of the form $x{:}\alpha$. A *formula* is an expression of sort $*$. A *sentence* is a closed formula. A *predicate* is an expression with a sort of the form $[\alpha_1, \ldots, \alpha_n, *]$.

An expression is said to be of *type* α if the type of its sort is α, and it is said to be of *kind* ι [respectively, $*$] if its type is of kind ι [respectively, $*$]. Notice that the well-formedness of an expression depends only on the types, and not directly on the *sorts*, of its components. Expressions of kind ι are used to refer to mathematical objects; they may be nondenoting. Expressions of kind $*$ are primarily used in making assertions about mathematical objects; they always have a denotation.

7.2.3 Alternate Notation

Expressions will usually be written in an abbreviated form as follows. Suppose A is an expression whose free variables are $x_1{:}\alpha_1,\ \ldots,\ x_n{:}\alpha_n$. Then A is abbreviated as

$$(\text{with } x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n,\ A'),$$

where A' is the result of replacing each (free or bound) occurrence of a variable $x_i{:}\alpha_i$ in A with x_i. This mode of abbreviation preserves meaning as

long as there are not two variables $x{:}\alpha$ and $x{:}\beta$ with $\alpha \neq \beta$ which are either both free in A or both in the scope of the same binder in A.

Parentheses in expressions may be suppressed when meaning is not lost. In addition, the following alternate notation may be used in the IMPS mathematical syntax:

(1) **prop** and **ind** in place of the base sorts $*$ and ι, respectively.

(2) $\alpha_1 \times \cdots \times \alpha_n \to \beta$ in place of a compound sort $[\alpha_1, \ldots, \alpha_n, \beta]$, and $\alpha_1 \times \cdots \times \alpha_n \rightharpoonup \beta$ in place of a compound sort $[\alpha_1, \ldots, \alpha_n, \beta]$ with β of kind ι.

(3) The symbol "." in place of "," at the end of variable-sort sequence; e.g., $\exists x{:}\alpha.x = x$ instead of $\exists x{:}\alpha, x = x$.

(4) $\Leftrightarrow$ in place of the biconditional $\equiv$.

(5) $\exists! x{:}\alpha, \varphi(x)$ in place of $\exists x{:}\alpha, (\varphi(x) \wedge \forall y{:}\alpha, (\varphi(y) \supset x = y))$.

(6) $A \neq B$ in place of $\neg(A = B)$.

7.3 Semantics

The semantics for a LUTINS language is defined in terms of models that are similar to models of many-sorted first-order logic. Let $\mathcal{L}$ be a LUTINS language, and let $\mathcal{S}$ denote the set of sorts of $\mathcal{L}$.

A *frame* for $\mathcal{S}$ is a set $\{\mathcal{D}_\alpha : \alpha \in \mathcal{S}\}$ of nonempty domains (sets) such that:

(1) $\mathcal{D}_* = \{\mathrm{T}, \mathrm{F}\}$ $(\mathrm{T} \neq \mathrm{F})$.

(2) If $\alpha \preceq \beta$, then $\mathcal{D}_\alpha \subseteq \mathcal{D}_\beta$.

(3) If $\alpha = [\alpha_1, \ldots, \alpha_n, \alpha_{n+1}]$ is of kind ι, then $\mathcal{D}_\alpha$ is the set of all partial functions $f : \mathcal{D}_{\alpha_1} \times \cdots \times \mathcal{D}_{\alpha_n} \to \mathcal{D}_{\alpha_{n+1}}$.

(4) If $\alpha = [\alpha_1, \ldots, \alpha_n, \alpha_{n+1}]$ is of kind $*$, then $\mathcal{D}_\alpha$ is the set of all total functions $f : \mathcal{D}_{\tau(\alpha_1)} \times \cdots \times \mathcal{D}_{\tau(\alpha_n)} \to \mathcal{D}_{\alpha_{n+1}}$ such that, for all $\langle b_1, \ldots, b_n \rangle \in \mathcal{D}_{\tau(\alpha_1)} \times \cdots \times \mathcal{D}_{\tau(\alpha_n)}$, $f(b_1, \ldots, b_n) = \mathrm{F}_{\tau(\alpha_{n+1})}$ whenever $b_i \notin \mathcal{D}_{\alpha_i}$ for at least one i with $1 \leq i \leq n$.

For a type $\alpha \in \mathcal{S}$ of kind $*$, F_α is defined inductively by:

(1) $\mathrm{F}_* = \mathrm{F}$.

(2) If $\alpha = [\alpha_1, \ldots, \alpha_{n+1}]$, then F_α is the function which maps every n-tuple $\langle a_1, \ldots, a_n \rangle \in \mathcal{D}_{\alpha_1} \times \cdots \times \mathcal{D}_{\alpha_n}$ to $\mathrm{F}_{\alpha_{n+1}}$.

A *model* for $\mathcal{L}$ is a pair $(\{\mathcal{D}_\alpha : \alpha \in \mathcal{S}\}, I)$ where $\{\mathcal{D}_\alpha : \alpha \in \mathcal{S}\}$ is a frame for $\mathcal{S}$ and I is a function which maps each constant of $\mathcal{L}$ of sort α to an element of $\mathcal{D}_\alpha$.

Let $\mathcal{M} = (\{\mathcal{D}_\alpha : \alpha \in \mathcal{S}\}, I)$ be a model for $\mathcal{L}$. A *variable assignment* into $\mathcal{M}$ is a function which maps each variable $x{:}\alpha$ of $\mathcal{L}$ to an element of $\mathcal{D}_\alpha$. Given a variable assignment φ into $\mathcal{M}$; distinct variables $x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n$ $(n \geq 1)$; and $a_1 \in \mathcal{D}_{\alpha_1}, \ldots a_n \in \mathcal{D}_{\alpha_n}$, let

$$\varphi[x_1{:}\alpha_1 \mapsto a_1, \ldots, x_n{:}\alpha_n \mapsto a_n]$$

be the variable assignment ψ such that

$$\psi(A) = \begin{cases} a_i & \text{if } A = x_i{:}\alpha_i \text{ for some } i \text{ with } 1 \leq i \leq n \\ \varphi(A) & \text{otherwise} \end{cases}$$

$V = V^{\mathcal{M}}$ is the partial binary function on variable assignments into $\mathcal{M}$ and expressions of $\mathcal{L}$ defined by the following statements:

(1) $V_\varphi(x{:}\alpha) = \varphi(x{:}\alpha)$

(2) If A is a constant, $V_\varphi(A) = I(A)$.

(3) $V_\varphi(\mathsf{T}) = \mathrm{T}$.

(4) $V_\varphi(\mathsf{F}) = \mathrm{F}$.

(5) $V_\varphi(\neg(A)) = \begin{cases} \mathrm{T} & \text{if } V_\varphi(A) = \mathrm{F} \\ \mathrm{F} & \text{otherwise} \end{cases}$

(6) $V_\varphi(A \supset B) = \begin{cases} \mathrm{T} & \text{if } V_\varphi(A) = \mathrm{F} \text{ or } V_\varphi(B) = \mathrm{T} \\ \mathrm{F} & \text{otherwise} \end{cases}$

(7) $V_\varphi(A \equiv B) = \begin{cases} \mathrm{T} & \text{if } V_\varphi(A) = V_\varphi(B) \\ \mathrm{F} & \text{otherwise} \end{cases}$

(8) If $\square$ is if-form or if, $V_\varphi(\square(A, B, C)) = \begin{cases} V_\varphi(B) & \text{if } V_\varphi(A) = \mathrm{T} \\ V_\varphi(C) & \text{otherwise} \end{cases}$

(9) $V_\varphi(A_1 \wedge \cdots \wedge A_n) = \begin{cases} \mathrm{T} & \text{if } V_\varphi(A_1) = \cdots = V_\varphi(A_n) = \mathrm{T} \\ \mathrm{F} & \text{otherwise} \end{cases}$

(10) $V_\varphi(A_1 \vee \cdots \vee A_n) = \begin{cases} \mathrm{T} & \text{if } V_\varphi(A_i) = \mathrm{T} \text{ for some } i \text{ with } 1 \leq i \leq n \\ \mathrm{F} & \text{otherwise} \end{cases}$

(11) $V_\varphi(\forall\, x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n,\ A) = \begin{cases} \mathrm{T} & \text{if } V_{\varphi[x_1:\alpha_1 \mapsto a_1, \ldots, x_n:\alpha_n \mapsto a_n]}(A) = \mathrm{T} \\ & \text{for all } \langle a_1, \ldots, a_n \rangle \in \\ & \mathcal{D}_{\alpha_1} \times \cdots \times \mathcal{D}_{\alpha_n} \\ \mathrm{F} & \text{otherwise} \end{cases}$

(12) $V_\varphi(\exists\, x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n,\ A) = \begin{cases} \mathrm{T} & \text{if } V_{\varphi[x_1:\alpha_1 \mapsto a_1, \ldots, x_n:\alpha_n \mapsto a_n]}(A) = \mathrm{T} \\ & \text{for some } \langle a_1, \ldots, a_n \rangle \in \\ & \mathcal{D}_{\alpha_1} \times \cdots \times \mathcal{D}_{\alpha_n} \\ \mathrm{F} & \text{otherwise} \end{cases}$

(13) $V_\varphi(A = B) = \begin{cases} \mathrm{T} & \text{if } V_\varphi(A) = V_\varphi(B) \\ \mathrm{F} & \text{otherwise} \end{cases}$

(14) Assume F is of kind ι. Then

$$V_\varphi(F(A_1, \ldots, A_n)) = V_\varphi(F)(V_\varphi(A_1), \ldots, V_\varphi(A_n))$$

if $V_\varphi(F), V_\varphi(A_1), \ldots, V_\varphi(A_n)$ are each defined; otherwise $V_\varphi(F(A_1, \ldots, A_n))$ is undefined.

(15) Assume F is of sort $[\alpha_1, \ldots, \alpha_{n+1}]$ of kind $*$. Then

$$V_\varphi(F(A_1, \ldots, A_n)) = V_\varphi(F)(V_\varphi(A_1), \ldots, V_\varphi(A_n))$$

if $V_\varphi(A_1), \ldots, V_\varphi(A_n)$ are each defined; otherwise $V_\varphi(F(A_1, \ldots, A_n)) = \mathrm{F}_{\tau(\alpha_{n+1})}$.

(16) Assume A is of sort α_{n+1} of kind ι. Then $V_\varphi(\lambda\, x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n,\ A)$ is the partial function $f : \mathcal{D}_{\alpha_1} \times \cdots \times \mathcal{D}_{\alpha_n} \to \mathcal{D}_{\alpha_{n+1}}$ such that

$$f(a_1, \ldots, a_n) = V_{\varphi[x_1:\alpha_1 \mapsto a_1, \ldots, x_n:\alpha_n \mapsto a_n]}(A)$$

if $V_{\varphi[x_1:\alpha_1 \mapsto a_1, \ldots, x_n:\alpha_n \mapsto a_n]}(A)$ is defined; otherwise $f(a_1, \ldots, a_n)$ is undefined.

(17) Assume A is of sort α_{n+1} of kind $*$. Then $V_\varphi(\lambda\ x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n,\ A)$ is the total function $f : \mathcal{D}_{\tau(\alpha_1)} \times \cdots \times \mathcal{D}_{\tau(\alpha_n)} \to \mathcal{D}_{\alpha_{n+1}}$ such that

$$f(a_1, \ldots, a_n) = V_{\varphi[x_1:\alpha_1 \mapsto a_1, \ldots, x_n:\alpha_n \mapsto a_n]}(A)$$

if $\langle a_1, \ldots, a_n \rangle \in \mathcal{D}_{\alpha_1} \times \cdots \times \mathcal{D}_{\alpha_n}$; otherwise $f(a_1, \ldots, a_n) = \mathrm{F}_{\tau(\alpha_{n+1})}$.

(18) $V_\varphi(\iota x{:}\alpha, A)$ is the unique element $a \in \mathcal{D}_\alpha$ such that $V_{\varphi[x:\alpha \mapsto a]}(A) = \mathrm{T}$; otherwise $V_\varphi(\iota x{:}\alpha, A)$ is undefined.

(19) $V_\varphi(\iota_\mathrm{p} x{:}\alpha, A)$ is the unique element $a \in \mathcal{D}_\alpha$ such that $V_{\varphi[x:\alpha \mapsto a]}(A) = \mathrm{T}$; otherwise $V_\varphi(\iota_\mathrm{p} x{:}\alpha, A) = \mathrm{F}_{\tau(\alpha)}$.

(20) $V_\varphi(A{\downarrow}) = \begin{cases} \mathrm{T} & \text{if } V_\varphi(A) \text{ is defined} \\ \mathrm{F} & \text{otherwise} \end{cases}$

(21) $V_\varphi(A \downarrow \alpha) = \begin{cases} \mathrm{T} & \text{if } V_\varphi(A) \text{ is defined and } V_\varphi(A) \in \mathcal{D}_\alpha \\ \mathrm{F} & \text{otherwise} \end{cases}$

(22) $V_\varphi(\bot_\alpha)$ is undefined.

Clearly, for each variable assignment φ into $\mathcal{M}$ and each expression A of $\mathcal{L}$,

(1) $V_\varphi(A) \in \mathcal{D}_\alpha$ if $V_\varphi(A)$ is defined, and

(2) $V_\varphi(A)$ is defined if A is of kind $*$.

$V_\varphi(A)$ is called the *value* or *denotation* of A in $\mathcal{M}$ with respect to φ, provided it is defined. When $V_\varphi(A)$ is not defined, A is said to be *nondenoting* in $\mathcal{M}$ with respect to φ.

7.4 Extensions of the Logic

The logic can be effectively extended by introducing "quasi-constructors." They are the subject of Chapter 10.

7.5 Hints and Cautions

(1) In most cases, the novice IMPS user is better off relying on his intuition instead of trying to fully comprehend the formal semantics of LUTINS.

(2) Although IMPS can handle expressions containing two variables with the same name (such as, $x{:}\alpha$ and $x{:}\beta$ with $\alpha \neq \beta$), they tend to be confusing and should be avoided if possible.

(3) LUTINS is a logic of two worlds: the ι world and the $*$ world. Expressions of kind ι are part of the ι world, and expressions of kind $*$ are part of the $*$ world. Although both worlds can be used for encoding mathematics, the ι world is much preferred for this purpose. As a general rule, expressions of kind $*$ should not be used to denote mathematical objects. Normally, formulas or predicates (i.e., expressions having a sort of the form $[\alpha_1, \ldots, \alpha_n, *]$) are the only useful expressions of kind $*$. Consequently, there is little support in IMPS for the constructor ι_{p}, the definite description operator for the $*$ world.

(4) The character `%` is often used in the names of variables and constants in the string syntax to denote a space (the character `_` is used to denote the start of a subscript). This character is printed in the mathematics syntax as either `%` or `_`.

Chapter 8

Theories

8.1 Introduction

As a mathematical object, an IMPS *theory* consists of a LUTINS language $\mathcal{L}$ and a set of sentences in $\mathcal{L}$ called *axioms*. The theory is the basic unit of mathematical knowledge in IMPS. In fact, IMPS can be described as a system for developing, exploring, and relating theories. How a theory is developed is the subject of several chapters; an overview is given in the next section. The principal technique for exploring a theory is to discover the logical implications of its axioms by stating and trying to prove conjectures; the IMPS proof system is described in Chapter 12. Two theories are related by creating an interpretation of one in the other; theory interpretations are discussed in Chapter 9.

We define a *theorem* of an IMPS theory $\mathcal{T}$ to be a formula of $\mathcal{T}$ which is valid in each model for $\mathcal{T}$. That is, a theorem of $\mathcal{T}$ is a semantic consequence of $\mathcal{T}$. The reader should note that our definition of a theorem does not depend on the IMPS proof system or any other proof system for LUTINS.

In the implementation, a theory is represented by a data structure which encodes the language and axioms of the theory. The data structure also encodes in procedural or tabular form certain consequences of the theory's axioms. This additional information facilitates various kinds of low-level reasoning within theories that are encapsulated in the IMPS expression simplifier, the subject of Chapter 13.

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be IMPS theories. $\mathcal{T}_1$ is *subtheory* of $\mathcal{T}_2$ (and $\mathcal{T}_1$ is a *supertheory* of $\mathcal{T}_2$) if the language of $\mathcal{T}_1$ is a sublanguage of the language of $\mathcal{T}_2$ and each axiom of $\mathcal{T}_1$ is a theorem of $\mathcal{T}_2$. Each IMPS theory $\mathcal{T}$ is

assigned a set of *component theories*, each of which is a subtheory of $\mathcal{T}$. $\mathcal{T}_1$ is *structural subtheory* of $\mathcal{T}_2$ (and $\mathcal{T}_1$ is a *structural supertheory* of $\mathcal{T}_2$) if $\mathcal{T}_1$ is either a component theory of $\mathcal{T}_2$ or a structural subtheory of a component theory of $\mathcal{T}_2$.

8.2 How to Develop a Theory

The user creates a theory and the IMPS objects associated with it by evaluating expressions called *definition forms* (or *def-forms* for short). There are approximately 30 kinds of def-forms, each described in detail in Chapter 17. The user interface provides templates to you for writing def-forms. The def-forms supplied by the system and created by you are stored in files which can be loaded as needed into a running IMPS process. In this section, we give an overview of the tasks that are involved in creating a well-developed theory. Along the way, we list the def-forms that are relevant to each task.

Task 1: The Primitive Theory

The first task in developing a theory is to build a primitive, bare bones theory $\mathcal{T}$ from a (possibly empty) set of theories, a language, and a set of axioms. This is done with the **def-theory** form; $\mathcal{T}$ is the smallest theory which contains the theories, the language, and the set of axioms. The theories are made the component theories of $\mathcal{T}$. A language is created with **def-language** or **def-sublanguage**. A primitive theory can also be created by instantiating an existing theory with **def-theory-instance**.

Task 2: Theorems

The information held in the axioms of the primitive theory is unlocked by proving theorems using the IMPS proof system (see Chapter 12). Theorem proving is crucial to developing a theory and is involved in most of the tasks below. Proven theorems are installed in the theory with **def-theorem** using various "usages." See Section 8.4 for details.

Task 3: Simplification

Once the primitive theory is built, usually the next task is to specify how simplification should be performed in the theory. There are three mechanisms for doing this that can be used separately or in combination:

(1) *Install a processor.* For theories of an algebraic nature (e.g., rings, fields, or modules over a ring), you may want to install an algebraic processor to perform rudimentary algebraic simplification. Processors are created with **def-algebraic-processor** and **def-order-processor**, and installed with **def-theory-processors**. The system will check that the theory contains the right theorems before anything is installed, so even if you build a weird processor (one that wants to treat ring multiplication as ring addition) it cannot be installed.

(2) *Install rewrite rules.* An ordinary rewrite rule is created and installed in the theory by installing a theorem with the usage **rewrite**. A transportable rewrite rule is created by installing a theorem with the usage **transportable-rewrite**, and is installed in the theory with **def-imported-rewrite-rules**.

(3) *Add information to the theory's domain-range handler.* Information about the domain and range of function constants can be added to a theory by installing theorems in the theory with the usages **d-r-convergence** and **d-r-value**.

See Chapter 13 for more details.

Task 4: Theory Interpretations

It is a good idea, early in the development of a theory, to create the principal theory interpretations involving the theory in either the role of source or target. Symmetry interpretations of theory in itself are often quite useful if they exist. Theory interpretations are created with **def-translation**. See Chapter 9 for details.

Task 5: Definitions

Atomic sorts and constants can be defined throughout the course of developing a theory. They are usually defined with **def-atomic-sort**, **def-constant**, and **def-recursive-constant**, but they can also be created by transporting definitions to the theory from some other theory with **def-transported-symbols**. An atomic sort with constants for representing a cartesian product with a constructor and accessors is created with **def-cartesian-product**. See Section 8.5 for details.

Task 6: Macetes

Theorems are applied semi-automatically in the course of a proof by means of procedures called *macetes*. A set of carefully crafted macetes is an essential part of a well-developed theory. An elementary macete is created whenever a theorem is installed, and a transportable macete is created by installing a theorem with the usage **transportable-macete**. Other kinds of macetes are created with **def-compound-macete** and **def-schematic-macete**. See Chapter 14 for details.

Task 7: Induction Principles

If your theory has a theorem which you think can be treated like an induction principle, then you can build an inductor with **def-inductor**. The name of the inductor can be used as an argument to the **induction** command. Note that most often you can get by with using **integer-inductor** which is already supplied with the system.

Task 8: Theory Ensembles

Sometimes copies and unions of copies of the theory are needed. These can be created and managed as an IMPS *theory ensemble* with **def-theory-ensemble**, **def-theory-ensemble-instances**, **def-theory-ensemble-overloadings**, and **def-theory-ensemble-multiple**. See Chapter 11 for details.

Task 9: Theory Extensions

Ordinarily an extension of the theory is created with **def-theory** and **def-theory-instance**, but the theory can also be extended by adding an abstract data type with **def-bnf** or a record with **def-record-theory**.

Task 10: Quasi-constructors

Quasi-constructors are global constants that can be used across a large class of theories. They are created with **def-quasi-constructor**. See Chapter 10 for details.

Task 11: Renamers

Renaming functions are used with **def-theory-instance** and **def-transported-symbols** to rename atomic sorts and constants. They are created with **def-renamer**.

Task 12: Syntax

Special parse and print syntax for the vocabulary of the theory is created independently from the development of the theory with **def-parse-syntax**, **def-print-syntax**, and **def-overloading**. See Chapter 16 for details.

Task 13: Sections

Theory library sections containing the theory or a part of the theory are created with **def-section**. See Section 8.6 for more information about sections.

8.3 Languages

An IMPS language is a representation of a LUTINS language. An IMPS language is built with **def-language** from a (possibly empty) set of languages, a set of base types, and a list of atomic sort declarations.

8.4 Theorems

Mathematically, a *theorem* of a theory is any logical consequence of the theory's axioms. One usually verifies that a formula A is a theorem of a theory $\mathcal{T}$ by constructing a proof of A in $\mathcal{T}$ using the IMPS proof system (see Chapter 12, particularly Section 12.3). A theorem is available to the user when working in a theory $\mathcal{T}$ only if it has been "installed" in $\mathcal{T}$. There are several ways a theorem can be installed:

- When a theory is built, all of its axioms are installed in it.
- One or two theorems are installed in a theory when a **def-theorem** form is evaluated. See Section 8.4 for details.

- When an atomic sort is defined in a theory, the new sort's defining axiom and a few special consequences of it are installed in the theory. Something similar takes place when constants are defined (either directly or recursively) in a theory.

- When a theorem A is transported from $\mathcal{T}_1$ to $\mathcal{T}_2$ via an interpretation Φ, the translation of A, $\Phi(A)$, is installed in $\mathcal{T}_2$.

When a theorem is installed in a theory $\mathcal{T}$, it is automatically installed in every structural supertheory of $\mathcal{T}$.

A theorem is installed with a list of *usages* that tells IMPS how to use the theorem. There are seven usages; they have the following effect when they are in the usage list of a theorem installed in a theory $\mathcal{T}$:

(1) **elementary-macete**. This causes an elementary macete to be created from the theorem. For a theorem which is given a name, this usage is always added by IMPS to the theorem's usage list, whether this usage is explicitly included by the user or not.

(2) **transportable-macete**. This causes a transportable macete to be created from the theorem.

(3) **rewrite**. This causes a rewrite rule to be created and installed in the transform table of $\mathcal{T}$. Although the theorem can have any form, usually this usage is used only with a theorem which is an equation, a quasi-equation, or a biconditional.

(4) **transportable-rewrite**. This causes a transportable rewrite rule to be created (but not installed). Transportable rewrite rules are installed (in $\mathcal{T}$ or other theories) with **def-imported-rewrite-rules**.

(5) **simplify-logically-first**. If **rewrite** and **transportable-rewrite** are in the usage list, this marks the generated rewrite and transportable rewrite rules so that logical simplification is used just before they are applied.

(6) **d-r-convergence**. If the theorem has an appropriate form, a convergence condition is created and installed in the domain-range handler of $\mathcal{T}$.

(7) **d-r-value**. If the theorem has an appropriate form, a value condition is created and installed in the domain-range handler of $\mathcal{T}$.

8.5 Definitions

IMPS supports four kinds of definitions: atomic sort definitions, constant definitions, recursive function definitions, and recursive predicate definitions. In the following let $\mathcal{T}$ be an arbitrary theory.

Atomic sort definitions are used to define new atomic sorts from nonempty unary predicates. They are created with the **def-atomic-sort** form. An *atomic sort definition* for $\mathcal{T}$ is a pair $\delta = (n, U)$ where n is a symbol intended to be the name of a new atomic sort of $\mathcal{T}$ and U is a nonempty unary predicate in $\mathcal{T}$ intended to specify the extension of the new sort. δ can be installed in $\mathcal{T}$ only if the formula $\exists x.U(x)$ is known to be a theorem of $\mathcal{T}$. As an example, the pair

$$(\mathbf{N},\ \lambda x : \mathbf{Z}\ .\ 0 \leq x)$$

defines $\mathbf{N}$ to be an atomic sort which denotes the natural numbers.

Constant definitions are used to define new constants from defined expressions. They are created with the **def-constant** form. A *constant definition* for $\mathcal{T}$ is a pair $\delta = (n, e)$ where n is a symbol intended to be the name of a new constant of $\mathcal{T}$ and e is a expression in $\mathcal{T}$ intended to specify the value of the new constant. δ can be installed in $\mathcal{T}$ only if the formula $e{\downarrow}$ is verified to be a theorem of $\mathcal{T}$. As an example, the pair

$$(\text{floor},\ \lambda x : \mathbf{R}\ .\ \iota z : \mathbf{Z}\ .\ z \leq x \wedge x < 1 + z)$$

defines the floor function on reals using the ι constructor.

Recursive function definitions are used to define one or more functions by simultaneous recursion. They are created with the **def-recursive-constant** form. The mechanism for recursive function definitions in IMPS is modeled on the approach to recursive definitions presented by Y. Moschovakis in [22]. A *recursive definition* for $\mathcal{T}$ is a pair $\delta = ([n_1, \ldots, n_k], [F_1, \ldots, F_k])$ where $k \geq 1$, $[n_1, \ldots, n_k]$ is a list of distinct symbols intended to be the names of k new constants, and $[F_1, \ldots, F_k]$ is a list of functionals (i.e., functions which map functions to functions) of kind ind in $\mathcal{T}$ intended to specify, as a system, the values of the new constants. δ can be installed in $\mathcal{T}$ only if the functionals $F_1, \ldots, F_k$ are verified to be monotone in $\mathcal{T}$ with respect to the subfunction order $\sqsubseteq$.[1] The names $[n_1, \ldots, n_k]$ then denote the simultaneous

[1] $f \sqsubseteq g$ iff $f(a_1, \ldots, a_m) = g(a_1, \ldots, a_m)$ for all m-tuples $\langle a_1, \ldots, a_m \rangle$ in the domain of f.

least fixed point of the functionals $F_1, \ldots, F_k$. As an example, the pair

$$(\text{factorial},\ \lambda f : \mathbf{Z} \rightharpoonup \mathbf{Z}\ .\ \lambda n : \mathbf{Z}\ .\ \text{if}(n = 0, 1, n * f(n-1)))$$

is a recursive definition of the factorial function in our standard theory of the real numbers.

This approach to recursive definitions is very natural in IMPS because expressions of kind **ind** are allowed to denote partial functions. Notice that there is no requirement that the functions defined by a recursive definition be total. In a logic in which functions must be total, a list of functionals can be a legitimate recursive definition only if it has a solution composed entirely of *total* functions. This is a difficult condition for a machine to check, especially when $k > 1$. Of course, in IMPS there is no need for a recursive definition to satisfy this condition since a recursive definition is legitimate as long as the defining functionals are monotone. IMPS has an automatic syntactic check sufficient for monotonicity that succeeds for many common recursive function definitions.

Recursive predicate definitions are used to define one or more predicates by simultaneous recursion. They are also created with the **def-recursive-constant** form. Recursive predicate definitions are implemented in essentially the same way as recursive function definitions using the order $\subseteq^2$ on predicates. The approach is based on the classic theory of positive inductive definitions (see [21]). For an example, consider the pair

$$([\text{even}, \text{odd}],\ [F_1, F_2]),$$

where:

- $F_1 = \lambda e, o : \mathbf{N} \to \text{prop}\ .\ \lambda n : \mathbf{N}\ .\ \text{if}(n = 0, \text{truth}, o(n-1))$.
- $F_2 = \lambda e, o : \mathbf{N} \to \text{prop}\ .\ \lambda n : \mathbf{N}\ .\ \text{if}(n = 0, \text{falsehood}, e(n-1))$.

It defines the predicates even and odd on the natural numbers by simultaneous recursion. As with recursive function definitions, there is an automatic syntactic check sufficient for monotonicity that succeeds for many recursive predicate definitions.

[2] $p \subseteq q$ iff $p(a_1, \ldots, a_m) \supset q(a_1, \ldots, a_m)$ for all m-tuples $\langle a_1, \ldots, a_m \rangle$ in the common domain of p and q.

8.6 Theory Libraries

A *theory library* is a collection of theories, theory constituents (definitions, theorems, proofs, etc.), and theory interpretations that serves as a database of mathematics. A theory library is composed of *sections*; each section is a particular body of knowledge that is stored in a set of files consisting of def-forms. A section can be loaded as needed into a running IMPS process. A section is built from a (possibly empty) set of subsections and files with **def-section**. The forms **load-section** and **include-files** will load a section and a list of files, respectively. These forms are useful for organizing sections.

Supplied with IMPS is an *initial theory library* which contains a variety of basic mathematics. In no sense is the initial theory library intended to be complete. We expect it to be extended by IMPS users. In the course of using IMPS, you should build your own theory library on top of the IMPS initial theory library. The initial theory library contains many examples that you can imitate. An outline of the initial theory library is given in Chapter 20.

8.7 Hints and Cautions

(1) The IMPS initial theory library contains no arithmetic theory weaker than **h-o-real-arithmetic**; e.g., there is no theory of Peano arithmetic. This is in accordance with our philosophy that there is rarely any benefit in building a theory with an impoverished arithmetic. Furthermore, since full-powered arithmetic is so useful and so basic, we advise the IMPS user to include **h-o-real-arithmetic** in every theory he or she builds.

(2) When building a theory, it is usually a good idea to minimize the primitive constants and axioms of the theory. This will make it easier to use the theory as a source of a theory interpretation.

(3) The best way to learn how to develop a theory is to study some of the theory-building techniques and styles which are demonstrated in the IMPS initial theory library.

(4) Since the sort of an expression gives immediate information about the value of the expression, it is often very advantageous to define new atomic sorts rather than work directly with unary predicates. Also, a nonnormal translation (see Section 9.2) can be "normalized," i.e.,

transformed into a normal translation, by defining new atomic sorts in the target theory of the translation.

Chapter 9

Theory Interpretations

9.1 Introduction

Theory Interpretations are translations which map the expressions from one theory to the expressions of another with the property that theorems are mapped to theorems. They serve as conduits to pass results from one theory to another. As such, they are an essential ingredient in the little theories version of the axiomatic method which IMPS supports (see Section 3.7). The IMPS notion of theory interpretation [5, 7] is modeled on the standard approach to theory interpretation in first-order logic (e.g., see [2, 19, 24]).

Most of the various ways that interpretations are used in IMPS are described in the penultimate section of this chapter.

9.2 Translations

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be LUTINS theories. A *translation from* $\mathcal{T}_1$ *to* $\mathcal{T}_2$ is a pair $\Phi = (\mu, \nu)$, where μ is a mapping from the atomic sorts of $\mathcal{T}_1$ to the sorts, unary predicates (quasi-sorts), and indicators (sets) of $\mathcal{T}_2$ and ν is a mapping from the constants of $\mathcal{T}_1$ to the expressions of $\mathcal{T}_2$, satisfying certain syntactic conditions. $\mathcal{T}_1$ and $\mathcal{T}_2$ are called the *source theory* and *target theory* of Φ, respectively. Given an expression e of $\mathcal{T}_1$, the *translation* of e via Φ, written $\Phi(e)$, is an expression of $\mathcal{T}_2$ defined from μ and ν in a manner that preserves expression structure. In particular, $\Phi(e)$ is a sentence when e is a sentence.

When μ maps an atomic sort α to a unary predicate U, the variable binding constructors—λ, $\forall$, $\exists$, and ι—are "relativized" when they bind a

variable of sort α. For example,

$$\Phi(\forall x{:}\alpha.\psi) = \forall x{:}\beta.U(x) \supset \Phi(\psi).$$

Something very similar happens when μ maps an atomic sort to an indicator. The constructor **defined-in** is also relativized when its second argument is α. A translation is *normal* if it maps each atomic sort of the source theory to a sort of the target theory. Normal translations do not relativize constructors.

Let Φ be a translation from $\mathcal{T}_1$ to $\mathcal{T}_2$. Φ is an *interpretation of* $\mathcal{T}_1$ *in* $\mathcal{T}_2$ if $\Phi(\varphi)$ is a theorem of $\mathcal{T}_2$ for each theorem φ of $\mathcal{T}_1$. In other words, an interpretation is a translation that maps theorems to theorems. An *obligation* of Φ is a formula $\Phi(\varphi)$ where φ is either:

(1) a primitive axiom of $\mathcal{T}_1$ (*axiom obligation*);

(2) a definition axiom of $\mathcal{T}_1$ (*definition obligation*);

(3) a formula asserting that a particular primitive atomic sort of $\mathcal{T}_1$ is nonempty (*sort nonemptiness obligation*);

(4) a formula asserting that a particular primitive constant of $\mathcal{T}_1$ is defined in its sort (*constant sort obligation*); or

(5) a formula asserting that a particular primitive atomic sort of $\mathcal{T}_1$ is a subset of its enclosing sort (*sort inclusion obligation*).

The following theorem gives a sufficient condition for a translation to be an interpretation:

Theorem 9.2.1 (Interpretation Theorem) *A translation Φ from $\mathcal{T}_1$ to $\mathcal{T}_2$ is an interpretation if each of its obligations is a theorem of T_2.*

See [5, 7], for a more detailed discussion of theory interpretations in LUTINS.

9.3 Building Theory Interpretations

The most direct way for you to build an interpretation is with the **def-translation** form. However, there are also several ways interpretations can be built automatically by IMPS with little or no assistance from you.

Building an interpretation with **def-translation** is generally a two-step process. The first step is to build a translation by evaluating a **def-translation** form which contains a specified name, source theory, target theory, sort association list, and constant association list. These latter lists, composed of pairs, specify respectively the two functions μ and ν discussed in the previous section. Each pair in the sort association list consists of an atomic sort of the source theory and a specification of a sort, unary predicate, or indicator of the target theory. Each pair in the constant association list consists of a constant of the source theory and a specification of an expression of the target theory. There does not need to be a pair in the sort association list for each atomic sort in the source theory; missing primitive atomic sorts are paired with themselves and missing defined atomic sorts are handled in the special way described in the next section. A similar statement is true about the constant association list.

When the **def-translation** form is evaluated, IMPS does a series of syntactic checks to make sure that the form correctly specifies a translation. If all the checks are successful, IMPS builds the translation (or simply retrieves it if it had been built previously).

The second step is to verify that the translation is an interpretation. IMPS first generates the obligations of the translation that are not trivially theorems of the target theory. Next IMPS removes from this set those obligations which are instances of installed theorems of the target theory. If the line

```
(theory-interpretation-check using-simplification)
```

is part of the **def-translation** form, IMPS will also remove obligations which are known to be theorems of the target theory by simplification. When IMPS is done working on the obligations, if there are none left the translation is marked as a theory interpretation. Otherwise, an error message is made that lists the outstanding obligations.

If there are outstanding obligations (and you believe that they are indeed theorems of the target theory), you will usually want to prove them in the target theory. Then, after installing them in the target theory, the **def-translation** form can be re-evaluated.

The following is a simple example of an interpretation built from a **def-translation** form:

```
(def-translation MONOID-THEORY-TO-ADDITIVE-RR
  (source monoid-theory)
```

```
(target h-o-real-arithmetic)
(fixed-theories h-o-real-arithmetic)
(sort-pairs
 (uu rr))
(constant-pairs
 (e 0)
 (** +)
(theory-interpretation-check using-simplification))
```

The purpose of the line

```
(fixed-theories h-o-real-arithmetic)
```

is to speed up the construction of the interpretation by telling IMPS ahead of time that the theory **h-o-real-arithmetic**, which is a subtheory of **monoid-theory**, is fixed by the translation, i.e., each expression of **h-o-real-arithmetic** is mapped to itself.

9.4 Translation of Defined Sorts and Constants

Let Φ be a translation from $\mathcal{T}_1$ to $\mathcal{T}_2$. Translations have been implemented in IMPS so that the defined sorts and constants of $\mathcal{T}_1$ are handled in an effective manner, even when they are defined after Φ is constructed. There are three possible ways that a defined sort or constant of $\mathcal{T}_1$ can be translated.

First, if the defined sort or constant is a member of one of the fixed theories of Φ, it is translated to itself. Second, if the defined sort or constant is mentioned (i.e., is the first component of a pair) in the sort or constant association list of Φ, it is translated to the second component of the pair. In practice, most defined sorts and constants are not mentioned in the sort and constant association lists. And, third, if the defined sort or constant is not mentioned in the sort and constant association lists and is not a member of a fixed theory, it is translated on the basis of its definiens (unary predicate, expression, or list of functionals). This last way of translation requires some explanation.

Suppose α is a defined sort of $\mathcal{T}_1$ which is not mentioned in the sort association list of Φ. Let U be the unary predicate that was used to define α. If there is a atomic sort β of $\mathcal{T}_2$ defined by a unary predicate U' such that U' and $\Phi(U)$ are alpha-equivalent, then α is translated as if the pair (α, β) were in the sort association list of Φ. Otherwise, it is translated as if the pair $(\alpha, \Phi(U))$ were in the sort association list of Φ.

Suppose a is a directly defined constant of $\mathcal{T}_1$ which is not mentioned in the constant association list of Φ. Let e be the expression that was used to define a. If there is a constant b of $\mathcal{T}_2$ directly defined by an expression e' such that e' and $\Phi(e)$ are alpha-equivalent, then a is translated to b. Otherwise, a is translated to $\Phi(e)$.

Suppose a_i is constant which is not mentioned in the constant association list of Φ, but which is a member of a list $a_1, \ldots, a_n$ of constants of $\mathcal{T}_1$ defined by recursion. Let $F_1, \ldots, F_n$ be the list of functionals that was used to define $a_1, \ldots, a_n$. If there is a list $b_1, \ldots, b_n$ of constants of $\mathcal{T}_2$ recursively defined by the list of functionals $F'_1, \ldots, F'_n$ such that F'_j and $\Phi(F_j)$ are alpha-equivalent for each j with $1 \leq j \leq n$, then a_i is translated to b_i. Otherwise, a_i is translated to an iota-expression which denotes the ith component of the minimal fixed point of $\Phi(F_1), \ldots, \Phi(F_n)$.

9.5 Reasoning and Formalization Techniques

This section briefly describes most of the ways interpretations are used in IMPS for formalizing mathematics and proving theorems.

9.5.1 Transporting Theorems

The most important use of theory interpretations is for transporting theorems from one theory to another. There are several ways that a theorem can be transported:

(1) The translation of a theorem of $\mathcal{T}_1$ via an interpretation of $\mathcal{T}_1$ in $\mathcal{T}_2$ can be installed in $\mathcal{T}_2$ using **def-theorem** with the modifier argument **translation**.

(2) In the course of a proof, the translation of a theorem can be added to the context of a sequent with the proof command **assume-transported-theorem**. Similarly, an instantiation of a translation of a theorem can be added with **instantiate-transported-theorem**. Both of these commands ask for the theory interpretation to be used. If no interpretation is given with the command **instantiate-transported-theorem**, IMPS will try to find or build an interpretation on its own using the information in the variable instances supplied by the user.

(3) When a transportable macete or transportable rewrite rule is applied, a theorem is effectively transported to the current theory (but it is not actually installed in the current theory).

Theorems are usually transported from an abstract theory to a more concrete theory.

9.5.2 Polymorphism

Constructors and quasi-constructors are polymorphic in the sense that they can be applied to expressions of several different types. This sort of polymorphism is not very useful unless we have results about constructors and quasi-constructors that could be used in proofs regardless of the actual types that are involved. For constructors, most of these "generic" results are coded in the form of the primitive inferences given in Chapter 19. Since quasi-constructors, unlike constructors, can be introduced by you, it is imperative that there is some way for you to prove generic results about quasi-constructors. This can be done by proving theorems about quasi-constructors in a theory of generic types, and then transporting these results as needed to theories where the quasi-constructor is used.

For example, consider the quasi-constructor **m-composition** defined by

```
(def-quasi-constructor M-COMPOSITION
  "lambda(f:[ind_2,ind_3],g:[ind_1,ind_2],
     lambda(x:ind_1, f(g(x))))"
  (language pure-generic-theory-3)
  (fixed-theories the-kernel-theory))
```

The basic properties about **m-composition**, such as associativity, can be proved in the theory **pure-generic-theory-4**, which has four base types but no constants, axioms, or other atomic sorts.

9.5.3 Symmetry and Duality

Theory interpretations can be used to formalize certain kinds of arguments involving symmetry and duality. For example, suppose we have proved a theorem in some theory and have noticed that some other conjecture follows from this theorem "by symmetry." This notion of symmetry can frequently be made precise by creating a theory interpretation from the theory to itself which translates the theorem to the conjecture.

As an illustration, consider the theory interpretation defined by

```
(def-translation MUL-REVERSE
  (source groups)
  (target groups)
  (fixed-theories h-o-real-arithmetic)
  (constant-pairs
   (mul "lambda(x,y:gg, y mul x)"))
  force-under-quick-load
  (theory-interpretation-check using-simplification))
```

This translation, which reverses the argument order of $*$ and holds everything else fixed in **groups**, maps the left cancellation law

```
(def-theorem LEFT-CANCELLATION-LAW
  "forall(x,y,z:gg, x mul y = x mul z iff y=z)"
  (theory groups)
  (usages transportable-macete)
  (proof
   (...)))
```

to the right cancellation law

```
(def-theorem RIGHT-CANCELLATION-LAW
  ;; "forall(x,y,z:gg, y mul x=z mul x iff y=z)"
  left-cancellation-law
  (theory groups)
  (usages transportable-macete)
  (translation mul-reverse)
  (proof existing-theorem))
```

Since this translation is in fact a theory interpretation, we need only prove the left cancellation law to show that both cancellation laws are theorems of **groups**.

9.5.4 Problem Transformation

Sometimes a problem is easier to solve if it is transformed into an equivalent, but more convenient form. For example, often geometry problems are easier to solve if they are transformed into algebra problems, and vice versa. Many problem transformations can be formalized as theory interpretations. The interpretation serves both as a means of transforming the problem and as a verification that the transformation is valid.

9.5.5 Definition Transportation

A list of definitions (atomic sort, directly defined constant or recursively defined constant) can be transported from T_1 to T_2 via an interpretation of $\mathcal{T}_1$ in $\mathcal{T}_2$ using **def-transported-symbols**.

For example, consider the following def-forms:

```
(def-translation ORDER-REVERSE
  (source partial-order)
  (target partial-order)
  (fixed-theories h-o-real-arithmetic)
  (constant-pairs
    (prec rev%prec)
    (rev%prec prec))
  (theory-interpretation-check using-simplification))

(def-renamer FIRST-RENAMER
  (pairs
   (prec%majorizes prec%minorizes)
   (prec%increasing rev%prec%increasing)
   (prec%sup prec%inf)))

(def-transported-symbols
  (prec%majorizes prec%increasing prec%sup)
  (translation order-reverse)
  (renamer first-renamer))
```

The **def-transported-symbols** form installs three new definitions—**prec%minorizes**, **rev%prec%increasing**, and **prec%inf**—in **partial-order**. These new definitions are created by translating the definiens of **prec%majorizes**, **prec%increasing**, and **prec%sup**, respectively, via **order-reverse**.

9.5.6 Theory Instantiation

As argued by R. Burstall and J. Goguen (e.g., in [14, 15]), a flexible notion of *parametric theory* can be obtained with the use of ordinary theories and theory interpretations. The key idea is that the primitives of a subtheory of a theory are a collection of parameters which can be instantiated as a group via a theory interpretation. In IMPS theories are instantiated using

def-theory-instance; each subtheory of a theory can serve as a parameter of the theory.

For example, consider the following def-forms:

```
(def-translation FIELDS-TO-RR
  (source fields)
  (target h-o-real-arithmetic)
  (fixed-theories h-o-real-arithmetic)
  (sort-pairs
   (kk rr))
  (constant-pairs
   (o_kk 0)
   (i_kk 1)
   (-_kk -)
   (+_kk +)
   (*_kk *)
   (inv "lambda(x:rr,1/x)"))
 (theory-interpretation-check using-simplification))

(def-theory-instance VECTOR-SPACES-OVER-RR
  (source vector-spaces)
  (target h-o-real-arithmetic)
  (translation fields-to-rr)
  (renamer vs-renamer))
```

The last def-form creates a theory of an abstract vector space over the field of real numbers by instantiating a theory of an abstract vector space over an abstract field. The parameter theory is **fields**, a subtheory of **vector-spaces** and the source theory of **fields-to-rr**.

For a detailed description of this technique, see [3, 6].

9.5.7 Theory Extension

Let $\mathcal{T}_i = (\mathcal{L}_i, \Gamma_i)$ be a LUTINS theory for $i = 1, 2$. $\mathcal{T}_2$ is an *extension* of $\mathcal{T}_1$ (and $\mathcal{T}_1$ is a *subtheory* of $\mathcal{T}_2$) if $\mathcal{L}_1$ is a sublanguage of $\mathcal{L}_2$ and $\Gamma_1 \subseteq \Gamma_2$. A very useful reasoning technique is to (1) add machinery to a theory by means of a theory extension and (2) create one or more interpretations from the theory extension to the original theory. This setup allows one to prove results in a an enriched theory and then transport them back to the unenriched theory.

For example, many theorems about groups are more conveniently proved about groups actions because several group theorems may be just different instantiations of a particular group action theorem. The following def-forms show how **group-actions** is an extension of **groups**:

```
(def-language GROUP-ACTION-LANGUAGE
  (embedded-language groups-language)
  (base-types uu)
  (constants
   (act "[uu,gg,uu]")))

(def-theory GROUP-ACTIONS
  (language group-action-language)
  (component-theories groups)
  (axioms
   (act-id
    "forall(alpha:uu,g:gg, act(alpha,e) = alpha)"
    rewrite transportable-macete)
   (act-associativity
    "forall(alpha:uu,g,h:gg,
       act(alpha,g mul h) = act(act(alpha,g),h))"
    transportable-macete)))
```

There are many natural interpretations of **group-actions** in **groups** such as:

```
(def-translation ACT->CONJUGATE
  (source group-actions)
  (target groups)
  (fixed-theories h-o-real-arithmetic)
  (sort-pairs
   (uu "gg"))
  (constant-pairs
   (act "lambda(g,h:gg, (inv(h) mul g) mul h)"))
  (theory-interpretation-check using-simplification))
```

Suppose $\mathcal{T}_2$ is an extension of $\mathcal{T}_1$ that is obtained by adding to $\mathcal{T}_1$ new constants and axioms relating the new constants of $\mathcal{T}_2$ to the old constants of $\mathcal{T}_1$. Then each theorem of $\mathcal{T}_2$ has a analogue in $\mathcal{T}_1$ obtained by generalizing over the new constants of $\mathcal{T}_1$. This notion of generalization is performed automatically in IMPS with an appropriate interpretation of $\mathcal{T}_2$ in $\mathcal{T}_1$.

For example, the def-forms

```
(def-language LCT-LANGUAGE
  (embedded-language groups)
  (constants
    (a "sets[gg]")
    (b "sets[gg]")))

(def-theory LCT-THEORY
  (language lct-language)
  (component-theories groups)
  (axioms
   (a-is-a-subgroup "subgroup(a)")
   (b-is-a-subgroup "subgroup(b)")))

(def-theorem LITTLE-COUNTING-THEOREM
  "f_indic_q{gg%subgroup}
     implies
   f_card{a set%mul b}*f_card{a inters b} =
   f_card{a}*f_card{b}"
  ;; "forall(b,a:sets[gg],
  ;;    subgroup(a) and subgroup(b)
  ;;     implies
  ;;    (f_indic_q{gg%subgroup}
  ;;     implies
  ;;    f_card{a set%mul b}*f_card{a inters b} =
  ;;    f_card{a}*f_card{b}))"
  (theory groups)
  (home-theory lct-theory)
  (usages transportable-macete)
  (proof (...)))
```

show that the "little counting theorem" is proved in an extension of **groups**, but is installed in **groups** by generalizing over the constants a and b.

9.5.8 Model Conservative Theory Extension

Suppose $\mathcal{T}_2$ is an extension of $\mathcal{T}_1$. $\mathcal{T}_2$ is a *model conservative extension* of $\mathcal{T}_1$ if every model of $\mathcal{T}_1$ "expands" to a model of $\mathcal{T}_2$. Model conservative

extensions are safe extensions since they add new machinery without compromising the old machinery. For instance, a model conservative extension preserves satisfiability. The most important model conservative extensions are *definitional extensions* which introduce new symbols that are defined in terms of old vocabulary. (Logically, IMPS definitions create definitional extensions.) Many natural methods of extending theories correspond to a class or type of model conservative extensions. For instance the theories created with the **def-bnf** are always model conservative extensions of the starting theory.

IMPS supports a general technique, based on theory interpretation and theory instantiation, for building safe theory extension methods. The idea is that a model conservative extension type can be formalized as an abstract theory with a distinguished subtheory. The theory can be shown to be a model conservative extension of the subtheory using

Theorem 9.5.1 (Model Conservative Verification Theorem) *Let $\mathcal{T}_2$ be an extension of $\mathcal{T}_1$. $\mathcal{T}_2$ is a model conservative extension of $\mathcal{T}_1$ if there is an interpretation of $\mathcal{T}_2$ in $\mathcal{T}_1$ which fixes $\mathcal{T}_1$.*

Then instances of the model conservative extension type are obtained by instantiating the theory. Each such instance is guaranteed to be a model conservative extension by

Theorem 9.5.2 (Model Conservative Instantiation Theorem) *Let $\mathcal{T}_1'$ be a model conservative extension of $\mathcal{T}_1$, and let $\mathcal{T}_2'$ be an instance of $\mathcal{T}_1'$ under the an interpretation of $\mathcal{T}_1$ in $\mathcal{T}_2$. Then $\mathcal{T}_2'$ is a model conservative extension of $\mathcal{T}_2$.*

For a detailed description of this technique, see [6].

9.5.9 Theory Ensembles

A *theory ensemble* consists of a base theory, copies of the base theory called *replicas*, and unions of copies of the base theory called *theory multiples*. The constituents of a theory ensemble are related to each other by theory interpretations. They allow you to make a definition or prove a theorem in just one place, and then transport the definition or theorem to other members of the theory ensemble as needed. They are used in a similar way to relate a theory multiple to one of its "instances." See Chapter 11 for a detailed description of the IMPS theory ensemble mechanism.

9.5.10 Relative Satisfiability

If there is a theory interpretation from a theory $\mathcal{T}_1$ to a theory $\mathcal{T}_2$, then $\mathcal{T}_1$ is satisfiable (in other words, semantically consistent) if $\mathcal{T}_2$ is satisfiable. Thus, theory interpretations provide a mechanism for showing that one theory is satisfiable relative to another. One consequence of this is that IMPS can be used as a *foundational system*. In this approach, whenever one introduces a theory, one shows it to be satisfiable relative to a chosen foundational theory (such as, perhaps, **h-o-real-arithmetic**).

9.6 Hints and Cautions

(1) The application of a nonnormal translation to an expression containing quasi-constructors will often result in a devastating explosion of quasi-constructors. One way of avoiding this problem is to "normalize" the translation by defining new atomic sorts in the translation's target theory to replace the unary predicates used in the translation.

(2) Suppose a defined atomic sort or constant a is transported to a new atomic sort or constant b via a theory interpretation Φ. From that point on, Φ will map a to b.

(3) Translations must be updated to take advantage of new definitions (see Section 9.4 above). The updating is called *translation enrichment* and it performed periodically by IMPS. For example, enrichment is done when a translation is evaluated and when the macete help mechanism is called. In rare occasions, IMPS will not work as expected because some interpretation has not been enriched. This may happen, for example, when a transportable macete is applied directly without using the macete help mechanism. In some cases, you can get around the problem by re-evaluating a relevant **def-translation** form.

Chapter 10

Quasi-Constructors

The purpose of this chapter is fourfold:

(1) To explain why quasi-constructors are desirable.

(2) To describe how they are implemented.

(3) To show how they are created and reasoned with.

(4) To point out the pitfalls associated with their use.

10.1 Motivation

The constructors of LUTINS are fixed, but you can define *quasi-constructors* which effectively serve as additional constructors. Quasi-constructors are desirable for several reasons:

(1) Once a quasi-constructor is defined, it is available in every theory whose language contains the quasi-constructor's home language. That is, a quasi-constructor is a kind of *global constant* that can be freely used across a large class of theories. (A constructor, as a *logical constant*, is available in *every* theory.)

(2) Quasi-constructors are *polymorphic* in the sense that they can be applied to expressions of several different types. (Several of the constructors, such as = and **if**, are also polymorphic in this sense.)

(3) The definition of a translation (see Chapter 9) is not directly dependent on the definition of any quasi-constructor. Consequently, translations

can be applied to expressions involving quasi-constructors that were defined after the translation itself was defined.

(4) Quasi-constructors can be defined without modifying the IMPS deductive machinery. (Note that the deductive machinery of a ordinary constructor is "hard-wired" into IMPS.)

(5) Quasi-constructors can be used to represent operators in nonclassical logics and operators on generic objects likes sets and sequences.

10.2 Implementation

Quasi-constructors are implemented as "macro/abbreviations." For example, the quasi-constructor **quasi-equals**[1] is defined by the following biconditional:

$$E_1 \simeq E_2 \equiv (E_1{\downarrow} \vee E_2{\downarrow}) \supset E_1 = E_2.$$

Hence, two expressions are quasi-equal if, and only if, they are either both undefined or both defined with the same value.

When the preparsed string representation of an expression ("preparsed string" for short) of the form $E_1 \simeq E_2$ is parsed, $\simeq$ is treated as a macro with the list of arguments E_1, E_2: an internal representation of the expression ("internal expression") is built from the preparsed string as if it had the form $(E_1{\downarrow} \vee E_2{\downarrow}) \supset E_1 = E_2$. When an internal expression of the form $(E_1{\downarrow} \vee E_2{\downarrow}) \supset E_1 = E_2$ is printed, $\simeq$ is used as an abbreviation: the printed string representation of the expression ("printed string") is generated from the internal expression as if the latter had the form $E_1 \simeq E_2$. In other words, in string representation of the expression looks like $E_1 \simeq E_2$, an operator applied to a list of two arguments, but it is actually represented internally as $(E_1{\downarrow} \vee E_2{\downarrow}) \supset E_1 = E_2$.

Abstractly, a quasi-constructor definition consists of three components: a name, a list of schema variables, and a schema. In our example above, **quasi-equals** is the name, E_1, E_2 is the list of schema variables, and the right-hand side of the biconditional above is the schema. A quasi-constructor is defined by a user with the **def-quasi-constructor** form. The list of schema variables and the schema are represented together by a lambda-expression which is specified by a string and a name of a language or theory.

[1] **quasi-equals** is written as `==` in the string syntax and as $\simeq$ in the mathematics syntax, infixed between its operands.

The variables of the lambda-expression represent the list of schema variables, and the body of the lambda-expression represents the schema.

For instance, **quasi-equals** could be defined by:

```
(def-quasi-constructor QEQUALS
  "lambda(e1,e2:ind, #(e1) or #(e2) implies e1 = e2)"
  (language the-kernel-theory))
```

The name of the quasi-constructor is **qequals**, and the lambda-expression is the expression specified by the string in **the-kernel-theory**. After this form has been evaluated, an expression of the form qequals (A, B) is well-formed in any language containing the language of **the-kernel-theory**, provided A and B are well-formed expressions of the same type of kind ι. Separate from the definition, special syntax may be defined for parsing and printing **qequals** (e.g., as the infix symbol $\simeq$).

The quasi-constructor **quasi-equals** is not actually defined using a def-quasi-constructor form in IMPS. It is one of a small number of "system quasi-constructors" which have been directly defined by the IMPS implementors. A system quasi-constructor cannot be defined using the **def-quasi-constructor** form because the schema of the quasi-constructor cannot be fully represented as a LUTINS lambda-expression. This is usually because one or more variables of the schema variables ranges over function expressions of variable arity or over expressions of both kind ι and $*$. For example, qequals (A, B) is not well-defined if A and B are of kind $*$, but quasi-equals (A, B) is well-defined as long as A and B have the same type (regardless of its kind). The major system quasi-constructors are listed in Table 10.1.

10.3 Reasoning with Quasi-Constructors

There are two modes for reasoning with a quasi-constructor Q. In the *enabled mode* the internal structure of Q is ignored. For example, in this mode, when an expression $Q(E_1, .., E_n)$ is simplified, only the parts of the internal expression corresponding to the arguments $E_1, .., E_n$ are simplified, and the part corresponding to Q is "skipped." Similarly, when a macete is applied to an expression $Q(E_1, .., E_n)$, subexpressions located in the part of the internal expression corresponding to Q are ignored. In this mode, quasi-constructors are intended to behave as if they were ordinary constructors.

In the *disabled mode*, the internal structure of Q has no special status. That is, deduction is performed on internal expressions as if Q was never

Quasi-constructor	Schema
quasi-equals (E_1, E_2)	$(E_1{\downarrow} \vee E_2{\downarrow}) \supset E_1 = E_2$
falselike $([\alpha_1, \ldots, \alpha_{n+1}])$	$\lambda x{:}\alpha_1, \ldots, x{:}\alpha_n, \text{falselike}\,(\alpha_{n+1})$
domain (f)	$\lambda x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n, f(x_1, , \ldots, x_n){\downarrow}$
total? $(f, [\beta_1, \ldots, \beta_{n+1}])$	$\forall x_1{:}\beta_1, \ldots, x_n{:}\beta_n, f(x_1, , \ldots, x_n){\downarrow}$
nonvacuous? (p)	$\exists x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n, p(x_1, , \ldots, x_n)$

Notes:

- E_1 and E_2 must have the same type.
- falselike $(*)$ represents the same internal expression as F.
- The sort of f is $[\alpha_1, \ldots, \alpha_{n+1}]$.
- $\tau([\alpha_1, \ldots, \alpha_{n+1}]) = \tau([\beta_1, \ldots, \beta_{n+1}])$.
- The sort of p is $[\alpha_1, \ldots, \alpha_n, *]$

Table 10.1: System Quasi-Constructors

defined. (However, in this mode expressions involving Q will still be printed using Q as an abbreviation.) The mode is activated by "disabling Q," and it inactivated by "enabling Q." The disabled mode is used for proving basic properties about Q. Once a good set of basic properties are proven and formalized as theorems, there is usually little need for the disabled mode. Most quasi-constructors are intended to be employed in multiple theories. Hence, you will usually want to install a theorem about a quasi-constructor with the usage **transportable-macete**.

For a few of the interactive proof commands, there is a dual command with the word "insistent" put somewhere into the command's name. For example, the dual of **simplify** is the **simplify-insistently**. Calling the dual of a command C is equivalent to disabling all quasi-constructors and then calling C itself. In other words, the dual commands behave as if there were no quasi-constructors at all. These "insistent" commands are intended to be used sparingly because they can disturb the internal structure corresponding to a quasi-constructor, creating a new internal expression that can no longer be abbreviated by the quasi-constructor. In visual terms, the quasi-constructor "explodes."

One of the advantages of working in a logic like LUTINS, with a rich structure of functions, is that generic objects like sets and sequences can be represented directly in the logic as certain kinds of functions. For instance, sets are represented in IMPS as *indicators*, which are similar to characteristic functions, except that x is a "member" of an indicator f iff $f(x)$ is *defined*. Operators on indicators and other functions representing generic objects are formalized in IMPS as quasi-constructors, and theorems about these operators are proved in "generic theories" that contain neither constants nor axioms (except for possibly the axioms of the theory **h-o-real-arithmetic**). Consequently, reasoning is performed in generic theories using only the purely logical apparatus of LUTINS (and possibly **h-o-real-arithmetic**). Moreover, theorems about generic objects are easy to apply in other theories since the operators, as quasi-constructors, are polymorphic and since the theory interpretations involved usually have no obligations to check.

10.4 Hints and Cautions

(1) Reasoning with quasi-constructors can be tricky because the internal representation of the quasi-constructor—the structure it abbreviates—

is hidden from view. The **view-expr** form (see Chapter 17) can be used to print an expression without quasi-constructor abbreviations. For example, when

```
(view-expr
  "1/0==?zz"
  (language-name h-o-real-arithmetic)
  no-quasi-constructors)
```

is evaluated, the expression $1/0 \simeq \perp_{\mathbf{Z}}$ is printed as

```
(#(1/0) or #(?zz)) implies 1/0=?zz.
```

(2) It is often easy to confuse quasi-constructors with constants. For instance, a common mistake is to try to unfold a quasi-constructor. To help you avoid such confusion, curly brackets can be used to surround the operands of a quasi-constructor instead of parentheses. That is, a string `q{e1,...,en}`, where `q` is a quasi-constructor, is parsed exactly like `q(e1,...,en)`. In addition, the internal expression is always printed like `q{e1,...,en}`, unless you have defined a special print syntax for `q`.

(3) It should always be remembered that the internal representation of an expression containing quasi-constructors might be much larger than what one might expect from the printed representation of the expression, especially if the quasi-constructors are defined in terms of other quasi-constructors. Consequently, the processing speed of IMPS on expressions containing quasi-constructors sometimes seems unjustifiably slow.

(4) An expression may sometimes "implode" when parsed. That is, quasi-constructors may appear in the printed string in places where they did not occur in the preparsed string. This happens because the IMPS printing routine will use quasi-constructors as abbreviations wherever possible, regardless of whether these quasi-constructors were used in the preparsed string. Expression implosion can be caused by simplification and macete application.

(5) Due to the implosion phenomenon, the preparsed and printed strings may not parse into the same internal expressions (but the two internal

expressions will always be α-equivalent). For example, consider the preparsed string S_1

```
with(f:[ind,ind], forall(x:ind, #(f(x))))
```

It parses into an internal expression E_1 which is printed as the string S_2

```
with(f:[ind,ind],total_q{f,[ind,ind]})
```

with quasi-constructor abbreviations and is printed as S_1 without quasi-constructor abbreviations. However, S_2 parses into an internal expression E_2 which is printed as S_2 with quasi-constructor abbreviations and is printed as the string S_3

```
with(f:[ind,ind],forall(x_0:ind,#(f(x_0))))
```

without quasi-constructor abbreviations. That is, the two internal expressions are the same except for the name of the single bound variable.

(6) One of the most devastating forms of quasi-constructor explosion can occur when a nonnormal translation (which maps some atomic sorts to unary predicates) is applied to an expression containing quasi-constructors. The quasi-constructor explosion happens because the binding expressions in the internal representation of a quasi-constructor will be rewritten when the translation is applied if the variables being bound involve a atomic sort which is mapped to a unary predicate.

(7) Often two different internal expressions containing quasi-constructors will be printed in exactly the same way. Usually this is due to the fact that some of the bound variables in the internal representation of some of the quasi-constructors have different names (i.e., the two expressions are α-equivalent). In this case, there is no problem; for the most part, IMPS will treat the expressions as if they were identical. However, the two expressions may differ in other ways, e.g., some of the corresponding bound variables may have different sorts. Hence, in rare occasions, it may happen that there are two internal expressions printed the same which are not known by IMPS to be equal (or quasi-equal). This situation can be very hard to deal with. It is often best to backup and try a different path that will avoid this confusing situation.

(8) A quasi-constructor can be disabled, and then later enabled, in the midst of a deduction using the appropriate proof commands. However, disabling a quasi-constructor in the midst of a proof can cause severe confusion if one forgets to later enable it. As a general rule, you should always try to find ways to avoid disabling quasi-constructors.

Chapter 11

Theory Ensembles

11.1 Motivation

Much of mathematics deals with the study of structures such as monoids, fields, metric spaces, or vector spaces, which consist of an underlying set **S**, some distinguished constants in **S**, and one or more functions on **S**. In IMPS, an individual structure of this kind can be easily formalized as a theory. For example, the IMPS theory for monoids is specified by two def-forms:

```
(def-language MONOID-LANGUAGE
  (embedded-languages h-o-real-arithmetic)
  (base-types uu)
  (constants
   (e uu)
   (** (uu uu uu))))

(def-theory MONOID-THEORY
  (component-theories h-o-real-arithmetic)
  (language monoid-language)
  (axioms
   (associative-law-for-multiplication-for-monoids
    "forall(z,y,x:uu, x**(y**z)=(x**y)**z)")
   ("forall(x:uu,x**e=x)")
   ("forall(x:uu,e**x=x)")))
```

In the mathematical syntax, the sort `uu` is written **U** and the constants `e`, `**` are written $e, \bullet$.

Notice that the theory of a monoid as presented here is suitable for concept formulation and proof in a *single* monoid. Here are some examples:

- One can prove uniqueness of the multiplicative identity:

$$\forall x{:}\mathbf{U}, (\forall y{:}\mathbf{U}, x \bullet y = y \bullet x = x) \equiv x = e$$

- A sequential product operator for sequences can be defined as the least solution of

$$\Pi = \lambda m, n{:}\mathbf{Z}, f{:}[\mathbf{Z}, \mathbf{U}], \mathrm{if}(m \leq n, \Pi(m, n-1, f) \bullet f(n), e)$$

- One can easily prove Π satisfies

$$\forall f{:}[\mathbf{Z}, \mathbf{U}], m, n{:}\mathbf{Z}, n \leq m \supset \Pi(n, m, f) \simeq \Pi(n, m-1, f) \bullet f(m)$$

- An monoid *endomorphism* (i.e., a self homomorphism) can be defined by the predicate

$$\lambda\varphi{:}[\mathbf{U}, \mathbf{U}], \varphi(e) = e \wedge \forall x, y{:}\mathbf{U}, \varphi(x \bullet y) = \varphi(x) \bullet \varphi(y)$$

However, there is no way to talk about monoid homomorphisms between different monoids.

Nevertheless, it is possible to produce a suitable theory in IMPS in which a general notion of monoid homomorphism can be defined. To do so requires at the very least a theory with two possibly distinct monoids. In general, multiple instances of structures can be dealt with in IMPS in two ways:

- In one approach, we first formalize set theory as an IMPS theory. This can be done by treating **Set** as a primitive sort and the membership relation ϵ as a primitive function symbol. It is then possible to define a structure of a particular kind by a predicate on **Set**. For instance, a monoid is any object in **Set** of the form $(\mathbf{U}, \bullet, e)$ where $\bullet$ is a binary associative operation on $\mathbf{U}$ for which e is a right and left identity.

- The other approach is to create a new theory which is the union of embedded replicas of a theory of a single structure. A replica of a theory $\mathcal{T}$ is a theory $\mathcal{T}'$, which differs from $\mathcal{T}$ only by an appropriate renaming of sorts and constants. By an embedding we mean a theory interpretation which maps distinct sorts and constants to distinct sorts and constants.

In this section we explain the theory ensemble mechanism, which implements the second approach above. To illustrate how the machinery works in the case of a monoid, let us define a new theory, **monoid-theory-1** obtained by renaming the sorts and constants of **monoid-theory** not in **h-o-real-arithmetic** by affixing the characters " _1."

```
(def-language MONOID-LANGUAGE-1
  (embedded-languages h-o-real-arithmetic)
  (base-types uu_1)
  (constants
    (e_1 uu_1)
    (**_1 (uu_1 uu_1 uu_1))))

(def-theory MONOID-THEORY-1
  (component-theories h-o-real-arithmetic)
  (language monoid-language-1)
  (axioms
    (associative-law-for-multiplication-for-monoids
     "forall(z,y,x:uu_1,
        x **_1 (y **_1 z)=(x **_1 y) **_1 z)")
    ("forall(x:uu_1,x **_1 e_1=x)")
    ("forall(x:uu_1,e_1 **_1 x=x)")))
```

Now define **monoid-pair** as the union of **monoid-theory** and **monoid-theory-1**:

```
(def-theory MONOID-PAIR
  (component-theories monoid-theory monoid-theory-1))
```

The theory ensemble facility automates these steps and also provides several bookkeeping devices for keeping track of theories and interpretations in the ensemble.

11.2 Basic Concepts

A copy of a theory $\mathcal{T}$ is a new theory $\mathcal{T}'$ obtained from $\mathcal{T}$ by applying a renaming function to the constant and sort symbols in $\mathcal{T}$. The only condition on the renaming function is that it not identify distinct constant names or distinct sort names. A typical renaming function is one which affixes an underscore followed by a number to a name. For each $n \geq 0$, let R_n be a renaming function.

- For each $n \geq 0$, let $\mathcal{T}_n$ be a copy of $\mathcal{T}$ obtained by applying the renamer R_n. $\mathcal{T}_n$ is called the n-th theory replica of $\mathcal{T}$.

- Let $\mathcal{U}_n$ be the union of the theories $\mathcal{T}_0, \ldots, \mathcal{T}_{n-1}$. $\mathcal{U}_n$ is called the n-th theory multiple of $\mathcal{T}$.

- For each $i \in \{0, \ldots n-1\}$ there is a theory interpretation Ψ_i of $\mathcal{T}$ in the replica $\mathcal{T}_i$. Ψ_i is called i-th replicating translation from the base theory. These translations are actually constructed by the system when a multiple is built. Note that Ψ_i composed with the inverse of Ψ_j is also an interpretation of $\mathcal{T}_j$ in $\mathcal{T}_i$; however, this translation is not constructed by the system automatically.

- Let $m < n$, and

  $$f : \{0, \ldots, m-1\} \to \{0, \ldots, n-1\}$$

  be an injective function. In our documentation we refer to f as a *permutation.* Then there is a unique interpretation $\Phi_f : \mathcal{U}_m \to \mathcal{U}_n$ with the property that, for each $i \in \{0, \ldots, m-1\}$, Φ_f extends the canonical interpretation of $\mathcal{T}_i$ in $\mathcal{T}_{f(i)}$. Φ_f is called the canonical interpretation associated to f.

- A defined constant is *natively defined* in $\mathcal{U}_m$ if and only if the following two conditions hold:

  - The constant is not the translation of a defined constant from a lower multiple by a canonical translation.
  - The constant is not the translation of a defined constant from a higher multiple by an existing interpretation.

11.3 Usage

There are four def-forms for building and manipulating theory ensembles.

- `def-theory-ensemble`. This form is used to build a theory ensemble from a given theory $\mathcal{T}$. The theory $\mathcal{T}$ is called the *base theory* of the ensemble.

- `def-theory-ensemble-multiple`. This form is used to build a theory $\mathcal{T}_n$ which is the union of n copies of $\mathcal{T}$.

- `def-theory-ensemble-instances`. This form is used to build a theory interpretation from a theory multiple $\mathcal{T}_n$ to a given theory.
- `def-theory-ensemble-overloadings`. This form establishes overloadings for the constants defined in $\mathcal{T}$ and all the multiples $\mathcal{T}_n$.

11.4 Def-theory-ensemble

This is the def-form for building a theory ensemble from an IMPS theory. In the simplest version, the form requires only one argument, the name of the theory ensemble which is then identical to the name of the base theory. For example:

```
(def-theory-ensemble METRIC-SPACES)
```

Evaluating the def-form builds a theory ensemble also named `METRIC-SPACES`. In general, three additional keyword arguments are allowed:

- `(base-theory` *theory-name*`)`. *theory-name* is the name of the base theory of the ensemble. This keyword arguiment is used when the name of the ensemble is different from that of the base theory.
- `(fixed-theories` $theory_1$ ... $theory_n$`)`. The presence of this argument causes IMPS to rename only those sorts and constants which do not belong to any of the theories $theory_1 \ldots theory_n$ when building replicas. The default value of this argument is the fixed-theories set at the time the form is evaluated.
- `(replica-renamer` *proc-name*`)`. *proc-name* is the name of a procedure of one integer argument, used to name sorts and constants of theory replicas. The default procedure is to affix an underscore followed by a nonnegative integer to a name. Most users should be content to use the default.

Another example which utilizes the fixed-theories option is the theory ensemble of vector spaces over a field:

```
(def-theory-ensemble VECTOR-SPACES
  (fixed-theories fields))
```

When a theory ensemble def-form is evaluated, IMPS builds a number of tables, all of which are initially empty. The table will be filled as multiples are built. The tables are:

- A table of the theory replicas. The key is an integer $k \geq 1$ and the entry is the k-th replica of the base theory.
- A table of the theory multiples. The key is an integer $k \geq 1$ and the entry is the k-th multiple of the base theory.
- A table of canonical interpretations between multiples. The key is a pair of integers (m, n) with $1 \leq m < n$ and the corresponding entry is a list of interpretations from the m-multiple to the n-multiple.

11.5 Def-theory-multiple

This form requires two arguments, the name of a theory ensemble and an integer n. For example,

```
(def-theory-ensemble-multiple metric-spaces 2)
```

When this form is evaluated, IMPS builds a 2-multiple of the theory **metric-spaces**. IMPS automatically names this theory **metric-spaces-2-tuples**.

Once the theory **metric-spaces-2-tuples** is available several important concepts can be defined. For instance,

```
(def-constant CONTINUOUS
  "lambda(f:[pp_0,pp_1],forall(v:sets[pp_1],
       open(v) implies open(inv_image(f,v))))"
  (theory metric-spaces-2-tuples))
```

11.6 Def-theory-ensemble-instances

This form is used to build interpretations from selected theory multiples and to transport natively defined constants and sorts from these multiples using these interpretations. The syntax for this def-form is very elaborate, so we refer you to Chapter 17 for its general use.

As an example, suppose we wanted to consider the pair consisting of a normed linear space and the real numbers as a special case of the theory **metric-spaces-2-tuples**. One reason for doing this is to make all the mathematical machinery developed in the abstract theory **metric-spaces-2-tuples** available in the particular case. This machinery includes definitions (such as the definition of continuity for mappings) and theorems (such as the various characterizations of continuity.)

We would like to build a theory interpretation from **metric-spaces-2-tuples** to the union of **normed-linear-spaces** and **h-o-real-arithmetic** which is **normed-linear-spaces** itself, since **normed-linear-spaces** is a super-theory of **h-o-real-arithmetic**. Moreover, we would like IMPS to translate all the defined sorts and constants in **metric-spaces-2-tuples** to **normed-linear-spaces**.

This can be accomplished by evaluating the form

```
(def-theory-ensemble-instances METRIC-SPACES
  (target-theories
     normed-linear-spaces
     h-o-real-arithmetic)
  (multiples 1 2)
  (sorts (pp uu rr))
  (constants
     (dist "lambda(x,y:uu, norm(x-y))"
           "lambda(x,y:rr, abs(x-y))")))
```

This does two things: First it builds an interpretation from **metric-spaces-2-tuples** into **normed-linear-spaces** in which

- `pp_0` goes to `uu`.
- `pp_1` goes to `rr`.
- `dist_0` goes to `lambda(x,y:uu, norm(x-y)`.
- `dist_1` goes to `lambda(x,y:rr, abs(x-y))`.

Secondly, it translates all the natively defined constants in **metric-spaces** and **metric-spaces-2-tuples** into **normed-linear-spaces**.

11.7 Def-theory-ensemble-overloadings

This form takes the following arguments:

- The name of the theory ensemble.
- Any number of integers `(`$N_1 \ldots N_k$`)`

Evaluation of the def-form causes IMPS to overload those constants natively defined in the theory multiples $N_1 \ldots N_k$. For example:

```
(def-theory-ensemble-overloadings metric-spaces 1 2)
```

This form makes the theory ensemble mechanism manageable to the user. When the system transports a natively defined constant c from a theory multiple, it will often give the new constant an unappealing name. For instance, the name might be very long with several indices as suffixes. Ideally, one would like to use the same name for all instances of c, and this is usually possible if the constant is a function symbol which occurs as an operator of an application. In this case, the appropriate function symbol may be unambiguously determined from the sorts of the arguments.

11.8 Hints and Cautions

(1) In principle, any theory can be used to build a theory ensemble. However, the theory ensemble facility is useful only if the following two conditions on the base theory are met:

- The base theory is fairly general so that there are likely to be numerous instances of the base theory. Thus it would make little sense to make a theory with essentially only one model into a theory ensemble.
- There is a need for dealing with at least two distinct instances of the base theory.

(2) There is a certain amount of overhead in building theory multiples. This overhead includes building and tabulating theory replicas and building canonical theory interpretations from the m-multiples to n-multiples, where $m < n$. Recall, that there is a canonical interpretation $\Phi_f : \mathcal{U}_m \to \mathcal{U}_n$ for each injective map

$$\{0, \ldots, m-1\} \to \{0, \ldots, n-1\}.$$

There are $n(n-1)\cdots(n-m+1)$ such mappings. This overhead can become forbidding when the multiples get too large. As a general rule, multiples of order greater than 4 should be avoided.

(3) Before the system builds an n-multiple it checks whether all j-multiples for $j < n$ already exist, building the missing multiples as needed.

Chapter 12

The Proof System

In purpose of this chapter is to explain:

- The IMPS proof system.
- A description of how you prove theorems in IMPS, in interactive mode and in script mode.
- A description of the proof-script language.

12.1 Proofs

A *proof* of a statement ψ is conclusive mathematical evidence of why ψ is true. A proof is usually thought of as a sequence of formulas $\varphi_1, \cdots, \varphi_n$ such that $\varphi_n = \psi$ and every φ_i is either:

(1) An axiom, that is, a formula which is assumed true to begin with;

(2) A theorem, that is, a formula whose truth has already been established by some other proof; or

(3) A consequence of one or more previous formulas in the sequence. This means that φ_i is related to these formulas by a relation called an *inference*.

The criterion for determining what constitutes an acceptable inference depends on the purpose of the proof. If the proof is for mechanical verification only (whether by a digital computer or a human being acting like one), then the criteria of acceptability can be specified unambiguously as a set of rules

for determining which sequences of symbols constitute proofs. This set of rules is called a *proof system.* For a human audience, on the other hand, it is usually a good idea to stress the essential or innovative steps in a proof and avoid routine computations. Moreover, for an expository article, a textbook, or a classroom lecture, it is acceptable, even desirable, that inference steps appeal to intuition.

12.2 The IMPS Proof System

In the traditional presentation of proofs in textbooks and journals, the deductive process appears as an inexorable progression from known facts to unknown ones. In contrast, the deductive process in IMPS begins with a conjecture and usually proceeds with a large amount of trial and error. This view of the deductive process leads naturally to the idea of a proof as a *graph.*

Proofs in IMPS are represented by a data structure called a *deduction graph,* which is a directed graph with two kinds of nodes: *inference nodes* and *sequent nodes.* A sequent node consists of a single formula called the *assertion* together with a *context.* The context is logically a set of assumptions, although the implementation caches various kinds of derived information with a context. An inference node i consists of a unique conclusion node c, a (possibly empty) set of hypothesis nodes $\{h_1, \cdots, h_n\}$, and the mathematical justification for asserting the relationship between c and $h_1, \cdots, h_n$. This justification is called an *inference rule.*

As an interactive process, a proof consists of a series of reductions of unproven sequent nodes of the graph (which we think of as goals) to new subgoals. Each reduction is created by a procedure called a *primitive inference procedure*, which takes a sequent node and possibly other arguments, returns an inference node **inf** and, as a side-effect, changes the deduction graph. These changes include:

- The addition of **inf** to the set of inference nodes of the deduction graph.
- The addition of the hypotheses of **inf** to the set of sequent nodes.

There are about 25 primitive inference procedures. *Primitive inference procedures are the only means by which inference nodes can be added to a deduction graph.* As a user of IMPS, however, you will never apply primitive

inference procedures directly; you will only apply them indirectly by applying *proof commands*—convenient procedures which add sequent nodes and inference nodes to a deduction graph by calling primitive inferences. Proof commands are documented in Chapter 18.

12.3 Theorem Proving

There are two modes of proving theorems in IMPS: *interactive mode* and *script mode.* Interactive mode is used for proof discovery and interactive experimentation with new proof techniques. Script mode is used primarily for proving theorems whose proofs are similar to previously constructed proofs or for checking a large number of proofs.

To begin an interactive proof, select the **Start dg** option in the menu or use the Emacs command `M-x imps-start-deduction`. IMPS will then prompt you in the minibuffer with the text `Formula or reference number:` which you can supply by clicking the *right* mouse button on the formula you want to prove (provided it is enclosed between double quotes) or by typing the reference number of the formula (provided it has already been built). All the proof commands and node-movement commands given during the course of a single proof are recorded as a component of the deduction graph. These commands can be inserted as text into a buffer (provided the major mode of the buffer is **scheme mode**) by selecting the option **Insert proof script** or by entering `C-c i`. The text that is inserted in the buffer is a script of proof commands which can be edited and used in other proofs. To run a command script, select the option **Execute region** or enter `C-c r`. This will run the commands in the current region. Note that an interactive proof can be combined with one or more proof segments run in script mode. This is especially useful for redoing similar arguments.

12.3.1 Checking Proofs in Batch

To check a large number of proofs, (for example, to check an addition to the theory library) proof scripts can be run in batch by using the shell command

```
$IMPS/../bin/run_test testfile logfile
```

where `testfile` is a text file which may contain any **def-form**. Typically, it will consist of a series of **include-files** and **load-section** forms. The file `logfile` will contain the output of the process.

For example, entering on a machine called veraguas the shell command

```
$IMPS/../bin/run_test testy loggy
```

where testy contains

```
(load-section abstract-calculus)
```

will produce a logfile `loggy` which will look something like:

```
HOST veraguas starting IMPS test.
File tested = testy
Executable = /imps/sys/executables/foundation
Mon May  3 18:21:56 EDT 1993

INITIAL THEORY NETWORK INFORMATION:

THEORIES = 12

THEORY-INTERPRETATIONS = 1

THEOREMS = 277

MACETES = 360

EXPRESSIONS = 3428

1. THEOREM-NAME: ANONYMOUS

forall(a,c:uu,
  forsome(b:uu,b prec a and c prec b) implies c prec a);

THEORY: PARTIAL-ORDER
SEQUENT NODES: 10
PROOF STEPS: 3
GROUNDED?: YES
CONTEXTS=4
ASSUMPTIONS/CONTEXT=1.0
virtual time = 0.16 seconds

   .
   .
```

```
   .


219. THEOREM-NAME: LOCALITY-OF-INTEGRALS

forall(phi,psi:[ii,uu],a,b:ii,
  a<b
   and
  integrable(phi)
   and
  integrable(psi)
   and
  forall(x:ii,a<=x and x<b implies phi(x)=psi(x))
   implies
  forall(x:ii,
    a<=x and x<b implies
    integral(a,x,phi)=integral(a,x,psi)));

THEORY: MAPPINGS-FROM-AN-INTERVAL-TO-A-NORMED-SPACE
SEQUENT NODES: 54
PROOF STEPS: 33
GROUNDED?: YES
CONTEXTS=29
ASSUMPTIONS/CONTEXT=5.551724137931035
virtual time = 25.05 seconds

0 errors during test.

0 failed proofs during test.

FINAL THEORY NETWORK INFORMATION:

THEORIES = 32

THEORY-INTERPRETATIONS = 110

THEOREMS = 915

MACETES = 1108
```

```
EXPRESSIONS = 89274

Mon May  3 21:06:38 EDT 1993
IMPS test completed.
```

12.4 The Script Language

Informally, a *script* is a sequence of proof commands and control statements. Essentially, the script facility in IMPS allows users to build programs to prove theorems. Scripts are a useful way to package and reuse common patterns of reasoning. The paper [8] presents some practical methods for exploiting the IMPS proof script mechanism.

A script is a kind of s-expression, that is, a symbol, a string, an integer, or a nested-list of such objects. To describe how IMPS interprets scripts to modify the state of the deduction graph, we first establish some terminology to designate certain classes of script components. Each script component is itself an s-expression.

- A *script form expression* (expression for short) is a string, a symbol, a number, or a list of expressions.
- An *expression operator* is one of the symbols `%`, `*`, `~*`.
- A *script variable* is a symbol beginning with $.
- A *keyword form* is a list whose first element is one of the following symbols (which we refer to as keywords): `move-to-sibling`, `move-to-ancestor`, `move-to-descendent`, `block`, `if`, `for-nodes`, `while`, `let-script`, `let-macete`, `let-val`, `script-comment`, `label-node`, `jump-to-node`, `skip`.
- A *command form* is an expression which is either a command name or a list whose first element is a command name.
- A *script* is a list of script form expressions.

In Figure 12.1 we provide an example of a script. This script can be used to prove the combinatorial identity discussed in Chapter 5 on the IMPS micro exercises. The script consists of a number of forms, each of which is applied to a unique node of the deduction graph called the *current node.*

```
(
  (unfold-single-defined-constant-globally comb)
  (apply-macete-with-minor-premises
    fractional-expression-manipulation)
  (label-node compound)
  direct-and-antecedent-inference-strategy
  (jump-to-node compound)
  (for-nodes
    (unsupported-descendents)
    (if (matches? "with(t:rr, #(t^[-1]))")
        (apply-macete-with-minor-premises
          definedness-manipulations)
        (block
          (apply-macete-with-minor-premises
            factorial-reduction)
          simplify)))
 )
```

Figure 12.1: An IMPS Proof Script

In interactive mode, the current node is the one which is displayed in the sequent node buffer.

(1) Expand the definition of **comb**, which in IMPS is given in terms of the factorial function. The resulting sequent consists of a context containing the formulas $k \leq m$ and $1 \leq k$ and the assertion

$$\frac{(1+m)!}{k! \cdot (1+m-k)!} = \frac{m!}{(k-1)!\,(m-(k-1))!} + \frac{m!}{k!\,(m-k)!}.$$

(2) Apply the macete fractional-expression-manipulation to the sequent above. Moreover, attach to the label **compound**, to the resulting node, but otherwise leave the deduction graph intact. This will allow us to subsequently return to this node.

(3) Apply the command direct-and-antecedent-inference-strategy to the result of the previous step. This adds three new sequents to the deduction graphs: two concerning definedness and one whose assertion is an equation.

(4) We will process these nodes in step by iterating through them with the **(for-nodes)** keyword form as follows:

   (a) For the sequents concerning definedness, use the macete definedness-manipulations. This combines a number of facts about the definedness of the arithmetic operations and the factorial function.

   (b) For the remaining sequent, use the macete factorial-reduction.

12.4.1 Evaluation of Script Expressions

Scripts are executed in an environment. This environment is an association of symbols to values. Every script expression has a value in the script's execution environment. The value of a script expression is determined as follows:

- If the expression is a script variable $\$n$ where n is an integer, then the value is the n-th argument in the script call.
- If the expression is a script variable $\$name$ where *name* is a symbol, its value is determined by the last assignment of that variable. If there is no assignment preceding the call, an error is raised.
- If the expression is a string, an integer, or a symbol which is not a variable, then the value of the expression is itself.
- If the expression is a list whose first element is an expression operator, (that is, one of the symbols `%`, `*`, `~*`) then that operator is applied to the values of the arguments as follows:
  - `(%` *format-string* $arg_1 \cdots arg_n$`)`. The first argument must be a string with exactly n occurrences of the characters `~A`. The remaining arguments arg_i are plugged in succession for each occurrence of `~A`.
  - `(%`sym *format-string* $arg_1 \cdots arg_n$`)`. The first argument must be a string with exactly n occurrences of the characters `~A`. The remaining arguments arg_i are plugged in succession for each occurrence of `~A` and the resulting string is read in as a symbol.
  - `(*` $arg_1 \cdots arg_n$`)`. The set of assumptions (regarded as strings) of the current sequent node which match any argument.

– `(~*` $arg_1 \cdots arg_n$`)`. The set of assumptions (regarded as strings) of the current sequent node which match no argument.

- If the expression is a list whose first element is not an expression operator, its value is the list of values of the arguments.

12.5 The Script Interpreter

A script is a list (*form*$_1$ ... *form*$_n$) of script expressions. The interpreter executes each form in the script in left-to-right order[1]. Once a form is executed by the script interpreter at a sequent node S, execution of the script continues at a new sequent node S_1 called the *natural continuation node* of the script. This node is usually the *first ungrounded relative* of S *after* the form is executed. The only exception is when the form is a node motion keyword form.

The rules for script interpreter execution are given below. Interpreting a script expression *form*$_i$ usually causes side effects on the current deduction graph and other structures associated to the current proof. The side-effects depend on the kind of form being handled by the interpreter.

Keyword Forms

Recall that a keyword form is a script expression which is a list whose first element is a keyword. The execution of keyword forms depends on the keyword:

- *Node motion forms.* These forms cause script execution to continue at a node other than the natural continuation node. The following are the node motion keyword forms:

  – `(move-to-sibling` n`)`. This causes execution to proceed at the n-th sibling node.

  – `(move-to-ancestor` n`)`. This causes execution to proceed at the n-th ancestor node.

  – `(move-to-descendent (`$l_1 \cdots l_n$`))`. This causes execution to proceed at the specified descendent node. Each l_i is an integer or a dotted pair $(a\ .\ b)$ of integers.

[1] Given the way scripts are usually presented on a page this really means from top to bottom.

- `(jump-to-node` *label*`)`. *label* is a deduction graph sequent node label.

- *Conditionals.* `(if` *test-form consequent alternative*`)` These are forms which provide conditional execution of command forms. The *test-form* can be of the following kinds:

  - `(matches?` *assertion assumption-list*$_1$ $\cdots$ *assumption-list*$_n$`)`, where *assertion* and *assumption-list*$_i$ are script expressions. The condition is true if the value of *assertion* matches the assertion of the current sequent node and every formula in the value of *assumption-list* matches an assumption of the current sequent node.
  - `(minor?)`. True if the current sequent node is the minor premise of some inference node.
  - `(major?)`. True if the current sequent node is the major premise of some inference node.
  - `(generated-by-rule?` *rule-name*`)`. True if the current sequent node is an assumption of an inference node with name *rule-name*.
  - `(succeeds?` *script-form*`)`. True if *script-form* applied to current sequent node grounds it. May cause side effects on the deduction graph (and is usually intended to do so.)
  - `(progresses?` *script-form*`)`. True if *script-form* applied to current sequent node makes any progress. May cause side effects on the deduction graph (and is usually intended to do so.)
  - `(not` *test-form*`)`. True if *test-form* fails. May cause side effects on the deduction graph.
  - `(and` *test-form*$_1 \cdots$ *test-form*$_n$ `)`. True if all *test-form*$_i$ are true. May cause side effects on the deduction graph.
  - `(or` *test-form*$_1 \cdots$ *test-form*$_n$`)`. True if at least one *test-form*$_i$ is true. May cause side effects on the deduction graph.

- *Blocks.* `(block` *script-form*$_1$ $\cdots$ *script-form*$_n$`)`. A block form merely provides a grouping for a list of script forms.

- *Iteration forms.* These are of two kinds:

  - Range iterations provide for execution over a specified range of nodes. These have the form `(for-nodes` *range-form script-form*`)`. *range-form* can be one of the following:
    * `(unsupported-descendents)`.
    * `(minor-premises)`.
    * `(node-and-siblings)`.
  - Conditional iterations provide for execution while a specified condition holds. These have the form `(while` *test-form script-form*`)`, where *test-form* is specified in the same way as for conditional forms.

- *Skip.* `(skip)`. This is a form used to insert in locations where at least one script form is required. It is essentially equivalent to `(block)`.

- *Assignments.*
  - `(let-script` *name arg-count script*`)`. This form adds a command with name $*name*. The value of $*name* is this name in the current execution environment.
  - `(let-macete` *name compound-macete-specification*`)`. This form adds a macete with name $*name*. The value of $*name* is this name in the current execution environment.
  - `(let-val` *name expression*`)`. The value of $*name* is this name in the current execution environment.

- *Deduction graph node labels.*
  - `(label-node` *name*`)`. This form assigns the label *name* to the current node.

  All these forms modify the current execution environment.

- *Comments.* `(script-comment` *comment-string*`)`. Adds a comment to the preceding history entry. This keyword form has no other effect.

Other Forms

Execution of any other form *form* in a script has the following result: The value of *form* must be a command form which is applied to the current node. In other words:

- If the value of *form* is a symbol, this value should be the name of a command which takes no arguments; otherwise an error results. The script interpreter then applies that command to the current sequent node.

- If the value of *form* is a list, the first expression of the list must be the name of a command, while the remaining elements should be arguments of the command; otherwise an error results. The script interpreter then applies that command with the resulting arguments to the current sequent node.

The following are possible arguments to a command:

- A symbol. Depending on the command and the location of the argument, this symbol may refer to a theorem, a macete, a translation, an inductor, a quasi-constructor, or a constant.

- A string. This string can be used to refer to an expression, a theorem, or a context.

- An integer. This integer can be used to refer to a sequent node, an occurrence number within an expression, or the number of an assumption.

- A list of strings.

- A list of integers.

12.6 Hints and Cautions

(1) To build a proof script for insertion in a **def-theorem** form, it is much easier to first prove the theorem interactively and then record the proof script using `C-c i`. A novice user should not attempt to build scripts from scratch without thoroughly understanding how scripts replay interactive proofs.

(2) More experienced users can build new scripts by modifying existing ones. For example, the following is a def-theorem form with a valid proof:

```
(def-theorem MIN-LEMMA
```

```
"forall(a,b,c:rr,
          a<b and a<c
           implies
          forsome(d:rr, a<d and d<=b and d<=c))"
(theory h-o-real-arithmetic)
(proof
 (
  direct-and-antecedent-inference-strategy
  (instantiate-existential ("min(b,c)"))
  (unfold-single-defined-constant-globally min)
  (case-split ("b<=c"))
  simplify
  simplify
  (apply-macete-with-minor-premises minimum-1st-arg)
  (apply-macete-with-minor-premises minimum-2nd-arg)
  )))
```

To get a valid proof for an analogous theorem where the arguments to $<$ and $\leq$ are interchanged, one can edit the previous script replacing **min** by **max** everywhere in the proof:

```
(def-theorem MAX-LEMMA
  "forall(a,b,c:rr,
            b<a and c<a
             implies
            forsome(d:rr, d<a and b<=d and c<=d))"
  (theory h-o-real-arithmetic)
  (proof
   (
    direct-and-antecedent-inference-strategy
    (instantiate-existential ("max(b,c)"))
    (unfold-single-defined-constant-globally max)
    (case-split ("b<=c"))
    simplify
    simplify
    (apply-macete-with-minor-premises maximum-1st-arg)
    (apply-macete-with-minor-premises maximum-2nd-arg)
    )))
```

Chapter 13

Simplification

13.1 Motivation

One of the goals of the IMPS system is to assist users in producing formal proofs in which the number of steps and the kinds of justification used for each step are close to those of a rigorous and detailed (but informal) proof. One of the main reasons that formal proofs are usually so long is that many proof steps involve replacement of an occurrence of a term t_1 with a term t_2 having the same value as t_1 and which is obtained from t_1 by a variety of theory-specific transformations. A formal proof might require hundreds of steps to justify such a replacement. In IMPS, the simplifier is designed to handle many of these inferences directly without user intervention and to bundle them into a single inference step. The simplifier accomplishes this as follows:

- By invoking a variety of theory-specific transformations on expressions, such as rewrite rules and simplification of polynomials (given that the theory has suitable algebraic structure, such as that of a field);
- By simplifying expressions based on their logical structure; and
- By discharging the great majority of definedness and sort-definedness assertions needed to apply many forms of inference.

The simplifier can be invoked directly by the user as an IMPS command. As such it is an exceedingly powerful tool. More significantly, the simplifier is a central element in the IMPS inference architecture, since it is routinely invoked by other commands.

The simplifier can be viewed as a function of a context Γ and an expression e which returns an expression $S(\Gamma, e)$. The following condition serves as the correctness requirement for the simplifier: For any context Γ (in a theory $\mathcal{T}$) and expression e, $\mathcal{T}$ and Γ must together entail $e \simeq S(\Gamma, e)$. That is to say, either e and e' are both defined and share the same denotation, or else they are both undefined.

13.2 Implementation

The simplifier is a highly recursive procedure. For instance, simplification of a compound expression may require prior simplification of all its components (that this is not always true reflects the fact that the simplification procedure for an expression depends on the constructor of the expression). Simplification of individual expression components is done with respect to the *local context* of that component in the expression. Thus, for instance, in simplifying an implication $A \supset B$, A may be assumed true in the local context relative to which B is simplified. Similarly, in simplifying the last conjunct C of a ternary conjunction $A \wedge B \wedge C$, A and B may be assumed in the local context. On the other hand, when a variable-binding operator is traversed, and there are context assumptions in which the bound variable occurs free, then the simplifier must either rename the bound variable or discard the offending assumptions. The strategy of exploiting local contexts is discussed in [20].

At any stage in this recursive descent, certain theory-specific procedures may be applied to a subexpression of the original expression or to a semantically equivalent form of the original expression. These procedures are called *transforms*. A transform T takes two arguments, a context Γ and an expression e, and returns two values, an expression $T(\Gamma, e)$ and a set of formulas $\{c_1(\Gamma, e), \ldots, c_n(\Gamma, e)\}$ called *convergence requirements.* To apply a transform T to a context Γ and an expression e means to do one of three things:

- If the simplifier can determine that all the convergence requirements are met in Γ, replace e with $T(\Gamma, e)$.
- If the simplifier can determine that one of the convergence requirements fails in Γ, replace e with an undefined expression of the same sort as e.
- Otherwise, leave e as it is.

This process is sound provided, for every context Γ and expression e, the following conditions hold:

- If all the convergence requirements $c_1(\Gamma, e), \cdots, c_n(\Gamma, e)$ are true in Γ, then $T(\Gamma, e) \simeq e$.
- If any of the convergence requirements is false in Γ, then $T(\Gamma, e)$ is undefined.

Each theory has a table of transforms indexed on two keys: (1) a constructor, a quasi-constructor, or the atom **nil** and (2) a constant or the atom **no-defined-constant**. The simplifier determines what transforms in the table to apply to an expression as follows:

- It first computes the expression's *lead constructor*. This is the quasi-constructor of the expression if it has one. Otherwise, it is the constructor of the expression (which is the atom **nil** for variables and constants).
- It then computes the given expression's *lead constant*. This is the first constant in a left-to-right traversal of the expression tree, if there is one, or **no-defined-constant** otherwise. Thus, the lead constant of $\lambda x, y{:}\mathbf{R}, x + y$ is $+$.
- Finally, it applies, in a nondeterministic order, the transforms in the table whose keys are the same as the expression's lead constructor and lead constant.

13.3 Transforms

The transforms used by the IMPS simplifier include:

(1) Algebraic simplification of polynomial expressions.

(2) A decision procedure for linear inequalities, based on the variable elimination method used in many other theorem provers.

(3) Rewrite rules for the current theory $\mathcal{T}$, or for certain theories $\mathcal{T}_0$ for which IMPS can find interpretations from $\mathcal{T}_0$ into $\mathcal{T}$).

The framework for applying rewrite rules is entirely general, and uses pattern matching and substitution in a familiar way. By contrast, the transforms that perform algebraic manipulation use specially coded procedures;

they are applied to expressions in a way that may not be easily expressed as patterns. Nevertheless, their validity, like the validity of rewrite rules, depends on theorems, many of which are universal, unconditional equalities (e.g., associative and commutative laws).

13.4 Algebraic Processors

Instead of providing a fixed set transforms for manipulating expressions in a limited class of algebraic structures, we have implemented a facility for automatically generating and installing such transforms for general classes of algebraic structures. This is possible since algorithmically the transforms are the same in many cases; only the names have to be changed, so to speak. The algebraic manipulation transform is one component of a data structure called an *algebraic processor.*

An algebraic processor has either two or three associated transforms. There is one transform for handling terms of the form $f(a, b)$ where f is one of the algebraic operations in the processor definition. A separate transform handles equalities of algebraic expressions. Frequently, when the structure has an ordering relation, there is a third transform for handling inequalities of algebraic expressions.

When an algebraic processor is created, the system generates a set of formulas which must hold in order for its manipulations to be valid. The processor's transforms are installed in a theory $\mathcal{T}$ only if the system can ascertain that each of these formulas is a theorem of $\mathcal{T}$.

The user builds an algebraic processor using a specification such as

```
(def-algebraic-processor FIELD-ALGEBRAIC-PROCESSOR
  (language fields)
  (base ((operations
          (+ +_kk)
          (* *_kk)
          (- -_kk)
          (zero o_kk)
          (unit i_kk))
         commutes)))
```

This specification has the effect of building a processor with a transform for handling terms of the form $f(a, b)$, where f is one the arithmetic functions $+_{\mathbf{K}}, *_{\mathbf{K}}, -_{\mathbf{K}}$. For example, the entry `(+ +_kk)` tells the transform to treat

the function symbol $+_{\mathbf{K}}$ as addition. The transforms are installed in the theory **fields** by the specification

```
(def-theory-processors FIELDS
 (algebraic-simplifier
   (field-algebraic-processor *_kk +_kk -_kk))
 (algebraic-term-comparator field-algebraic-processor))
```

When this form is evaluated, IMPS checks that the conditions for doing algebraic manipulation do indeed hold in the theory **fields**.

13.5 Hints and Cautions

(1) Though simplification is a powerful user command, there are several reasons why you do not want to invoke it indiscriminately at every sequent node:

- If the sequent node context is very large or the expression is large (a fact which may be masked by the presence of quasi-constructors), simplification may take considerable time.
- Simplification may transform an expression into a form which may be easier for IMPS to work with, but may be harder for you to visualize. For instance, the expression $x - y$ simplifies to $x + [-1]y$.

(2) The simplifier may at times appear to exhibit "temperamental" behavior. At one extreme, it may seem to the user that the simplifier does too little or that it has ignored an assumption that would have immediately grounded the sequent. At the other extreme, the simplifier may do too much work, reducing the assertion in unintuitive or unusable ways.

What may be especially infuriating to the user is that the presence of additional assumptions may actually cause the simplifier to return a much less usable answer. For instance, in the theory **h-o-real-arithmetic**, the simplifier will reduce the formula

$$\forall x, y{:}\mathbf{R}, 1 < 2xy \supset 0 < 2xy$$

to T. However,

$$\forall x, y{:}\mathbf{R}, 0 < y \wedge 1 < 2xy \supset 0 < 2xy$$

simplifies to

$$\forall x, y{:}\mathbf{R}, 0 < y \wedge 1 < 2xy \supset 0 < 2x.$$

This discrepancy results from its factoring the positive y in $0 < 2xy$ before applying the decision procedure for linear inequalities.

Chapter 14

Macetes

In addition to its simplifier, IMPS also provides a separate mechanism for manipulating formal expressions by applying theorems. Unlike simplification, it is under the control of the user. The basic idea is that some expression manipulation procedures—for instance, conditional rewrite rules—are in fact direct applications of theorems. Other expression manipulation procedures can be formed from existing ones using a simple combination language. These expression manipulation procedures are called *macetes.*[1] Formally, a macete is a function M which takes as arguments a context Γ and an expression e (called the *source expression*) and returns another expression $M(\Gamma, e)$ (called the *replacement expression*). In general, the replacement expression does not have to be equal to the source expression.

Macetes serve two somewhat different purposes:

- They are a straightforward way to apply a theorem or a collection of theorems to a sequent in a deduction graph.
- They are also a mechanism for computing with theorems.

14.1 Classification

A macete can be *atomic* or *compound*. Compound macetes are obtained by applying a *macete constructor* to a list of one or more macetes.

Atomic macetes are classified as:

[1] In Portuguese, a macete is an ingenious trick. "Macete" is pronounced with a soft c. Do not pronounce it as in "Government supporters dropped confetti on the man waving the machete."

- *Schematic macetes.* A schematic macete is one in which the replacement expression is obtained from the source expression by matching and substitution. Schematic macetes can be thought of as conditional rewrite rules.

- *Procedural macetes.* These are macetes which compute the replacement expression by a more complicated procedure. There are only a few such procedures:

  – simplify

  – beta-reduce

  – beta-reduce-insistently

  Having these procedures as macetes allows you to build more powerful and useful compound macetes.

Macetes can be also classified according to the relation between the source and replacement expressions.

- *Bidirectional.* Whatever the context and source expression, the replacement expression is quasi-equal to the source expression, that is, given any legitimate model in which the context assumptions are true, either both source expression and replacement expression have no value or their value is identical.

- *Backward-directional.* Whatever the context and source expression, the replacement expression is quasi-equal to or implies the source expression. Notice that implication only makes sense for formulas of sort $*$.

- *Nondirectional.* There are contexts and source expressions in which the target expression is neither quasi-equal to nor implies the source expression.

14.2 Atomic Macetes

14.2.1 Schematic Macetes

A schematic macete can be associated to any formula in two ways depending on the kind of matching procedure used. IMPS has two general forms of matching called *expression matching* and *translation matching.*

Let us consider first the case of expression matching. To any IMPS formula we can associate:

- A source pattern. This is an expression s.
- A replacement pattern. This is an expression r.
- A list of condition patterns. This is a list of expressions $c_1, \cdots, c_n$.
- A direction specifier: this specifier can be either *bidirectional* or *backward-directional.*

Suppose the formula is $\forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi$, where φ itself is not a universal formula. The patterns are extracted from φ depending on the form of φ as described by the following table:

Form	Source	Replacement	Conditions	Spec.
$\varphi_1 \wedge \cdots \wedge \varphi_n \supset t_1 \simeq t_2\,\dagger$	t_1	t_2	$\varphi_1, \ldots, \varphi_n$	Bid.
$\varphi_1 \wedge \cdots \wedge \varphi_n \supset t_1 = t_2\,\dagger$	t_1	t_2	$\varphi_1, \ldots, \varphi_n$	Bid.
$\varphi_1 \wedge \cdots \wedge \varphi_n \supset \psi_1 \Leftrightarrow \psi_2$	ψ_1	ψ_2	$\varphi_1, \ldots, \varphi_n$	Bid.
$\varphi \supset \psi$	ψ	φ	None	Back.
$t_1 \simeq t_2\,\dagger$	t_1	t_2	None	Bid.
$t_1 = t_2\,\dagger$	t_1	t_2	None	Bid.
$\psi_1 \Leftrightarrow \psi_2$	ψ_1	ψ_2	None	Bid.
$\neg\psi$	ψ	F	None	Bid.
ψ	ψ	T	None	Bid.

Notes

- If the replacement pattern φ according to the above table is of sort $*$ and contains free variables $y_1, \ldots, y_m$ not occurring freely in the source pattern ψ, then consider instead the replacement pattern

$$\exists y_1{:}\delta_1, \ldots, y_m{:}\delta_n, \varphi.$$

- Nested implications $\varphi_1 \supset (\varphi_2 \supset (\cdots \varphi_n \supset \psi))$ are treated as single implications $\varphi_1 \wedge \cdots \wedge \varphi_n \supset \psi$ when applying the above table.
- In those cases marked with †, if t_2 contains free variables which do not occur freely in t_1, then consider instead the source pattern $t_1 = t_2$ or $t_1 \simeq t_2$ and replacement pattern $\varphi_1 \wedge \cdots \wedge \varphi_n$ or T.

We now describe how a schematic macete M is obtained from the source, replacement, condition patterns, and direction specifier of a formula. Given a context Γ and an expression t, the result of applying M to Γ and t is determined as follows: Let $\mathcal{P}$ be the set of paths $l = (l_1, \ldots, l_n)$ to subexpressions e_l of t which satisfy all of the following conditions:

- If the directional specifier is backward-directional, the parity of l must be $+1$; otherwise, no condition is imposed on the path.
- The source pattern s matches e_l. Thus there is a substitution $\rho_l = \{\langle x_1, t_1\rangle, \ldots, \langle x_n, t_n\rangle\}$ such that ρ_l applied to s gives e_l.
- The validity of all the convergence requirements of the substitution ρ_l and all the formulas $\rho_l(c_i)$ in the context Γ localized at the path l are called the *minor premises* generated by the macete application. These formulas may or may not generate applicability conditions:
  - If the macete is being applied with the accumulation of minor premises, then these formulas are posted as new sequents and must be proven separately. No new applicability conditions are generated in this case.
  - Otherwise, the minor premises must be recognized as valid by the simplifier.
- No smaller path $(l_1, \ldots, l_k)$ satisfies the preceding conditions.

The replacement formula is obtained by replacing each subexpression e_l by $\rho_l(e_l)$. Macetes obtained in this way are called *elementary macetes.*

Whenever you enter a theorem into a theory, the associated schematic macete is automatically installed, so that you can use it in deductions by hitting the key `m`.

The case of translation matching is similar, but instead of considering only convergence requirements, we need to insure that the translation component of the translation match is a theory interpretation. Macetes obtained in this way are called *transportable macetes.* The use of a theorem as a transportable macete must be entered into its usage list when the theorem is entered.

14.2.2 Nondirectional Macetes

Frequently it is useful to construct a schematic macete from an arbitrary formula by using a matching and substitution procedure which does not obey

the variable capturing protections and other restrictions of normal matching. We refer to this as *nullary* matching and substitution. For instance, to apply a theorem which involves quantification over functions, one often has to build a lambda-abstraction of a sub-expression. We shall see later on how this can be done.

14.3 Compound Macetes

Compound macetes are built from atomic macetes and other compound macetes using macete constructors. The constructors are:

- *Series:* This constructor takes a list $M_1, \ldots, M_n$ of macetes as arguments and yields a macete Series$(M_1, \ldots, M_n)$ which applies every macete in the list once, from left to right. Thus given a context Γ and an expression e,

$$\text{Series}(M_1, \ldots, M_n)(\Gamma, e) = M_n(\Gamma, M_{n-1}(\Gamma, \ldots, M_1(\Gamma, e)\ldots))$$

- *Sequential:* This constructor takes any number of macetes as arguments. It works like the series constructor, except that when a macete fails to change an expression, subsequent macetes on the list are not applied.

- *Parallel:* This constructor takes a list $M_1, \ldots, M_n$ of macetes as arguments and yields a macete Parallel$(M_1, \ldots, M_n)$ which applies macetes in the list from left to right until a macete changes the expression. Thus given a context Γ and an expression e,

$$\text{Parallel}(M_1, \ldots, M_n)(\Gamma, e) = M_i(\Gamma, e)$$

  provided $M_i(\Gamma, e) \neq e$ and $M_j(\Gamma, e) = e$ for all $1 \leq j < i$.

- *Repeat:* Takes any number of macetes $(M_1, \ldots, M_n)$ as arguments and yields a macete in the following way: The series composition M of $(M_1, \ldots, M_n)$ is applied repeatedly to the expression e until no more change occurs. It may possibly compute forever. Thus the macete terminates if

$$M(\Gamma, M(\Gamma, \ldots, M(\Gamma, e)\ldots))$$

  stabilizes to some expression t. This expression is the value of

$$\text{Repeat}(M_1, \ldots, M_n)(\Gamma, e).$$

- *Sound:* This constructor takes a list M_1, M_2, M_3 of three macetes and yields a macete characterized as follows: Given a context Γ and an expression e, $\text{Sound}(M_1, M_2, M_3)(\Gamma, e)$ is
  - $M_1(\Gamma, e)$ if $M_3(\Gamma, M_1(\Gamma, e))$ is alpha-equivalent to $M_2(\Gamma, e)$
  - e otherwise.
- *Without-minor-premises:* This constructor takes a single argument macete M. It returns a new macete which applies M without minor premises. That is, macete substitution is only done when the macete requirements are satisfied.

Now whether the resulting macete is bidirectional or backward-directional, can be determined by considering the bidirectionality or backward-directionality each of the components.

- Series, sequential, parallel, repeat, and without-minor-premises macete constructors: If all the M_i are bidirectional macetes then the compound macete
$$\text{Constructor}(M_1, \ldots, M_n)(\Gamma, e)$$
is bidirectional. Similarly, if all the arguments M_i are backward-directional macetes then the compound macete is also backward-directional.
- Sound macete constructor: If M_2, M_3 are bidirectional, then the compound macete
$$\text{Sound}(M_1, M_2, M_3)(\Gamma, e)$$
is always bidirectional, regardless of M_1. If M_2 is backward-directional and M_3 is bidirectional, then the compound macete is backward-directional. In all other cases, the compound macete is nondirectional.

14.4 Building Macetes

Schematic macetes are for the most part installed by the system when theorems installed.

(1) Elementary macetes are automatically installed when a theorem is added to a theory. The name of the installed macete is the same as the name of the theorem which generates it.

(2) Transportable macetes are installed when the **transportable-macete** modifier is present in the usage list of a theorem or axiom. Moreover, in this case, the name of the installed macete is the name of theorem which generates it with the prefix **tr%**.

Macetes are also automatically built when atomic sort and constant definitions are made by the user.

(1) For directly defined constants, the macete corresponding to the constant defining equation is installed. This macete is named

constant-name-**equation**_*theory-name*.

If the usage list of the definition includes the **rewrite** modifier and the defined constant is a function, the constant defining equation in applied form is also include as a macete named

constant-name-**applied-equation**_*theory-name*.

(2) For recursively defined constants, a number of macetes are installed:

(a) The equation axioms named
constant-name-**equation**_*theory-name*

(b) The minimality axiom named
definition-name-**strong-minimality**_*theory-name*

(c) The minimality theorem named
definition-name-**minimality**_*theory-name*

(3) For defined atomic sorts a number of macetes are installed:

(a) The sort defining axiom named
sort-name-**defining-axiom**_*theory-name*.

(b) Some auxiliary theorems named
sort-name-**in-quasi-sort**_*theory-name*
and
sort-name-**in-quasi-sort-domain**_*theory-name*

Nondirectional schematic macetes usually are not associated to theorems. The form **def-schematic-macete** can be used for building schematic macetes of all kinds. You can build compound macetes using the form **def-compound-macete**.

14.5 Applicable Macetes

A macete is "applicable" to a formula if applying the macete modifies the formula in some way. The only sure-fire way of determining whether a macete is applicable, is to actually apply the macete and see the result. However, there are simple heuristic conditions which can be used to quickly determine that some macetes are not applicable. For example, for an elementary bidirectional macete, if the left hand side of the macete does not match any subexpression of the formula, then clearly the macete it is not applicable. For transportable macetes other similar inapplicability conditions are used. Macetes are tabulated in such a way that it is possible to quickly throw out inapplicable ones using these heuristic conditions. IMPS exploits this to provide users with a dynamic help facility for selecting macetes. The use of this facility is explained more extensively in Chapter 4.

14.6 Hints and Cautions

(1) Conceptually the macete mechanism is extremely simple. Users however, should not underestimate its power, usefulness, and flexibility. The utility of macetes hinges on three facts:

- The way a theorem is converted into a macete depends on the syntactic form of the the theorem as explained earlier in this section. In particular, it is important that theorems which are implications can be used as macetes along with those that are conditional equalities. Moreover, the "conversion algorithm" is based on a number of heuristics that in our experience work very well.
- Macetes can be combined using macete constructors. It has been our experience that carefully building a collection of compound macetes is an important part of developing a theory.
- Macetes are tabulated in such a way that the macetes which are applicable to a given formula can be retrieved very effectively and displayed to the user in a menu. In situations where over 500 macetes are installed, the menu usually contains less than twenty entries and takes less than ten seconds to compute.

(2) Though there are various modes of applying macetes, the default mode, **with minor premises**, is the one you want to use in most

cases. The without-minor-premises macete constructor is often useful for building compound macetes to be used in this mode.

Chapter 15

The Iota Constructor

15.1 Motivation

The **iota** or ι constructor is a definite description operator for objects of kind ι. Given a variable x of sort α of kind ι and a unary predicate φ,

$$\iota x{:}\alpha, \varphi(x)$$

denotes the *unique* x that satisfies φ if there is such an x and is undefined otherwise. For example,

$$\iota x{:}\alpha, 0 \leq x \wedge x * x = 2$$

denotes $\sqrt{2}$ but

$$\iota x{:}\alpha, x * x = 2$$

is undefined.

The **iota** constructor is very useful for specifying functions, especially partial functions. For example, ordinary division of sort $[\mathbf{R}, \mathbf{R}, \mathbf{R}]$ (which is undefined whenever its second argument is 0) can be defined from the times function $*$ by the lambda-expression

$$\lambda x, y{:}\mathbf{R}, \iota z{:}\mathbf{R}, x * z = y.$$

Of course, this kind of definite description operator is only possible in a logic that admits undefined expressions.

15.2 Reasoning with Iota

An *iota-expression* is any expression which begins with the **iota** constructor (without any abbreviations by quasi-constructors). Proving a sequent with an assertion containing an iota-expression can be tricky. The key idea is to reduce the sequent to a new sequent with one less iota-expression. This is called "eliminating iota." There are two commands and one macete for eliminating iota, each of which is discussed below in a separate subsection.

15.2.1 eliminate-defined-iota-expression

This command, which is described in Chapter 18, is the best iota-elimination tool in IMPS. Generally, you should use it whenever you want to eliminate an iota-expression whose definedness is implied by the sequent's context. It can also be used effectively in some cases on an iota-expression E whose definedness is not implied by the sequent's context. This is done by applying **case-split** with $E{\downarrow} \vee \neg(E{\downarrow})$, followed by applying **eliminate-defined-iota-expression** on the $E{\downarrow}$ case.

The **eliminate-defined-iota-expression** command is multi-inference; it adds about 20 new sequent nodes to a deduction graph.

15.2.2 eliminate-iota

This command is also described in Chapter 18. It is a single-inference command that is applicable to just atomic and negated atomic formulas. You should use it when you want to eliminate an iota-expression whose definedness is not implied by the sequent's context.

15.2.3 eliminate-iota-macete

This is a compound macete with built-in abstraction specified by:

```
(def-compound-macete ELIMINATE-IOTA-MACETE
  (sequential
   iota-abstraction-macete
   (series
    tr%defined-iota-expression-elimination
    tr%negated-defined-iota-expression-elimination
    tr%left-iota-expression-equation-elimination
    tr%right-iota-expression-equation-elimination)
```

```
beta-reduce))
```

The four elimination theorems are in the file **$IMPS/theories/generic-theories/iota.t**. You should try **eliminate-iota-macete** when the assertion has one of the following forms:

(1) $(\iota x{:}\alpha, \varphi(x))\downarrow$

(2) $\neg((\iota x{:}\alpha, \varphi(x))\downarrow)$

(3) $(\iota x{:}\alpha, \varphi(x)) = E$

(4) $(\iota x{:}\alpha, \varphi(x)) \neq E$

(5) $E = (\iota x{:}\alpha, \varphi(x))$

(6) $E \neq (\iota x{:}\alpha, \varphi(x))$

(7) $(\iota x{:}\alpha, \varphi(x)) \simeq E$

(8) $E \simeq (\iota x{:}\alpha, \varphi(x))$

For forms (1)–(6), the macete can be applied immediately, but for forms (7) and (8), the command **insistent-direct-inference** must be applied first. Beware that, for various reasons, **eliminate-iota-macete** may fail to do anything, in which case you should use one of the two iota-elimination commands.

15.3 Hints and Cautions

(1) Suppose E is an expression of sort α. Sometimes it is convenient to have a new expression E' of sort β (where $\tau(\beta) = \tau(\alpha)$ but $\beta \neq \alpha$) such that E and E' have the same denotation. Such an expression can be easily constructed from E and β using **iota**: $\iota x{:}\beta, x = E$ is quasi-equal to E and is of sort β. For example, $\iota x{:}\mathbf{N}, x = 2$ denotes the *natural number* 2. It is important to mention here that the IMPS simplifier reduces an iota-expression of the form $\iota x{:}\beta, x = E$ to the conditional $\mathrm{if}(E \downarrow \beta, E, \bot_\beta)$, and further to E or $\bot_\beta$ if it can decide the formula $E \downarrow \beta$.

(2) Suppose f is a function constant defined to be

$$\lambda x{:}\alpha, \iota y{:}\beta, \varphi(x, y)$$

such that the following "existence implies uniqueness" theorem holds for φ:

$$\forall x{:}\alpha, (\exists y{:}\beta, \varphi(x,y)) \supset (\exists! y{:}\beta, \varphi(x,y)).$$

By virtue of this theorem, there is an "iota-free characterization" of f,

$$\forall x{:}\alpha, y{:}\beta, f(x) = y \equiv \varphi(x,y),$$

which enables one to prove a formula $f(a) = b$ by showing only existence—without employing any of the aforementioned tools for eliminating **iota**.

As an example, consider the limit operator **lim%rr** on sequences of reals defined by:

```
(def-constant LIM%RR
  "lambda(s:[zz,rr], iota(x:rr, forall(eps:rr,
     0<eps
      implies
     forsome(n:zz, forall(p:zz,
       n<=p implies abs(x-s(p))<=eps)))))"
  (theory h-o-real-arithmetic))
```

Since a sequence always has at most one limit point, there is an iota-free characterization of **lim%rr**:

```
(def-theorem CHARACTERIZATION-OF-REAL-LIMIT
  "forall(s:[zz,rr],
     x:rr,lim%rr(s)=x
      iff
     forall(eps:rr,
       0<eps
        implies
       forsome(n:zz, forall(p:zz,
         n<=p implies abs(x-s(p))<=eps))))"
  (theory h-o-real-arithmetic))
```

(3) It is rarely a good idea to unfold a constant defined as an iota-expression, if it has an iota-free characterization. Instead, the following steps are suggested. For concreteness, let us assume the constant we

are unfolding is a function constant f as in the previous item. Our aim is to reduce all occurrences of f to occurrences of the form $f(a) = b$,

- If all occurrences in the formula are of the form $f(a) = b$ then apply the iota-free characterization of f as a macete.
- Otherwise, for every occurrence $f(a)$, where a is a free variable, cut with the formula

  $$\exists y{:}\beta, f(a) = y.$$

  Each application of **cut** adds two new sequent nodes to the deduction graph:
  - The cut major premise, which is the original sequent plus the cut formula as another assumption.
  - The cut minor premise, which has the cut formula as the assertion. The minor premise can be proved by instantiating y with $f(a)$. To proceed with this part of the proof you will need to establish the definedness of $f(a)$.

  For a precise description of the sequent nodes added by cut, see the documentation on the command **cut-with-single-formula**.
- To deal with the major premise, do an antecedent inference on the cut formula to remove its existential quantifier. This yields a new goal with an assumption of the form $f(a) = v$.
- Backchain repeatedly on $f(a) = v$. This replaces all occurrences of $f(a)$ with v.
- Incorporate the antecedent $f(a) = v$ and apply the iota-free characterization of f as a macete. This should replace all such occurrences with some condition not involving f, and more importantly, not involving **iota**.

(4) The **iota-p** constructor is a definite description operator for objects of kind $*$. It has a different semantics than **iota**: if there is no unique x of sort α satisfying a unary predicate φ, then $\iota_{\mathrm{p}} x{:}\alpha, \varphi$ is *defined* (like all expressions of kind $*$) but has the value $\mathrm{F}_{\tau(\alpha)}$. There is currently no support in IMPS for reasoning about the **iota-p**. This should not cause you any concern because **iota-p** has little practical value.

Chapter 16

Syntax: Parsing and Printing

The purpose of this chapter is threefold:

(1) To explain the relation between syntax and IMPS expressions.

(2) To explain the default syntax, which we refer to as the "string syntax," and briefly to explain how you can extend or modify this syntax.

(3) To explain how you can redefine the syntax altogether.[1]

16.1 Expressions

As explained in Chapter 7, expressions are variables, constants, and compound expressions formed by applying constructors to other expressions. Expressions can thus be represented as Lisp s-expressions:

- Formal symbols (i.e., constants or variables) correspond to atoms.
- Compound expressions correspond to certain lists of the form

  $$(\text{constructor } c_1 \ \cdots \ c_n)$$

  where each c_i is itself an expression.

In the implementation, an expression is a data structure which also caches a large amount of information such as the free and bound variables of the expression, the constants contained in the expression, the home language

[1] IMPS is designed so that it does not impose a syntax on the user.

of the expression, and so on. For the IMPS user, an expression is typically something which can be represented as text, for instance $\int_a^b \ln x dx$. The correspondence of an expression as a data structure to an external representation for input or output is determined by the user syntax which is employed. IMPS allows multiple user syntaxes, so for example, the syntax that is used for reading in expressions (usually text) may be different from the syntax used to display expressions (which could be text or text interspersed with TEX commands which would make the output suitable for input to the TEX processor.) This flexible arrangement means users can freely change from one syntax to another, even during the course of an interactive proof.

16.2 Controlling Syntax

The process of building an IMPS expression from user input can be broken up into two steps:

(1) Parsing a string into an s-expression **s**.

(2) Building an IMPS expression from the s-expression **s**.

Similarly, the process of displaying an expression also consists of two parts:

(1) Building an s-expression **s** from an IMPS expression.

(2) Providing some representation of **s** as a string.

The phases "string to s-expression" for reading and "s-expression to string" for printing are under direct user control. They can be modified by resetting the following switches:

- `(imps-reader)` is a T procedure which takes a language and a port and returns an s-expression. If `proc` is a procedure of this type, you can reset the **imps-reader** switch by evaluating

  ```
  (set (imps-reader) proc)
  ```

- `(imps-printer)` is a procedure which takes an s-expression and a port and prints something to this port. If `proc` is a procedure of this type, you can reset the **imps-printer** switch by evaluating

  ```
  (set (imps-printer) proc)
  ```

To write your own reader and printer procedures, you will have to do some programming in the T language. However, some reader and printer procedures are particularly easy to write. For example to write your own reader and printer for s-expression syntax, you could evaluate the following s-expressions:

```
(set (imps-reader)
     (lambda (l port) (read port)))

(set (imps-printer)
     (lambda (s-exp port) (pretty-print s-exp port)))
```

In the next section we describe the default syntax which is similar to the syntax of some computer algebra systems.

16.3 String Syntax

We begin with some notation: if "Term" is a class of strings,

Term*	=	zero or more comma-separated members of Term
Term+	=	one or more comma-separated members of Term.

A string is *accepted* by the string syntax exactly when IMPS can build an s-expression **s** from it. Otherwise, IMPS will give you a parsing error. However, to say the string is accepted does not mean that the system can build an IMPS expression from **s**. A number of things could go wrong:

- An atom in **s** has no sort specification, so that IMPS assumes it is the name of a constant, but no such constant exists in the language.
- The arguments of a function are of the wrong type, or there are too many or too few arguments.
- A constructor was given components of the wrong type, or the number of components is wrong.

The tables 16.1, 16.2, and 16.3 are a description of those sequences of characters accepted by the string syntax. This description is fairly close to a BNF grammar, but fails to be one since some of the expression categories (such as the category of binary infix operators or the category of quasi-constructors) are only specified partially or not at all. This is because these

expression categories depend on parts of the state of the IMPS system, which are likely to change from user to user.

We also want to stress that these tables say nothing about how strings are parsed into nested lists (and much less about what the strings mean mathematically.) For example, one cannot say how argument associations are disambiguated from these tables alone. The table of operator precedences in the next section specifies this information. The syntax tables are thus of limited value and should only be consulted to get a very rough idea of what expressions look like.

16.4 Parsing

The process of building an s-expression from an input string, itself breaks up into two parts:

- Tokenization of the input string. This can be thought of as dividing the input string into a list of substrings called *tokens*.
- Parsing of the list of tokens. This can be thought of as transforming a linear structure of tokens into a tree structure of tokens.

The table below gives some examples of how a string s is translated into an s-expression $\Phi(s)$ according to the string syntax.

String	Reads
p_1 `and` $\cdots$ `and` p_n	(and $\Phi(p_1)$ $\cdots$ $\Phi(p_n)$)
p_1 `or` $\cdots$ `or` p_n	(or $\Phi(p_1)$ $\cdots$ $\Phi(p_n)$)
`if`(x, y, z)	(if $\Phi(x)$ $\Phi(y)$ $\Phi(z)$)
`if`(x, y, z, w, v)	(if $\Phi(x)$ $\Phi(y)$ (if $\Phi(z)$ $\Phi(w)$ $\Phi(v)$))
`#`(t, s)	(defined-in $\Phi(t)$ $\Phi(s)$)
`#`(t)	(is-defined $\Phi(t)$)
$x + y$	(apply $+$ $\Phi(x)$ $\Phi(y)$)
$x - y$	(apply $-$ $\Phi(x)$ $\Phi(y)$)
$f(x_1, \ldots, x_n)$	(apply $\Phi(f)$ $\Phi(x_1)$ $\cdots$ $\Phi(x_n)$)

To build an s-expression from the input tokens, it is necessary to disambiguate the tokens which are operators[2] from the tokens which are arguments. The association of tokens to operators is determined by the operator

[2]Note that the word "operator" is being used here in a different sense than it is used in logic: an operator is simply a token which is parsed as the first element of a list.

Exp ::=	Aexp	
	Oxp(Oxp*)	Function application
	Exp BInfxOp Exp	Binary infix application
	(Oxp)	Parenthesized Oxp
	Exp oo Exp	Composition
	Oxp=Oxp	
	Oxp==Oxp	
	`not` Exp	
	Exp `and` Exp	
	Exp `or` Exp	
	Exp `implies` Exp	
	Exp `iff` Exp	
	Binder(Binding* , Exp)	Binders (including `with`)
	`iota`(Atom:Sort, Exp)	
	`if`(Exp, Exp, Exp+)	Requires an odd number of Exp
	Qc{Exp+}	Prefix quasi-constructor Exp
	Qc{Exp+,Sort}	Prefix quasi-constructor Exp
	?Sort	Undefined from sort
	`#`(Exp,Sort)	Sort definedness assertion
	`#`(Exp)	Definedness assertion
Oxp ::=	Exp	Operator or expression
	PrefxOp	
	LogfxOp	
	BInfxOp	
	PostfxOp	
Aexp ::=	Num	Numerical constants
	Atom	Variables & other constants

Table 16.1: Description of String Syntax 1

Sort	::=	Asort [Sort,Sort+]	These are base sorts N-ary function sort constructor.

Binder	::=	`lambda`
		`forall`
		`forsome`
		`with`	

Binding	::=	Sortdecl+	
Sortdecl	::=	Atom+ : Sort	
PrefxOp	::=	`comb` `lim`	Combinatorial coefficients Limit operator for sequences
LogfxOp	::=	`-`	

BInfxOp	::=	`*`
		`+`
		`<`
		`<=`
		`i-in`
		`i-subset`	

PostfxOp	::=	`!`	

Table 16.2: Description of String Syntax 2

Atom	::=	TextChar *followed by* TextChar's *or* Digit's SpecialSeq SpecialSeq *followed by* `_` *and* TextChar's *or* Digit's
TextChar	::=	`a` | $\cdots$ | `z` | `_` | `%` | `$` | `&`
Digit	::=	`0` | $\cdots$ | `9`
SpecialSeq	::=	`^` | `*` | `+` | `-` | `^^` | `**` | `++` | `!` | `<` | `<=` | `>` | `>=`

Table 16.3: Description of Atoms

Operator	Binding
`+`	100
`-`	110
`*`	120
`/`	121
`^`	140
`!`	160
`=`	80
`==`	80
`i-subset`	101
`i-in`	101
`>`	80
`>=`	80
`<`	80
`<=`	80
`not`	70
`iff`	65
`implies`	59
`and`	60
`or`	50

Table 16.4: Operator Binding Powers

binding powers. For example, the string `x+2*3` is parsed as

$$(\text{apply} + x\ (\text{apply} * 2\ 3))$$

because the operator `*` has a higher binding power than `+`. Table 16.4 is a list of the bindings of some common operators.

16.5 Modifying the String Syntax

The string syntax can be modified or extended using the def-forms **def-parse-syntax** and **def-print-syntax**. The use of def-forms is documented in Chapter 17.

16.6 Hints and Cautions

(1) A *parsing error* is indicated by an error message

```
**Error: Parsing error:
```

This means IMPS was unable to build an s-expression from your input. IMPS will provide some indication as to where the error occurred. Following are some examples of common parsing errors:

```
> (qr "with(a,b:rr a=b)")
** Error: Parsing error: Illegal token A encountered.
with(a,b:rr a <<== =b).
```

This error occurred because the variable sort specifications have to be followed by a comma.

```
> (qr "forall(a,b:rr,a=b")
** Error: Parsing error: Expecting ":" or "," or ")".
forall(a,b:rr,a=b <<== .
```

This error results from a missing right parenthesis.

(2) An error message of one of the following kinds is usually an indication that the current theory (this is the value of the switch **(current-theory)**) is different from what you expected. This typically happens when you begin an interactive proof of a formula without having set the current theory to its desired value.

```
> (qr "with(x:rr,x=1)")
** Error: QUANTIFIER-DECODE-VAR-SUB-LIST:
RR is not a sort in
#{IMPS-basic-language 12: THE-KERNEL-LANGUAGE}.
```

This error message indicates that IMPS was unable to build an expression in the current theory because there is no sort named **rr**.

```
> (qr "with(x:ind, x+x)")
** Error: SEXP->EXPRESSION-1:
Cannot locate + in
#{IMPS-basic-language 12: THE-KERNEL-LANGUAGE}.
```

This error message indicates that IMPS was unable to build an expression in the current theory because there is no constant named +. To remedy an error of the above type, reset the **(current-theory)** switch by using the menu.

Part III

Reference Manual

Chapter 17

The IMPS Special Forms

In this chapter, we document a number of user forms for defining and modifying IMPS objects, for loading sections and files, and for presenting expressions. We will use the following template to describe each one of these forms.

Positional Arguments. This is a list of arguments that must occur in the order given, and must precede the modifier arguments and the keyword arguments. All IMPS definition forms have a name as first argument or, a list of names as first argument.

Modifier Arguments. Each modifier argument is a single symbol called the *modifier.*

Keyword Arguments. Each keyword argument is a list whose first element is a symbol called the *keyword.* Keyword arguments have one of two forms:

- (`keyword` *arg-component*).
- (`keyword` $\mathit{arg\text{-}component}_1$ ... $\mathit{arg\text{-}component}_n$).

Note: By default, nearly every special form in this chapter is allowed to have a keyword argument of the form

(`syntax`*syntax-name*)

where *syntax-name* is the name of a syntax (e.g., `string-syntax` or `sexp-syntax`) which specifies what syntax to use when reading the

form. If this keyword argument is missing, the current syntax is used when reading the form.

Modifier and keyword arguments can occur in any order.

Description. A brief description of the definition form.

Remarks: Hints or observations that we think you should find useful.

Examples. Some example forms.

17.1 Creating Objects

def-algebraic-processor

Positional Arguments:

- *processor-name.*

Modifier Arguments:

- `cancellative`. This modifier argument tells the simplifier to do multiplicative cancellation.

Keyword Arguments:

- `(language` *language-name*`)`. Required.
- `(base` *spec-forms*`)`. Required.
- `(exponent` *spec-forms*`)`. Instructs processor how to simplify exponents if there are any.
- `(coefficient` *spec-forms*`)`. Instructs processor how to simplify coefficients for modules.

In all of the above cases, *spec-forms* is either a name of an algebraic processor or is a list with keyword arguments and modifier arguments of its own. The keyword arguments for *spec-forms* are:

- `(scalars` *numerical-type*`)`.
- `(operations` *operation-alist*$_1$ ... *operation-alist*$_n$`)`. *operation-alist* is a list of entries (*operation-type operator*). *operation-type* is one of the symbols `+ * - / ^ sub zero unit`. *operator* is the name of a constant in the processor language.

The modifier arguments for *spec-forms* are:

- `use-numerals-for-ground-terms`.
- `commutes`.

Description:

This form builds an object called a *processor* for algebraic simplification. Processors are used by IMPS to generate algebraic and order simplification procedures used by the simplifier for the following purposes:

- To utilize theory-specific information of an algebraic nature in performing simplification. For example, the expression $x + x - x$ should simplify to x, or $x < x + 1$ should simplify to true.

- To reduce any expression built from formal constants whose value can be algebraically computed from the values of the components. Thus the expression $2 + 3$ should reduce to 5.

In order for simplification to reduce expressions involving only formal constants (for instance, $2 + 3$), a processor associates such IMPS terms to executable T expressions involving algebraic procedures and objects from some data type **S**. This type is specified by the `scalars` declaration of the **def-algebraic-processor** form. We refer to this data type as a *numerical type.* Each formal constant o which denotes an algebraic function (such as $+$ or $-$) is associated to an operation $f(o)$ on the data type. This correspondence is specified by the `operations` keyword of processor. Finally, certain terms must be associated to elements of **S**. This is accomplished by a predicate and two mappings constructed by IMPS when the processor is built:

- A predicate that singles out certain terms as numerical constants. These are called *numerical* terms. For the processor used in the theory **h-o-real-arithmetic** the numerical constants are those formal constants whose names are rational numbers.

- A function $m \mapsto T(m)$ which maps certain elements of **S** to terms.

- A function $t \mapsto N(t)$ which maps certain terms to **S**.

The simplifier uses these functions to reduce expressions containing numerical terms, according to the following rule: If f denotes a processor operation, and s, t denote numerical terms, and $o(s, t)$ is defined, then $o(s, t)$ is replaced by $T(f(o)(N(s), N(t)))$.

Remarks:

Evaluating a **def-algebraic-processor** form by itself will not affect theory simplification in any way. To add algebraic simplification to a theory, evaluate the form **def-theory-processors**. It is also important to note that an algebraic processor can only be installed in the simplifier when certain theorems generated by the processor are valid in the theory.

Examples:

```
(def-algebraic-processor RR-ALGEBRAIC-PROCESSOR
  (language numerical-structures)
  (base ((scalars *rational-type*)
         (operations
          (+ +)
          (* *)
          (- -)
          (^ ^)
          (/ /)
          (sub sub))
         use-numerals-for-ground-terms
         commutes)))

(def-algebraic-processor VECTOR-SPACE-ALGEBRAIC-PROCESSOR
  (language real-vector-language)
  (base ((operations
          (+ ++)
          (* **)
          (zero v0))))
  (coefficient rr-algebraic-processor))
```

def-atomic-sort

Positional Arguments:

- *sort-name.* The name of the atomic sort to be defined. *sort-name* also is the name of the newly built definition.

- *quasi-sort-string.*

Keyword Arguments:

- `(theory` *theory-name*`)`. Required.
- `(usages` $symbol_1$ ... $symbol_n$`)`.
- `(witness` *witness-expr-string*`)`.

Description:

This form creates a new atomic sort σ called *sort-name* from a unary predicate p of sort $[\sigma, *]$ specified by *quasi-sort-string* and adds a set of new theorems with usage list $symbol_1, \ldots, symbol_n$. The new atomic sort and new theorems are added to the theory T named *theory-name* provided:

- *sort-name* is not the name of any current atomic sort of T or of a structural supertheory of T; and
- the formula $\exists x{:}\sigma, p(x)$ is known to be a theorem of T.

If there is a `witness` argument, the systems tries to verify condition (2) by simplifying $p(a)$, where a is the expression specified by *witness-expr-string*.

Examples:

```
(def-atomic-sort NN
  "lambda(x:zz, 0<=x)"
  (theory h-o-real-arithmetic)
  (witness "0"))
```

def-bnf

Positional Arguments:

- *name*. A symbol to serve as the name of the theory that will be created as a consequence of this def-form. In addition, a second implementation object with this name is also created; it represents the BNF definition itself together with the axioms and theorems that it generates.

Keyword Arguments:

- `(theory` *theory-name*`)`. Required. The theory that will be conservatively extended by adding the new datatype. We will refer to this theory as the *underlying* theory.

- `(base-type` *type-name*`)`. Required. The name of the new data type.

- `(sorts (`$subsort_1$ $enclosing\text{-}sort_1$`) ... (`$subsort_n$ $enclosing\text{-}sort_n$`))`.

  Subsorts of the new datatype. All these subsorts must be included within the new base type. Thus, the enclosing sort of the first subsort must be the new base type, and the enclosing sort of a later subsort must be either the base type or a previously mentioned subsort of it.

- `(atoms (`$atom_1$ $sort_1$`) ... (`$atom_n$ $sort_n$`))`. Individual constants declared to belong to the base type (or one of its subsorts).

- `(constructors (`$constr_1$ `(`$sort_1$ `...`$sort_n$`) (selectors` s_1 `...`s_{n-1}`)))`.

  Function constants declared to construct elements of the new type (or its subsorts). Each range sort must be the new base type or one of its subsorts. Each domain sort may belong to the underlying theory, or alternatively it may be the new base or one of its subsorts. It may not be a higher sort involving the new type.

- `(sort-inclusions (`$subsort_1$ $supersort_1$`) ... (`$subsort_n$ $supersort_n$`))`. Declare that every element of *subsort* is also a member of *supersort*. This is not needed when *subsort* is a syntactic subsort of *supersort* in the sense that there are zero or more intermediate sorts $\alpha_1, \ldots, \alpha_n$ such that the enclosing sort of *subsort* is α_1, the enclosing sort of α_n is *supersort*, and the enclosing sort of α_i is α_{i+1} for each i with $1 \leq i < n$.

Description:

The purpose of the `def-bnf` form is to define a new recursive data type to be added (as a conservative extension) to some existing theory. The members of the new data type are the atoms in the declaration together with the objects returned ("constructed") by the constructor functions. We will use the τ to refer to the base type; symbols such as α_i will stand for any sort included in the base type (such as the base sort τ itself); β_i will be used for any sort of the underlying theory.

The logical content of a data type definition of this kind is determined by two main ideas:

- **No Confusion.** If c_0 and c_1 are two different constructor functions, then the range of c_0 is disjoint from the range of c_1. If a and b are two atoms, then $a \neq b$. Moreover $a \notin \text{ran}(c_0)$.
- **No Junk.** Everything in τ apart from the atoms is generated by the constructor functions. This is in fact a form of induction. It allows us to infer that a property holds of all members of the data type on two assumptions:

  (1) The property holds true of each atom;

  (2) The property holds true of a value returned by any constructor function c, assuming that it holds true of those arguments that belong to τ (rather than to some sort β of the underlying theory).

The implementation ensures that the resulting theory will be a model conservative extension of the underlying theory. That is, given any model M of the underlying theory, we can augment M with a new domain to interpret τ and provide interpretations of the new vocabulary so that the resulting M' is a model of extended theory.

In the case where τ contains no subtype, it is clear that to ensure conservatism, it is enough to establish that τ will be non-empty. The syntax of the declaration suffices to determine whether this condition is met: either there must be an atom, or else at least one constructor function must take all of its arguments from types β_i of the underlying theory.

If τ does have subtypes, the implementation must establish that each subtype is non-empty. That will hold if there is an ordering of the new sorts α_i such that, for each α_i, at least one of the following conditions holds:

(1) There is an atom declared to be of sort α_i; or

(2) There is a sort α_j occurring earlier in the ordering, and α_j is declared to be included within α_i; or

(3) There is at least one constructor c declared with range sort α_i, and every domain sort of c is either some β_i belonging to the underlying theory or some α_j occurring earlier in the ordering that α_i.

When this condition is not met, the implementation raises an error and identifies the uninhabited sort (or sorts) for the user.

Axioms Generated. The BNF mechanism generates six categories of axioms:

Constructor definedness A constructor is well-defined for every selection of arguments of the syntactically declared sorts.

Disjointness If a and b are atoms and c_0 and c_1 are constructors, then $a \neq b$, $a \notin \mathrm{ran}(c_0)$, and $\mathrm{ran}(c_0)$ is disjoint from $\mathrm{ran}(c_1)$.

Selector/constructor If s is the ith selector for the constructor c, then $s(c(\vec{x})) = x_i$. Moreover, if $s(y)\downarrow$, then $y = c(\vec{x})$ for some $\vec{x}$.

Sort Inclusions $\forall x{:}\alpha_0, x \downarrow \alpha_1$, when α_0 is specified as included within α_1.

Induction The type τ is the smallest containing the atoms and closed under the constructor functions in the sense given above ("No Junk").

Case Analysis If $x \downarrow \alpha$, where α is the new type or one of its subsorts, then one of the following holds:

(1) x is one of the atoms declared with sort α;

(2) x is in the range of some constructor declared with range α;

(3) $x \downarrow \alpha'$, where either α is the enclosing sort of α', or else the inclusion of α' within α is declared as a sort inclusion.

Strictly speaking this should be a theorem rather than an axiom, as it follows from the induction principle.

Primitive Recursion. A data type introduced via the BNF mechanism justifies a schema for defining functions by primitive recursion. Roughly speaking, a function g of sort $\tau \to \sigma$ is uniquely determined if a sort σ is given, together with the following:

- For each atom, a value of sort σ;
- For each constructor c, with sort

$$\alpha_1 \times \ldots \times \alpha_n \times \beta_1 \times \ldots \times \beta_m \to \alpha_0,$$

a functional Φ_c of sort

$$(\sigma \times \ldots \times \sigma \times \beta_1 \times \ldots \times \beta_m \to \sigma)$$

is given. The primitive recursively defined function g will have the property that $g(c(x_1, \ldots, x_n, y_1, \ldots, y_m))$ is quasi-equal to

$$\Phi_c(g(x_1), \ldots, g(x_n), y_1, \ldots, y_m).$$

Thus in effect Φ_c says how to combine the results of the recursive calls (for arguments in the primary type) with the values of the parameters (from sorts β_i of the underlying theory).

Additional Theorems. A number of consequences of the axioms are installed.

Examples:

The simplest example possible is:

```
(def-bnf nat
  (theory pure-generic-theory-1)
  (base-type nat)
  (atoms (zero nat))
  (constructors (succ (nat nat) (selectors pred))))
```

The axioms generated from this definition are displayed in Figure 17.1 on page 177.

As a more complicated example, consider a programming language with phrases of three kinds, namely identifiers, expressions, and statements. The data type to be introduced corresponds to set of phrases of all three syntactic categories; each syntactic category will be represented by a subsort. A BNF for the language might take the form given in Figure 17.2, with x ranging over numbers and s ranging over strings. A corresponding IMPS def form is shown in Figure 17.3, where it is assumed that the underlying theory *String-theory* contains a sort *string* representing ASCII strings, as well as containing **R**.

bnf data type nat:

Constructor definedness axioms: Theorem **succ-definedness_nat** is:
$\text{total}(\text{succ}, [\text{nat} \rightharpoonup \text{nat}])$.

Disjointness axioms: Theorem **succ-zero-distinctness_nat** is:
$\forall y_0{:}\text{nat}, \quad \neg(\text{succ}(y_0) = \text{zero})$.

Selector constructor axioms: Theorem **succ-pred_nat** is:
$\forall y_0{:}\text{nat}, \quad \text{pred}(\text{succ}(y_0)) = y_0$.

Selector undefinedness axioms: Theorem **pred-definedness_nat** is:
$\forall x{:}\text{nat}, \quad \text{pred}(x) \downarrow\supset (\exists y_0{:}\text{nat}, \quad \text{succ}(y_0) = x)$.

Sort case axioms: Theorem **nat-cases_nat** is:
$\forall e{:}\text{nat}, \quad e = \text{zero} \vee \text{succ}(\text{pred}(e)) = e$.

Induction axiom: Theorem **nat-induction_nat** is:
$\forall \varphi{:}\text{nat} \rightharpoonup *,$
$\quad (\forall x_0{:}\text{nat}, \quad \varphi(x_0))$
$\quad \iff$
$\quad (\varphi(\text{zero}) \wedge (\forall \text{y\_succ}_0{:}\text{nat}, \quad \varphi(\text{y\_succ}_0) \supset \varphi(\text{succ}(\text{y\_succ}_0))))$.

Figure 17.1: Axioms for Nat, as Introduced by BNF

S ::=	I:=E	| while E do S od	| if E then S_1 else S_2
	| skip		
E ::=	I	| E_1 + E_2	| $E_1 * E_2$
	| Num(x)		
I ::=	Ide(s)		

Figure 17.2: BNF Syntax for a Small Programming Language

```
(def-bnf pl-syntax
 (theory string-theory)
 (base-type phrase)
 (sorts (statement phrase)
        (expression phrase)
        (identifier expression))
 (atoms (skip statement))
 (constructors
  (assign   "[identifier, expression, statement]"
            (selectors lvar rexp))
  (while    "[expression, statement, statement]"
            (selectors while%test body))
  (if       "[expression, statement, statement, statement]"
            (selectors if%test consequent alternative))
  (plus     "[expression, expression, expression]"
            (selectors first%addend second%addend))
  (times    "[expression, expression, expression]"
            (selectors first%factor second%factor))
  (numconst "[rr, expression]"
            (selectors num%value))
  (ide      "[string, identifier]"
            (selectors ide%name))))
```

Figure 17.3: IMPS def-bnf Form for Programming Language Syntax

def-cartesian-product

Positional Arguments:

- *name*. A symbol to name the product sort.
- (*sort-name*$_1$ ... *sort-name*$_m$). *sort-name* is the name of a sort α.

Keyword Arguments:

- (`constructor` *constructor-name*).
- (`accessors` *accessor-name*$_1$ ... *accessor-name*$_n$).
- (`theory` *theory-name*). Required.

Description:

This form creates a new atomic sort σ called *name*. This sort can be regarded as the cartesian product of the sorts $\alpha_1 \ldots \alpha_m$. The sort actually created is a subsort of the function sort

$$\alpha_1 \times \cdots \times \alpha_m \rightharpoonup \text{unit\_sort}$$

- constructor-name is the name of the canonical "n-tuple" mapping:

$$\alpha_1 \times \cdots \times \alpha_m \rightharpoonup \sigma.$$

- accessor-name$_i$ is the name of the canonical projection onto the i-th coordinate: $\sigma \rightharpoonup \alpha_i$.

Examples:

```
(def-cartesian-product complex
  (rr rr)
  (constructor make%complex)
  (accessors real imaginary)
  (theory h-o-real-arithmetic))
```

def-compound-macete

Positional Arguments:

- *name.*
- *spec.* A compound macete specification is defined recursively as either:
  - A symbol, *macete-name.*
  - A list of the form:
    * `(series` $spec_1$ ... $spec_n$`)`.
    * `(repeat` $spec_1$ ... $spec_n$`)`.
    * `(sequential` $spec_1$ ... $spec_n$`)`.
    * `(sound` $spec_1$ $spec_2$ $spec_3$`)`.
    * `(parallel` $spec_1$ ... $spec_n$`)`.
    * `(without-minor-premises` *spec*`)`.

    where *spec* is itself a macete specification.

Keyword Arguments:

None.

Description:

This form adds a compound macete with the given name and components. See the section on macetes for the semantics of macetes.

Examples:

```
(def-compound-macete FRACTIONAL-EXPRESSION-MANIPULATION
  (repeat
    beta-reduce
    addition-of-fractions
    multiplication-of-fractions
    multiplication-of-fractions-left
    multiplication-of-fractions-right
    equality-of-fractions
    left-denominator-removal-for-equalities
    right-denominator-removal-for-equalities
    strict-inequality-of-fractions
```

```
    right-denominator-removal-for-strict-inequalities
    left-denominator-removal-for-strict-inequalities
    inequality-of-fractions
    right-denominator-removal-for-inequalities
    left-denominator-removal-for-inequalities))

(def-compound-macete ELIMINATE-IOTA-MACETE
  (sequential
    (sound
     tr%abstraction-for-iota-body
     beta-reduce
     beta-reduce)
    (series
     tr%defined-iota-expression-elimination
     tr%negated-defined-iota-expression-elimination
     tr%left-iota-expression-equation-elimination
     tr%right-iota-expression-equation-elimination)
    beta-reduce))
```

def-constant

Positional Arguments:

- *constant-name*. The name of the constant to be defined. *constant-name* is also the name of the newly-built definition.
- *defining-expr-string*.

Keyword Arguments:

- `(theory` *theory-name*`)`. Required.
- `(sort` *sort-string*`)`.
- `(usages` $symbol_1$ ... $symbol_n$`)`.

Description:

This form creates a new constant c called *constant-name* and a new axiom of the form $c = e$ with usage list $symbol_1, \ldots, symbol_n$, where e is the expression

specified by *defining-expr-string*. The sort σ of c is the sort specified by *sort-string* if there is a `sort` argument and otherwise is the sort of e. The new constant and new axiom are added to the theory T named *theory-name* provided:

- *constant-name* is not the name of any current constant of T or of a structural supertheory of T; and
- the formula $(e \downarrow \sigma)$ is known to be a theorem of T.

Examples:

```
(def-constant ABS
  "lambda(r:rr, if(0<=r,r,-r))"
  (theory h-o-real-arithmetic))

(def-constant >
  "lambda(x,y:rr,y<x)"
  (theory h-o-real-arithmetic)
  (usages rewrite))

(def-constant POWER%OF%TWO
  "lambda(x:zz,2^x)"
  (sort "[zz,zz]")
  (theory h-o-real-arithmetic))
```

def-imported-rewrite-rules

Positional Arguments:

- *theory-name.*

Keyword Arguments:

- `(source-theory` *source-theory-name*`)`.
- `(source-theories` *source-theory-name*$_1$ ... *source-theory-name*$_n$`)`.

Description:

This form imports into the theory named *theory-name* all the transportable rewrite rules installed in the theory named *source-theory-name* (or in the theories named *source-theory-name*$_1, \ldots,$ *source-theory-name*$_n$).

def-inductor

Positional Arguments:

- *inductor-name.*
- *induction-principle.* Can be a string representing the induction principle in the current syntax or a name of a theorem.

Keyword Arguments:

- `(theory` *theory-name*`)`. Required.
- `(translation` *name*`)`. *name* is the name of a theory interpretation. If this keyword is present, the formula used in constructing the induction principle is actually the translation of *induction-principle*.
- `(base-case-hook` *name*`)`. *name* is the name of a macete or a command.
- `(induction-step-hook` *name*`)`. *name* is the name of a macete or a command.
- `(dont-unfold` $name_1$ ... $name_n$`)`. *name* is the name of a recursive constant in the theory or the symbol `#t`. This form instructs the inductor not to unfold the recursive constant named *name* when processing the induction step. If `#t` is specified, then no recursive definitions will be unfolded in the induction step.

Remarks:

The induction principle used for constructing an inductor must satisfy a number of requirements.

- The induction principle must be a theorem.
- The induction principle must be a universally quantified biconditional of the form

  $$\forall p{:}[\alpha, *], x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n, \varphi \Leftrightarrow \psi_0 \wedge \psi_1.$$

  with none of the sorts $\alpha_1, \ldots, \alpha_n$ being of kind $*$. ψ_0 is referred to as the *base case* and ψ_1 is the *induction step.*

Description:

This form builds an inductor from the formula φ represented by *induction-principle* or if the `translation` keyword is present, from the translation of φ under the translation named by *name*. The base-case and induction-step hooks are macetes or commands that provide additional handling for the base and induction cases.

Examples:

```
(def-inductor TRIVIAL-INTEGER-INDUCTOR
  "forall(s:[zz,prop],m:zz,
     forall(t:zz,m<=t implies s(t))
      iff
     (s(m) and
     forall(t:zz,m<=t implies (s(t) implies s(t+1)))))"
  (theory h-o-real-arithmetic))

(def-inductor NATURAL-NUMBER-INDUCTOR
  "forall(s:[nn,prop],
     forall(t:nn, s(t))
      iff
     (s(0) and forall(t:nn,s(t) implies s(t+1))))"
  (theory h-o-real-arithmetic))

(def-inductor INTEGER-INDUCTOR
  induct
  (theory h-o-real-arithmetic)
  (base-case-hook unfold-defined-constants-repeatedly))

(def-inductor SM-INDUCT
  "forall(p:[state,prop],
     forall(s:state, accessible(s) implies p(s))
      iff
     (forall(s:state, initial(s) implies p(s)) and
      forall(s_1, s_2:state, a:action,
        (accessible(s_1) and p(s_1) and tr(s_1, a, s_2))
         implies
        p(s_2))))"
  (theory h-o-real-arithmetic)
```

```
  (base-case-hook
    eliminate-nonrecursive-definitions-and-simplify))

(def-inductor SEQUENCE-INDUCTOR
  "forall(p:[[nn,ind_1],prop],
     forall(s:[nn,ind_1], f_seq_q(s) implies p(s))
      iff
     (p(nil(ind_1)) and
      forall(s:[nn,ind_1], e:ind_1,
        f_seq_q(s) and p(s) implies p(cons{e,s}))))"
 (theory sequences)
 (base-case-hook simplify))
```

def-language

Positional Arguments:

- *language-name*. A symbol.

Keyword Arguments:

- `(embedded-languages` $name_1$ ... $name_n$`)`. *name* is the name of a language or theory.
- `(embedded-language` *name*`)`. *name* is the name of a language or theory.
- `(base-types` $type\text{-}name_1$ ... $type\text{-}name_n$`)`.
- `(sorts` $sort\text{-}spec_1$ ... $sort\text{-}spec_n$`)`. *sort-spec* is a list of the form `(`*sort enclosing-sort*`)`, called a sort-specification. *sort* must be a symbol, and *enclosing-sort* must satisfy one of the following:
  - It occurs in a previous sort specification.
  - It is a sort in one of the embedded languages.
  - It is a compound sort which can be built up from sorts of the preceding kinds.
- `(extensible` $type\text{-}sort\text{-}alist_1$ ... $type\text{-}sort\text{-}alist_n$`)`. *type-sort-alist* is a list of the form `(`*numerical-type-name sort*`)`. *numerical-type-name* is the name of a *numerical type,* that is, a class of objects defined in

T. For example, `*integer-type*` and `*rational-type*` are names of numerical types. This keyword argument is used for defining self-extending languages. This is a language that incorporates new formal symbols when the expression reader encounters numerical objects of the given type.

- `(constants` *constant-spec*$_1$ ... *constant-spec*$_n$`)`. *constant-spec* is a list `(`*constant-name sort-or-compound-sort*`)`.

Description:

This form builds a language with name *language-name* satisfying the following properties:

- This language contains the sorts and constants given by the sort and constant specification subforms.
- It contains the languages named *name*$_i$ as sublanguages.
- If the language contains the `extensible` keyword, then the language may contain an infinite number of constants; these constants are in one-to-one correspondence with elements of the numerical types present in the *type-sort-alist*.

Examples:

```
(def-language NUMERICAL-STRUCTURES
  (extensible (*integer-type* zz) (*rational-type* qq))
  (sorts (rr ind) (qq rr) (zz qq))
  (constants
   (^ (rr zz rr))
   (+ (rr rr rr))
   (* (rr rr rr))
   (sub (rr rr rr))
   (- (rr rr))
   (/ (rr rr rr))
   (<= (rr rr prop))
   (< (rr rr prop))))

(def-language METRIC-SPACES-LANGUAGE
  (embedded-language h-o-real-arithmetic)
```

```
  (base-types pp)
  (constants
   (dist (pp pp rr))))

(def-language REAL-VECTOR-LANGUAGE
  (embedded-language h-o-real-arithmetic)
  (base-types uu)
  (constants
   (++ (uu uu uu))
   (v0 uu)
   (** (rr uu uu))))

(def-language HAM-SANDWICH-LANGUAGE
  (base-types sandwich)
  (embedded-language h-o-real-arithmetic)
  (constants
   (ham%sandwich  sandwich)
   (tuna%sandwich sandwich)
   (refrigerator (sandwich unit%sort))))
```

def-order-processor

Positional Arguments:

- *name*.

Keyword Arguments:

- `(algebraic-processor` *name*`)`. The name of an algebraic processor.
- `(operations` *operation-alist*$_1$ ... *operation-alist*$_n$`)`. *operation-alist* is a list of entries `(`*operation-type operator*`)`, where *operation-type* is one of: `<`, `<=`. *operator* is the name of a constant in the language of the associated algebraic processor.
- `(discrete-sorts` *sort-spec*$_1$ ... *sort-spec*$_n$`)`.

Description:

This form builds an object called an *order processor* for simplification of formulas involving order relations. Processors are used by IMPS to generate

algebraic and order simplification procedures used by the simplifier for the following purposes:

- To utilize theory-specific information of an algebraic nature in performing simplification. For example, the expression $x + x - x$ should simplify to x, or $x < x + 1$ should simplify to true.
- To reduce any formula built from formal constants whose value can be algebraically computed from the values of the components. Thus the expression $2 < 3$ should reduce to true.

Examples:

```
(def-order-processor RR-ORDER
  (algebraic-processor rr-algebraic-processor)
  (operations (< <) (<= <=))
  (discrete-sorts zz))
```

def-primitive-recursive-constant

Positional Arguments:

- *constant name*: The name of the function (or predicate) constant to be introduced by the def-form.
- *bnf*: The BNF object furnishing the primitive recursive schema for the definition (see **def-bnf**). The base type of this BNF will also be the domain of the constant.

Keyword Arguments:

- `(theory` *theory-name*`)`: Required. The theory to which the defined constant will be added. This may be an extension of the theory of the BNF with respect to which the recursion is performed.
- `(range-sort` *range-sort-form*`)`: Required. This specifies the range of the function or predicator being introduced; since the domain will be the base sort of the BNF, this fixes the sort of the constant.
- `(`*atom-name value*`)`: Required, one for each BNF atom. This specifies one base case of the recursive definition.

- (*constructor-name var-names operator-body*): Required, one for each BNF constructor. This specifies one recursive step of the recursive definition. The *operator-body* specifies how the results of the recursive subcalls should be combined (possibly with parameters not belonging to the datatype) to produce the value of the recursive operator for an argument of this form.

Description:

Each BNF is automatically equipped with a principle of definition by structural (primitive) recursion. A **def-primitive-recursive-constant** form instantiates this principle by stipulating the intended range sort, the values for atoms of the BNF data type, and how the recursively defined operator combines its results on the arguments of each datatype constructor.

Examples:

See, for instance, the *compiler* exercise file.

def-quasi-constructor

Positional Arguments:

- *name.* The name of the quasi-constructor.
- *lambda-expr-string.*

Keyword Arguments:

- `(language` *language-name*`)`. Required. *language-name* may also be the name of a theory.
- `(fixed-theories` *theory-name*$_1$ ... *theory-name*$_n$`)`.

Description:

This form builds a quasi-constructor named *name* from the schema specified by *lambda-expr-string*, which is a lambda-expression in the language named *language-name*. The sorts in the theories named *theory-name*$_1$, ... *theory-name*$_n$ are held fixed.

Examples:

```
(def-quasi-constructor PREDICATE-TO-INDICATOR
  "lambda(s:[uu,prop],
     lambda(x:uu, if(s(x),an%individual, ?unit%sort)))"
  (language indicators)
  (fixed-theories the-kernel-theory))

(def-quasi-constructor GROUP
  "lambda(a:sets[gg],
          mul%:[gg,gg,gg],
          e%:gg,
          inv%:[gg,gg],
     forall(x,y:gg,
       (x in a and y in a) implies mul%(x,y) in a) and
     e% in a and
     forall(x:gg, x in gg% implies inv%(x) in gg%) and
     forall(x:gg, x in a implies mul%(e%,x)=x) and
     forall(x:gg, x in a implies mul%(x,e%)=x) and
     forall(x:gg, x in a implies mul%(inv%(x),x)=e%) and
     forall(x:gg, x in a implies mul%(x,inv%(x))=e%) and
     forall(x,y,z:gg,
       (x in a) and (y in a) and (z in a)
        implies
       mul%(mul%(x,y),z) = mul%(x,mul%(y,z))))"
  (language groups))
```

def-record-theory

Positional Arguments:

- *theory-name*. The name of a theory.

Keyword Arguments:

- (`type` *symbol*). Required.
- (`accessors` *accessor-spec*$_1$ ... *accessor-spec*$_n$). *accessor-spec* is a list

  (*accessor-name target-sort*).

Description:

NO DESCRIPTION.

Examples:

```
(def-record-theory ENTITIES-AND-LEVELS
  (type access)
  (accessors
    (read "prop")
    (write "prop")))
```

def-recursive-constant

Positional Arguments:

- *names.* The name or lists of names of the constant or constants to be defined.
- *defining-functional-strings.* A string or list of strings specifying the defining functionals of the definition.

Keyword Arguments:

- `(theory` *theory-name*`)`. Required.
- `(usages` $symbol_1$ ... $symbol_n$`)`.
- `(definition-name` *def-name*`)`. The name of the definition.

Description:

Let T be the theory named *theory-name*; $\textit{const-name}_1,\ldots,\textit{const-name}_n$ be the names given by *names*; and $f_1,\ldots,f_n$ be the functionals specified by the strings in *defining-functional-strings*. This form creates a list of new constants $c_1,\ldots,c_n$ called $\textit{const-name}_1,\ldots,\textit{const-name}_n$ and a set of new axioms with usage list $\textit{symbol}_1,\ldots,\textit{symbol}_n$. The axioms say that $c_1,\ldots,c_n$ are a minimal fixed point of the family $f_1,\ldots,f_n$ of functionals. The new constants and new axioms are added to T provided:

- each $\textit{const-name}_i$ is not the name of any current constant of T or of a structural supertheory of T; and

- each functional f_i is known to be monotone in T.

If the `definition-name` keyword is present, the name of the definition is *def-name*; otherwise, it is *const-name*$_1$.

Examples:

```
(def-recursive-constant SUM
  "lambda(sigma:[zz,zz,[zz,rr],rr],
     lambda(m,n:zz,f:[zz,rr],
       if(m<=n,sigma(m,n-1,f)+f(n),0)))"
  (theory h-o-real-arithmetic)
  (definition-name sum))

(def-recursive-constant (EVEN ODD)
  ("lambda(even,odd:[nn,prop],
      lambda(n:nn, if_form(n=0, truth, odd(n-1))))"
   "lambda(even,odd:[nn,prop],
      lambda(n:nn, if_form(n=0, falsehood, even(n-1))))")
  (theory h-o-real-arithmetic)
  (definition-name even-odd))

(def-recursive-constant OMEGA%EMBEDDING
  "lambda(f:[nn,nn], a:sets[nn],
     lambda(k:nn,
       if(k=0,
          iota(n:nn, n in a and
            forall(m:nn, m<n implies not(m in a))),
          lambda(z:nn,
            iota(n:nn, n in a and z<n
              forall(m:nn, (z<m and m<n) implies
                (not(m in a)))))(f(k-1)))))"
  (theory h-o-real-arithmetic)
  (definition-name omega%embedding))
```

def-renamer

Positional Arguments:

- *name.* A symbol naming the renamer.

Keyword Arguments:

- `(pairs` $pair_1$ ... $pair_n$`)`. *pair* is of the form `(`*old-name new-name*`)`.

Description:

This form defines a T procedure named *name* which, given a symbol x, returns the symbol y if $(x\ y)$ is one of the pairs, and otherwise returns x.

Examples:

```
(def-renamer SB-RENAMER
  (pairs
   (last%a%index last%b%index)
   (a%inf b%inf)
   (a%even b%even)
   (a%odd b%odd)))
```

def-schematic-macete

Positional Arguments:

- *macete-name.* A symbol naming the schematic macete.
- *formula.* A string representing the formula using the current syntax.

Modifier Arguments:

- `null`. The presence of this modifier means the macete is nondirectional. This means that schematic macete which is built uses a matching and substitution procedure which does not obey the variable capturing protections and other restrictions of normal matching and substitution.
- `transportable`. The presence of this modifier means the macete is transportable.

Keyword Arguments:

- `(theory` *theory-language-name*`)`. Required. The name of the language in which expression should be read in.

Examples:

```
(def-schematic-macete ABSTRACTION-FOR-DIFF-PROD
  "with(a,b:rr,y:rr,
     diff(lambda(x:rr,a*b))(y)=
     diff(lambda(x:rr,
       lambda(x:rr,a)(x)*lambda(x:rr,b)(x)))(y))"
  null
 (theory calculus-theory))
```

def-script

Positional Arguments:

- *script-name*. The name of the new command being created.
- *N*. The number of arguments required by the script.
- (*form*$_1$ ... *form*$_m$). A proof script consisting of 0 or more forms.

Keyword Arguments:

- (`retrieval-protocol` *proc*). *proc* is the name of an Emacs procedure which IMPS uses to interactively request command arguments from the user. The default procedure is **general-argument-retrieval-protocol** which requires the user to supply all the arguments in the minibuffer.
- (`applicability-recognizer` *proc*). *proc* is the name of a T predicate or is the symbol `#t`. The inclusion of this argument will cause the command name to appear as a selection in the command menu whenever the current sequent node satisfies the predicate. Supplying `#t` as an argument is equivalent to supplying a predicate which is always true.

Description:

This form is used to build a new command named *script-name*. This new command can be used in either interactive mode or in script mode. The script defining the command is a list of forms, each of which is a *command*

form or a *keyword form*. See a description of the proof script language in Chapter 12.

Examples:

```
(def-script EPSILON/N-ARGUMENT 1
  ((instantiate-universal-antecedent
     "with(p:prop, forall(eps:rr, 0<eps implies p))" ($1))))
```

def-section

Positional Arguments:

- *section-name*.

Keyword Arguments:

- `(component-sections` *section-name*$_1$ ... *section-name*$_n$`)`.
- `(files` *file-spec*$_1$ ... *file-spec*$_n$`)`.

Description:

This form builds a section which, when loaded, loads the sections with names *section-name*$_1, \ldots,$ *section-name*$_n$ and then loads the files with specifications *file-spec*$_1, \ldots,$ *file-spec*$_n$. A section can be loaded with the form **load-section**.

Examples:

```
(def-section BASIC-REAL-ARITHMETIC
  (component-sections reals)
  (files
   (imps theories/reals/some-lemmas)
   (imps theories/reals/arithmetic-macetes)
   (imps theories/reals/number-theory)))

(def-section FUNDAMENTAL-COUNTING-THEOREM
  (component-sections
   basic-cardinality
   basic-group-theory)
```

```
(files
 (imps theories/groups/group-cardinality)
 (imps theories/groups/counting-theorem)))
```

def-sublanguage

Positional Arguments:

- *sublanguage-name.*

Keyword Arguments:

- `(superlanguage` *superlanguage-name*`)`. Required. The name of the language containing the sublanguage.
- `(languages` *language-name*$_1$ ... *language-name*$_n$`)`.
- `(sorts` *sort-name*$_1$ ... *sort-name*$_n$`)`.
- `(constants` *constant-name*$_1$ ... *constant-name*$_n$`)`.

Description:

This form finds the smallest sublanguage of the language L named *superlanguage-name* which contains the sublanguages of L named *language-name*$_1$, ..., *language-name*$_n$; the atomic sorts of L named *sort-name*$_1$, ..., *sort-name*$_n$; and the constants of L named *constant-name*$_1, \ldots,$ *constant-name*$_n$.

Each of *superlanguage-name*, *language-name*$_1, \ldots,$ *language-name*$_n$ may be the name of a theory instead of a language.

Examples:

```
(def-sublanguage REAL-VECTOR-LANGUAGE
  (superlanguage vector-spaces-over-rr-language)
  (languages h-o-real-arithmetic)
  (sorts uu)
  (constants ++ v0 **))
```

def-theorem

Positional Arguments:

- *name.* The name of the theorem which may be `()`.
- *formula-spec.* A string representing the formula using the current syntax or the name of a theorem.

Modifier Arguments:

- `reverse`.
- `lemma`. Not loaded when quick-loading.

Keyword Arguments:

- `(theory` *theory-name*`)`. Required. The name of the theory in which to install the theorem.
- `(usages` $symbol_1$ ... $symbol_n$`)`.
- `(translation` *translation-name*`)`. The name of a theory interpretation.
- `(macete` *macete-name*`)`. The name of a bidirectional macete.
- `(home-theory` *home-theory-name*`)`. The name of the home theory of the formula specified by *formula-spec*.
- `(proof` *proof-spec*`)`.

Description:

This form installs a theorem in three steps. Let Φ be the theory interpretation named *translation-name* when there is a `translation` argument, and let M be the bidirectional macete named *macete-name* when there is a `macete` argument. Also, let T be the theory named *theory-name*, and let H be the theory named by *home-theory-name*. If there is no `home-theory` argument, then H is the source theory of Φ if there is a `translation` argument and otherwise $H = T$. Finally, let φ be the formula specified by *formula-spec* in H.

Step 1. IMPS verifies that φ is a theorem of H. If *proof-spec* is the symbol `existing-theorem`, IMPS checks to see that φ is alpha-equivalent to some existing theorem of H. If the theorem is being installed in "batch mode" (see Section 12.3.1), IMPS verifies that φ is a theorem of H by running the script *proof-spec*. Otherwise, IMPS simply checks that φ is a formula in H, and it is assumed that the user has verified that φ is a theorem of H.

Step 2. The theorem φ' to be installed is generated from φ as follows:

- If there is no `translation` or `macete` argument and $T = H$, then $\varphi' = \varphi$.
- If there is no `translation` or `macete` argument and T is a subtheory of H, φ' is a formula of T which is the result of generalizing the constants of H which are not in T (assuming each sort of T is also a sort of H).
- If there is a `translation` argument but no `macete` argument, then φ' is the translation of φ by Φ.
- If there is a `macete` argument but no `translation` argument, then φ' is the result of applying M to φ.
- If there is both a `translation` and `macete` argument, then φ' is the result of applying M to the translation of φ by Φ.

Step 3. φ' is installed as a theorem in T with usage list $symbol_1$, ..., $symbol_n$.

If the modifier argument `reverse` is present, evaluating this form is equivalent to evaluating the form without `reverse` followed by evaluating the form again without `reverse` but with:

- *name* changed to the concatenation of `rev%` with *name*, and
- *formula-spec* changed to a string which specifies the "reverse" of the formula specified by *formula-spec*.

In other words, the `reverse` argument allows you to install both a theorem and its reverse. It is very useful for building a macete (or transportable macete) and its reverse by evaluating one form. Of course, it would not be wise to use the `reverse` argument when `rewrite` is a usage.

Examples:

```
(def-theorem ABS-IS-TOTAL
  "total_q(abs,[rr,rr])"
  (theory h-o-real-arithmetic)
  (usages d-r-convergence)
  (proof
   (
    (unfold-single-defined-constant (0) abs)
    insistent-direct-inference
    beta-reduce-repeatedly
    )))

(def-theorem RIGHT-CANCELLATION-LAW
  left-cancellation-law
  (theory groups)
  (translation mul-reverse)
  (proof existing theorem))

(def-theorem FUNDAMENTAL-COUNTING-THEOREM
  "f_indic_q{gg%subgroup}
     implies
   f_card{gg%subgroup} =
     f_card{stabilizer(zeta)}*f_card{orbit(zeta)}"
  ;; "forall(zeta:uu,
  ;;    f_indic_q{gg%subgroup}
  ;;      implies
  ;;    f_card{gg%subgroup} =
  ;;      f_card{stabilizer(zeta)}*f_card{orbit(zeta)})"
  (theory group-actions)
  (home-theory counting-theorem-theory))
```

def-theory

Positional Arguments:

- *theory-name.*

Keyword Arguments:

- `(language` *language-name*`)`.

- `(component-theories` *theory-name*$_1$ ... *theory-name*$_n$`)`.
- `(axioms` *axiom-spec*$_1$ ... *axiom-spec*$_n$`)`. *axiom-spec* is a list

  (*name formula-string usage*$_1$ ... *usage*$_n$)

  where *name*, a symbol, is the optional name of the axiom and *formula-string* is a string representing the axiom in the syntax of the language of the theory.
- `(distinct-constants` *distinct*$_1$ ... *distinct*$_n$`)`. *distinct* is a list

  (*constant-name*$_1$ ... *constant-name*$_n$).

Description:

This form builds a theory named *theory-name*. The language of this theory is the union of the language *language-name* and the languages of the component theories. The axioms of the theory are the formulas represented by the strings in the list *axiom-spec*. The *distinct-constants* list implicitly adds new axioms asserting that the constants in each list *distinct* are not equal.

Examples:

```
(def-theory METRIC-SPACES
  (component-theories h-o-real-arithmetic)
  (language metric-spaces-language)
  (axioms
    (positivity-of-distance
      "forall(x,y:pp, 0<=dist(x,y))"
      transportable-macete)
    (point-separation-for-distance
      "forall(x,y:pp, x=y iff dist(x,y)=0)"
      transportable-macete)
    (symmetry-of-distance
      "forall(x,y:pp, dist(x,y) = dist(y,x))"
      transportable-macete)
    (triangle-inequality-for-distance
      "forall(x,y,z:pp, dist(x,z)<=dist(x,y)+dist(y,z))")))
```

```
(def-theory VECTOR-SPACES-OVER-RR
  (language real-vector-language)
  (component-theories generic-theory-1)
  (axioms
    ("forall(x,y,z:uu, (x++y)++z=x++(y++z))")
    ("forall(x,y:uu, x++y=y++x)")
    ("forall(x:uu, x++v0=x)")
    ("forall(x,y:rr, z:uu, (x*y)**z=x**(y**z))")
    ("forall(x,y:uu,a:rr,a**(x++y)=(a**x)++(a**y))")
    ("forall(a,b:rr, x:uu,(a+b)**x=(a**x)++(b**x))")
    ("forall(x:uu, 0**x=v0)")
    ("forall(x:uu, 1**x=x)")))
```

def-theory-ensemble

Positional Arguments:

- *ensemble-name*. The name of the theory ensemble. This will also be the name of the base theory unless the `base-theory` keyword argument is provided.

Keyword Arguments:

- (`base-theory` *theory-name*) *theory-name* is the name of the base theory of the ensemble. If this keyword argument is not included then the base theory is that theory with the name *ensemble-name*.

- (`fixed-theories` $theory_1$ ... $theory_n$). The theories listed are not replicated when the theory replicas and multiples are built. If this argument is not provided, then the fixed theories are those in the global fixed theories list at the time the form is evaluated.

- (`replica-renamer` *proc-name*). *proc-name* is the name of a procedure of one integer argument, used to name sorts and constants of theory replicas. The default procedure is to subscript the corresponding sort and constant of the base theory.

Description:

This form builds a theory ensemble such that

(1) The base theory is the theory with name *theory-name* if the `base-theory` keyword argument is provided or *ensemble-name* otherwise.

(2) The fixed theories are those given in the fixed-theories keyword argument list if the `fixed-theories` keyword argument is provided or those theories in the global fixed theories list at the time the form is evaluated.

Remarks:

If there is already a theory ensemble with name *ensemble-name* but with different fixed-theories set or different renamer an error will result.

Examples:

```
(def-theory-ensemble METRIC-SPACES)
```

def-theory-ensemble-instances

Positional Arguments:

- *ensemble-name.* An error will result if no ensemble exists with *ensemble-name.*

Keyword Arguments:

- `(target-theories` $name_0$ ... $name_{N-1}$`)`. For each i, $name_i$ must be the name of a theory $\mathcal{T}_i$ or an error will result.
- `(target-multiple` N`)`. N must be an integer. This is equivalent to giving a keyword argument

  `(target-theories` $name_0 \ldots name_{N-1}$`)`

  where $name_i$ is the name of the i-th theory replica. The `target-multiple` and `target-theories` keyword arguments are exclusive.
- `(sorts` $sort\text{-}assoc_1$ ... $sort\text{-}assoc_n$`)`. *sort-assoc* is a list

  $$(sort\ sort_0\ \ldots\ sort_{N-1})$$

  where *sort* is an atomic sort in the base theory of the ensemble and $sort_i$ is either

– A sort specification in the theory $\mathcal{T}_i$*or*

– A string representing a quasi-sort in $\mathcal{T}_i$.

In particular, the length of each *sort-assoc* entry must be one more than the number N of theory entries given in the `target-theories` keyword argument. This argument is valid only in case the `target-theories` keyword is given.

- `(constants` *const-assoc*$_1$ ... *const-assoc*$_p$`)`. *const-assoc* is a list

$$(const\ \ expr_0\ \ \ldots\ \ expr_{N-1})$$

where *const* is a constant in the base theory of the ensemble and $expr_i$ is either

– A constant in the theory $\mathcal{T}_i$ or

– A string representing an expression in $\mathcal{T}_i$.

In particular, the length of each *const-assoc* entry must be one more than the number N of theory entries. This argument is valid only in case the `target-theories` keyword is given.

- `(multiples` M_1 ... M_n`)`. Each argument M is an integer not exceeding the number N of theory entries. This instructs IMPS to translate constants defined in the ensemble M-multiple.

- `(permutations` P_1 ... P_n`)`. Each argument P is a list $(\rho_1, \ldots \rho_\ell)$ of integers in $\{0, \ldots N-1\}$. The presence of this list instructs IMPS to translate constants defined in the ensemble ℓ-multiple into the theory generated by the theories

$$\mathcal{T}_{\rho_1}, \ldots, T_{\rho_\ell}.$$

- `(special-renamings` *renaming*$_1$ ... *renaming*$_n$`)`. A list of entries (*name new-name*) if you want to override the system's naming conventions. Note that *name* is the full constant name, not just the root name.

Description:

This form is used to build translations from selected theory multiples and to transport natively defined constants and sorts from these multiples. To describe the effects of evaluating this form, we consider a number of cases and subcases determined by the presence of certain keyword arguments.

- `(target-theories` $name_0$ ... $name_{N-1}$`)`. Let $\mathcal{T}_i$ be the theory with name $name_i$. In this case, evaluation of the form builds translations and transports natively defined constants and sorts by these translations from selected theory multiples into the theory union

  $$\mathcal{T}_0 \cup \cdots \cup \mathcal{T}_{N-1}.$$

  The theory multiples that are selected as translation source theories are determined by the `multiples` or `permutations` keyword arguments as follows:

  - `(permutations` P_1 ... P_n`)`. Each argument P is a list $(\rho_1, \ldots \rho_\ell)$ of integers in $\{0, \ldots N-1\}$, which instructs IMPS to translate constants defined in the ensemble ℓ-multiple into the theory union

    $$\mathcal{T}_{\rho_1} \cup \cdots \cup \mathcal{T}_{\rho_\ell}.$$

    The translation data are provided by the `sorts` and `constants` keyword arguments.
  - `(multiples` M_1 ... M_n`)`. This case can be reduced to the case of the `(permutations` P_1 ... P_n`)` argument by considering all lists of length M_i of nonnegative integers $\leq N-1$.

- `(target-multiple` n`)`. This case can be reduced to the case of the `(target-theories` $name_1$ ... $name_n$`)` argument by letting $name_i$ be the name of the i-th theory replica.

Examples:

```
(def-theory-ensemble-instances METRIC-SPACES
  (target-theories h-o-real-arithmetic metric-spaces)
  (multiples 1 2)
  (sorts (pp rr pp))
  (constants (dist "lambda(x,y:rr,abs(x-y))" dist)))
```

def-theory-ensemble-multiple

Positional Arguments:

- *ensemble-name.* An error will result if no ensemble exists with *ensemble-name.*

- *N.* An integer.

Description:

Builds a theory which is an N-multiple of the theory ensemble base, together with N theory interpretations from the base theory into the multiple. All existing definitions of the base theory are translated via these interpretations. The N-multiple is constructed to be a subtheory of the $(N+1)$-multiple.

Examples:

```
(def-theory-ensemble-multiple metric-spaces 2)
```

def-theory-ensemble-overloadings

Positional Arguments:

- *ensemble-name.* An error will result if no ensemble exists with *ensemble-name.*

- `(`$N_1 \ldots N_k$`)`. Each N is an integer

Description:

Installs overloadings for constants natively defined in the theory multiples $N_1 \ldots N_k$.

Examples:

```
(def-theory-ensemble-overloadings metric-spaces 1 2)
```

def-theory-instance

Positional Arguments:

- *name.* The name of the theory to be created.

Keyword Arguments:

- `(source` *source-theory-name*`)`. Required.
- `(target` *target-theory-name*`)`. Required.
- `(translation` *trans-name*`)`. Required.
- `(fixed-theories` *theory-name*$_1$ ... *theory-name*$_n$`)`.
- `(renamer` *renamer*`)`. Name of a renaming procedure.
- `(new-translation-name` *new-trans-name*`)`. Name of the the translation to be created.

Description:

Let T_0 and T_0' be the source and target theories, respectively, of the translation Φ named *trans-name*. Also, let T_1 and T_1' be the theories named *source-theory-name* and *target-theory-name*, respectively. Lastly, let $\mathcal{F}$ be the set of theories named *theory-name*$_1, \ldots,$ *theory-name*$_n$.

Suppose that T_0 and T_0' are subtheories of T_1 and T_1', respectively, and that every member of $\mathcal{F}$ is a subtheory of T_1. The theory named *name* is an extension U of T_1' built as follows. First, the primitive vocabulary of T_1 which is outside of T_0 and $\mathcal{F}$ is added to the language of T_1'; the vocabulary is renamed using the value of *renamer*. Next, the translations of the axioms of T_1 which are outside of T_0 and $\mathcal{F}$ are added to the axioms of T_1'; the axioms are renamed using the value of *renamer*. U is union of the resulting theory and the members of $\mathcal{F}$. The translation Φ' from T_1 to U extending Φ is created with name *new-trans-name*.

Examples:

```
(def-theory-instance METRIC-SPACES-COPY
  (source metric-spaces)
  (target the-kernel-theory)
```

```
  (translation the-kernel-translation)
  (fixed-theories h-o-real-arithmetic))

(def-theory-instance VECTOR-SPACES-OVER-RR
  (source vector-spaces)
  (target h-o-real-arithmetic)
  (translation fields-to-rr)
  (renamer vs-renamer))
```

def-theory-processors

Positional Arguments:

- *theory-name*.

Keyword Arguments:

- `(algebraic-simplifier` $spec_1 \ \ldots \ spec_n$`)`.
  Each entry *spec* is a list

  $$(\textit{processor-name}\ op_1\ \ldots op_n),$$

  where *processor-name* is the name of an algebraic processor and

  $$op_1\ \ldots op_n$$

  are constant names denoting functions. We will say that the operators op_i are within the scope of the specified processor.

- `(algebraic-order-simplifier` $spec_1 \ \ldots \ spec_n$`)`.
  Each *spec* is a list

  $$(\textit{processor-name}\ pred_1\ \ldots pred_n),$$

  where *processor-name* is the name of an order processor and

  $$pred_1\ \ldots pred_n$$

  are constant names denoting two-place predicates. We will say that the predicates $pred_i$ are within the scope of the specified order processor.

- `(algebraic-term-comparator` $spec_1 \ \ldots \ spec_n$`)`. *spec* is just the name of an algebraic or order processor. We will say that the equality constructor is within the scope of the specified processor.

Description:

This form causes the simplifier of the theory named *theory-name* to simplify certain terms using the specified algebraic and order processors as follows:

- All applications of an operator or predicate within the scope of a processor are handled by that processor. An operation or predicate may be within the scope of more than one processor.
- All equalities are handled by processors which contain the equality constructor within its scope.

Examples:

```
(def-theory-processors H-O-REAL-ARITHMETIC
  (algebraic-simplifier
    (rr-algebraic-processor * ^ + - / sub))
  (algebraic-order-simplifier (rr-order < <=))
  (algebraic-term-comparator rr-order))

(def-theory-processors VECTOR-SPACES-OVER-RR
  (algebraic-simplifier
    (vector-space-algebraic-processor ** ++))
  (algebraic-term-comparator
    vector-space-algebraic-processor))
```

def-translation

Positional Arguments:

- *name*: The name of the translation.

Modifier Arguments:

- `force`.
- `force-under-quick-load`.
- `dont-enrich`.

Keyword Arguments:

- `(source` *source-theory-name*`)`. Required.
- `(target` *target-theory-name*`)`. Required.
- `(assumptions` *formula-string*$_1$ ... *formula-string*$_n$`)`.
- `(fixed-theories` *theory-name*$_1$ ... *theory-name*$_n$`)`.
- `(sort-pairs` *sort-pair-spec*$_1$ ... *sort-pair-spec*$_n$`)`.
  Each *sort-pair-spec* has one of the following forms:
  - `(`*sort-name sort-name*`)`.
  - `(`*sort-name sort-string*`)`.
  - `(`*sort-name* `(pred` *expr-string*`))`.
  - `(`*sort-name* `(indic` *expr-string*`))`.
- `(constant-pairs` *const-pair-spec*$_1$ ... *const-pair-spec*$_n$`)`, where *const-pair-spec* has one of the following forms:
  - `(`*constant-name constant-name*`)`.
  - `(`*constant-name expr-string*`)`.
- `(core-translation` *core-translation-name*`)`. The name of a theory interpretation.
- `(theory-interpretation-check` *check-method*`)`.

Description:

This form builds a translation named *name* from the source theory S named *source-theory-name* to the target theory T named *target-theory-name*. (The target context in T is the set of assumptions specified by *formula-string*$_1$, ..., *formula-string*$_n$.) The translation is specified by the set of fixed theories, the set of sort pairs, and the set of constant pairs.

If there is a `core-translation` argument, an extension of the translation named *core-translation-name* is build using the information given by the other arguments. The argument `theory-interpretation-check` tells IMPS how to check if the translation is a theory interpretation (relative to the set of assumptions).

If the modifier argument `force` is present, the obligations of the translation are not generated, and thereby the translation is forced to be a theory interpretation. Similarly, if the modifier argument `force-under-quick-load` is present and the switch `quick-load?` is set to true, the obligations of the translation are not generated. The former lets you build a theory interpretation when the obligations are very hard to prove, and the latter is useful for processing the def-form faster, when it has been previously determined that the translation is indeed a theory interpretation.

If the modifier argument `dont-enrich` is present, the translation will not be enriched. A translation is enriched at various times to take into account new definitions in the source and target theories. If one expects there to be many untranslated definitions after enrichment, it may be computationally beneficial to use this modifier argument.

Examples:

```
(def-translation MONOID-THEORY-TO-ADDITIVE-RR
  (source monoid-theory)
  (target h-o-real-arithmetic)
  (fixed-theories h-o-real-arithmetic)
  (sort-pairs
   (uu rr))
  (constant-pairs
   (e 0)
   (** +))
  (theory-interpretation-check using-simplification))

(def-translation MUL-REVERSE
  (source groups)
  (target groups)
  (fixed-theories h-o-real-arithmetic)
  (constant-pairs
    (mul "lambda(x,y:gg, y mul x)"))
  (theory-interpretation-check using-simplification))

(def-translation GROUPS->SUBGROUP
  force
  (source groups)
  (target groups)
```

```
(assumptions
  "with(a:sets[gg], nonempty_indic_q{a})"
  "with(a:sets[gg],
     forall(g,h:gg,
       (g in a) and (h in a)
         implies
       (g mul h) in a))"
   "with(a:sets[gg],
     forall(g:gg, (g in a) implies (inv(g) in a)))")
(fixed-theories h-o-real-arithmetic)
(sort-pairs
 (gg (indic "with(a:sets[gg], a)")))
(constant-pairs
 (mul "with(a:sets[gg],
         lambda(x,y:gg,
           if((x in a) and (y in a), x mul y, ?gg)))")
 (inv "with(a:sets[gg],
         lambda(x:gg, if(x in a, inv(x), ?gg)))"))
(theory-interpretation-check using-simplification))
```

def-transported-symbols

Positional Arguments:

- *names.* The name or lists of names of defined atomic sorts and constants to be transported.

Keyword Arguments:

- `(translation` *translation-name*`)`. Required.
- `(renamer` *renamer.*`)` Name of a renaming procedure.

Description:

Let T and T' be the source and target theories, respectively, of the translation Φ named *translation-name*, and let sym_1,...,sym_n be the symbols whose names are given by *names*. For each sym_i which is a defined symbol in T, this form creates a corresponding new defined symbol sym'_i in T' by

translating the defining object of sym_i via Φ. If Φ already translates sym_i to some defined symbol, the new symbol is not created.

Examples:

```
(def-transported-symbols
  (last%a%index a%inf a%even a%odd)
  (translation schroeder-bernstein-symmetry)
  (renamer sb-renamer))

(def-transported-symbols
 (prec%increasing prec%majorizes prec%sup)
 (translation order-reverse)
 (renamer first-renamer))
```

17.2 Changing Syntax

def-overloading

Positional Arguments:

- *symbol*. A symbol to overload (that is, to have multiple meanings which are determined by context.)

- *theory-name-pair*$_1$ ... *theory-name-pair*$_m$ *theory-name-pair* is a list (*theory-name symbol-name*).

Examples:

```
(def-overloading *
  (h-o-real-arithmetic *)
  (normed-linear-spaces **))
```

def-parse-syntax

Positional Arguments:

- *constant-name*. This is a symbol or a list of symbols.

Keyword Arguments:

- `(token` *spec*`)`. *spec* is a symbol or a string. If this argument is omitted, by default it is *constant-name*.
- `(left-method` *proc-name*`)`. *proc-name* is the name of a procedure for left parsing of the token.
- `(null-method` *proc-name*`)`. *proc-name* is the name of a procedure for null parsing of the token.
- `(table` *table-name*`)`. *table-name* is the name of the parse-table being changed. If this argument is omitted, the default value is the global parse table `*parse*`.
- `(binding` *N*`)`. Required. *N* is the binding or precedence of the token.

Description:

This form allows you to set (or reset) the parse syntax of a constant. In particular, you can change the precedence, parsing method, and token representation of a constant. The most common parsing methods are:

- **infix-operator-method**. For example multiplication and addition use this method.
- **prefix-operator-method**. **comb**, the binomial coefficient function, uses this method.
- **postfix-operator-method**. **!** uses this method.

Note that an operator can have both a null-method and a left-method.

Examples:

```
(def-parse-syntax +
  (left-method infix-operator-method)
  (binding 100))

(def-parse-syntax factorial
  (token !)
  (left-method postfix-operator-method)
  (binding 160))
```

def-print-syntax

Positional Arguments:

- *constant-name*. This is a symbol.

Modifier Arguments:

- `tex`. If this argument is present, then the syntax is added to the global TEX print table.

Keyword Arguments:

- `(token` *spec*`)`. *spec* is a symbol, a string or a list of such. If this argument is omitted, by default it is *constant-name*.
- `(method` *proc-name*`)`. *proc-name* is the name of a procedure for printing of the token.
- `(table` *table-name*`)`. *table-name* is the name of the parse-table being changed. If this argument is omitted, and the `tex` modifier argument is not given, the default value is the global print table `*form*`.
- `(binding` *N*`)`. Required. *N* is the binding or precedence of the token.

Description:

This form allows you to set (or reset) the print syntax of a constant. In particular, you can change the precedence, parsing method, and token representation of a constant.

Examples

```
(def-print-syntax +
  (method present-binary-infix-operator)
  (binding 100))

(def-print-syntax factorial
  (token !)
  (method present-postfix-operator)
  (binding 160))
```

17.3 Loading Sections and Files

load-section

Positional Arguments:

- *section-name*.

Modifier Arguments:

- `reload`. Causes the section to be reloaded if the section has already been loaded.
- `reload-files-only`. Causes the files of the section (but not the component sections) to be reloaded if the section has already been loaded.
- `quick-load`. Causes the section to be quick-loaded.

Keyword Arguments:

None.

Description:

If there are no modifier arguments, this form will simply load the component sections and files of the section named *section-name* which have not been loaded.

include-files

Positional Arguments:

None.

Modifier Arguments:

- `reload`. Causes a file to be reloaded if the file has already been loaded.
- `quick-load`. Causes the files to be quick-loaded.

Keyword Arguments:

- `(files` *file-spec*$_1$ ... *file-spec*$_n$`)`.

Description:

If there are no modifier arguments, this form will simply load the files with specifications *file-spec*$_1, \ldots,$ *file-spec*$_n$ which have not been loaded.

17.4 Presenting Expressions

view-expr

Positional Arguments:

- *expression-string.* A string representing the expression to be built and viewed.

Modifier Arguments:

- `fully-parenthesize`. Causes the expression to be viewed fully parenthesized.
- `fully`. Same as `fully-parenthesize`.
- `no-quasi-constructors`. Causes the expression to be viewed without quasi-constructor abbreviations.
- `no-qcs`. Same as `no-quasi-constructors`.
- `tex`. Causes the expression to be xviewed (i.e., printed in TEX and displayed in an X window TEX previewer).

Keyword Arguments:

- `(language` *language-name*`)`. *language-name* is the name of a language or theory in which to build the expression.

Chapter 18

The Proof Commands

This chapter is intended to provide users with documentation for IMPS proof commands. However, it is not suggested that users read this chapter before using IMPS.

Commands can be used in two modes:

- Interactive mode. In this mode an individual command is invoked by supplying the command's name. The system will then prompt you for additional arguments.

- Script mode. In this mode a command is invoked by a *command form.* Command forms are s-expressions

  $$(\textit{command-name}\ a_1 \ldots\ a_n)$$

  To use commands in this way, you must know the possible command arguments. Commands in script mode can be invoked line by line, by region, or in batch mode. Moreover, if the command requires no arguments, then the name of the command by itself is a command form.

We will use the following template to describe use of individual commands. (The last three entries are optional.)

Script usage Describes the use of the command in scripts. Arguments such as theories, macetes, and theorems are specified by name. Some arguments can be specified in various ways. When an argument is an assumption, it can be specified as follows:

- A nonnegative integer i. This denotes the i-th assumption.
- A string σ. If σ denotes an expression which is alpha-equivalent to an assumption, then it denotes that assumption. If σ matches one or more assumptions, then it refers to the first of those assumptions. Matching here is done in a way that does not preserve scopes of binding constructors and places only type constraints on the matching of variables.

Interactive argument retrieval. This tells you how arguments are requested in the minibuffer when used interactively, with a brief description of each argument. In cases where IMPS can determine that there is only one possible choice for an argument, for example, if the argument is an index number for an assumption when there is exactly one assumption, then IMPS will make this choice and not return for additional input.

Command Kind. Null-inference, single-inference, or multi-inference. A *null-inference* command never adds any inference nodes to the deduction graph. A *single-inference* commands adds exactly one inference node to the deduction graph, when it is successful. A *multi-inference* command adds more than one inference node to the deduction graph in some cases.

Description. A brief description of the command. Some commands can be described as rules of inference in a logical calculus. For each such command we use a table composed of the following units:

Conclusion	TEMPLATE
Premise	TEMPLATE

where the item TEMPLATE for the conclusion refers to the form for the given goal (sequent), while the TEMPLATE item for the premise indicates the form for the resulting subgoal. In general, there will be more than one premise; moreover, in some cases we distinguish one particular premise as a *major premise*. In so doing, we regard the remaining premises as *minor premises*.

Related commands. These are other commands you might consider applying because, for example, they are quicker or more effective for your task at hand.

Remarks. Hints or observations that we think you should find useful.

Key binding. A single key to invoke the command.

Command Application in Interactive Mode

To apply a command in interactive mode to the current sequent node (that is, the one visible in the *sequent node* buffer), hit `!`. You can also apply a command by selecting it from the command menu as follows:

- For Emacs version 19, click on the entry **Extend-DG** in the menubar and select the option **Commands.**
- For Emacs version 18, click right on the **Command menu** item in the **Extending the Deduction Graph** pane. You can also invoke the command menu directly by pressing the key `F3`.

You will be presented with a well-pruned menu of those commands which are applicable to the given sequent node. Once you select a command, the system will request the additional arguments it needs to apply the command. You will notice that for a given command, the system will sometimes request additional arguments and at other times not do so. In the cases where the system fails to make a request for arguments, the system determines what these additional arguments should be on its own.

Command Application in Script Mode

For the precise definition of what a script is, see the section on the proof script language in Chapter 12. Essentially a proof script is a sequence of forms of the following kinds:

- (*keyword* $a_1 \ldots\ a_n$)
- (*command-name* $a_1 \ldots\ a_n$)

Each command form (that is, a form which begins with a command name) instructs IMPS to do the following two things:

- Apply the command with name *command-name* to the current sequent node with the arguments $a_1 \ldots a_n$.
- Reset the current sequent node to be the first ungrounded relative.

To apply a single command in script mode to the current sequent node, place the cursor on the first line of the command form and type `C-c l`. In order for the interface software to recognize the form to execute, there cannot be more than one command form per line. However, a single command form can occupy more than one line.

To apply in sequence the command forms within a region, type `C-c r`.

antecedent-inference

Script usage: `(antecedent-inference` *assumption*`).` *assumption* can be given as an integer i or a string σ.

Interactive argument retrieval:

- `0-based index of antecedent formula` – `(`$i_1\ i_2\ \cdots\ i_n$`):` i.
  The $i_1, i_2, \ldots, i_n$ are the indices of the assumptions of sequent node on which antecedent inferences can be done, that is an implication, conjunction, disjunction, biconditional, conditional-formula, or an existential formula. In case there is only one such formula, this argument request is omitted.

Command kind: Single-inference.

Description:

Conclusion	$\Gamma \cup \{\varphi \supset \psi\} \Rightarrow \theta$
Premises	$\Gamma \cup \{\neg\varphi\} \Rightarrow \theta$
	$\Gamma \cup \{\psi\} \Rightarrow \theta$
Conclusion	$\Gamma \cup \{\varphi_1 \wedge \cdots \wedge \varphi_n\} \Rightarrow \psi$
Premise	$\Gamma \cup \{\varphi_1, \ldots, \varphi_n\} \Rightarrow \psi$
Conclusion	$\Gamma \cup \{\varphi_1 \vee \cdots \vee \varphi_n\} \Rightarrow \psi$
Premises ($1 \leq i \leq n$)	$\Gamma \cup \{\varphi_i\} \Rightarrow \psi$
Conclusion	$\Gamma \cup \{\varphi \equiv \psi\} \Rightarrow \theta$
Premises	$\Gamma \cup \{\varphi, \psi\} \Rightarrow \theta$
	$\Gamma \cup \{\neg\varphi, \neg\psi\} \Rightarrow \theta$
Conclusion	$\Gamma \cup \{\text{if-form}(\varphi, \psi, \theta)\} \Rightarrow \xi$
Premises	$\Gamma \cup \{\varphi, \psi\} \Rightarrow \xi$
	$\Gamma \cup \{\neg\varphi, \theta\} \Rightarrow \xi$
Conclusion	$\Gamma \cup \{\exists x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi\} \Rightarrow \psi$
Premise	$\Gamma \cup \{\varphi'\} \Rightarrow \psi$

Notes:

- Each sequent in the preceding table is of the form $\Gamma \cup \{\varphi\} \Rightarrow \psi$.
- The index of the assumption φ in context $\Gamma \cup \{\varphi\}$ is i.

- φ' is obtained from φ by renaming the variables among $x_1, \ldots, x_n$ which are free in both φ and the original sequent.

Related commands:
antecedent-inference-strategy
direct-and-antecedent-inference-strategy
direct-inference

Remarks: Keep in mind that any antecedent inference produces subgoals which together are equivalent to the original goal. Thus, it is always safe to do antecedent inferences, in the sense that they do not produce false subgoals from true ones.

Key binding: `a`

antecedent-inference-strategy

Script usage: `(antecedent-inference-strategy` *assumption-list*`)`.
assumption-list can be given as a list of numbers, a list of strings, or an assumption-list form.

Interactive argument retrieval:

- `List of formula indices for antecedent inferences --` `(`i_1 i_2 $\cdots$ i_m`):`j_1 j_2 $\cdots$ j_n. The $i_1, i_2, \ldots, i_m$ are the indices of the assumptions of sequent node on which antecedent inferences can be done, that is an implication, conjunction, disjunction, biconditional, conditional-formula, or an existential formula. In case there is only one such formula, this argument request is omitted.

Command kind: Multi-inference.

Description: Call an assumption of the goal sequent node *fixed* if its index is not among $j_1, \ldots, j_n$. This command repeatedly applies the antecedent-inference command to the goal node and resulting subgoal nodes until the only possible antecedent inferences are on fixed assumptions.

Related commands:
antecedent-inference
direct-and-antecedent-inference-strategy

Remarks: This command is called as a subroutine by a number of other commands including **instantiate-theorem** and **instantiate-universal-antecedent**.

Key binding: `A`

apply-macete

Script usage: `(apply-macete` *macete*`)`.

Interactive argument retrieval:

- `Macete name:` The name of a macete. Formally, a *macete* is a function which takes as arguments a context and an expression and returns an expression. Macetes are used to apply a theorem or a collection of theorems to a sequent in a deduction graph. In order to use them effectively, read the section on macetes in the manual.

Command kind: Single-inference.

Description: This command applies the argument macete to the given sequent node.

Related commands:
apply-macete-locally
apply-macete-locally-with-minor-premises
apply-macete-with-minor-premises

Remarks: Macetes are an easy and very effective way of applying lemmas in a proof. In fact, in the course of developing a theory, we suggest that some effort be expended in formulating lemmas with a view to applying them as macetes.

Key binding: `m`

apply-macete-locally

Script usage: `(apply-macete-locally` *macete expression occurrences*`).`

Interactive argument retrieval:

- `Macete name:` The name of a macete.
- `Expression to apply macete:` A subexpression of the sequent assertion to which you want to apply the macete.
- `Occurrences of expression (0-based):` The list occurrences of the expression you want to apply the macete to.

Command kind: Single-inference.

Description: This command applies the argument macete to the given sequent node at those occurrences of the expression supplied in response to the minibuffer prompt.

Related commands:
apply-macete
apply-macete-locally-with-minor-premises
apply-macete-with-minor-premises

Remarks: This command is useful when you need to control where a macete is applied, in cases where it applies at several locations. For example, if the sequent assertion is $|x + y^2| = |y^2 + x|$, then applying the macete **commutative-law-for-addition** to the 0-th occurrence of $x + y^2$ yields a new sequent with assertion $|y^2+x| = |y^2+x|$. If the macete had been applied globally, the resulting assertion would have been $|y^2 + x| = |x + y^2|$.

apply-macete-locally-with-minor-premises

Script usage: `(apply-macete-locally-with-minor-premises` *macete expression occurrences*`).`

Interactive argument retrieval:

- `Macete name:` The name of a macete.
- `Expression to apply macete:` A subexpression of the sequent assertion to which you want to apply the macete.
- `Occurrences of expression (0-based):` The list occurrences of the expression you want to apply the macete to.

Command kind: Single-inference.

Description: This command applies the argument macete to the given sequent node, in the same way as apply-macete-locally, but with the following important difference: whenever the truth or falsehood of a definedness or sort-definedness assertion cannot be settled by the simplifier, the assertion is posted as a additional subgoal to be proved.

Related commands:
apply-macete
apply-macete-locally
apply-macete-with-minor-premises

apply-macete-with-minor-premises

Script usage: `(apply-macete-with-minor-premises` *macete*`)`.

Interactive argument retrieval:

- `Macete name:` The name of a macete.

Command kind: Single-inference.

Description: This command applies the argument macete to the given sequent node, in the same way as apply-macete, but with the following important difference: whenever the truth or falsehood of a convergence requirement cannot be settled by the simplifier, the assertion is posted as a additional subgoal to be proved.

Related commands:
apply-macete
apply-macete-locally
apply-macete-locally-with-minor-premises

Remarks: The convergence requirements which are posted as additional subgoals arise from two different sources:

(1) Convergence requirements are generated by the simplifier in the course of certain reductions, including algebraic simplification and beta-reduction.

(2) Convergence requirements are also generated by checks that instantiations of universally valid formulas used by the macete are sound.

You should never assume that subgoals generated by this command are always true. As always for nonreversible commands, you should inspect the new subgoals to insure IMPS is not misdirecting your proof.

assume-theorem

Script usage: `(assume-theorem` *theorem*`)`.

Interactive argument retrieval:

- `Theorem name:` The name of a theorem in the deduction graph theory.

Command kind: Single-inference.

Description: This command adds the argument theorem to the context of the given sequent.

Related commands:
apply-macete
assume-transported-theorem
instantiate-theorem

Remarks: To apply a theorem to a sequent, you will usually want to use **apply-macete** or **instantiate-theorem** instead of **assume-theorem**.

assume-transported-theorem

Script usage: `(assume-transported-theorem` *theorem interpretation*`)`.

Interactive argument retrieval:

- `Theorem name:` The name of a theorem in T.
- `Theory interpretation:` The name of a theory interpretation of T in the deduction graph's theory.

Command kind: Single-inference.

Description: This command transports the argument theorem to the deduction graph's theory via the argument theory interpretation. Then the transported theorem is added to the context of the given sequent using **assume-theorem**.

Related commands:
apply-macete
assume-theorem
instantiate-transported-theorem

Remarks: To apply a theorem to a sequent from outside the deduction graph's theory, you will usually want to use **apply-macete** or **instantiate-transported-theorem** instead of **assume-transported-theorem**.

auto-instantiate-existential

Script usage: `auto-instantiate-existential`.

Interactive argument retrieval: None.

Command Kind: Multi-inference.

Description: This command tries to instantiate an existential assertion with terms from the context of the given sequent.

Related commands:
auto-instantiate-universal-antecedent
instantiate-existential

auto-instantiate-universal-antecedent

Script usage: `(auto-instantiate-universal-antecedent` *assumption*`)`. *assumption* can be given as an integer i or a string σ.

Interactive argument retrieval:

- `0-based index of universal antecedent formula --` (i_1 i_2 $\cdots$ i_n)`:` i. The $i_1, i_2, \ldots, i_n$ are the indices of the universal assumptions of the sequent node. In case there is only one universal antecedent formula, this argument request is omitted.

Command Kind: Multi-inference.

Description: This command tries to instantiate the i-th assumption of the context of the given sequent with terms from the context.

Related commands:
auto-instantiate-existential
instantiate-universal-antecedent

backchain

Script usage: `(backchain` *assumption*`)`. *assumption* can be given as an integer i or a string σ.

Interactive argument retrieval:

- `0-based index of antecedent formula:` If there is only one assumption, this argument request is omitted.

Command Kind: Single-inference.

Description: This command includes the behavior described under **backchain-through-formula**. However, for a part of the assumption of one of the forms $s = t$, $s \simeq t$, $s \equiv t$, if a subexpression of the assertion matches s, then it is replaced by the corresponding instance of t.

Related commands:
backchain-backwards
backchain-repeatedly
backchain-through-formula

Key Binding: `b`

backchain-backwards

Script usage: `(backchain-backwards` *assumption*`)`. *assumption* can be given as an integer i or a string σ.

Interactive argument retrieval:

- `0-based index of antecedent formula:`

Command Kind: Single-inference.

Description: Backchain-backwards differs from `backchain` in that subformulas of the assumption of the forms $s = t$, $s \simeq t$, $s \equiv t$ are used from right to left.

Related commands:
backchain
backchain-repeatedly
backchain-through-formula

backchain-repeatedly

Script usage: `(backchain-repeatedly` *assumption-list*`)`. *assumption-list* can be given as a list of numbers, of strings or an assumption-list form.

Interactive argument retrieval:

- `List of 0-based indices of antecedent formulas:` If there is only one assumption, this argument request is omitted.

Command Kind: Multi-inference.

Description: A succession of backchains are performed using the indicated assumptions. Execution terminates when every backchaining opportunity against those assumptions has been used up.

Related commands:
backchain
backchain-backwards
backchain-through-formula

backchain-through-formula

Script usage: `(backchain-through-formula` *assumption*`)`. *assumption* can be given as an integer i or a string σ.

Interactive argument retrieval:

- `0-based index of antecedent formula:` If there is only one assumption, this argument request is omitted.

Command kind: Single-inference.

Description: This command attempts to use the assumption with the given index to replace the assertion to be proved. In the simplest case, if the given assumption is $\varphi \supset \psi$ and the assertion is ψ, then backchain-through-formula replaces the assertion with φ. Similarly, if the assertion is $\neg\varphi$, it is replaced with $\neg\psi$. The command extends this simplest case in four ways:

- If the assumption is universally quantified, then matching is used to select a relevant instance.

- If the assumption is a disjunction

$$\bigvee_{i \in i} \varphi_i,$$

  then, for any $j \in i$, it may be treated as

$$\left(\bigwedge_{i \in i, i \neq j} \neg \varphi_i \right) \supset \varphi_j.$$

- If the assumption is a conjunction, each conjunct is tried separately, in turn.
- These rules are used iteratively to descend through the structure of the assumption as deeply as necessary.

If IMPS cannot recognize that the terms returned by a match are defined in the appropriate sorts, then these assertions are returned as additional minor premises that must later be grounded to complete the derivation.

Related commands:
backchain
backchain-backwards
backchain-repeatedly

beta-reduce

Script usage: `beta-reduce`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command attempts to beta-reduce each lambda-application in the assertion of the given sequent.

Related commands:
beta-reduce-antecedent
beta-reduce-insistently
beta-reduce-repeatedly
beta-reduce-with-minor-premises
simplify

Remarks: Since beta-reduction can often be applied several times in a row, the command **beta-reduce-repeatedly** is usually preferable to this command. Beta-reduction is also performed by **simplify**.

beta-reduce-antecedent

Script usage: `(beta-reduce-antecedent` *assumption*`)`. *assumption* can be given as an integer i or a string σ.

Interactive argument retrieval:

- `0-based index of antecedent-formula:` i. If there is only one assumption, this argument request is omitted.

Command kind: Multi-inference.

Description: This command is used for beta-reducing an assumption in the context of the given sequent. It is equivalent to the following sequence of commands: **incorporate-antecedent** applied to the given sequent with argument i; **beta-reduce-repeatedly** applied to the sequent yielded by the previous command; and **direct-inference** applied to the sequent yielded by the previous command. The commands halts if **beta-reduce-repeatedly** grounds the first produced sequent.

Related commands::
beta-reduce-repeatedly
simplify-antecedent

Remarks: The implementation of this command has the side effect that the assertion is beta-reduced as well as the antecedent formula.

beta-reduce-insistently

Script usage: `beta-reduce-insistently`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command is equivalent to disabling all quasi-constructors and then calling **beta-reduce**.

Related commands:
beta-reduce
insistent-direct-inference
simplify-insistently

Remarks: There is rarely any need to use this command. It has the disagreeable effect of exploding quasi-constructors.

beta-reduce-repeatedly

Script usage: `beta-reduce-repeatedly`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command attempts to beta-reduce each lambda-application in the assertion of the given sequent repeatedly until there are no longer any lambda-applications that beta-reduce.

Related commands:
beta-reduce
beta-reduce-antecedent

Remarks: Since beta-reduction can often be applied several times in a row, this command is usually preferable to the command **beta-reduce**.

Key binding: `C-c b`

beta-reduce-with-minor-premises

Script usage: `beta-reduce-with-minor-premises`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command is the same as **beta-reduce** except that, instead of failing when convergence requirements are not verified, it posts the unverified convergence requirements as additional subgoals to be proved.

Related commands:
beta-reduce
simplify-with-minor-premises

Remarks: This command is useful for determining why a lambda-application does not beta-reduce.

case-split

Script usage: `(case-split` *list-of-formulas*`)`.

Interactive argument retrieval:

- `First formula:` φ_1.
- `Next formula (<RET> if done):` φ_1.
  $\vdots$
- `Next formula (<RET> if done):` φ_n $(n \geq 1)$.

Description: This command considers all the different possible cases for $\varphi_1, \ldots, \varphi_n$.

Conclusion	$\Gamma \Rightarrow \psi$
Premises ($A \subseteq \{1, \ldots, n\}$)	$\Gamma \cup \{\varphi_1^A \wedge \cdots \wedge \varphi_n^A\} \Rightarrow \psi$

Notes:

- For each $A \subseteq \{1, \ldots, n\}$, φ_i^A is φ_i if $i \in A$, and $\neg\varphi_i$ otherwise.

case-split-on-conditionals

Script usage: `(case-split-on-conditionals` *occurrences*`)`. *occurrences* is a list of integers.

Interactive argument retrieval:

- `Occurrences of conditionals to be raised (0-based):` A list l given in the form $l_1 \ \cdots \ l_n$.

Command kind: Multi-inference.

Description: This command applies **case-split** to the first components of the conditional expressions in the given sequent specified by l. The command **simplify** is then applied to each of the newly created sequents.

Related commands::
case-split
raise-conditional

Remarks: Avoid using this command with more than two occurrences of conditionals.

choice-principle

Script usage: `choice-principle`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command implements a version of the axiom of choice.

Conclusion	$\Gamma \Rightarrow \exists f{:}[\sigma_1, \ldots, \sigma_n, \tau], \forall x_1{:}\sigma'_1, \ldots, x_n{:}\sigma'_n, \varphi$
Premise	$\Gamma \Rightarrow \forall x_1{:}\sigma'_1, \ldots, x_n{:}\sigma'_n, \exists y_f{:}\tau, \varphi[y_f/f(x_1, \ldots, x_n)]_{\text{all}}$

Notes:

- σ_i and σ'_i have the same type for all i with $1 \leq i \leq n$.
- The command fails if there is any occurrence of f in φ which is not in an application of the form $f(x_1, \ldots, x_n)$.

Remarks: The existential sequence is often introduced into the deduction graph using **cut-with-single-formula**.

contrapose

Script usage: `(contrapose` *assumption*`)`. *assumption* can be given as an integer i or a string σ.

Interactive argument retrieval:

- `0-based index of antecedent-formula:` i. If there is only one assumption, this argument request is omitted.

Command kind: Single-inference.

Description: This command interchanges the given sequent's assertion and its assumption given by i.

Conclusion	$\Gamma \cup \{\varphi\} \Rightarrow \psi$
Premise	$\Gamma \cup \{\neg\psi\} \Rightarrow \neg\varphi$

Notes:

- The index of φ in context $\Gamma \cup \{\varphi\}$ is i.

Remarks: Use **contrapose** if you want to do "proof by contradiction." However, if there is nothing to contrapose with (because the sequent has no assumptions), you will have to add an assumption by using **cut-with-single-formula** or **instantiate-theorem**.

Key binding: `c`

cut-using-sequent

Script usage: `(cut-using-sequent` *sequent-node*`)`. *sequent-node* can be given as an integer (referring to the sequent node with that number) or as a list `(`*context-string assertion-string*`)`.

Interactive argument retrieval:

- `Major premise number:` The number of a sequent node.

Command kind: Single-inference.

Description: This command allows you to add some new assumptions to the context of the given sequent. Of course, you are required to show separately that the new assumptions are consequences of the context. The node containing the major premise sequent must exist before the command can be called. Usually you create this sequent node with **edit-and-post-sequent-node**.

Conclusion	$\Gamma \Rightarrow \varphi$
Major Premise	$\Gamma \cup \{\psi_1, \ldots, \psi_n\} \Rightarrow \varphi$
Minor Premises ($1 \leq i \leq n$)	$\Gamma \Rightarrow \psi_i$

Related commands: **cut-with-single-formula**.

Remarks: To cut with one formula, you should usually use **cut-with-single-formula**.

cut-with-single-formula

Script usage: `(cut-with-single-formula` *formula*`)`.

Interactive argument retrieval:

- `Formula to cut:` String specifying a formula ψ.

Command kind: Single-inference.

Description: This command allows you to *add* a new assumption to the context of the given sequent. Of course, you are required to show separately that the new assumption is a consequence of the context.

Conclusion	$\Gamma \Rightarrow \varphi$
Major Premise	$\Gamma \cup \{\psi\} \Rightarrow \varphi$
Minor Premise	$\Gamma \Rightarrow \psi$

Related commands: **cut-using-sequent**.

Remarks: To cut with several formulas at the same time, use **cut-using-sequent**.

Key binding: `&`

definedness

Script usage: `definedness`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command applies to a sequent whose assertion is a definedness statement of the form $t\downarrow$. The command first tests whether the context entails the definedness of t. If the test fails, the command then tries to reduce the sequent to a set of simpler sequents.

Related commands: **sort-definedness**.

Remarks: This command is mainly useful when t is an application or a conditional.

direct-and-antecedent-inference-strategy

Script usage: `direct-and-antecedent-inference-strategy`.

Interactive argument retrieval: None.

Command kind: Multi-inference.

Description: This command repeatedly applies (1) **direct-inference** to the given sequent and resulting sequents and (2) **antecedent-inference** to newly created antecedents of the given sequent and resulting sequents until no more such direct and antecedent inferences are possible.

Related commands:
antecedent-inference-strategy
direct-and-antecedent-inference-strategy-with-simplification
direct-inference-strategy

Key binding: `D`

direct-and-antecedent-inference-strategy-with-simplification

Script usage:
`direct-and-antecedent-inference-strategy-with-simplification`.

Interactive argument retrieval: None.

Command kind: Multi-inference.

Description: This command first applies **direct-and-antecedent-inference-strategy** to the given sequent, and then applies **simplify** to all resulting sequents.

Related commands:
antecedent-inference-strategy
direct-and-antecedent-inference-strategy
direct-inference-strategy

direct-inference

Script usage: `direct-inference`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command applies an analogue of an introduction rule of Gentzen's sequent calculus (in reverse), based on the leading constructor of the assertion of the given sequent.

Conclusion	$\Gamma \Rightarrow \varphi \supset \psi$
Premise	$\Gamma \cup \{\varphi\} \Rightarrow \psi$
Conclusion	$\Gamma \Rightarrow \varphi_1 \wedge \cdots \wedge \varphi_n$
Premises $(1 \leq i \leq n)$	$\Gamma \cup \{\varphi_1, \ldots, \varphi_{i-1}\} \Rightarrow \varphi_i$
Conclusion	$\Gamma \Rightarrow \varphi_1 \vee \cdots \vee \varphi_n$
Premise	$\Gamma \cup \{\neg\varphi_1, \ldots, \neg\varphi_{n-1}\} \Rightarrow \varphi_n$
Conclusion	$\Gamma \Rightarrow \varphi \equiv \psi$
Premises	$\Gamma \cup \{\varphi\} \Rightarrow \psi$
	$\Gamma \cup \{\psi\} \Rightarrow \varphi$
Conclusion	$\Gamma \Rightarrow \text{if-form}(\varphi, \psi, \theta)$
Premises	$\Gamma \cup \{\varphi\} \Rightarrow \psi$
	$\Gamma \cup \{\neg\psi\} \Rightarrow \theta$
Conclusion	$\Gamma \Rightarrow \forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi$
Premise	$\Gamma \Rightarrow \varphi'$

Notes:

- φ' is obtained from φ by renaming the variables among $x_1, \ldots, x_n$ which are free in both φ and the context Γ.

Related commands:
antecedent-inference
direct-inference-strategy
unordered-direct-inference

Remarks: Keep in mind that any direct inference produces subgoals which together are equivalent to the original goal. Thus it is always safe to do direct inferences, in the sense that they do not produce false subgoals from true goals.

Key binding: `d`

direct-inference-strategy

Script usage: `direct-inference-strategy`.

Interactive argument retrieval: None.

Command kind: Multi-inference.

Description: This command repeatedly applies **direct-inference** to the given sequent and resulting sequents until no more direct inferences are possible. In other words, it adds to the deduction graph the smallest set of sequents containing the given sequent and closed under direct inferences.

Related commands:
antecedent-inference-strategy
direct-inference
direct-and-antecedent-inference-strategy

disable-quasi-constructor

Script usage: `(disable-quasi-constructor` *quasi-constructor*`)`.

Interactive argument retrieval:

- `Quasi-constructor name:` The name of a quasi-constructor in the expression.

Command kind: Null-inference.

Description: This command disables the quasi-constructor given as argument.

edit-and-post-sequent-node

Script usage: `(edit-and-post-sequent-node` *context-string assertion-string*`)`.

Interactive argument retrieval:

- `Edit sequent and C-c C-c when finished.` A sequent presented as a string of formulas using `=>` to separate the sequent assumptions from the sequent assertion.

Command kind: Null-inference.

Description: This commands adds the sequent given as an argument to the deduction graph.

Key binding: `e`

eliminate-defined-iota-expression

Script usage: `(eliminate-iota` *index symbol*`)`.

Interactive argument retrieval:

- `0-based index of iota-expression occurrence:` i.
- `Name of replacement variable:` y

Command kind: Multi-inference.

Description: This command replaces the i-th occurrence of an iota-expression in the given sequent's assertion with the variable y. The command is predicated upon the iota-expression being defined with respect to the sequent's context. (An *iota-expression* is an expression whose lead constructor is **iota** or a quasi-constructor that builds an iota-expression.)

Conclusion	$\Gamma \Rightarrow \psi$
Major Premise	$\Gamma \cup \{\varphi[y/x]_{\text{free}}, \forall z{:}\sigma, (\varphi[z/x]_{\text{free}} \supset z = y)\} \Rightarrow \psi'$
Minor Premise	$\Gamma \Rightarrow (\iota x{:}\sigma, \varphi)\downarrow$

Notes:

- $\iota x{:}\sigma, \varphi$ is the i-th iota-expression occurrence in ψ.
- ψ' is the result of replacing the i-th iota expression occurrence in ψ with y.

Related commands: **eliminate-iota**.

Remarks: This command is very useful for dealing with iota-expressions that are known to be defined with respect to the sequent's context. The minor premise is grounded automatically if the sequent's context contains $(\iota x{:}\sigma, \varphi)\downarrow$.

eliminate-iota

Script usage: `(eliminate-iota` *index*`)`.

Interactive argument retrieval:

- `0-based index of iota-expression occurrence:` i.

Command kind: Single-inference.

Description: This command applies to a sequent whose assertion is an atomic formula or negated atomic formula containing a specified occurrence of an iota-expression. The command reduces the sequent to an equivalent sequent in which the specified iota-expression occurrence is "eliminated." (An *iota-expression* is an expression whose lead constructor is **iota** or a quasi-constructor that builds an iota-expression.)

Conclusion	$\Gamma \Rightarrow \psi$
Major Premise	$\Gamma \Rightarrow \exists y{:}\sigma, (\varphi[y/x]_{\text{free}} \wedge \forall z{:}\sigma, (\varphi[z/x]_{\text{free}} \supset z = y) \wedge \psi')$

Conclusion	$\Gamma \Rightarrow \neg\psi$
Major Premise	$\Gamma \Rightarrow \exists y{:}\sigma, (\varphi[y/x]_{\text{free}} \wedge \forall z{:}\sigma, (\varphi[z/x]_{\text{free}} \supset z = y)) \supset$ $\exists y{:}\sigma, (\varphi[y/x]_{\text{free}} \supset \neg\psi')$

Notes:

- ψ is an atomic formula.
- $\iota x{:}\sigma, \varphi$ is the i-th iota-expression occurrence in ψ, an atomic formula.
- ψ' is the result of replacing the i-th iota-expression occurrence in ψ with y.
- The occurrence of $\iota x{:}\sigma, \varphi$ in ψ is within some argument component s of ψ. Moreover, the occurrence is an extended application component of s, where a is an *extended application component* of b if either a is b or b is an application of kind ι and a is an extended application component of a component of b.

Related commands: **eliminate-defined-iota-expression**.

Remarks: If ψ is an equation or a definedness expression, the macete **eliminate-iota-macete** is more direct than **eliminate-iota**.

enable-quasi-constructor

Script usage: `(enable-quasi-constructor` *quasi-constructor*`)`.

Interactive argument retrieval:

- `Quasi-constructor name:` The name of a quasi-constructor in the expression.

Command kind: Null-inference.

Description: This command enables the quasi-constructor given as argument.

extensionality

Script usage: `extensionality`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: If the sequent node assertion is an equality or quasi-equality or the negation of an equality or quasi-equality of n-ary functions, this command applies the extensionality principle.

Conclusion	$\Gamma \Rightarrow f = g$
Major Premise	$\Gamma \Rightarrow \forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, f(x_1, \cdots, x_n) \simeq g(x_1, \cdots, x_n)$
Minor Premise	$\Gamma \Rightarrow f\downarrow$
Minor Premise	$\Gamma \Rightarrow g\downarrow$
Conclusion	$\Gamma \Rightarrow f \simeq g$
Major Premise	$\Gamma \Rightarrow \forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, f(x_1, \cdots, x_n) \simeq g(x_1, \cdots, x_n)$
Minor Premise	$\Gamma \cup \{g\downarrow\} \Rightarrow f\downarrow$
Minor Premise	$\Gamma \cup \{f\downarrow\} \Rightarrow g\downarrow$
Conclusion	$\Gamma \Rightarrow \neg f = g$
Premise	$\Gamma \Rightarrow \exists x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \neg f(x_1, \cdots, x_n) \simeq g(x_1, \cdots, x_n)$
Conclusion	$\Gamma \Rightarrow \neg f \simeq g$
Premise	$\Gamma \Rightarrow \exists x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \neg f(x_1, \cdots, x_n) \simeq g(x_1, \cdots, x_n)$

Notes:

- If the sorts of f and g are $[\alpha_1, \ldots, \alpha_n, \alpha_{n+1}]$ and $[\beta_1, \ldots, \beta_n, \beta_{n+1}]$, respectively, then $\sigma_i = \alpha_i \sqcup \beta_i$ for all i with $1 \leq i \leq n$.
- A sequent node corresponding to a minor premise is not created if the truth of the minor premise is immediately apparent to IMPS.
- If f and g are of kind $*$, then "$=$" is used in place of "$\simeq$" in the premise.

Remarks: By substitution of quasi-equals for quasi-equals, both $f = g$ and $f \simeq g$ imply

$$\forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, f(x_1, \cdots, x_n) \simeq g(x_1, \cdots, x_n).$$

So in this direction extensionality is nothing new.

Key binding: ~

force-substitution

Script usage: `(force-substitution` *expression replacement occurrences*`).` *expression* and *replacement* are denoted by strings, and *occurrences* is a list of integers.

Interactive argument retrieval:

- `Expression to replace:` The subexpression e of the sequent node assertion you want to replace.
- `Replace it with:` The new expression r you want to replace it with.
- `0-based indices of occurrences to change:` A list l given in the form $l_1 \ \cdots \ l_n$.

Command kind: Single-inference.

Description: This command allows you to change part of the sequent assertion. It yields two or more new subgoals. One subgoal is obtained from the original sequent assertion by making the requested replacements. The other subgoals assert that the replacements are sound.

Conclusion	$\Gamma \Rightarrow \varphi$
Major Premise	$\Gamma \Rightarrow \varphi[r/e]_l$
Minor Premises ($1 \leq i \leq n$)	$\Gamma_i \Rightarrow \varphi_i$

Notes:

- Γ_i is the local context of $\Gamma \Rightarrow \varphi$ at the location of the l_i-th occurrence of e in φ.
- φ_i is
  - $e \equiv r$ if e is a formula and the parity of the path l_i is 0.
  - $r \supset e$ if e is a formula and the parity of the path l_i is 1.
  - $e \supset r$ if e is a formula and the parity of the path l_i is -1.
  - $e \simeq r$ in all other cases.

Key binding: `f`

generalize-using-sequent

Script usage: `(generalize-using-sequent` *major-premise*`)`. *major-premise* can be given as an integer (referring to the sequent node with that number) or as a list `(`*context-string assertion-string*`)`.

Interactive argument retrieval:

- `Major premise number:` The number of a sequent node.

Command kind: Single-inference.

Description: This command generalizes the assertion of the given sequent. The node containing the major premise sequent must exist before the command can be called. Usually you create this sequent node with **edit-and-post-sequent-node**.

Conclusion	$\Gamma \Rightarrow \varphi$
Major Premise	$\Gamma \Rightarrow \forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi'$
Minor Premises ($1 \leq i \leq n$)	$\Gamma \Rightarrow t_i \downarrow \sigma_i$

Notes:

- φ is the result of simultaneously replacing each free occurrence of x_i in φ' with t_i for all i with $1 \leq i \leq n$.
- t_i is free for x_i in φ' for each i with $1 \leq i \leq n$.

incorporate-antecedent

Script usage: `(incorporate-antecedent` *assumption*`)`. *assumption* can be given as an integer i or a string σ.

Interactive argument retrieval:

- `0-based index of antecedent-formula:` i. If there is only one assumption, this argument request is omitted.

Command kind: Single-inference.

Description: This commands does the reverse of **direct-inference** on an implication.

Conclusion	$\Gamma \cup \{\varphi\} \Rightarrow \psi$
Premise	$\Gamma \Rightarrow \varphi \supset \psi$

Notes:

- The index of φ in the context $\Gamma \cup \{\varphi\}$ is i.

Related commands: **direct-inference**.

Remarks: This command is used for incorporating an assumption of a sequent's context into the sequent's assertion. It is particularly useful as a preparation for applying such commands as **apply-macete**, **beta-reduce**, and **simplify**.

Key binding: `@`

induction

Script usage: `(induction` *inductor variable*`)`. *inductor* is a symbol naming an inductor, and *variable* is a string or `()`. If `()` is supplied as argument, IMPS will choose the variable of induction itself.

Interactive argument retrieval:

- `Inductor:` An *inductor* is an IMPS structure which has an associated induction principle together with heuristics (in the form of macetes and commands) to handle separately the base and induction cases.
- `Variable to induct on (<RET> to use IMPS default):` IMPS will choose a variable of the appropriate sort to induct on. Thus, in general, there is no danger that it will attempt to induct on a variable for which induction makes no sense. However, if various choices for an induction variable are possible, the choice is more or less arbitrary. In these cases, you should be prepared to suggest the variable.

Command kind: Multi-inference.

Description: This command attempts to apply a formula called an *induction principle* to a sequent and applies some auxiliary inferences as well. The induction principle used by IMPS is determined by the inductor you give as an argument. When using the induction command, the system will attempt to reformulate the sequent in a form which matches the induction principle. In particular, you can use the induction command directly in cases where the goal sequent does not exactly match the induction principle.

Remarks: In a formal sense "induction" refers to an axiom or theorem in a particular theory (for example, the formula of weak induction below), and in this sense, "applying induction," means applying this formula to a sequent. However, induction alone rarely gets the job done. Other tricks, such as algebraic simplification, beta-reduction, and unfolding of definitions, especially recursive ones, are usually needed. In this broader sense, induction can be thought of as a proof technique incorporating a large number of fairly standard inferences.

Despite its simplicity, induction is one of the most powerful proof techniques in mathematics, especially in proofs for formulas involving the integers. Moreover, forms of induction are also used in theories of syntax or theories of lists. An induction principle in one of these theories is usually referred to as *structural induction.*

The induction principle for the inductor **integer-inductor** is the formula

$$\forall s{:}[\mathbf{z}, *], m{:}\mathbf{z},$$
$$\forall t{:}\mathbf{z}, (m \leq t \supset s(t) \equiv (s(m) \wedge \forall t{:}\mathbf{z}, m \leq t \supset (s(t) \supset s(t+1)))).$$

This formula is usually referred to as the *principle of weak induction.* You might also be familiar with the *principle of strong induction*; in fact, both induction principles are equivalent. However, some formulas cannot be proved *directly* with the principle of weak induction, because the induction hypothesis is not sufficiently strong to prove the induction conclusion. In such cases, instead of using a different induction principle, you first prove a stronger formula by using the same weak induction principle.

You can build inductors using the form **def-inductor**. See Section 17.1.

Key binding: `i`

insistent-direct-inference

Script usage: `insistent-direct-inference`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command is equivalent to disabling all quasi-constructors and then calling **direct-inference**.

Related commands:
direct-inference
insistent-direct-inference-strategy.

Remarks: There is rarely any need to use this command. It has the disagreeable effect of exploding quasi-constructors.

insistent-direct-inference-strategy

Script usage: `insistent-direct-inference-strategy`.

Interactive argument retrieval: None.

Command kind: Multi-inference.

Description: This command repeatedly applies the **insistent-direct-inference** command to the goal node and resulting subgoal nodes until no more insistent direct inferences are possible. In other words, it adds to the deduction graph the smallest set of sequents containing the goal sequent and closed under insistent direct inferences. This command is equivalent to disabling all quasi-constructors and then calling **direct-inference-strategy**.

Related commands:
direct-inference-strategy
insistent-direct-inference

Remarks: Like **insistent-direct-inference**, this command has the disagreeable effect of exploding quasi-constructors.

instantiate-existential

Script usage: `(instantiate-existential` *instantiations*`)`. *instantiations* is a list of strings denoting expressions.

Interactive argument retrieval:

- `Instance for variable` x_1 `of sort` σ_1`:` t_1.
  ⋮
- `Instance for variable` x_n `of sort` σ_n`:` t_n $(n \geq 1)$.

Command kind: Multi-inference.

Description: This command instantiates an existential assertion with the specified terms.

Conclusion	$\Gamma \Rightarrow \exists x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi$
Major Premise	$\Gamma \Rightarrow \varphi'$
Minor Premises $(1 \leq i \leq n)$	$\Gamma \Rightarrow t_i \downarrow \sigma_i$

Notes:

- φ' is the result of simultaneously replacing each free occurrence of x_1 in φ with t_i, for each i with $1 \leq i \leq n$, if necessary renaming bound variables in φ to avoid variable captures.

Related commands: **auto-instantiate-existential**.

instantiate-theorem

Script usage `(instantiate-theorem` *theorem instantiations*`)`. *theorem* is the name of a theorem, and *instantiations* is a list of strings denoting expressions.

Interactive argument retrieval:

- `Theorem name:` The name of a theorem $\forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \psi$ in the deduction graph's theory.
- `Instance for variable` x_1 `of sort` σ_1`:` t_1.
  $\vdots$
- `Instance for variable` x_n `of sort` σ_n`:` t_n $(n \geq 1)$.

Command kind: Multi-inference.

Description: This command instantiates the argument theorem with the specified terms $t_1, \ldots, t_n$ and then adds the resulting formula to the context of the given sequent.

Conclusion	$\Gamma \Rightarrow \varphi$
Major Premise	$\Gamma \cup \{\psi'\} \Rightarrow \varphi$
Minor Premises $(1 \leq i \leq n)$	$\Gamma \Rightarrow t_i \downarrow \sigma_i$

Notes:

- ψ' is the result of simultaneously replacing each free occurrence of x_i in ψ with t_i, for each i with $1 \leq i \leq n$, if necessary renaming bound variables in ψ to avoid variable captures.

Related commands:
assume-theorem
instantiate-transported-theorem

instantiate-transported-theorem

Script usage: `(instantiate-transported-theorem` *theorem interpretation instantiations*`)`. *theorem* is the name of a theorem, *interpretation* is the name of a theory interpretation, and *instantiations* is a list of strings denoting expressions.

Interactive argument retrieval:

- `Theorem name:` The name of a theorem $\forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \psi$ in T.
- `Theory interpretation (<RET> to let IMPS find one):` The name of a theory interpretation i of T in the deduction graph's theory.
- `Instance for variable` x_1 `of sort` σ_1`:` t_1.
  ⋮
- `Instance for variable` x_n `of sort` σ_n`:` t_n $(n \geq 1)$.

Command kind: Multi-inference.

Description: This command transports the argument theorem to the deduction graph's theory via i. Then, the transported theorem is instantiated with the specified terms $t_1, \ldots, t_n$. And, finally, the resulting formula is added to the context of the given sequent.

If no theory interpretation name is given, IMPS will try to find a theory interpretation to play the role of i using the sort information contained in the terms $t_1, \ldots, t_n$.

Related commands:
assume-transported-theorem
instantiate-theorem

Remarks: IMPS can often find an appropriate theory interpretation automatically when the home theory of the theorem is a generic theory, i.e., a theory which contains neither constants nor axioms.

instantiate-universal-antecedent

Script usage: `(instantiate-universal-antecedent` *assumption instantiations*`)`. *assumption* can be given as an integer i or a string σ, and *instantiations* is a list of strings denoting expressions.

Interactive argument retrieval:

- `0-based index of universal antecedent formula` `(`$i_1\ i_2\ \cdots\ i_n$`):` i. The $i_1, i_2, \ldots, i_n$ are the indices of the universal assumptions of the sequent node. In case there is only one universal antecedent formula, this argument request is omitted.
- `Instance for variable` x_1 `of sort` σ_1`:` t_1.
  $\vdots$
- `Instance for variable` x_n `of sort` σ_n`:` t_n $(n \geq 1)$.

Command kind: Multi-inference.

Description: This command instantiates the i-th assumption of the context of the given sequent (provided the assumption is a universal statement).

Conclusion	$\Gamma \cup \{\forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \psi\} \Rightarrow \varphi$
Major Premise	$\Gamma \cup \{\psi'\} \Rightarrow \varphi$
Minor Premises $(1 \leq i \leq n)$	$\Gamma \Rightarrow t_i \downarrow \sigma_i$

Notes:

- The index of the assumption $\forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \psi$ in the context $\Gamma \cup \{\forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \psi\}$ is i.
- ψ' is the result of simultaneously replacing each free occurrence of x_1 in ψ with t_i, for each i with $1 \leq i \leq n$, if necessary renaming bound variables in ψ to avoid variable captures.

Related commands:
auto-instantiate-universal-antecedent
instantiate-universal-antecedent-multiply

instantiate-universal-antecedent-multiply

Script usage: `(instantiate-universal-antecedent-multiply` *assumption lists-of-instantiations*`)`. *assumption* can be given as an integer i or a string σ, and *lists-of-instantiations* is a list of lists of strings which specify the expressions that instantiate the universally quantified assumption.

Interactive argument retrieval:

- `0-based index of antecedent-formula:` i. If there is only one assumption, this argument request is omitted.
- `First instance term:` t_1.
- `Next instance term (<RET> if done):` t_2.
  ⋮
- `Next instance term (<RET> if done):` t_n $(n \geq 1)$.
- `Input terms for another instance? (y or n)`
  ⋮

Command kind: Multi-inference.

Description: This command produces one or more instances of the i-th assumption of the context of the given sequent (provided the assumption is a universal statement) in the same way that **instantiate-universal-antecedent** produces one instance of the assumption.

Related commands:
auto-instantiate-universal-antecedent
instantiate-universal-antecedent

prove-by-logic-and-simplification

Script usage: `(prove-by-logic-and-simplification` *persistence*`)`.

Interactive argument retrieval:

- `Backchaining persistence:` An integer, which is 3 by default.

Command kind: Multi-inference.

Description This command tries to ground a sequent by applying a list inference procedures (some of which have a sequent assumption as an additional argument) to the goal node and recursively to the generated subgoal nodes. See Table 18.1. When an inference procedure is applied to the goal sequent or to a subgoal sequent, two possibilities can occur:

(1) The inference procedure fails. In this case, **prove-by-logic-and-simplification** tries the same inference procedure with the next assumption as argument (if this makes sense), the next inference procedure on the list, if one exists, or otherwise backtracks.

(2) The inference procedure succeeds, either grounding the node or generating one or more new subgoals to prove. In this case we attempt to prove each new subgoal by applying the primitive inferences in order.

Remarks: This command is an example of an ending strategy, that is, a proof construction procedure to be used when all that remains to be done is completely straightforward reasoning. It should be used with great caution since in the cases in which it fails to ground a node it may generate a large number of irrelevant subgoals.

Key binding: None, lest you accidently hit it.

raise-conditional

Script usage: `(raise-conditional` *occurrences*`)`. *occurrences* is given by a list of integers.

Interactive argument retrieval:

- `Occurrences of conditionals to be raised (0-based):` A list l given in the form $l_1 \cdots l_n$.

Command kind: Single-inference.

Condition	Procedure	Action
do-simplify?	simplify	lower do-simplify?
0 < persist	backchain-through-formula	lower persist raise do-simplify?
	antecedent-inference for existentials and conjunctions	
	direct-inference	
	antecedent-inference for other assumptions	raise do-simplify?
0 < persist	backchain-inference	
	sort-definedness	
	definedness	
	extensionality	raise do-simplify?
	conditional-inference followed by direct-inference	raise do-simplify?

Notes:

(1) The variable *persist* is the backchaining persistence supplied as an argument to the command. *lower persist* means reduce backchaining persistence by 1.

(2) *do-simplify?* is a boolean flag which starts off as *true* when the command is called.

Table 18.1: Search Order for **prove-by-logic-and-simplification**

Description: This command will "raise" a subset of the conditional expressions (i.e., expressions whose lead constructor is **if**), specified by l, in the assertion of the given sequent $\Gamma \Rightarrow \varphi$.

For the moment, let us assume that $n = 1$. Suppose the l_1-th occurrence of a conditional expression in φ is the a-th occurrence of $s = \text{if}(\theta, t_1, t_2)$ in φ; the smallest formula containing the a-th occurrence of s in φ is the b-th occurrence of ψ in φ; and the a-th occurrence of s in φ is the c-th occurrence of s in ψ. If every free occurrence of a variable in s is also a free occurrence in ψ, the command reduces the sequent to

$$\Gamma \Rightarrow \varphi[\text{if-form}(\theta, \psi[t_1/s]_c, \psi[t_2/s]_c)/\psi]_b,$$

and otherwise the command fails.

Now assume $n > 1$. The command will then simultaneously raise each specified occurrence of a conditional expression in φ, in the manner of the previous paragraph, if there are no conflicts between the occurrences. The kinds of conflicts that can arise and how they are resolved are listed below:

- If two or more specified occurrences of a conditional expression s in φ are contained in same smallest formula, they are raised together.
- If two or more specified occurrences of distinct conditional expressions in φ are contained in same smallest formula, at most one of the occurrences is raised.
- If ψ_1 and ψ_2 are the smallest formulas respectively containing two specified occurrences of a conditional expression in φ and ψ_1 is a proper subexpression of ψ_2, then the first specified conditional expression is not raised.

Related commands: **case-split-on-conditionals**.

Remarks: This command can be used to change a conditional expression $\text{if}(\theta, \varphi_1, \varphi_2)$ in a sequent's assertion, where φ_1 and φ_2 are formulas, to the conditional formula $\text{if-form}(\theta, \varphi_1, \varphi_2)$.

Key binding: `r`

simplify

Script usage: `simplify`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command simplifies the assertion of the given sequent with respect to the context of the sequent. It uses both theory-specific and general logical methods to reduce the sequent to a logically equivalent sequent. The theory-specific methods include algebraic simplification, deciding rational linear inequalities, and applying rewrite rules.

Related commands:
beta-reduce
simplify-antecedent
simplify-insistently
simplify-with-minor-premises

Remarks: This is a very powerful command that can be computationally expensive. Computation can often be saved by using the weaker command **beta-reduce**.

Key binding: `C-c s`

simplify-antecedent

Script usage: `(simplify-antecedent` *assumption*`)`. *assumption* can be given as an integer i or a string σ.

Interactive argument retrieval:

- `0-based index of antecedent-formula:` i. If there is only one assumption, this argument request is omitted.

Command kind: Multi-inference.

Description: This command is used for simplifying an assumption in the context of the given sequent with respect to this context. It is equivalent to the following sequence of commands: **contrapose** applied to the given sequent with argument i; **simplify** applied to the sequent yielded by the previous command; and **contrapose** applied to the sequent yielded by the previous command with the index of the negated assertion of the original sequent. The commands halts if **simplify** grounds the first produced sequent.

Related commands::
beta-reduce-antecedent
simplify

simplify-insistently

Script usage: `simplify-insistently`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command is equivalent to disabling all quasi-constructors and then calling **simplify**.

Related commands:
beta-reduce-insistently
insistent-direct-inference
simplify

Remarks: There is rarely any need to use this command. It has the disagreeable effect of exploding quasi-constructors.

simplify-with-minor-premises

Script usage: `simplify-with-minor-premises`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command is the same as **simplify** except that, instead of failing when convergence requirements are not verified, it posts the unverified convergence requirements as additional subgoals to be proved.

Related commands::
beta-reduce-with-minor-premises
simplify

Remarks: This command is useful for identifying convergence requirements that the simplifier cannot verify.

sort-definedness

Script usage: `sort-definedness`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: This command applies to a sequent whose assertion is a definedness statement of the form $(t \downarrow \sigma)$. The command first tests whether the context entails the definedness of t in σ. If the test fails, the command then tries to reduce the sequent to a set of simpler sequents. In particular, when t is a conditional term $\text{if}(\varphi, t_0, t_1)$, it distributes the sort definedness assertion into the consequent and alternative. If t_0 and t_1 are not themselves conditional terms, the new subgoal has the assertion $\text{if-form}(\varphi, (t_0 \downarrow \sigma), (t_1 \downarrow \sigma))$. If one or both of them is a conditional term, then the sort definedness assertion is recursively distributed into the consequents and alternatives.

Related commands: **definedness**.

Remarks: This command is mainly useful when t is an application, a function, or a conditional term.

sort-definedness-and-conditionals

Interactive argument retrieval: None.

Command kind: Multi-inference.

Description: This strategy invokes sort-definedness. Insistent direct inference and repeated beta reduction are then invoked, followed by case-split-on-conditionals, applied to the first conditional term.

Related commands: **sort-definedness**.

Remarks: This command is useful for a goal $(t \downarrow \sigma)$ when the definition of σ involves a conditional term.

unfold-defined-constants

Script usage: `unfold-defined-constants`.

Interactive argument retrieval: None.

Command kind: Multi-inference.

Description: This command replaces every occurrence of a defined constant by its respective unfolding. The command **beta-reduce-repeatedly** is called after all the unfoldings are performed.

Related commands:
unfold-defined-constants-repeatedly
unfold-directly-defined-constants
unfold-recursively-defined-constants
unfold-single-defined-constant

Key binding: `C-c u`

unfold-defined-constants-repeatedly

Script usage: `unfold-defined-constants-repeatedly`.

Interactive argument retrieval: None.

Command kind: Multi-inference.

Description: This command replaces every occurrence of a defined constant by its respective unfolding, repeatedly, until there are no occurrences of defined constants. The command **beta-reduce-repeatedly** is called after all the unfoldings are performed.

Related commands:
unfold-defined-constants
unfold-directly-defined-constants-repeatedly
unfold-recursively-defined-constants-repeatedly
unfold-single-defined-constant

Remarks: If there are occurrences of recursively defined constants, this command can run forever.

unfold-directly-defined-constants

Script usage: `unfold-directly-defined-constants`.

Interactive argument retrieval: None.

Command kind: Multi-inference.

Description: This command replaces every occurrence of a directly defined constant by its respective unfolding. (A *directly defined constant* is a constant defined nonrecursively.) The command **beta-reduce-repeatedly** is called after all the unfoldings are performed.

Related commands:
unfold-defined-constants
unfold-directly-defined-constants-repeatedly
unfold-recursively-defined-constants
unfold-single-defined-constant

unfold-directly-defined-constants-repeatedly

Script usage: `unfold-directly-defined-constants-repeatedly`.

Interactive argument retrieval: None.

Command kind: Multi-inference.

Description: This command replaces every occurrence of a directly defined constant by its respective unfolding, repeatedly, until there are no occurrences of directly defined constants. (A *directly defined constant* is a constant defined nonrecursively.) The command **beta-reduce-repeatedly** is called after all the unfoldings are performed.

Related commands:
unfold-defined-constants-repeatedly
unfold-directly-defined-constants
unfold-recursively-defined-constants-repeatedly
unfold-single-defined-constant

Remarks: This command will always terminate, unlike **unfold-defined-constants-repeatedly** and **unfold-recursively-defined-constants-repeatedly**.

unfold-recursively-defined-constants

Script usage: `unfold-recursively-defined-constants`.

Interactive argument retrieval: None.

Command kind: Multi-inference.

Description: This command replaces every occurrence of a recursively defined constant by its respective unfolding. The command **beta-reduce-repeatedly** is called after all the unfoldings are performed.

Related commands:
unfold-defined-constants
unfold-directly-defined-constants
unfold-recursively-defined-constants-repeatedly
unfold-single-defined-constant

unfold-recursively-defined-constants-repeatedly

Script usage: `unfold-recursively-defined-constants-repeatedly`.

Interactive argument retrieval: None.

Command kind: Multi-inference.

Description: This command replaces every occurrence of a recursively defined constant by its respective unfolding, repeatedly, until there are no occurrences of recursively defined constants. The command **beta-reduce-repeatedly** is called after all the unfoldings are performed.

Related commands:
unfold-defined-constants-repeatedly
unfold-directly-defined-constants-repeatedly
unfold-recursively-defined-constants
unfold-single-defined-constant

Remarks: This command may run forever.

unfold-single-defined-constant

Script usage: `(unfold-single-defined-constant` *occurrences constant*`)`. *occurrences* is a list of integers, and *constant* is denoted by the constant name.

Interactive argument retrieval:

- `Constant name:` *c*.

- `Occurrences to unfold (0-based):` A list l given in the form $l_1 \ \cdots \ l_n$.

Notice that the order in which arguments are requested is different than the order for script usage.

Command kind: Multi-inference.

Description: This command replaces each specified occurrence of the defined constant c by its unfolding e:

Conclusion	$\Gamma \Rightarrow \varphi$
Premise	$\Gamma \Rightarrow \varphi[e/c]_l$

The command **beta-reduce-repeatedly** is called after all the unfoldings are performed.

Related commands:
unfold-defined-constants
unfold-defined-constants-repeatedly
unfold-directly-defined-constants
unfold-directly-defined-constants-repeatedly
unfold-recursively-defined-constants
unfold-recursively-defined-constants-repeatedly
unfold-single-defined-constant-globally

Remarks: The related commands are all elaborations of this command.

Key binding: `u`

unfold-single-defined-constant-globally

Script usage: `(unfold-single-defined-constant-globally` *constant*`)`.

Interactive argument retrieval:

- `Constant name:` c.

Command kind: Multi-inference.

Description: This command replaces every occurrence of the defined constant c by its unfolding e:

Conclusion	$\Gamma \Rightarrow \varphi$
Premise	$\Gamma \Rightarrow \varphi[e/c]_{\text{all}}$

The command **beta-reduce-repeatedly** is called after all the unfoldings are performed.

Related commands: **unfold-single-defined-constant**.

Key binding: `U`

unordered-direct-inference

Script usage: `unordered-direct-inference`.

Interactive argument retrieval: None.

Command kind: Single-inference.

Description: If the sequent node assertion is a conjunction, this command does a direct inference without strengthening the context.

Conclusion	$\Gamma \Rightarrow \varphi_1 \wedge \cdots \wedge \varphi_n$
Premises ($1 \leq i \leq n$)	$\Gamma \Rightarrow \varphi_i$

Related commands: **direct-inference**.

weaken

Script usage: `(weaken` *assumption-list*`)`. *assumption-list* can be given as a list of numbers, a list of strings, or an assumption-list form.

Interactive argument retrieval:

- List of formula indices to omit (0-based): A list l given in the form $l_1 \ \cdots \ l_n$.

Command Kind: Single-inference.

Description: This command *removes* one or more assumptions you specify from the context of the given sequent.

Conclusion	$\Gamma \cup \Delta \Rightarrow \varphi$
Premise	$\Gamma \Rightarrow \varphi$

Notes:

- The indices of the members of Δ in the context $\Gamma \cup \Delta$ are given by l.

Remarks: You might wonder why you would ever want to remove an assumption, but in fact, in many cases this is a natural step to take:

- If an assumption is irrelevant, then removing it will do no harm and will make the job of the simplifier a lot easier,
- It is often the case that sequent nodes are identical except for the addition of "irrelevant" assumptions. In this case, removing the irrelevant assumptions allows you to ground both sequents by just grounding one.

Key binding: `w`

Chapter 19

The Primitive Inference Procedures

In this chapter we list and document each of the primitive inference procedures. We will use the following format to describe them.

Parameters. A list of the arguments (other than the sequent node) required by procedure. These arguments can be of the following types:

- A formula.
- A list of formulas.
- A path represented as a list of nonnegative integers.
- A list of paths.
- A constant.
- A macete.
- Another sequent node (usually referred to as the *major premise*).

Description. A brief description of the primitive inference. Some of primitive inference procedures have descriptions which are identical to the description of the corresponding interactive proof command.

antecedent-inference

Parameters: A sequent node assumption.

Description: The effect of applying this primitive inference procedure is given by the following table.

Conclusion	$\Gamma \cup \{\varphi \supset \psi\} \Rightarrow \theta$
Premises	$\Gamma \cup \{\neg\varphi\} \Rightarrow \theta$
	$\Gamma \cup \{\psi\} \Rightarrow \theta$
Conclusion	$\Gamma \cup \{\varphi_1 \wedge \cdots \wedge \varphi_n\} \Rightarrow \psi$
Premise	$\Gamma \cup \{\varphi_1, \ldots, \varphi_n\} \Rightarrow \psi$
Conclusion	$\Gamma \cup \{\varphi_1 \vee \cdots \vee \varphi_n\} \Rightarrow \psi$
Premises $(1 \leq i \leq n)$	$\Gamma \cup \{\varphi_i\} \Rightarrow \psi$
Conclusion	$\Gamma \cup \{\varphi \equiv \psi\} \Rightarrow \theta$
Premises	$\Gamma \cup \{\varphi, \psi\} \Rightarrow \theta$
	$\Gamma \cup \{\neg\varphi, \neg\psi\} \Rightarrow \theta$
Conclusion	$\Gamma \cup \{\text{if-form}(\varphi, \psi, \theta)\} \Rightarrow \xi$
Premises	$\Gamma \cup \{\varphi, \psi\} \Rightarrow \xi$
	$\Gamma \cup \{\neg\varphi, \theta\} \Rightarrow \xi$
Conclusion	$\Gamma \cup \{\exists x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi\} \Rightarrow \psi$
Premise	$\Gamma \cup \{\varphi'\} \Rightarrow \psi$

Notes:

- Each sequent in the preceding table is of the form $\Gamma \cup \{\varphi\} \Rightarrow \psi$, where $\{\varphi\}$ is the parameter to the primitive inference procedure.
- φ' is obtained from φ by renaming the variables among $x_1, \ldots, x_n$ which are free in both φ and the original sequent.

backchain-inference

Parameters: A sequent node assumption.

Description This primitive inference procedure attempts to use the assumption given as argument to replace the assertion to be proved. In the simplest case, if the given assumption is $\varphi \supset \psi$ and the assertion is ψ, then backchain-inference replaces the assertion with φ. Similarly, if the assertion is $\neg\varphi$, it is replaced with $\neg\psi$. The command extends this simplest case in four ways:

- If the assumption is universally quantified, then matching is used to select a relevant instance.
- If the assumption is a disjunction

$$\bigvee_{i \in I} \varphi_i,$$

  then, for any $j \in I$, it may be treated as

$$(\bigwedge_{i \in I, i \neq j} \neg \varphi_i) \supset \varphi_j.$$

- If the assumption is of one of the forms $s = t$, $s \simeq t$, $s \equiv t$, if a subexpression of the assertion matches s, then it is replaced by the corresponding instance of t.
- If the assumption is a conjunction, each conjunct is tried separately, in turn.
- These rules are used iteratively to descend through the structure of the assumption as deeply as necessary.

If IMPS cannot recognize that the terms returned by a match are defined in the appropriate sorts, then these assertions are returned as additional minor premises that must later be grounded to complete the derivation.

backchain-backwards-inference

Parameters: A sequent node assumption.

Description: This primitive inference procedure works the same way as **backchain**, except that subformulas of the assumption of the forms $s = t$, $s \simeq t$, $s \equiv t$ are used from right to left.

backchain-through-formula-inference

Parameters: A sequent node assumption.

Description: Similar to **backchain**, except that it does not backchain through equivalences, i.e., formulas of the form $s = t$, $s \simeq t$, or $s \equiv t$.

choice

Parameters: None.

Description: This primitive inference procedure implements a version of the axiom of choice.

Conclusion	$\Gamma \Rightarrow \exists f{:}[\sigma_1, \ldots, \sigma_n, \tau], \forall x_1{:}\sigma'_1, \ldots, x_n{:}\sigma'_n, \varphi$
Premise	$\Gamma \Rightarrow \forall x_1{:}\sigma'_1, \ldots, x_n{:}\sigma'_n, \exists y_f{:}\tau, \varphi[y_f/f(x_1, \ldots, x_n)]_{\text{all}}$

Notes:

- σ_i and σ'_i have the same type for all i with $1 \leq i \leq n$.
- The command fails if there is any occurrence of f in φ which is not in an application of the form $f(x_1, \ldots, x_n)$.

contraposition

Parameters: A sequent node assumption φ.

Description: This primitive inference procedure interchanges the given sequent's assertion and the assumption φ.

Conclusion	$\Gamma \cup \{\varphi\} \Rightarrow \psi$
Premise	$\Gamma \cup \{\neg\psi\} \Rightarrow \neg\varphi$

cut

Parameters: A sequent node to be the major premise.

Description:

Conclusion	$\Gamma \Rightarrow \varphi$
Major Premise	$\Gamma \cup \{\psi_1, \ldots, \psi_n\} \Rightarrow \varphi$
Minor Premises ($1 \leq i \leq n$)	$\Gamma \Rightarrow \psi_i$

Notes:

- The major premise is the sequent $\Gamma \cup \{\psi_1, \ldots, \psi_n\} \Rightarrow \varphi$.

definedness

Parameters: None.

Description: This primitive inference procedure applies to a sequent whose assertion is a definedness statement of the form $t\downarrow$. The primitive inference procedure first tests whether the context entails the definedness of t. If the test fails, the primitive inference then tries to reduce the sequent to a set of simpler sequents.

defined-constant-unfolding

Parameters:

- A list of paths $(p_1, \ldots, p_n)$ to occurrences of a defined constant c.
- The constant c itself.

Description: This primitive inference procedure replaces occurrences of the defined constant c at the paths p_i by its unfolding e.

direct-inference

Parameters: None.

Description: This primitive inference procedure applies an analogue of an introduction rule of Gentzen's sequent calculus (in reverse), based on the lead constructor of the assertion of the given sequent.

Conclusion	$\Gamma \Rightarrow \varphi \supset \psi$
Premise	$\Gamma \cup \{\varphi\} \Rightarrow \psi$
Conclusion	$\Gamma \Rightarrow \varphi_1 \wedge \cdots \wedge \varphi_n$
Premises $(1 \leq i \leq n)$	$\Gamma \cup \{\varphi_1, \ldots, \varphi_{i-1}\} \Rightarrow \varphi_i$
Conclusion	$\Gamma \Rightarrow \varphi_1 \vee \cdots \vee \varphi_n$
Premise	$\Gamma \cup \{\neg\varphi_1, \ldots, \neg\varphi_{n-1}\} \Rightarrow \varphi_n$
Conclusion	$\Gamma \Rightarrow \varphi \equiv \psi$
Premises	$\Gamma \cup \{\varphi\} \Rightarrow \psi$
	$\Gamma \cup \{\psi\} \Rightarrow \varphi$
Conclusion	$\Gamma \Rightarrow \text{if-form}(\varphi, \psi, \theta)$
Premises	$\Gamma \cup \{\varphi\} \Rightarrow \psi$
	$\Gamma \cup \{\neg\psi\} \Rightarrow \theta$
Conclusion	$\Gamma \Rightarrow \forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi$
Premise	$\Gamma \Rightarrow \varphi'$

Notes:

- φ' is obtained from φ by renaming the variables among $x_1, \ldots, x_n$ which are free in both φ and the context Γ.

disjunction-elimination

Parameters: A sequent node.

Description: NO DESCRIPTION.

eliminate-iota

Parameters: A path p to an iota-expression occurrence in the assertion.

Description: This primitive inference procedure applies to a sequent whose assertion is an atomic formula or negated atomic formula containing a specified occurrence of an iota-expression. The command reduces the sequent to an equivalent sequent in which the specified iota-expression occurrence is "eliminated."

Conclusion	$\Gamma \Rightarrow \psi$
Major Premise	$\Gamma \Rightarrow \exists y{:}\sigma, (\varphi[y/x]_{\text{free}} \wedge \forall z{:}\sigma, (\varphi[z/x]_{\text{free}} \supset z = y) \wedge \psi')$

Conclusion	$\Gamma \Rightarrow \neg\psi$
Major Premise	$\Gamma \Rightarrow \exists y{:}\sigma, (\varphi[y/x]_{\text{free}} \wedge \forall z{:}\sigma, (\varphi[z/x]_{\text{free}} \supset z = y)) \supset$ $\exists y{:}\sigma, (\varphi[y/x]_{\text{free}} \supset \neg\psi')$

Notes:

- ψ is an atomic formula.
- $\iota x{:}\sigma, \varphi$ is the expression located at the path p in ψ.
- ψ' is the result of replacing the i-th iota-expression occurrence in ψ with y.
- The occurrence of $\iota x{:}\sigma, \varphi$ in ψ is within some argument component s of ψ. Moreover, the occurrence is an extended application component of s, where a is an *extended application component* of b if either a is b or b is an application of kind ι and a is an extended application component of a component of b.

existential-generalization

Parameters: A sequent node $\Gamma \Rightarrow \varphi$.

Description: This primitive inference procedure proves an existential assertion by exhibiting witnesses.

Conclusion	$\Gamma \Rightarrow \exists x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi$
Premise	$\Gamma \Rightarrow \varphi$

Notes:

- The command will fail unless the parameter sequent node is of the form $\Gamma \Rightarrow \varphi$.

extensionality

Parameters: None.

Description: If the sequent node assertion is an equality or quasi-equality or the negation of an equality or quasi-equality of n-ary functions, this primitive inference applies the extensionality principle.

Conclusion	$\Gamma \Rightarrow f = g$
Major Premise	$\Gamma \Rightarrow \forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, f(x_1, \cdots, x_n) \simeq g(x_1, \cdots, x_n)$
Minor Premise	$\Gamma \Rightarrow f\downarrow$
Minor Premise	$\Gamma \Rightarrow g\downarrow$

Conclusion	$\Gamma \Rightarrow f \simeq g$
Major Premise	$\Gamma \Rightarrow \forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, f(x_1, \cdots, x_n) \simeq g(x_1, \cdots, x_n)$
Minor Premise	$\Gamma \cup \{g\downarrow\} \Rightarrow f\downarrow$
Minor Premise	$\Gamma \cup \{f\downarrow\} \Rightarrow g\downarrow$

Conclusion	$\Gamma \Rightarrow \neg f = g$
Premise	$\Gamma \Rightarrow \exists x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \neg f(x_1, \cdots, x_n) \simeq g(x_1, \cdots, x_n)$

Conclusion	$\Gamma \Rightarrow \neg f \simeq g$
Premise	$\Gamma \Rightarrow \exists x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \neg f(x_1, \cdots, x_n) \simeq g(x_1, \cdots, x_n)$

Notes:

- If the sorts of f and g are $[\alpha_1, \ldots, \alpha_n, \alpha_{n+1}]$ and $[\beta_1, \ldots, \beta_n, \beta_{n+1}]$, respectively, then $\sigma_i = \alpha_i \sqcup \beta_i$ for all i with $1 \leq i \leq n$.
- A sequent node corresponding to a minor premise is not created if the truth of the minor premise is immediately apparent to IMPS.
- If f and g are of kind $*$, then "=" is used in place of "$\simeq$" in the premise.

force-substitution

Parameters:

- Paths. A list of paths $p_1, \ldots, p_n$.
- Replacements. A list of expressions $r_1, \ldots, r_n$.

Description: This primitive inference procedure changes part of the sequent assertion. It yields two or more new subgoals. One subgoal is obtained from the original sequent assertion by replacing the subexpression e_i at the path p_i with r_i. The other subgoals assert that the replacements are sound.

Conclusion	$\Gamma \Rightarrow \varphi$
Major Premise	$\Gamma \Rightarrow \varphi[r_i/e_i]$
Minor Premises $(1 \leq i \leq n)$	$\Gamma_i \Rightarrow \varphi_i$

Notes:

- Γ_i is the local context of $\Gamma \Rightarrow \varphi$ at the path p_i.
- φ_i is:
  - $e_i \equiv r_i$ if e_i is a formula and the parity of the path p_i is 0.
  - $r_i \supset e_i$ if e_i is a formula and the parity of the path p_i is 1.
  - $e_i \supset r_i$ if e_i is a formula and the parity of the path p_i is -1.
  - $e_i \simeq r_i$ in all other cases.

incorporate-antecedent

Parameters: A sequent node assumption φ.

Description: This primitive inference procedure does the reverse of **direct-inference** on an implication.

Conclusion	$\Gamma \cup \{\varphi\} \Rightarrow \psi$
Premise	$\Gamma \Rightarrow \varphi \supset \psi$

insistent-direct-inference

Parameters: None.

Description: This primitive inference procedure is equivalent to disabling all quasi-constructors and then calling **direct-inference**.

insistent-simplification

Parameters: None.

Description: This primitive inference procedure is equivalent to disabling all quasi-constructors and then calling **simplification**.

macete-application-at-paths

Parameters:

- A list of paths $(p_1, \ldots, p_n)$.
- A macete.

Description: This primitive inference procedure applies the argument macete to the sequent node assertion at those occurrences specified by the paths $p_1, \ldots, p_n$.

macete-application-with-minor-premises-at-paths

Parameters:

- A list of paths $(p_1, \ldots, p_n)$.
- A macete

Description: This primitive inference procedure applies the argument macete to the given sequent node, in the same way as macete-application, but with the following important difference: whenever the truth or falsehood of a convergence requirement cannot be determined by the simplifier, the assertion is posted as a additional subgoal to be proved.

raise-conditional-inference

Parameters: A list of paths $(p_1, \ldots, p_n)$ to conditional subexpressions.

Description: This primitive inference procedure will "raise" a subset of the conditional expressions (i.e., expressions whose lead constructor is **if**), specified by the paths p_i, in the assertion of the given sequent $\Gamma \Rightarrow \varphi$.

For the moment, let us assume that $n = 1$. Suppose the conditional expression in φ located at p_1 is the a-th occurrence of $s = \text{if}(\theta, t_1, t_2)$ in φ; the smallest formula containing the a-th occurrence of s in φ is the b-th occurrence of ψ in φ; and the a-th occurrence of s in φ is the c-th occurrence of s in ψ. If every free occurrence of a variable in s is also a free occurrence in ψ, the primitive inference procedure reduces the sequent to

$$\Gamma \Rightarrow \varphi[\text{if-form}(\theta, \psi[t_1/s]_c, \psi[t_2/s]_c)/\psi]_b,$$

and otherwise the primitive inference procedure fails.

Now assume $n > 1$. The primitive inference procedure will then simultaneously raise each specified occurrence of a conditional expression in φ, in the manner of the previous paragraph, if there are no conflicts between the occurrences. The kinds of conflicts that can arise and how they are resolved are listed below:

- If two or more specified occurrences of a conditional expression s in φ are contained in same smallest formula, they are raised together.
- If two or more specified occurrences of distinct conditional expressions in φ are contained in same smallest formula, at most one of the occurrences is raised.
- If ψ_1 and ψ_2 are the smallest formulas respectively containing two specified occurrences of a conditional expression in φ and ψ_1 is a proper subexpression of ψ_2, then the first specified conditional expression is not raised.

simplification

Parameters: None.

Description: This primitive inference procedure simplifies the assertion of the given sequent with respect to the context of the sequent. It uses both theory-specific and general logical methods to reduce the sequent to a logically equivalent sequent. The theory-specific methods include algebraic simplification, deciding rational linear inequalities, and applying rewrite rules.

simplification-with-minor-premises

Parameters: None.

Description: This primitive inference procedure is the same as **simplification** except that, instead of failing when convergence requirements are not verified, it posts the unverified convergence requirements as additional subgoals to be proved.

sort-definedness

Parameters: None.

Description: This primitive inference procedure applies to a sequent whose assertion is a definedness statement of the form $(t \downarrow \sigma)$. The primitive inference procedure first tests whether the context entails the definedness of t in σ. If the test fails, the primitive inference procedure then tries to reduce the sequent to a set of simpler sequents. In particular, when t is a conditional term $\text{if}(\varphi, t_0, t_1)$, it distributes the sort definedness assertion into the consequent and alternative. If t_0 and t_1 are not themselves conditional terms, the new subgoal has the assertion $\text{if-form}(\varphi, (t_0 \downarrow \sigma), (t_1 \downarrow \sigma))$. If one or both of them is a conditional term, then the sort definedness assertion is recursively distributed into the consequents and alternatives.

theorem-assumption

Parameters: A formula which must be a theorem of the context theory.

Description: This primitive inference procedure adds the argument theorem to the context of the given sequent.

universal-instantiation

Parameters: A sequent node $\Gamma \Rightarrow \forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi$.

Description: This primitive inference procedure proves a formula by proving a universal.

Conclusion	$\Gamma \Rightarrow \varphi$
Premise	$\Gamma \Rightarrow \forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi$

Notes:

- The command will fail unless the parameter sequent node is of the form $\Gamma \Rightarrow \forall x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi$.

unordered-conjunction-direct-inference

Parameters: None.

Description: If the sequent node assertion is a conjunction, this primitive inference procedure does a direct inference without strengthening the context.

Conclusion	$\Gamma \Rightarrow \varphi_1 \wedge \cdots \wedge \varphi_n$
Premises ($1 \leq i \leq n$)	$\Gamma \Rightarrow \varphi_i$

weakening

Parameters: A set of formulas Δ.

Description: This primitive inference procedure *removes* one or more assumptions you specify from the context of the given sequent.

Conclusion	$\Gamma \cup \Delta \Rightarrow \varphi$
Premise	$\Gamma \Rightarrow \varphi$

Chapter 20

The Initial Theory Library: An Overview

A theory library is a collection of theories, theory interpretations, and theory constituents (e.g., definitions and theorems) which serves as a database of mathematics. The basic unit of a theory library is a *section*; each section is a body of mathematics. Each section consists of a sequence of def-form specifications which are stored in a set of text files.

The IMPS initial theory library contains a large variety of basic mathematics. It is intended to provide you with a starting point for building your own theory library. It also is a source of examples that illustrates some of the diverse ways mathematics can be formulated in IMPS.

This chapter contains only a sampling of the sections that are contained in the IMPS initial theory library with a partial description of each section. Some of the descriptions contain a listing of the theories and some of the more important theorems and definitions included in that section.

Each section description begins with the symbol § followed by the section name. Each theory description begins with the symbol ¶ followed by the theory name.

§ abstract-calculus

The main theory of this section is **normed-spaces**. The section develops the basic notions of calculus (e.g., the derivative and the integral) in the context of normed spaces.

§ banach-fixed-point-theorem

This section has a proof of the Banach contractive mapping fixed point theorem, which states that any contractive mapping on a complete metric space has a unique fixed point.

§ basic-cardinality

This section introduces the basic notions of cardinality. It also proves some basic theorems about finite cardinality such as the fact that every subset of a finite set is itself finite.

§ basic-fields

The main theory of this section is **fields**. The section develops the machinery of field theory sufficiently for installing an algebraic processor for simplification.

§ basic-group-theory

This section contains two principle theories: **groups** and **group-actions**. The section includes a proof that the quotient of a group by a normal subgroup is itself a group. Also, several interpretations of **group-actions** in **groups** are defined.

§ basic-monoids

This section contains the theories **monoid-theory** and **commutative-monoid-theory**. In **monoid-theory** a constant **monoid%prod** is defined recursively as the iterated product of the primitive monoid operation. Basic properties of this constant are proved in **monoid-theory** and then are transported to theories with their own iterated product operators, such as **h-o-real-arithmetic** with the operators $\sum$ and $\prod$.

§ binary-relations

Binary relations are represented in this section as certain partial functions in the same way that sets are represented as indicators. Several basic operations on binary relations are formalized as quasi-constructors. As an exercise, the transitive closure of the union of two equivalence relations is shown to be an equivalence relation.

§ binomial-theorem

This section proves the combinatorial identity and the binomial theorem for fields.

§ counting-theorems-for-groups

This section contains a proof of the fundamental counting theorem for group theory. Several consequences of this theorem are proved including Lagrange's theorem.

§ foundation

This section is always loaded with IMPS. It contains the theory **h-o-real-arithmetic** and a number of definitions, theorems, and macetes in this theory. It also adds a number of definitions and theorems in the "generic theories" named **generic-theory-1**, **generic-theory-2**, **generic-theory-3** and **generic-theory-4** and the theory **indicators**.

Definitions

The following constants in **h-o-real-arithmetic** are among those included in the section **foundation**.

- **sum** (denoted $\sum$ in the mathematical syntax) is the least fixed point of the functional:

$$\sigma \mapsto \lambda m, n{:}\mathbf{Z}, f{:}\mathbf{Z} \rightharpoonup \mathbf{R}, \mathrm{if}(m \leq n, \sigma(m, n-1, f) + f(n), 0)$$

- **prod** (denoted $\prod$ in the mathematical syntax) is the least fixed point of the functional:

$$\pi \mapsto \lambda m, n{:}\mathbf{Z}, f{:}\mathbf{Z} \rightharpoonup \mathbf{R}, \mathrm{if}(m \leq n, \pi(m, n-1, f)\, f(n), 1)$$

- **abs** (denoted $|\cdot|$ in the mathematical notation) is

$$\lambda r{:}\mathbf{R}, \mathrm{if}(0 \leq r, r, -r)$$

- **floor**: $\lambda x{:}\mathbf{R}, \iota z{:}\mathbf{Z}, z \leq x \wedge x < 1 + z$

§ groups-as-monoids

This section interprets **monoid-theory** in **groups** and then proves the telescoping product formula.

§ knaster-fixed-point-theorem

The main interest of this section is the statement and proof of the Knaster fixed point theorem which states that on a complete partial order with a least element and a greatest element, any monotone mapping has a fixed point. One important application of this result included in this section is a proof of the Schröder-Bernstein theorem.

§ metric-spaces

metric-spaces is the only new theory included in this section.

¶ metric-spaces

Contains **h-o-real-arithmetic** as a component theory.

Language:

- *Sorts*: **pp** (**P** in the mathematical syntax) denotes the underlying set of points of the metric space.
- *Constants*: **dist** denotes a real-valued, two-place function on **pp**.

Axioms:

(1) *Positivity.* $\forall x, y{:}\mathbf{P}, 0 \leq \mathrm{dist}(x, y)$

(2) *Separation.* $\forall x, y{:}\mathbf{P}, x = y \equiv \mathrm{dist}(x, y) = 0$

(3) *Symmetry.* $\forall x, y{:}\mathbf{P}, \mathrm{dist}(x, y) = \mathrm{dist}(y, x)$

(4) *Triangle Inequality.* $\forall x, y, z{:}\mathbf{P}, \mathrm{dist}(x, z) \leq \mathrm{dist}(x, y) + \mathrm{dist}(y, z)$

§ metric-space-pairs

This section introduces the theory **metric-space-pairs**.

¶ metric-space-pairs

This theory is defined by the theory ensemble mechanism. Alternatively, it can be described as follows:

Language: The language contains two copies of the language for metric spaces. Notice, however, that it does *not* contain two copies of the reals.

- *Sorts*:
  - **pp_0** ($\mathbf{P}_0$ in the mathematical syntax).
  - **pp_1** ($\mathbf{P}_1$ in the mathematical syntax).
- *Constants*:
  - **dist_0** (dist_0 in the mathematical syntax).
  - **dist_1** (dist_1 in the mathematical syntax).

Axioms: The axioms are the positivity, separation, symmetry, and triangle inequality axioms for both functions **dist_0** and **dist_1**. (See [13] for some example proofs.)

§ metric-space-continuity

This section develops continuity and related notions in the context of metric spaces.

§ number-theory

This section develops the rudiments of number theory. The fundamental theorem of arithmetic and the infinity of primes are proved.

§ pairs

A pair of elements a, b of sort α, β is represented as a function whose domain equals the singleton set $\{\langle a, b\rangle\}$. The basic operations on pairs are formalized as quasi-constructors.

§ partial-orders

This section contains the following theories:

- **partial-order**
- **complete-partial-order**

¶ partial-order

Language:

- *Base Types*: **uu** (**U** in the mathematical syntax) denotes the underlying set of points.
- *Constants*: **prec** ($\prec$ in the mathematical syntax) denotes a binary relation on **uu**.

Axioms:

- *Transitivity.* $\forall a, b, c{:}\mathbf{U}, a \prec b \wedge b \prec c \supset a \prec c$
- *Reflexivity.* $\forall a{:}\mathbf{U}, a \prec a$
- *Anti-symmetry.* $\forall a, b{:}\mathbf{U}, a \prec b \wedge b \prec a \supset a = b$

¶ complete-partial-order

Axioms: In addition to the axioms for the theory **partial-order**, this theory has the completeness axiom which states that any nonempty set with an upper bound has a least upper bound. In a complete partial order it is also true that any nonempty set bounded below has a greatest lower bound.

§ pre-reals

Two IMPS theories of the real numbers are included in the **pre-reals** section:

- **complete-ordered-field**
- **h-o-real-arithmetic**

These theories are equivalent in the sense that each one can be interpreted in the other; moreover, the two interpretations compose to the identity. These interpretations are constructed in the section using the IMPS translation machinery.

¶ complete-ordered-field

In this theory, the real numbers are specified as a complete ordered field and the rational numbers and integers are specified as substructures of the real numbers. Exponentiation to an integer power is a defined constant denoting a partial function.

Language:

- *Sorts*:
  - **zz** (**Z** in the mathematical syntax) denotes the set of integers.
  - **qq** (**Q** in the mathematical syntax) denotes the set of rational numbers.
  - **rr** (**R** in the mathematical syntax) denotes the set of real numbers.
- *Constants*:
  - `+` denotes real addition.

– * (in the mathematical syntax, denoted by juxtaposition of its arguments) denotes rel multiplication.
– - denotes sign negation.
– / denotes division.
– < denotes the binary predicate "less than."

There are also an infinite number of constants, one for each rational number. Thus **8**, **[-9]**, **[21/4]** denote rational numbers.

Axioms: The following is the list of all the axioms of the IMPS theory **complete-ordered-field** in the mathematical syntax.

(1) *Trichotomy.* $\forall y, x{:}\mathbf{R}, x < y \vee x = y \vee y < x$

(2) *Irreflexivity.* $0 \not< 0$

(3) *Strict Positivity for Products.* $\forall y, x{:}\mathbf{R}, 0 < x \wedge 0 < y \supset 0 < xy$

(4) $<$ *Invariance .* $\forall z, y, x{:}\mathbf{R}, x < y \equiv x + z < y + z$

(5) *Transitivity.* $\forall z, y, x{:}\mathbf{R}, x < y \wedge y < z \supset x < z$

(6) *Negative Characterization.* $\forall x{:}\mathbf{R}, x + (-x) = 0$

(7) *Characterization of 0.* $\forall x{:}\mathbf{R}, x + 0 = x$

(8) *Associative Law for Multiplication.* $\forall z, y, x{:}\mathbf{R}, (xy)z = x(yz)$

(9) *Left Distributive Law.* $\forall z, y, x{:}\mathbf{R}, x(y + z) = xy + xz$

(10) *Multiplicative Identity.* $\forall x{:}\mathbf{R}, 1\, x = x$

(11) *Commutative Law for Multiplication.* $\forall y, x{:}\mathbf{R}, xy = yx$

(12) *Associative Law for Addition.* $\forall z, y, x{:}\mathbf{R}, (x + y) + z = x + (y + z)$

(13) *Commutative Law for Addition.* $\forall y, x{:}\mathbf{R}, x + y = y + x$

(14) *Characterization of Division.* $\forall a, b{:}\mathbf{R}, b \neq 0 \supset b(a/b) = a$

(15) *Division by Zero Undefined.* $\forall a, b : ind, b = 0 \supset \neg a/b \downarrow$

(16) $\mathbf{Z}$ *Additive Closure.* $\forall x, y : \mathbf{Z}, x + y \downarrow \mathbf{Z}$

(17) **Z** *Negation Closure.* $\forall x : \mathbf{Z}, -x \downarrow \mathbf{Z}$

(18) *Induction.*

$$\forall s : \mathbf{Z} \to *, \forall t : \mathbf{Z},$$
$$0 < t \supset s(t) \equiv (s(1) \wedge \forall t : \mathbf{Z}, 0 < t \supset (s(t) \supset s(t+1)))$$

(19) *Characterization of [-1].* $[-1] = -1$

(20) **Q** *is the field of fractions of* **Z**. $\forall x{:}\mathbf{R}, x \downarrow \mathbf{Q} \equiv \exists a, b : \mathbf{Z}, x = a/b$

(21) *Order Completeness.*

$$\forall p{:}\mathbf{R} \to *,$$
$$\text{nonvacuous?}\,(p) \wedge \exists \alpha{:}\mathbf{R}, (\forall \theta{:}\mathbf{R}, p(\theta) \supset \theta \leq \alpha) \supset$$
$$\exists \gamma{:}\mathbf{R}, (\forall \theta{:}\mathbf{R}, p(\theta) \supset \theta \leq \gamma) \wedge$$
$$\forall \gamma_1{:}\mathbf{R}, (\forall \theta{:}\mathbf{R}, p(\theta) \supset \theta \leq \gamma_1) \supset \gamma \leq \gamma_1$$

¶ h-o-real-arithmetic

This is another axiomatization of the real numbers which we consider to be our working theory of the real numbers. The axioms of **h-o-real-arithmetic** include the axioms of **complete-ordered-field**, formulas characterizing exponentiation as a primitive constant and formulas which are theorems proven in **complete-ordered-field**. These theorems are needed for installing an algebraic processor and for utilizing the definedness machinery of the simplifier. The proofs of these theorems in the theory **complete-ordered-field** require a large number of intermediate results with little independent interest. The use of two equivalent axiomatizations frees our working theory of the reals from the burden of recording these uninteresting results.

The theory **h-o-real-arithmetic** is equipped with routines for simplifying arithmetic expressions and rational linear inequalities. These routines allow the system to perform a great deal of low-level reasoning without user intervention. The theory contains several defined entities; e.g., the natural numbers are a defined sort and the higher-order operators Σ and Π are defined recursively.

h-o-real-arithmetic is a useful building block for more specific theories. If a theory has **h-o-real-arithmetic** as a subtheory, the theory can be developed with the help of a large portion of basic, everyday mathematics. For example, in a theory of graphs which includes **h-o-real-arithmetic**, one could introduce the concept of a weighted graph in which nodes or edges are assigned real numbers. We imagine that **h-o-real-arithmetic** will be a subtheory of most theories formulated in IMPS.

§ sequences

Sequences over a sort α are represented as partial functions from the natural numbers to α. Lists are identified with sequences whose domain is a finite initial segment of the natural numbers. The basic operations on sequences and lists, including **nil**, **car**, **cdr**, **cons**, and **append** are formalized as quasi-constructors.

Glossary

Alpha-equivalence Two **expressions** are alpha-equivalent if one can be obtained from the other by renaming bound variables.

Application An expression whose lead constructor is named apply. Applications must have at least two components. The first component of an application is called the *operator* and the remaining components are called *arguments*. Applications are usually denoted by prefixing the operator to the arguments enclosed in parentheses as $f(x_1, \ldots, x_n)$. However, other forms (such as infix for algebraic operations) are also used.

Beta-reduction An inference rule which plugs in the values of **lambda-applications**. For example, beta-reduction transforms the expression

$$\lambda(x, y{:}bfZ, x^3 + 6 * y)(1, 2)$$

to 13.

Buffer An Emacs data structure for representing portions of text, for example, text files which are being edited. To the user, a buffer has the appearance of a scrollable page. More than one buffer may exist at a given time in an Emacs session. Users will also use buffers to build, edit and evaluate **def-forms**.

Command form An s-expression

$$(\textit{command-name } \textit{arg}_1 \ldots \textit{ arg}_n)$$

used in proof scripts. The number and type of arguments is command-dependent. If the command takes no arguments, then the form *command-name* is also valid.

Compound expression An expression with a **constructor** and a (possibly empty) list of components. $1 + 2$ is a compound expression whose constructor is named apply and whose components are $+, 1, 2$.

Constant A kind of IMPS **expression**. For example, in the theory named h-o-real-arithmetic, the expressions $1, 2, +$ are constants.

Constructor One of the 19 logical constants of **LUTINS**. Constructors are used to build compound expressions.

Context An IMPS data structure representing a set of of formulas to be used as assumptions. Contexts contain various kinds of cached information (for example, about definedness of terms), used by the IMPS simplification machinery.

Core All the basic logical and deductive machinery of IMPS on which the soundness of the system depends.

Deduction graph An IMPS data structure representing a proof. Deduction graphs contain two kinds of nodes: **inference nodes** and **sequent nodes**.

Def-form An **s-expression** whose evaluation has some effect on the IMPS environment, such as building a theory, adding a theorem to a theory, making a definition in a theory or building a translation between theories. There are about 30 such def-forms.

Emacs A text editor. Although various implementations of Emacs exist, in this manual we mean exclusively GNU Emacs, which is the extensible display editor developed by the Free Software Foundation. Because of the many extension facilities it provides (including a programming language with many string-processing functions), GNU Emacs is well-suited for developing interfaces.

Expression An IMPS data structure representing an element of a language. Expressions are used to describe mathematical entities or to make logical

assertions. Two examples (printed in TEX) are $2^{2^n} - 1$ and

$$\forall k, m : \mathbf{Z}, (1 \leq k \wedge k \leq m) \supset \binom{1+m}{k} = \binom{m}{k-1} + \binom{m}{k}.$$

An expression is either a **formal symbol** or a **compound expression.**

Evaluate To cause an s-expression to be read by the **T** process, thereby creating an internal representation of a program, which is then executed by **T**. This sequence of events usually has a number of side-effects on the IMPS environment. For example, to define a constant named square in the theory h-o-real-arithmetic, evaluate the s-expression:

```
(def-constant SQUARE
  "lambda(x:rr, x^2)"
  (theory h-o-real-arithmetic))
```

Formal symbol A primitive IMPS expression. A formal symbol has no components, although it is possible for an expression with no components to be a compound expression. A formal symbol can be a **variable** or a **constant.** Every formal symbol has a *name* which is used for display. The name is a Lisp object such as a symbol or a number.

Formula Intuitively, an expression having a truth value, for example,

$$\forall x, y, z : \mathbf{R}, x < y \wedge y < z \supset x < z.$$

In IMPS a formula is an **expression** whose sort is $*$.

IMPS A formal reasoning system and theorem prover developed at The MITRE Corporation.

Inference node A node in a deduction graph, connecting a conclusion sequent node with zero or more hypothesis sequent nodes, that represents an instance of a rule of inference.

Iota-expression A compound expression whose lead constructor is named iota, denoted in the mathematical syntax by ι. An iota-expression has the form $\iota x{:}\sigma, \varphi$. It denotes the unique object in the sort σ which satisfies the condition φ, if such an object exists. Otherwise, the iota-expression is undefined.

Kind Each sort and expression is of kind ι (also written as iota) or $*$ (prop). Expressions of kind ι are used to refer to mathematical objects; they may be undefined. Expressions of kind $*$ are primarily used in making assertions about mathematical objects; they are always defined.

Lambda-application An **application** whose operator is a **lambda-expression**. For example $\lambda(x, y{:}\mathbf{R}, x^2 + 3 * y)(1, z)$.

Lambda-expression A compound expression whose lead constructor is named lambda, denoted in the mathematical syntax by λ. A lambda-expression has the form $\lambda x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n, \varphi$. It denotes the function of n arguments whose value at $x_1, \ldots, x_n$ is given by the value of term φ, provided the x_i are of sort σ_i.

Language Mathematically, a language in IMPS consists of **expressions**, **sorts**, and user-supplied information about sort inclusions. At the implementation level, a language is a Lisp data structure having additional structure which procedurally encodes this information. This is meant to facilitate various kinds of lower level reasoning, such as reasoning about membership of an expression in a sort.

Little theories A version of the axiomatic method in which mathematical reasoning is distributed over a network of theories linked to one another by theory interpretations, in contrast to the "big theory" approach in which all reasoning is performed within a single, usually very expressive theory.

Local context The set of assumptions relevant to a particular location in an expression.

LUTINS A version of type theory with partial functions and subtypes; the logic of IMPS.

Macete A user-defined extension of the simplifier. When a theorem is installed, it is also automatically installed as a macete. The way the corresponding macete works depends on the syntactic form of the theorem. In the simplest case in which the theorem is a universally quantified equality, the theorem is used as a rewrite rule. Macetes can be composed using the **def-form** named def-compound-macete. The word "macete" comes from

Brazilian Portuguese, where it means "clever trick." Theorem macetes can also be made into **transportable macetes**.

Mode Line The line at the bottom of each buffer window. It provides information about the displayed buffer such as its name and mode.

Parser A program which takes user input, usually represented as text, and builds a data structure suitable for computer processing.

Primitive inference procedure An IMPS procedure that implements one of the primitive rules of inference of the IMPS proof system. IMPS has about 30 primitive inference procedures.

Proof Conclusive mathematical evidence of the validity of some assertion. The rules for admissible evidence are given by a *proof system.*

Quasi-constructor A user-defined macro/abbreviation for building expressions. Quasi-constructors are used much like **constructors**.

Quasi-equality A quasi-constructor, written as infix $\simeq$ in the mathematical syntax. $t \simeq s$ means that either t and s are both undefined or they are equal.

S-expression The external representation of the basic data structure of Lisp-like languages including **T**. S-expressions are usually represented as nested lists such as `(fuba (abacaxi farofa) alface)`.

Sequent node A node in a deduction graph that represents a sequent, a formula composed of a set of *assumptions* and an *assertion.*

Sort A sort denotes a nonempty set of mathematical entities. Every **expression** has a sort which restricts the value of that expression.

Syntax Rules for parsing and printing **expressions**. The syntaxes that are currently used in IMPS are the *string syntax* which allows for infix and postfix notation, the *s-expression syntax* which provides a uniform prefix syntax for all **constructors** and operators, and the *tex syntax* for displaying (but not parsing) in TEX.

T A dialect of Lisp developed at the Yale University Computer Science Department in which the core machinery of IMPS is implemented. T is very similar to the Scheme programming language, familiar to readers of Abelson and Sussman's book [1].

Tea An alias for the **T** programming language.

Theory The term "theory" as used in this manual, consists of a **language** and a set of closed formulas in the language called *axioms*. A theory is implemented as a record structure having slots for the language, the list of axioms, and other relevant information. There is also a slot for a table of procedures which the simplifier uses to apply the axioms. This is meant to facilitate various kinds of lower-level reasoning, such as algebraic or order simplification. In a different sense, theory is also commonly used to refer to a collection of related facts about a given mathematical structure, such as the theory of rings or the theory of ordinary differential equations.

Theory ensemble A data structure which bundles together theories which are unions of copies of a single base theory. Theory ensembles are mainly used to treat multiple instances of structures, such as triples of metric spaces or pairs of vector spaces.

Theory interpretation A **translation** that maps theorems to theorems.

Translation An IMPS data structure, created with the **def-form** named def-translation, that is used to translate expressions from one theory to another.

Transportable macete A theorem macete that can be used in a theory different from the one in which it was installed. Transportable macetes are named by the system by prefixing `tr%` to the name of the theorem.

Type A **sort** that is maximal with respect to the subsort relation.

Variable A kind of IMPS **expression**. An occurrence of a variable v within an expression is *bound* if it occurs with the body of a binding constructor which includes v as a binding variable. Otherwise, the occurrence of v is *free.*

Xview An IMPS procedure for printing **expressions** in TEX and then displaying them in an X window TEX previewer.

Bibliography

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[3] W. M. Farmer. Abstract data types in many-sorted second-order logic. Technical Report M87-64, The MITRE Corporation, 1987.

[4] W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55:1269–91, 1990.

[5] W. M. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64:211–240, 1993.

[6] W. M. Farmer. A technique for safely extending axiomatic theories. Technical report, The MITRE Corporation, 1993.

[7] W. M. Farmer. Theory interpretation in simple type theory. In *Proceedings, International Workshop on Higher Order Algebra, Logic and Term Rewriting (HOA '93), Amsterdam, The Netherlands, September 1993*, Lecture Notes in Computer Science. Springer-Verlag, 1994. Forthcoming.

[8] W. M. Farmer, J. D. Guttman, M. E. Nadel, and F. J. Thayer. Proof script pragmatics in IMPS. In *12 International Conference on Automated Deduction (CADE-12)*, Lecture Notes in Computer Science. Springer-Verlag, 1994. Forthcoming.

[9] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: System description. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume

607 of *Lecture Notes in Computer Science*, pages 701–705. Springer-Verlag, 1992.

[10] W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.

[11] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: an Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[12] W. M. Farmer, J. D. Guttman, and F. J. Thayer. Reasoning with contexts. In A. Miola, editor, *Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 216–228. Springer-Verlag, 1993.

[13] W. M. Farmer and F. J. Thayer. Two computer-supported proofs in metric space topology. *Notices of the American Mathematical Society*, 38:1133–1138, 1991.

[14] J. A. Goguen. Principles of parameterized programming. Technical report, SRI International, 1987.

[15] J. A. Goguen and R. M. Burstall. Introducing institutions. In *Logic of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer-Verlag, 1984.

[16] J. D. Guttman. A proposed interface logic for verification environments. Technical Report M91-19, The MITRE Corporation, 1991.

[17] J. D. Guttman. Building verification environments from components: A position paper. In *Proceedings, Workshop on Effective Use of Automated Reasoning Technology in System Development*, pages 4–17, Naval Research Laboratory, Washington, D.C., April 1992.

[18] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for scheme. In *Proceedings of the* SIGPLAN *'86 Symposium on Compiler Construction*, volume 21, pages 219–233, 1986. Proceedings of the '86 Symposium on Compiler Construction.

[19] J. D. Monk. *Mathematical Logic*. Springer-Verlag, 1976.

[20] L. G. Monk. Inference rules using local contexts. *Journal of Automated Reasoning*, 4:445–462, 1988.

[21] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, 1974.

[22] Y. N. Moschovakis. Abstract recursion as a foundation for the theory of algorithms. In *Computation and Proof Theory, Lecture Notes in Mathematics 1104*, pages 289–364. Springer-Verlag, 1984.

[23] J. A. Rees, N. I. Adams, and J. R. Meehan. *The T Manual*. Computer Science Department, Yale University, fifth edition, 1988.

[24] J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[25] R. M. Stallman. GNU *Emacs Manual (Version 18)*. Free Software Foundation, sixth edition edition, 1987.

[26] F. J. Thayer. Obligated term replacements. Technical Report MTR-10301, The MITRE Corporation, 1987.