An Introduction to ULTRA

Interactive Transformation of Functional Programs

— Version 2.2 —
# Contents

8 How To ................................................. 85
  8.1 Load Multiple Theory Files ......................... 85
  8.2 Start a Derivation ................................ 85
  8.3 Keep Lambda Abstractions Explicit at the Outer-Most Level ................................................. 85

9 Future .................................................. 87
  9.1 Automation ........................................... 87
  9.2 Reduction Mode ..................................... 87
  9.3 Limitations .......................................... 87

Appendices ............................................... 89
  A I got an error message and don’t know why ............... 89
  B Predefined Constructs ................................. 93
    B.1 Built-in Functions ................................ 93
    B.2 Descriptive Specification ......................... 94
  C The Ultra System ........................................ 97
    C.1 Structure of the System .......................... 97
    C.2 Files in the Distribution ......................... 97
    C.3 Customization ..................................... 99
    C.4 Mouse Use .......................................... 99
  D Theory Files ............................................ 101
    D.1 Syntax ............................................. 101
    D.2 Standard Theories .................................. 102
    D.3 Theory for the Sum-Squares Derivation .......... 103
    D.4 Theory for the Maximum-Segment-Sum Derivation .. 103
  E Glossary ............................................... 107

References ............................................... 109

Index ..................................................... 111
1 Notices

The following terms in this publication are trademarks as follows:

<table>
<thead>
<tr>
<th>Term</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td>Apple Computer Incorporated</td>
</tr>
<tr>
<td>Solaris</td>
<td>Sun Microsystems Incorporated</td>
</tr>
<tr>
<td>Virtual PC</td>
<td>Connectix Corporation</td>
</tr>
<tr>
<td>Windows</td>
<td>Microsoft Corporation</td>
</tr>
</tbody>
</table>

The Ultra team:

- Prof. Dr. Helmuth Partsch
- Dr. rer. nat. habil. Wolfram Schulte
- Dr. Ton Vullinghs
- Dr. Mark Dettinger
- Dipl. Inf. Joachim Schmid
- Dipl. Inf. Thorsten Quell
- Dipl. Inf. Tobias Häberlein
- Walter Guttmann

We appreciate your comments — find contact addresses on the Ultra home page:

http://www.informatik.uni-ulm.de/pm/ultra/

Contact the author of this document at

walter.guttmann@informatik.uni-ulm.de

Ultra is a medium developed at the University of Ulm.
2 About this Manual

This manual introduces the transformation system Ultra 2.2 and helps you get started using it. After working through this manual you should be able to use all features of the system and perform transformation tasks on your own.

2.1 Who Should Read this Manual

Anyone who will be using Ultra 2.2 should read this manual. It explains the basic and the advanced features of the transformation system and gives further links to related topics of general or special interest.

Teachers may want to use Ultra as a tool for a course (e.g. transformational programming or functional programming). Students learning a modern functional programming language may want to experiment with functions and equational reasoning. The working computer scientist may want to use Ultra in the development of programs. Researchers may profit from the benefits of interactive design of transformation rules.

What are the prerequisites for reading this manual? You should have some knowledge of predicate logic, transformational programming (see e.g. [9]), and a modern functional programming language (see e.g. [2]). Part of this knowledge is provided in a chapter devoted to the theoretical background.

We assume that you are familiar with the operating system you intend to run Ultra on, its file-system, and the use of some text editor.

2.2 Structure of this Manual

This manual has 9 chapters and 5 appendices.

- The introduction gives an overview of the system and describes its installation.
- Some theoretical background on the transformation of functional programs is provided then.
- The First Steps chapter guides the reader through the system by realizing a small transformation task.
- The Next Steps chapter demonstrates further capabilities by realizing another (slightly more complex) transformation task. For another sample session we refer to [8].
- The functionality of Ultra is described by the System Reference.
- A chapter giving advice for a few frequently occurring tasks follows then.
- We conclude by pointing out further directions of evolution for Ultra.
- The appendices cover technical details of Ultra.

A bibliography and an index completes the manual.
2.3 Online- and Related Information

The latest version of this manual can be downloaded from the Internet:

http://www.informatik.uni-ulm.de/pm/ultra/

Further online documentation is not available yet. Related information can be found in the literature cited throughout this manual.
3 Introduction

This chapter introduces to the Ultra system. Read it prior to installation.

3.1 What is Ultra

The Ultra program transformation system is the result of several projects considering functional and transformational programming that have been carried out at the University of Ulm under the supervision of Prof. Partsch (see [4], [12], [11], and [5]). The Ultra system is intended to support the transformational development of functional programs. To this end, the system supports interactive manipulation of programs by application of correctness preserving transformation rules.

Primary goal is to support all the basic steps of the Burstall/Darlington unfold-fold system [3]. The basic transformation steps are unfolding, folding, instantiation, abstraction, rule application and the introduction of new function definitions. These steps allow new equations to be introduced as consequences of existing ones.

In addition, Ultra provides also advanced strategies in the form of specialized transformation rules (e.g. the manipulation of language-specific constructs) or combinations of transformation rules (e.g. to perform automatic simplifications). Furthermore, Ultra supports the development of operational algorithms from descriptive (non-deterministic) specifications.

Keep in mind that Ultra is a research tool rather than a full-featured product and we cannot guarantee that everything always works to the extent you might expect.

3.2 Components

Ultra source code is written in the functional programming language Gofer [7]. The different files are dealt with in detail in the appendix (C.2). At run-time a few bitmap resource files are loaded.

For program transformation predefined theory files are accessible. For a reference, see the appendix (D.2). Users will incrementally create additional theory files.

3.3 System Requirements

Ultra can be used on every system capable of running the large version of TkGofer 2.0 [13]. The large TkGofer uses a version of Gofer that can handle more type constructors and classes than a usual Gofer system. TkGofer in turn needs Tk and Tcl - refer to the TkGofer documentation. A distribution can be obtained from the Internet:

http://www.informatik.uni-ulm.de/pm/ftp/tkgofer.html

TkGofer runs at least on the following platforms: Linux, Solaris, Windows. We have even succeeded to run it on an Apple notebook with the software PC emulator “Virtual PC”. Since Ultra has no built-in editor, you will have to use your favorite one to edit theory or source files.
3.4 Installation

We assume a completely installed release of TkGofer 2.0. Get the latest version of *Ultra* from:

http://www.informatik.uni-ulm.de/pm/ultra/

Unpack the files from the archive to a directory of your choice. We recommend adding the TkGofer binaries directory to your `PATH` environment variable or creating a link or short cut to the TkGofer executable from within your *Ultra* directory.

3.5 System Startup

Change to the *Ultra* directory and run the TkGofer interpreter. Load the project file `ultra.p` by typing the command

```
:p ultra.p
```

at the prompt of the interpreter. After the source files are loaded, start *Ultra* by calling for evaluation of the function

```
main
```

The *Ultra* graphical user interface pops up in a window.

![Ultra GUI](image)

Figure 1: The *Ultra* GUI

The recommended way to exit *Ultra* is to select *Quit* from the *File* menu. On most systems `CTRL-C` will interrupt the interpreter. Then, you can restart the system by calling `main` or quit the interpreter by typing the command

```
:q
```

If you find the GUI too small or too large, you can adjust its size by modifying the variable `f_fontPixelSize` in the file `setup.gs`, see also the appendix (C.3).
4 Theoretical Background

In this chapter we give brief description of the three bases Ultra is built upon: functional programming, the calculus for transformations, and the transformational way to software development.

4.1 Functional Programming

In this manual, when we use the term functional programming language, we assume a language with features as follows:

- referential transparency
- lazy evaluation
- strongly typed
- higher-order polymorphic functions

For example, Haskell fits this description whereas Lisp does not. Programming in a functional language [2] — or rather in a functional style — has a number of benefits.

- Referential transparency implies that the meaning of an expression is denoted by its value and that there are no side effects if we try to compute this value. The value of an expression depends only on the value of its constituents. As a consequence, subexpressions may be replaced by other expressions having the same value, thus providing a simple but sound basis for equational reasoning.

- Lazy evaluation (sometimes called call-by-need) combines the benefits of a call-by-value and a call-by-name semantics for the costs of a small overhead. It enables dealing with infinite-sized data structures. Terminating programs examine only a finite part of them, of course.

- A strongly typed language requires that you get the types right before running a program. Most interpreters and compilers for functional languages provide a type inference mechanism. You are forced to think about the type of your functions. This leads to cleaner programs and prohibits many kinds of errors.

- Higher-order functions are treated as first-class citizens. This is an extension to the type systems of most imperative languages and eliminates unnecessary restrictions — why should a function not return another function?

Imperative languages have borrowed some of these concepts (e.g. procedure variables in newer Pascal implementations or function objects in the C++ Standard Template Library) but they are used in at least fussy or even inconsistent ways.

A functional program is a set of mutually recursive function definitions. Functions support multiple arguments and their definition typically uses pattern matching (cf. the definition of the function \texttt{length} below).

From a set of basic data types the programmer can construct algebraic data structures. Pattern matching occurs over the constructors. Modern compilers use this to check the totality of function definitions. Recursive data types are defined easily
without the use of a pointer type. The single most prominent data structure is the list container.

Parametric polymorphism is used to define infinite sets of data types and functions operating on them. Compare this feature (which in a similar way is realized by the template mechanism in C++) to defining over and over functions for types with similar structure. For example, the length of a list does not depend on the type of the stored elements:

\[
\text{length} :: [a] \rightarrow \text{Int} \\
\text{length} \ [\] = 0 \\
\text{length} \ (x:xs) = 1 + \text{length} \ xs
\]

Ultra is completely written in the functional language Gofer. It uses TkGofer [14] for the functional implementation of the graphical user interface. Modern concepts of functional languages like constructor classes and monads [6] were used in the implementation.

Programs that can be manipulated in Ultra are also functional. For their specification an extended subset of Gofer can be applied, see (9.3).

4.2 Transformation Calculus

We now give an introduction to our transformation calculus and describe briefly how it is used by Ultra. This calculus has its roots in the transformation semantics of the CIP system [1, 10] and is based on a two-level Horn clause logic. The top-level of implicational formulas is provided by inferences of the following form

\([C_1, C_2, \ldots, C_n] \models D\]

where \(C_i\) (\(1 \leq i \leq n\)) (called premises) and \(D\) (called conclusion) are clauses. Clauses build the lowest level of positive implicational formulas and have the following form:

\([A_1, A_2, \ldots, A_m] \models B\]

with \(A_i\) (\(1 \leq i \leq m\)) (called antecedents) and \(B\) (called consequent) being atomic formulas. An atomic formula is a semantic predicate over program schemes. Terms are expressions that can be formulated using a programming language (in our case the extended subset of Gofer). Program schemes are a generalization of terms. They may additionally contain free variables. As an example take the declaration of the well-known function map:

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{map} \ f \ [] = [] \\
\text{map} \ f \ (x:xs) = f \ x : \text{map} \ f \ xs
\]

This function applies a function \(f\) of type \(a \rightarrow b\) to every element of a list of type \([a]\) resulting in a list of type \([b]\). Then \(\text{map} \ g \ [1,2,3]\) is a program scheme with free variable \(g\). A program scheme can be instantiated by mapping its free variables to program schemes. For example, \(\text{map} \ \text{incr} \ [1,2,3]\) is an instance of the above program scheme where \(\text{incr}\) might be the function which adds 1 to its argument.

A transformation rule is a special kind of an inference, where the conclusion is a clause with empty antecedents and its consequent either denotes an equivalence
4.2 Transformation Calculus

or descendance relation between two program schemes. The premises denote the *applicability conditions* for the rule. They usually restrict the possible values of the variables in the schemes by some semantic predicate. The generic form of a transformation rule for an equivalence of two schemes \( i \) and \( o \) is:

$$\text{rule_name} = [C_1, C_2, \ldots, C_n] \models ([] |- i \equiv o)$$

You can see the applicability conditions \( C_i \), the input scheme \( i \) and the output scheme \( o \). For a descendance, the operator \( \equiv \) is replaced by \( \Rightarrow \). As an example, let’s define the Map Identity law:

$$\text{map_identity} = [] \models ([] |- \text{map id} \Rightarrow \text{id})$$

Since the antecedent clause of the conclusion of a transformation rule is always empty, we normally use \( f \leftrightarrow g \) as an abbreviation for \( [] |- f \equiv g \), and similarly \( f \Rightarrow g \) as an abbreviation for \( [] |- f \Rightarrow g \), so that we can denote the above rule as:

$$\text{map_identity} = [] \models (\text{map id} \leftrightarrow \text{id})$$

The operators \( \models \) and \( \leftrightarrow \) are handled as ordinary functions. Because equality on functions is not computable in general, there is no way to give a useful definition for \( \equiv \). However, the operators are useful if we like to check the rules for type-correctness. For this reason, the operators have the following types:

\[
\begin{align*}
\equiv & : \text{Term} \to \text{Term} \to \text{Formula} \\
\Rightarrow & : \text{Term} \to \text{Term} \to \text{Formula} \\
\leftrightarrow & : \text{Term} \to \text{Term} \to \text{Clause} \\
\Rightarrow & : \text{Term} \to \text{Term} \to \text{Clause} \\
\models & : [\text{Formula}] \to \text{Formula} \to \text{Clause} \\
\models & : [\text{Clause}] \to \text{Clause} \to \text{Inference}
\end{align*}
\]

The definitions of the operators and data types are not important, they are merely used for type checking the rules. For example the rule Map Distribution has the following definition:

$$\text{map_distribution} : : (b \to c) \to (a \to b) \to \text{Inference}$$

$$\text{map_distribution} f g = [] \models \text{map } f . \text{map } g \leftrightarrow \text{map } (f \cdot g)$$

In the sequel we do consider the signatures of transformation rules. Note the role of the parameters \( f \) and \( g \) in the above declaration. They have to be listed because otherwise the transformation rule would not be a valid Gofer declaration. On the other hand, they exactly correspond to the free variables in the input and output scheme of the transformation rule (assuming that there exists a declaration of \( \text{map} \) somewhere else in the visible scope). In general, the scheme variables of a transformation rule are exactly those variables that are listed as parameters in the rule’s declaration. They are instantiated by first-order pattern matching when the rule is applied.

Basic transformation rules are either given by the user — who then takes responsibility for their correctness — or are derived from the programming language semantics (e.g. \( \beta \)-reduction). These rules are used in a transformation session to derive new transformation rules which can be stored for future use.
The above-mentioned transformation rules map-identity and map-distribution are both rules without applicability conditions. The transformation rule

\[
\text{foldr
dl} :: (a \to a \to a) \to a \to \text{Inference}
\]

\[
\text{foldr
dl} f e = [[[] |\neg \text{monoid} f e] \implies \text{foldr} f e \iff \text{folddl} f e
\]

has an applicability condition. It states that it is safe to replace \text{foldr} by \text{folddl} (and vice versa) if \(f\) and \(e\) form a \textit{monoid}, i.e., \(f\) is associative with \(e\) as a neutral element. Just as \(\iff\), \textit{monoid} is a semantic predicate on terms. By using a prolog-like syntax, we can formulate additional algebraic properties of terms. For example, the \textit{monoid} property is defined in the following way:

\[
\text{monoid} :: (a \to a \to a) \to a \to \text{Clause}
\]

\[
\text{monoid} \ op \ n :- \text{associative} \ op, \text{neutral} \ op \ n
\]

Likewise, we can define the predicate of being a neutral element:

\[
\text{neutral} :: (a \to a \to a) \to a \to \text{Clause}
\]

\[
\text{neutral} \ op \ n :- \text{lneutral} \ op \ n, \text{rneutral} \ op \ n
\]

We can define instances of these properties by declaring \textit{facts}. For example, list concatenation (++) is associative and has as its neutral element the empty list \([\,]\). This is described by following facts:

\[
\text{associative} \ (++)
\]

\[
\text{lneutral} \ (++) \ [\,]
\]

\[
\text{rneutral} \ (++) \ [\,]
\]

User-defined facts are stored in the knowledge-base of \textit{Ultra} and can be retrieved automatically for the instantiation of semantic predicates. Facts, rules, and programs are logically organized into \textit{theories}. They can be made accessible to the system through theory files. See the appendix (D.2) for examples of theory files.

The application of a transformation rule can be regarded as a simple \textit{tactic} of transforming programs. Tactic combinators are means of composing elementary tactics to create powerful transformation \textit{strategies}. These can be used to partly automate transformation tasks.
4.3 Software Development by Transformational Programming

Transformational Programming [9] can be used to derive correct and efficient programs from formal specifications. First, show the correctness of an initial non-operational and maybe inefficient program. Then, repeatedly apply correctness-preserving transformation rules until an efficient operational version is reached. The following diagram shows the general strategy and gives a simple example:

![Diagram of Transformational Programming]

Using a concrete functional language for the programs to be transformed has the benefit of getting executables as the result of transformations (which enables prototyping). Since the entire development process works only with formal objects, one of the basic requirements for machine support is fulfilled.

Transformational programming is in essence the manipulation of terms. This can be performed interactively using Ultra which partially automates transformation tasks and provides bookkeeping and development navigation assistance.
5 First Steps

We will now guide you through Ultra by working through a small-sized derivation step by step. You should get familiar with the user interface and notice some of the essential tasks you can perform with Ultra.

First, we give a rough overview of the graphical user interface (GUI) of the system. The intention is to provide several definitions necessary to carry out the sample derivation. As introductory example we chose to derive a tail-recursive algorithm for the Sum-Squares problem. After a short review of the problem, we start our derivation.

5.1 Overview of the Graphical User Interface

This is not a complete technical description of the user interface (which will be given later) but introduces the main components.

The Editor is the place where most of the actions happen. Currently, Ultra can be operated in two modes. On start-up the system is in command-line mode. Much like the Gofer system, you can issue commands to Ultra in the editor. Entering a valid expression at the prompt starts a derivation process which puts the system into derivation mode. The displayed term is manipulated by selecting parts of it and applying transformations.

The Controls are enabled only during a derivation. The buttons support the basic tasks of the unfold-fold paradigm. Further buttons perform primitive navigational tasks such as undoing a transformation step. Other system tasks are reachable through the menus. They include file management, accessing the information about the current derivation, and elaborate tactics.

The Database allows you to access objects of a transformation session. Currently there are three components:

- The Theory List shows the loaded theory files and allows to select a theory for further operation.
- The Rule List shows the rules from the currently selected theory. The rules can be displayed and applied to selected terms during a derivation.
• The Term List shows the terms from the currently selected theory. The terms can be displayed and used as parameters in transformation steps.

If you already know the structure of a theory file you might wonder why there is no Clause List. The reason is that clauses from all loaded theories are gathered in a single database which can be displayed by selecting Show Clauses from the Clauses menu.

5.2 The Sum-Squares Problem

Now that you know where to find terms, rules and controls for derivations, you are almost prepared for doing your first derivation using Ultra: from an already operational program for the Sum-Squares problem, we will derive a tail-recursive version.

Here is a description of the problem: given a sequence of integer numbers, calculate the sum of the squares of the numbers in the sequence. Take a look at the following sequence:

\[ [5, -3, -7, 8, 1] \]

The sum of the squares of the numbers is \( 5^2 + (-3)^2 + (-7)^2 + 8^2 + 1^2 = 25 + 9 + 49 + 64 + 1 = 148. \) This reasoning can be generalized to a straight-forward solution for this problem. First take the squares of all numbers, then calculate the sum of the resulting numbers. In Gofer this reads

\[
\begin{align*}
\text{sumsquares } xs & = \text{sum } (\text{squares } xs) \\
\text{sum } [] & = 0 \\
\text{sum } (x:xs) & = x + \text{sum } xs \\
\text{squares } [] & = [] \\
\text{squares } (x:xs) & = \text{square } x : \text{squares } xs \\
\text{square } x & = x * x
\end{align*}
\]

This definition uses recursive implementations for the \textit{sum} and the \textit{squares} function. It is our goal to derive a tail-recursive version of the \textit{sumsquares} function that is independent of auxiliary functions.

5.3 Sample Derivation

We will guide you through the derivation step by step (paragraphs marked with STEP). For each step, corresponding screen-shots will illustrate what’s going on, additionally to the explanations in the text (paragraphs marked with EXPL), which can occur as the introduction to the step or as its conclusion. Wherever appropriate, we will hint on alternative actions (paragraphs marked with OPTION).

STEP 1 Start Ultra as explained in (3.5). The GUI appears.

EXPL There is already one theory loaded: \texttt{precompiled.thy}. Much like the Gofer prelude it contains definitions of frequently used functions. For our specific derivation we will need additional functions which are defined in the theory file \texttt{sumsquares.thy} that is located in the sub-directory \texttt{Theories}. 
STEP 2 Add this theory to the system by typing

:a Theories/sumsquares.thy

into the editor. A listing of sumsquares.thy is given in the appendix (D.3).

OPTION Alternatively to typing you can select Add Theory from the File menu which opens a dialog where you can select the theory file you want to add to the system. First, select the directory Theories, then the file sumsquares.thy.
EXPL When *Ultra* has finished loading the theory, it will tell you so in the status line at the bottom of the user interface. *Ultra* will also automatically select the newly loaded theory from the theory list and thus show you the defined terms.

The `sumsquares` function is the implementation of the straight-forward algorithm for the Sum-Squares problem as described in the previous section. It appears in the term list.

**STEP 3** Start the derivation by typing `sumsquares` at the command prompt.

![Ultra Interface](image)

**Figure 5:** Starting the Derivation on `sumsquares`
EXPL  Ultra recognizes that this is not a command but a valid expression and switches from command-line mode to derivation mode. This has two effects:

First, the control buttons are enabled now along with several tasks that can be selected from the menu. At this point, just notice that the Back button from the control bar will undo the last transformation step (this is good when you have made a mistake) and the Forward button will redo it (when it wasn’t a mistake at last). You can undo or redo transformation steps several times. Furthermore, when you suddenly want to abort a transformation session, you have to select Cancel Derivation from the Derivation menu before you can quit Ultra.

Second, you no longer can type commands into the editor. In derivation mode the contents of the editor are changed only by performing a transformation task. The only thing you can do directly in the editor is to select a part of the displayed term. The selected sub-term will then be the input to the next transformation task.
STEP 4   Go ahead and select the term \texttt{sumsquares} by pointing to it with your mouse and pressing the left button. We will now perform an unfold step. This will replace the just selected term by its definition. Press the \textbf{Unfold} button from the control bar.

![Diagram showingUltra 2.2 interface with terms and rules selected, and a status line indicating "derivation mode: sumsquares".](image)

Figure 6: Unfold \texttt{sumsquares}

**EXPL**  Don’t worry about changes in the status line: after each transformation step, \textit{Ultra} tells you about the simplifications it performed automatically, see 7.4.1. The editor now shows the straight-forward algorithm: \texttt{squares} generates the squares of the numbers of a list, and \texttt{sum} calculates the sum over all numbers from a list.
STEP 5  Select the sub-term \((\text{squares } xs)\) by pointing to the white-space right between \text{squares} and \text{xs} with your mouse and pressing the left button. Unfold this sub-term, too.

Figure 7: Unfold \text{squares } xs

**EXPL**  \textit{Ultra} has replaced the call to the function \text{squares} by its definition, and immediately substituted \text{xs} for the first (and in this case only) formal parameter.
STEP 6  We will now distribute the call to the function \texttt{sum} over the two branches of the \texttt{case} expression. To this end, select the sub-term containing \texttt{sum} and the \texttt{case} expression by pointing to the white-space immediately before the left parenthesis preceding the \texttt{case}. In case you have difficulties, try to point right below the letter \texttt{s} of the term \texttt{sum}. Now open the Tactics menu and select the task \textbf{Lift Let and Case}. The result is:

![Diagram showing the distribution of sum over the case expression]

Figure 8: Distributing \texttt{sum} over the \texttt{case}
EXPL This transformation has lifted the case expression up the syntax-tree to become the new root of the selected sub-term. In this particular case, the built-in simplifier would have generated the same outcome.

OPTION Try it: press the button Back from the control bar, which will undo the last transformation. Select the same sub-term again, and this time use Simplify from the control bar. Although you get the same result as with the Lift Let and Case task in this case, choosing Simplify generally tries to perform several kinds of complexity-reducing transformations.
STEP 7    Either way, we can proceed with transforming the two branches of the `case` expression. For the first branch, select `sum []` by pointing to the white-space between `sum` and `[]`. Unfolding this term leads to the expected result of 0.

![Figure 9: Unfold sum in the first branch of the case](image)

EXPL    You might have already noticed that selecting a sub-term is quite different from selecting part of the contents of a usual editor. That’s because of the differing underlying structure. For many editors, the contents are just a sequence of characters, so it is natural to select a range from the start to the end. Ultra, however, knows about the syntax-tree structure of the displayed term. Since sub-
terms are also trees, it is natural to select a sub-tree. This is done by pointing to the root which unambiguously identifies the sub-tree. But the nodes of the syntax-tree are not explicitly visible in the editor – this might make the novice user feel uncomfortable during the selection of a term.

**STEP 8** Continuing our transformation, we will now focus on the second branch of the case expression. Select the right side of the second branch by pointing to the white-space just to the right of the sum sub-term. Press the button **Zoom In** from the control bar.

![Figure 10: Zooming in on the second branch of the case expression](image-url)
What has happened? The previously selected sub-term now occupies the whole editor (well, there is actually a lot of free space left). Don’t worry, the rest of the expression has not been removed, since this was no real transformation. It just isn’t displayed any longer. You can use this feature if you want to concentrate on a sub-term where you will apply several transformations that don’t influence the rest of the expression. Later on, you can return to larger parts of the expression.

**STEP 9** First, unfold the definition of `sum` by pointing to the white-space just to the right of the `sum` sub-term (which selects the whole displayed expression) and using **Unfold** from the control bar.

---

**Figure 11: Unfold sum**
STEP 10  Now, eliminate the call to \texttt{square} by unfolding \texttt{square} \( x \) (point to the white-space between \texttt{square} and \( x \)).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure12.png}
\caption{Unfold \texttt{square} \( x \)}
\end{figure}
STEP 11  The expression to the right of the addition operator + matches the definition of the \textit{sumsquares} function exactly. This calls for a \textbf{Fold} operation. Select the sub-term by pointing to the white-space just to the right of the \textit{sum} sub-term, and apply \textbf{Fold} from the control bar.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure13.png}
\caption{Fold \textit{sum (squares b)} to \textit{sumsquares b}}
\end{figure}
STEP 12  After pressing the **Fold** button, a dialog will be opened to let you select the right instance to fold. This always happens, when **Ultra** finds more than one possible way of folding the selected sub-term. Here, select **sumsquares** to fold:

![Select function to fold dialog](image)

**Figure 14:** Selecting the right instance for the **Fold** task

**EXPL**  We have almost finished the transformation. Remember that we have focused on the second branch of the **case** expression several steps before. Now it’s time to **Zoom Out** again. In contrast to the **Zoom In** task, we cannot select the sub-term that should occupy the editor for the next step (after all, it is not displayed). So zooming out will always take you one level up the syntax-tree.
STEP 13 To return to the upper-most level, it may happen, that you have to press the **Zoom Out** button from the control bar several times: in this case you must press it twice.

![Image of ULTRA 2-2 interface with text: x ^ 2 + x * a + a * b]

Figure 15: Zooming out twice

**OPTION** We have now completed the derivation. From the **Derivation** menu, **Show Derived Rule** opens a window which displays the rule generated during the transformation session. From the same menu, **Show Protocol** opens a window that lists all the transformation steps you performed.
STEP 14 To let Ultra know that we finished our session, choose **Stop Derivation** from the same menu and confirm the warning message. The system then asks you for a name – this name is used to store the generated rule in a theory for further reference. Enter `sumsquares_rule`.

---

**EXPL** Ultra switches back to command-line mode, and displays the last term of the derivation. The rule is stored in a theory called `system.thy`. This theory is not stored persistently in a file, so if you quit Ultra now, the rule will be lost. You can store its contents into a file by selecting **Save System Theory** from the **File**
menu. Typically you will store the system theory at the end of a run of Ultra to a temporary file, and then move the rules to a place of your choice using an external editor.

**OPTION** If you want to know what transformation you have done now, you can display the definition of the generated rule by selecting it from the rule list and choosing **Show Rule** from the **Rules** menu or alternatively by pressing the right button of your mouse in the rule list.

![Figure 17: Display the definition of sumsquares_rule](image)

**STEP 15** Now, we have a rule that can transform a call to sumsquares into another call. To generate the corresponding tail-recursive function, select the rule from the rule list, and use **Rule to Term** from the **Rules** menu. Ultra will ask you for a substitution for the name of the function. Since sumsquares is already defined (it’s the old definition), enter e.g. newsumsquares.

![Figure 18: Enter a new name for the new function](image)

**EXPL** Now, Ultra generated the corresponding function and stored it in the system theory as well.
OPTION  Similarly to viewing a rule, you can select a term in the term list and display its definition by choosing Show Term from the Terms menu or alternatively by pressing the right button in the term list.

![Figure 19: Display the definition of newsumsquares](image)

EXPL  Though you can continue the derivation with one or more opened windows showing definitions of rules or terms, it may be distracting to have too many of them opened simultaneously.

You may wonder where the functions of the sumsquares derivation have gone. They are still loaded in the system, but in another theory, namely sumsquares.thy. When you finished the derivation and the new rule was generated, Ultra automatically switched to display system.thy since only one theory can be displayed at a time. To switch between theories, use the Theory List which is located above the rule list. At last you have finished your first derivation!
6 Next Steps

We have prepared another sample derivation for you. It will demonstrate more complex tasks you can perform with *Ultra*. Feel free to have a look into the system reference before or while performing this derivation. We assume that you have already performed the derivation of the Sum-Squares function as shown in the previous section.

From an already operational but inefficient program for the *Maximum-Segment-Sum* problem, we will derive an efficient version. After a short review of the problem, we start our derivation.

6.1 The Maximum-Segment-Sum Problem

Here is a description of the problem: given a sequence of integer numbers, find the maximum sum of a segment of the sequence. A segment is a possibly empty sub-sequence without gaps. By definition the sum of an empty segment (or sequence) is 0. Take a look at the following sequence:

\[
\begin{array}{c}
5, -3, -7, 8, 1, -2, -3, 3, 6, -2
\end{array}
\]

Its maximum segment sum is 13 and a segment where this sum is achieved is shown. Generally there may be many segments with the maximum sum (for our example the shown segment is actually the only one). Take a moment to think about a straight-forward (but not necessarily efficient) solution for this problem.

A naïve algorithm could go like this: first, generate all possible segments of the given sequence. Then, calculate the sum of each segment. Finally, find the maximum of these numbers.

We will represent sequences and segments as lists – the predominant data structure in functional languages. Then, recursive implementations of the three sub-tasks (generating all segments, calculating the sum and the maximum of a sequence) can be easily obtained and proved correct by induction. The tasks are sequentialized by function composition.

Since there are \(\Theta(n^2)\) segments for a sequence of length \(n\), and the calculation of the sum for each segment takes linear time, this algorithm has a run-time of \(O(n^3)\). Detailed analysis shows that the run-time behaviour is indeed \(\Theta(n^3)\).

6.2 Sample Derivation

**STEP 1** Start *Ultra*. For our specific derivation we will need additional functions which are defined in the theory file `mss.thy` that is located in the sub-directory `Theories`. This file also contains all the rules we are going to apply during the transformation session. Add this theory to the system by typing

```
:a Theories/mss.thy
```

into the editor or by using *Add Theory* from the *File* menu. A listing of `mss.thy` is given in the appendix (D.4).
EXPL When Ultra has finished loading the theory, it will tell you so in the status line at the bottom of the user interface. Ultra will also automatically select the newly loaded theory from the theory list and thus show you the defined rules and terms. Don’t worry about the asterisks appended to the names of the rules: we’ll describe them later (7.1.3).

The mss function is the implementation of the naïve algorithm for the Maximum-Segment-Sum problem as described in the previous section. It appears in the term list.
STEP 2  Start the derivation by typing `mss` at the command-line prompt.

![Figure 21: Starting the Derivation on mss](image)

EXPL  *Ultra* switches from command-line mode to derivation mode. The control buttons are enabled now. Remember that if you want to quit *Ultra* you first have to return to command-line mode (e.g. by using `Cancel Derivation` from the Derivation menu).
STEP 3   Select the term mss by pointing to it with your mouse and pressing the left button. We will now perform an unfold step. This will replace the just selected term by its definition. Press the Unfold button from the control bar.

EXPL   The editor now shows the naïve algorithm: segs generates all possible segments of a sequence, sum calculates the sum of a sequence, and maximum finds a maximal element of a sequence of integers. As in Gofer, function composition is denoted by the infix . operator, and map is the well-known functor (from a set to its Kleene closure) which applies a function to every element of a sequence, see (4.2).
STEP 4 Select the \texttt{segs} sub-term and unfold it, too. The result is:

Figure 23: Unfold \texttt{segs}

\textbf{EXPL} The definition of \texttt{segs} uses three functions that may be new to you: \texttt{concat} (also known as \textit{flatten} or \textit{join}) is given a list of lists, and returns the concatenation of all lists as a single list. \texttt{tails} resp. \texttt{inits} are similar to the \texttt{segs} function, only that they do not generate all segments, but only those that end at the end resp. start at the beginning of the given sequence.
Since function composition is associative, there is no need to write parentheses. However its syntax tree has to be represented one way or the other internally. As in Gofer, function composition is right associative in Ultra, so that by default

\[ f \circ g \circ h = f \circ (g \circ h) \]

**EXPL** As the next step, we want to apply a transformation rule to the term `(map sum . concat)`. But in its current form, there is no proper sub-tree of the syntax tree that represents this sub-expression. We will have to re-order the expression by inserting parentheses appropriately. In the control bar you find the button **Assoc/Comm** which will do the required task. As input it requires a sequence of sub-term selections - it will then try to re-order the expression following the user supplied sequence.

How do you select multiple sub-terms? Whereas the first term is selected by pressing the left button of the mouse only, to select further terms you must hold the *Shift* key on your keyboard depressed while left-clicking the desired sub-term.
STEP 5  Select the $\text{map sum}$ sub-term by pointing to the white-space right between $\text{map}$ and $\text{sum}$. Then hold down the Shift key and select the $\text{concat}$ sub-term. Release the Shift key. Press the $\text{Assoc/Comm}$ button to insert the parentheses.

Figure 24: Apply Assoc/Comm to Multiple Selection
STEP 6  Continuing our transformation, we will now apply the rule *Map Promotion* to the sub-term (*map sum . concat*). Select this term by pointing to the white-space just before *concat*. Select the rule from the rule list. Press the *Apply* button.

**Figure 25: Apply the Rule map.promotion**

**OPTION**  If you want to know what transformation you have done now, you can display the definition of the rule by selecting it and choosing *Show Rule* from the *Rules* menu or alternatively by pressing the right button of your mouse in the rule list.
The function `concat`, when followed by the function `maximum`, can be replaced by multiple calls to `maximum`. The rule *Fold Promotion* does exactly this. However, before we can apply this rule, we need two preparatory steps.

**STEP 7** Use **Assoc/Comm** to bring `maximum` and `concat` together (remember that you have to select both terms).

---

**Figure 26:** Apply Assoc/Comm to `maximum` and `concat`
STEP 8  As the second preparatory step, unfold `maximum`.

![Figure 27: Unfold maximum](image)

**EXPL** The unfold step is necessary because *Ultra* currently supports only first order pattern matching. When you take a look at the definition of *Fold Promotion*

\[
\text{fold\_promotion } f\ a = \mathbb{N} \big\uparrow \text{monoid } f\ a \big\downarrow = \text{foldl } f\ a \cdot \text{concat} \leftrightarrow \text{foldl } f\ a \cdot \text{map} \ (\text{foldl } f\ a)
\]

you see that the input scheme requires a `foldl`. Systems which support higher order pattern matching could realize that `maximum` is actually defined by a `foldl`
and apply the rule immediately. For systems without higher order pattern matching, the other possibility would be to give a specialized version of the rule:

\[
\text{maximum\_promotion} = \\
\text{[]} \to \text{maximum . concat} \leftrightarrow \text{maximum . map maximum}
\]

This, of course, is not in the spirit of functional programming.

**STEP 9** Finally, apply the rule *Fold Promotion* to \((\text{foldl max 0 . concat})\).

Figure 28: Apply the Rule *fold\_promotion*
The fold function (also known as reduce or apply) applies a binary function repeatedly to the elements of a list, until the list is collapsed into a single element. It comes in two flavours, foldl and foldr which indicate the direction of the application (and coincide if the underlying structure is a monoid).

You already saw the rule Map Distribution as an example in the section about the transformation calculus. We will now apply this rule to combine all the calls to map in the middle of the current term into a single call. On its own, Ultra cannot generalize the rule to apply to three sub-terms simultaneously, so we will have to do it in two steps. Fortunately, the order of application does not matter (for this rule). Before reading on, try it by yourself.
STEP 10  Here’s one way: select the three sub-terms in left-to-right order by pointing to the white-spaces immediately after `map`. Don’t forget to hold the `Shift` key depressed during the selection of the second and the third sub-term. Use `Assoc/Comm` to insert the appropriate parentheses.

Figure 29: Apply Assoc/Comm to three Terms
STEP 11 Select the sub-term representing the composition of the second and the third call to `map` by pointing to the white-space directly before `map tails`. Select the rule `Map Distribution` and press the **Apply** button.

Figure 30: Apply the Rule `map_distribution` (first time)
STEP 12  Repeat the application of the rule to the remaining two sub-terms that call the `map` function (point to the white-space after the composition operator).

![Figure 31: Apply the Rule `map_distribution` (second time)]
Once again, because we lack higher order pattern matching, it is necessary to unfold the term \texttt{sum} before we can apply the rule \textit{Horner Scheme}. This rule is a generalized version of the familiar scheme for speeding up the evaluation of a polynomial. It works for arbitrary operations (replacing the usual multiplication and addition) that satisfy certain algebraic properties.

**STEP 13** Unfold \texttt{sum}.

*Figure 32: Unfold \texttt{sum}*
STEP 14 Then, apply the rule *Horner Scheme* to the argument of the outer call of the *map* function (for selection, point to the white-space after the first composition operator of the argument).

EXPL To further increase the efficiency of the algorithm, we will collapse the many calls to the *foldl* function (one for each element of the *inits* sequence) into a single call to *scanl* which eliminates duplicate calculations. The rule *Scan Lemma* will do this for us:

\[
\text{scan\_lemma}\; f\; a = [] \implies \text{map}\; (\text{foldl}\; f\; a)\; .\; \text{inits} \iff \text{scanl}\; f\; a
\]
It may be seen as a specification for the `scan1` function, for which there exists a linear-time implementation (see the Gofer prelude).

**STEP 15** Select the term that matches the input scheme of the rule (point to the white-space before `inits`) and apply *Scan Lemma*.

![Figure 34: Apply the Rule `scan_lemma`](image)

**EXPL** We have now derived an algorithm for Maximum-Segment-Sum which has linear run-time behaviour. This is asymptotically optimal for a deterministic algorithm. All we can further do is to improve on the constant factors. However, as is often the case with small improvements, the program gets a little more complex.
The `foldl` and `scanl` functions sequentially perform two runs over a list. We can combine the operations into a single run. The rule which formalizes this is called *Fold-Scan Fusion*.

**STEP 16** Select the whole term (by pointing to the white-space after the composition operator) and the mentioned rule, and apply the rule.

![Image of UltraIDE with the rule application process]

**EXPL** As you can see in the status line, the system has automatically performed some simplifications. These were very basic simplifications which is reflected in the fact that *Ultra* even left `max 0 0` unsimplified.
STEP 17  We can instruct Ultra to make up this bit of simplification by selecting this term (point to the white-space between the two zeros) and applying Solve from the control bar. As one would expect, $\max 0 0 = 0$.

Figure 36: Applying Solve on $\max 0 0$
STEP 18  Finally, we introduce a common name for the expression \(\max(v+x)\ 0\) which occurs two times. Make a multiple selection composed of the two occurrences (point each time to the white-space before 0) and press the Let Abstract button.

![Image of the UTV 2-2 interface with Let Abstraction applied to \(\max(v+x)\ 0\)]

Figure 37: Let Abstraction of \(\max(v+x)\ 0\)

OPTION  We have now completed the derivation. From the Derivation menu, Show Derived Rule opens a window which displays the rule generated during the transformation session. From the same menu, Show Protocol opens a window that lists all the transformation steps you performed.
STEP 19 To let Ultra know that we finished our session, choose **Stop Derivation** from the same menu and confirm the warning message. The system then asks you for a name – this name is used to store the generated rule in the system theory for further reference. Enter `mss_transformation`.

![Ultra interface](image)

![Ultra interface](image)

**Figure 38: Stop the Derivation of mss**

**EXPL** Ultra switches back to command-line mode, and displays the last term of the derivation. Remember that the system theory is not persistent, until you choose **Save System Theory** from the File menu, and is lost when you quit Ultra.

At last you have finished your second derivation!
7 System Reference

This chapter provides a complete functional description of Ultra 2.2. The order of presentation is loosely related to the graphical user interface. We will first describe theories, the objects they consist of, and the operations you can perform on them. Then, we turn our attention to the process of a derivation. Finally, we discuss the building-blocks of a derivation: transformation steps and advanced tactics.

Specific tasks are described in detail in the next chapter. If some error occurs while performing a task and the reason is not obvious, a look into the appendix A might help.

For some of Ultra’s most common tasks we have defined short-cuts using the right or the middle button of the mouse. The short-cuts are described when we talk about the corresponding tasks. For an overview, see the appendix (C.4). Don’t worry if your mouse has less than three buttons: you can invoke all tasks through the menus anyhow.

7.1 Theories

Theories are organizational entities that group together the different objects that can be manipulated by Ultra: facts, transformation rules, data structures, and programs. Usually, the formulation of a problem theory precedes a transformation session. There, new functions and rules can be created and stored for future sessions.

7.1.1 Theory Files and the File Menu

Theories are persistent in theory files that are readable and editable using any text editor. On a first glance, a theory file is like a Gofer script with additional sections specifying facts and rules. The complete syntax of theory files is described in the appendix (D.1).

Theory files (not their contents) are manipulated by commands from the File menu.

```
Add Theory...
Load Theory...
Reload Theory

Save System Theory...
Clear System Theory

Load Project...
Compile Theory...

Quit
```

Figure 39: The File Menu

The operations Add Theory, Load Theory and Reload Theory are similar to the corresponding operations of the Gofer interpreter.
• **Add Theory**
  
  opens a dialog where you can navigate through the file system and select a theory file. Since the names of theories have to be unique, you cannot add a theory with the same name as an already loaded theory — even if it has a different path specification. Then, the system tries to include the selected theory into its database. For this to happen, it is necessary that the file can be correctly parsed, i.e. it must conform to the syntax of theory files, see the appendix (D.1).

  Furthermore, several semantic checks are performed to guarantee that no undefined names are used and that terms and rules have correct types. The new theory must not depend on names that are defined in the system theory (for the system theory might be cleared afterwards). Only if the file passes all checks successfully, it is added to the system. Otherwise, an error message is displayed indicating the cause for failure of adding the theory.

• **Load Theory**
  
  opens a dialog for file selection as well. Then, it first throws out all theories you have added since the start of Ultra or the last load operation and clears the system theory. Finally the system tries to include the selected theory into its (clean) database.

  The last operation is essentially an Add Theory operation so the same checks are performed. Use Load Theory if you want to start a new transformation session with Ultra’s database being cleared.

• **Reload Theory**
  
  will work only if the currently selected theory is neither the system theory nor the precompiled theory. It will throw away the selected theory and all other theories that depend on it. Here, Ultra conservatively assumes that a theory $T$ depends on another theory $U$, if and only if $T$ was added to the system after $U$. The contents of the system theory are saved in a file.

  Then, the system tries to reload the selected theory and the depending theories. This happens in the same order in which they were added to the system prior to the reload operation. If all theories could successfully be integrated into the system, it will try to re-add the contents of the system theory.

  Another way of reloading a theory is to select it in the theory list and to double-click with the middle button of the mouse into the field right next to the theory list where the selected theory is shown.

A special theory is the **System Theory**. Objects that are generated during a transformation session are stored there. Currently, these can be either terms that you have defined, or rules that are created as the result of a transformation.

• **Ultra** offers no way to directly move these objects between theories. That’s why you will typically gather the rules and terms in the system theory, save the theory to a file, and then manually move the objects to the desired locations.

  The system theory behaves in most ways just like any other theory, meaning that rules and terms in the system theory can be used the same way as rules and terms in any other theory. The real difference is that the system theory can be modified – on the one hand objects are being added to it, on the other
hand it is sometimes emptied – either implicitly through a **Load Theory** operation or explicitly by the user.

Objects in the system theory cannot be referenced by other theories. They can be used only in the system theory itself and in derivations. This guarantees that at any time there are no undefined functions in *Ultra*'s database.

We'll now have a look at the two operations on the system theory from the **File** menu.

- **Save System Theory**
  makes the contents of the system theory persistent. A file dialog pops up that lets you enter the name of the file to save the system theory to. After the save operation the selected file contains the terms and rules from the system theory in a readable (ASCII) format. Moreover, the format adheres to the syntax of theory files, so that you can load the file just like any other theory file. The contents of the system theory are not affected by the save operation.

- **Clear System Theory**
  erases all objects defined in the system theory. This is an irreversible operation. You will usually use this after you have saved the system theory to a file.

If you are dealing with large transformation tasks, you will benefit from modularizing your project and using only those theory files that are needed in each session. To reduce your labour with loading the desired theories (and paying attention to load them in the right order) you can collect sets of theory files into a project file.

- A project file is an ASCII file that contains on every line the file-name of a theory. The file-names must be either relative to the location of the project file or absolute paths.

- **Load Project**
  prompts you to select a project file. *Ultra* will then try to load the theories designated in the project file. However, before that happens, the system is reset by throwing away all loaded theories and clearing the system theory. Thus the loading of a project is equivalent to a **Load Theory** operation on the first theory of the project followed by **Add Theory** operations for every remaining theory of the project.

All operations that can have a destructive effect on the system theory are disabled in derivation mode since the derivation might use names defined in the system theory. The affected operations are **Load Theory**, **Reload Theory**, **Clear System Theory**, and **Load Project**.

**Advanced Topic: Compiled Theories** If you are dealing permanently with the same theory files, you can speed up the loading process by compiling the theories and integrating it directly into the *Ultra* system. The theory is then already loaded at system startup. By default there is one compiled theory, *precompiled.thy*, that contains most of the functions you know from the Gofer prelude. It is not difficult to generate a new precompiled theory: when you select **Compile Theory** a dialog pops up that lets you select the theories you want to compile and the file they should be written to. The figure shows an example where the theories *tree.thy*, *algebra.thy*, and *precompiled.thy* are selected for compilation.
Figure 40: Compiling several theories into the Gofer readable file `tree.gs`

Press the Compile button to start compilation. Press the Done button to close the dialog box. In most cases it makes sense to include the currently precompiled theory into the set of theories to compile. Then, you can replace Ultra’s `precompiled.gs` by your newly created compiled theory. For seeing the effects you will have to quit Ultra, perform a reload operation in Gofer, and then restart Ultra. All the theories you have compiled are already loaded into the system now. By changing the Ultra sources it is possible to have several compiled theories loaded at the start of the system:

- Include the names of the newly generated compiled theory files (that are actually Gofer scripts) into the `ultra.p` project file right after the already available `precompiled.gs` entry.
- In each of the new files, rename the (only) defined function, so that all functions are assigned unique names that are not defined elsewhere (e.g. use `precompiled_XXX` where `XXX` is a unique identifier).
- Modify the function `s_process_initial_theories` in the file `s_neutral.gs` so that it loads the new theories and adds them to the internal database. To this end, add a new function call `s_load_compiled_theories precompiled_XXX` for each new compiled theory.

Since operations with precompiled theories go right to the heart of Ultra remember to always make a backup copy of Gofer sources you are going to modify (including `precompiled.gs`).

We conclude this section with the Quit option from the File menu. This is the recommended way of quitting Ultra. Don’t forget to save the system theory if it contains important objects. The Quit operation can be performed in command-line mode only. When you perform a derivation, you will have to stop the derivation first.

### 7.1.2 The Theory List

You can view the list of currently loaded theories by selecting the Theory button which is located below the File Menu. The Theory List opens as a pull-down menu.

![Figure 41: The Theory List with `precompiled.thy` selected](image)

From this list you can select a theory that you will use for further operations. There is always exactly one theory selected and its name is shown in the field to the right...
of the **Theory** button. When a new theory is selected both the term list and the
rule list are updated.
A double-click with the middle button of the mouse into the aforementioned field
triggers the **Reload Theory** task which will try to reload the currently selected
theory. A double-click with the right button of the mouse will start the **Apply
Catalog** operation, which we will describe in the next section.

### 7.1.3 The Rule List and the Rules Menu

The theory that is selected in the theory list may define rules. If so, they are shown
in the **Rule List**.

![Figure 42: The Rule List showing the rules of algebra.thy](image)

When performing a derivation you can apply a rule on some part of the derived
term. One way to do this is to select the rule you want to apply and double-click
with the left button of the mouse onto it. On the other hand a double-click with the
middle button applies the selected rule backwards. We will discuss the application
of rules in more detail when talking about derivations.

![Figure 43: The Rules Menu](image)

Other operations with rules are accessible via the **Rules** menu.

- To have a look at the definition of a rule, select the rule and use **Show Rule**.
  Alternatively select the rule and press the right button of the mouse in the rule
  list. In both cases a window will open that allows you to view the definition
  and save it to a file in ASCII format.
A very simple kind of rule — where the left hand side is a single variable —
can be converted to a term. Why would you want to do so? Let’s suppose you
have a function slow_func with some definition:

```
slow_func :: a -> b
slow_func = \x -> slow_def
```

Starting a derivation on slow_func you succeed to replace the definition by a
more efficient algorithm. The derivation results in a rule:

```
make_func_faster =
[] |- slow_func <=> \x -> fast_def
```

Now, the only way to use the new definition is to use slow_func and to ap-
ply the rule make_func_faster to every occurrence of it. This may become
tiresome, so Ultra helps you. Select the rule and apply Rule to Term from
the Rules menu. The system will prompt you for a substitution for the name
slow_func. Enter fast_func, and a new term will be created and stored in
the system theory:

```
fast_func :: a -> b
fast_func = \x -> fast_def
```

From now on, you will be able to use fast_func in your derivations right from
the start.

A rule can be converted to a term regardless of its applicability conditions.
Rules where the single variable of the left hand side appears on the right hand
side as well can be converted to a term, too. The resulting function is recursive
then.

For the application of the Apply Catalog operation a subset of the rules from
the currently selected theory can be chosen. Rules to be used are marked with
an asterisk * after their name. By selecting a rule and using Mark/Unmark
Rule you can toggle the status of each rule individually. For the same effect
you can select the rule and press the middle button of the mouse in the rule
list.

Then, Apply Catalog will try to perform a series of transformations on
the selected sub-term using the marked rules of the currently selected theory.
Although the status of each rule as being marked or unmarked does not change
when you switch between theories, only the marked rules of the currently selected theory are used for these transformations.

In contrast to the other operations of the Rules menu, Apply Catalog can be started in derivation mode only. The marked rules of the currently selected theory are applied only in left-to-right direction. Ultra considers each rule only if all of its applicability conditions can be resolved, which implies that no new applicability conditions are introduced.

The group of marked rules must be confluent in the sense that (mutually) recursive application does not lead to non-termination. It is your responsibility to assure this — otherwise, Ultra will interrupt the application at some level and give you an error message.

7.1.4 Clauses

You know that every transformation rule is an inference. The premises are called applicability conditions. Syntactically a premise is a Horn clause. When a rule is applied during a transformation, Ultra checks the applicability conditions. For this, a Prolog-like resolution algorithm is used. The facts and clauses the system can choose from to resolve the applicability conditions are stored in the internal clause database.

You can add clauses to the database by declaring them in the clauses section of theory files, see appendix (D.1). Unlike rules and terms, the clauses of the selected theory are not shown explicitly in a clause list. Instead, you can inspect them by selecting Show Clauses from the Clauses menu.

Figure 45: The (rather small) Clauses Menu

A new window will open that shows all clauses known to the system in lexicographically sorted order. Although Ultra knows which theory each clause came from, there is currently no way for you to see this. This means that the contents of the clause database are updated as theories come and go, too.

Figure 46: Show the clause database

The above figure shows a small clause database from which Ultra can deduce that (++) with [] forms a monoid. The actual contents of the window depend on what theories you have loaded.
The clauses shown in the window can be saved to a file in ASCII format. Moreover, the syntax of the file meets the specification for the syntax of the clauses section of a theory file, so that you can include the data into a theory file.

7.1.5 The Term List

The theory that is selected in the theory list may define terms. If so, they are shown in the Term List.

Figure 47: The Term List showing some terms from precompiled.thy

Terms are Gofer functions that can be used as sub-expressions in derivations and in the definition of clauses and rules. To have a look at the definition of a term, select it in the term list and use Show Term from the Terms menu.

Figure 48: The Terms Menu

Alternatively, after selection press the right button of the mouse in the term list. In both cases, a window will pop up with the definition of the selected term. Additionally, its type is displayed.

Figure 49: Definition and type of iterate
You can save the contents of this window to a file in ASCII format. The type and
definition as displayed correspond to the syntax of theory files. Of course, any used
functions must be defined if you copy the contents to a theory file.
When Ultra is in derivation mode, you can find out the type of any sub-term of
the expression you are working on. Do this by selecting the term and using Show
Type. This will open a window that displays the selected term and its type and
lets you save it to a file.

The displayed type is not the most general type that can be assigned to the selected
term, but the most general type that can be assigned in the given context.

7.2 Command-Line Mode

Ultra is operated either in command-line mode or in derivation mode. At any time,
the system is in exactly one of these modes. You can deduce which mode is active
by looking at the editor. The editor is the large white-coloured rectangle-shaped
part of the graphical user interface. Whenever there is a blinking cursor prompt,
Ultra is in command-line mode. Otherwise Ultra is in derivation mode.

When you start Ultra, the system comes up in command-line mode. This mode is
inspired by the Gofer interpreter’s traditional command-line interface. That means
that you can type in commands into the editor at the prompt. Most of the commands
are reachable through the menus as well — so they are just short cuts for users who
prefer using the keyboard instead of the mouse.
All the commands start with a colon : followed by a one-character symbol. Let’s
give a summary of the commands that Ultra understands:
- :?
  Displays the list of valid commands, each with a short description.

- :q
  Exits Ultra immediately. This has the same effect as Quit from the File menu except that you are not asked for confirmation.

- :a name.thy
  Adds the theory name.thy to the system. This is the same as Add Theory from the File menu. For the :a command exactly one argument is mandatory. It will be used to find the theory in the file system, so you will have to include a directory path if necessary.

- :l name.thy
  Loads the theory name.thy to the system. This is the same as Load Theory from the File menu. If necessary, path specifications have to be included in the name. The single argument to the :l command is optional. If omitted, the system is reset (i.e. all theories are thrown away and the system theory is cleared) without loading a new theory.

- :r name.thy
  Reloads the theory name.thy. This has the same effect as selecting the theory and performing Reload Theory from the File menu. The argument for the :r command is mandatory. It shall call a theory by its short name, i.e. without any path specifications.

- :p name.prj
  Loads the project name.prj. This is the same as Load Project from the File menu. For the :p command exactly one argument is mandatory that must include the path if necessary.

- :t expr
  Shows the type of the expression expr. The argument to the :t command is mandatory. It is not restricted to be a term name, but may be any valid expression. To reach this operation through the menu, you would have to start a derivation on the expression, select the whole term, and use Show Type from the Terms menu.

- :i name
  Displays the definition of the term name. The argument to the :i command is mandatory and must be the name of an existing term. This has the same effect as selecting the theory where the term is defined, selecting the term in the term list, and use Show Term from the Terms menu.

- :n pattern
  Lists the terms that match pattern together with their types. In addition to the usual characters that appear in identifier names, the pattern may contain the wildcard character * which stands for a possibly empty sequence of arbitrary characters. The wildcard character may occur multiple times. The single argument to the :n command is optional. If omitted, all terms are displayed.
7.3 Derivations

To start a derivation, you have to enter a valid expression at the command-line prompt. Ultra then switches from command-line mode to derivation mode. The blinking cursor disappears, the editor window is cleared, and the expression you entered at the prompt is displayed.

- You can start a derivation only on closed expressions, i.e. all sub-terms must be defined. From this requirement we will now deduce a special treatment for certain kinds of expressions. Say you want to start a derivation on the map function. But as first argument to map you don’t want a concrete function (e.g. head). Rather you want to start a derivation on map f, parameterized over f. Alas, map f is not a closed expression (assuming f is not defined somewhere else). The solution is to lambda abstract from f. So you start a derivation on \f -> map f.

- In general, for every parameter of your expression you will have to introduce a lambda abstraction. However, during a derivation you don’t want to do anything with these lambdas since they are only the parameters! Through the whole derivation, the lambdas would stay in front of the expression you really manipulate.

Ultra solves this problem by treating all lambda abstractions at the outer-most level of an expression with which a derivation is started as global parameters. The first expression that is not a lambda abstraction is used for the derivation. From the point of view of this expression, all parameters look as if they had been globally defined. The implicit lambdas remain undisplayed through the derivation. As an example, consider a derivation started on \f g xs -> map f (map g xs)

There are three lambda abstractions at the outer-most level. These will not be displayed during the derivation. Therefore, when Ultra switches to derivation mode, it will show

map f (map g xs)

- You may wonder, why we made the assumption that a derivation can only be started on a closed expression. It should be clear, that you cannot have any undefined terms during a derivation, e.g. you don’t know their type. So the only alternative would have been to declare any free occurrence of undefined names as global parameters.

However, for all typing mistakes in the expression, we would now introduce parameters. Since the lambda abstractions would not be shown anyhow, the typing mistakes could remain unnoticed for a long time into the derivation, which had to be restarted then.
7.3.1 The Derivation Menu

Operations concerning derivations at large are found in the Derivation menu. They are available whenever Ultra is in derivation mode.

A derivation is a sequence of transformations. In each step a transformation rule is applied to a sub-term of the current expression. The result of a derivation is itself a rule that shows how the term at the start of the derivation can be transformed into the term at the end of the derivation. The applicability conditions of the transformation rules of the individual steps are combined and form the applicability conditions of the resulting rule.

What kind of transformations can be performed in the derivation steps we’ll have a look at in the next section. For this section, let’s stay with the abstract idea of a derivation step.

- At each stage of a derivation Ultra can show you what rule it would generate if you stopped the derivation at that moment. Show Derived Rule opens a window with the virtual rule. This is essentially the same window as you would get by a Show Rule on the generated rule after stopping the derivation. Again, you can save the contents to a file.

- If you do not just want to look at the final result, but you like to see the whole derivation process summarized, use Show Protocol. A window will pop up showing in textual format the individual steps. The contents can be saved to a file in ASCII format.
7.3 Derivations

In *Ultra* the protocol is a derivation-time only object. If the derivation is stopped or aborted, the protocol is gone. If you need the contents (e.g. for documentation purposes) you will have to save them prior to ending the derivation.

- There are two ways of bringing a derivation to an end. You can abort the derivation immediately by **Cancel Derivation**. Without any further confirmation *Ultra* switches back to command-line mode. All that remains from the derivation is the last term which *Ultra* displays in the editor before showing the command-line prompt. This is an irreversible operation.

- Usually, you will end the derivation regularly by **Stop Derivation**. *Ultra* asks for confirmation. If you answer in the affirmative you are prompted to give a name to the new-born rule. The names of rules must be unique in a theory only. Different rules may have the same name if they are in different theories (it is another matter if this is a good idea). Finally the rule is stored in the system theory under the name you entered. It can be used from now on just like any other rule. Don’t forget that the system theory is a temporary place — your rule might get deleted depending on your further operations if you don’t save the system theory in time.

### 7.3.2 Selecting Terms

In most cases you will apply a transformation rule to a part of the displayed term only. To tell *Ultra* which part should be affected, you need to select sub-terms. In derivation mode the editor is configured to allow only the selection of well-formed parts of the term. A well-formed part is a complete sub-tree of the syntax-tree of the displayed expression. To select a sub-tree you have to point to the root and press the left button of your mouse.

- Unfortunately the structure of the syntax-tree is not explicitly visible in the *Ultra* editor. Moreover, the order of terms changes if infix operators are involved, as we’ll see later. For now, let’s start with the usual functional (prefix) notation.
As an example, consider the term \( \text{map } f \ (\text{map } g \ x) \) during a derivation. For the following discussion, suppose that the free variables \( f \), \( g \), and \( x \) are defined by lambda abstractions at the outer-most level. They are taken as global parameters then, and remain undisplayed.

- Take a look at the underlying syntax tree. The leaves correspond to the term names that occur in the expression. Inner nodes \( \oplus \) are the application nodes. You can read them as “apply the left sub-tree to the right sub-tree”.

![Syntax tree of \( \text{map } f \ (\text{map } g \ x) \)](image)

The tree has 9 nodes, so there are 9 different sub-terms in the expression that can be selected. Which are the 9 positions you have to point to for selecting each sub-term? To the 5 leaves the 5 term names of the expression correspond. To the 4 inner nodes the 4 white-space regions between the term names correspond.

- We can say more formally how to find the root of a tree in the editor. A tree is either a term name (a leaf in the syntax-tree) or the concatenation of its left sub-tree, the following white-space region, and its right sub-tree. The root of a tree corresponds to the white-space region between the left and the right sub-trees. To select a tree, point at this white-space region.

- As the next step, let’s consider the special treatment of infix operators now. Say, you want to start a derivation on

\[
f \rightarrow \text{const} \ 1 \ . \ \text{head} \ . \ \text{iterate} \ f
\]

If there was no support for infix operators, you would have to enter (and Ultra would have to display) *Lots of Irritating Superfluous Parentheses*:

\[
f \rightarrow (.) \ (\text{const} \ 1) \ ((.) \ \text{head} \ (\text{iterate} \ f))
\]

- Obviously, it is more comfortable to work with the infix representation. Again, when *Ultra* switches to derivation mode, the outer-most lambda abstraction
remains undisplayed. Take a look at the underlying syntax tree. Remember that the function composition operator \((.)\) is a right associative infix operator.

![Syntax tree of `const 1 . head . iterate f`](image)

Figure 56: Syntax tree of `const 1 . head . iterate f`

If an infix operator is to be displayed, *Ultra* first displays its first argument, then the operator itself, followed by the second argument. Compared with the usual prefix notation, the operator and its first argument are *swapped* in infix notation. Of course, the internal representation of the syntax-tree remains the same. Take a look at the left-most @ operator in the syntax-tree. The whitespace region corresponding to this node is the whitespace between `const 1` and . (the function composition operator).

This tree has 13 nodes, so there are 13 different sub-terms in the expression that can be selected. To the 7 leaves the 7 term names of the expression correspond. To the 6 inner nodes the 6 whitespace regions between the term names correspond.

- When associative operations occur in expressions, it can be the case that the syntax-tree does not exactly fit to be a valid input for a transformation rule. This happens when the (implicitly set) parentheses combine the wrong sub-trees. In this case it is necessary that you re-order the internal representation by using the *Assoc/Comm* task which we will describe in the next section. *Assoc/Comm* — as well as some other operations provided by *Ultra* — require two or more sub-terms to be marked (in this case they define the desired order of the sub-terms after rearrangement).

You can select multiple sub-trees by selecting them sequentially while holding the Shift key depressed. Pressing the mouse button while the Shift key is depressed does not toggle the status of a sub-tree between selected and not selected, but means always “add this sub-tree to the selection”. Note that there is always an order implied by the sequence of selections. Whether the selected sub-trees are interpreted as a set or as an ordered sequence depends on the task you issue after the selection.

The best way to get comfortable with the term selection mechanism is to experiment with it.
7.3.3 Navigation

As you already saw, a derivation is a sequence of transformation steps. The system remembers all transformations you applied since the start of the derivation. This means that you can undo one or more of your steps. It is also possible to redo a step, i.e. to undo one or more undo steps. The undo/redo feature in Ultra behaves the same way as undo buffers in most other applications.

- Before you undo a step, all the transformations and the resulting terms are gathered in the *derivation history*. When you undo a step the most recent element of the history will become the current term while the current term moves into the *derivation future*. Further undo steps move more terms from the history to the future. Similarly, when you redo a step, the most recent element of the future will become the current term while the current term moves back to the history. Further redo steps move more terms from the future to the history. If you made several undo steps (i.e. your future is not empty) and decide to make a new transformation instead of a redo step, the derivation future is cleared. It is no longer possible to get back to these derivation states.

- In the GUI, navigational tasks are found at the right end of the control bar which is above the editor.

\[\begin{array}{|c|c|}
\hline
\text{Zoom in} & \text{Back} \\
\hline
\text{Zoom out} & \text{Forward} \\
\hline
\end{array}\]

Figure 57: Navigational Tasks in the Control Bar

**Back** performs an undo step (moves back in history). If you just started the derivation or moved all the way back to the start, **Back** is not allowed any more. *Ultra* gives you an error message if you still try.

**Forward** performs a redo step (moves forward into the future). If there is no derivation future, you will get an error message as well.

- If you deal with large transformation sessions, the notion of a sub-derivation will help you managing complexity. In a sub-derivation you concentrate on a sub-term of your expression. All the transformations you apply to the sub-term have only effects local to it. The context remains unaffected, except for certain applicability conditions that must be valid for the whole term. *Ultra* does not know what a sub-derivation is, but enables you to concentrate on part of your term by **zooming**.

- If you select a sub-term and use **Zoom In**, the selected term will be displayed in the editor as if it would be the whole expression on which the derivation was started. You can then perform transformations as usual. If you need to, you can further zoom in for sub-sub-derivations.

As soon as you finished the sub-derivation you can display the whole expression again with **Zoom Out**. Zooming out is not exactly the inverse operation of zooming in. *Ultra* does not remember exactly from where you zoomed in, but
with once pressing the **Zoom Out** button, one more level of the syntax-tree will be displayed. This means that you may have to do several times a zoom out to get back to where you once belonged.

- If you are already at the outer-most level, i.e. the complete syntax-tree is displayed, **Zoom Out** will bring you an error message. **Zoom In** will bring you an error message only if you didn’t mark a term.

### 7.4 Transformations

Every step of a derivation is the application of a transformation rule to a part of the displayed expression. In **Ultra** there are user-defined transformation rules and built-in transformation rules. We will discuss user-defined transformation rules when we talk about the **Apply** task. Built-in transformations are designed to support the basic tasks of the unfold-fold paradigm [3]. These are: folding, unfolding, function definition, abstraction, instantiation and rule application.

From another point of view, transformation rules are functions that map a term to a new term. We call such functions *tactics*. Powerful tactics can be built from elementary ones by means of *tactic combinators* [8]. **Ultra** defines several general tactics, some of which are: simplify, solve, and rearrange associative or commutative operators. These general tactics along with the unfold-fold and related transformations are accessible through the **Control Bar**. Further elaborate and specialized tactics are available as well. They are reachable via the **Tactics** menu.

#### 7.4.1 The Control Bar

The **Control Bar** is the rectangle-shaped collection of buttons above the editor. Besides the navigational tasks we discussed in the precious section, the control bar contains unfold-fold transformations and general tactics.

![Figure 58: Transformational Tasks in the Control Bar](image)

All tasks in the control bar are enabled only during a derivation. Most tasks require that you select a term before calling them. The selected term then serves as input for the task. We will now describe the unfold-fold tasks in detail:

- **Unfold**: to unfold the occurrence of a term means to replace it by the right hand side of its defining equation. The input for the unfold operation is a sub-tree where the left-most leaf is the name of a function. **Ultra** will look up the definition of this function in its database which includes the function sections of all loaded theories. If the occurrence of the function name is bound (e.g. by a **let** expression), **Ultra** will look for a local definition instead. If a definition is found, the occurrence is replaced. Then, optionally some automatic simplification may be carried out on the selected sub-tree. We will discuss automatic simplification at the end of this section.

If no definition is found — because you are trying to unfold a primitive function like (+) or a lambda-defined variable — you will get an error message.
The same happens if you try to unfold a sub-tree that does not have a function name as its left-most leaf, but e.g. a lambda abstraction or the constant 1. As an example, suppose we start a derivation on

\[ \text{\textbackslash}xs \rightarrow \text{any} \text{null} \text{xs} \]

Remember that the outer-most lambda abstraction is not displayed. If you mark any and press Unfold, the system will unfold the definition of this function and you’ll get

\[ (\text{\textbackslash}p \rightarrow \text{or} . \text{map} \text{p}) \text{null} \text{xs} \]

You can then further simplify this term by applying a \( \beta \)-reduction (with the Simplify task). Alternatively you could have marked any null and pressed Unfold to get

\[ (\text{or} . \text{map} \text{null}) \text{xs} \]

immediately. This is an example of the automatic simplification feature which carried out the \( \beta \)-reduction for you.

- **Fold**: to fold a term means to replace its occurrence by a call to a function. The fold operation is the opposite of the unfold task: the right hand side of a defining equation is replaced by its left hand side. The input for the fold operation is an arbitrary sub-tree. Normally, Ultra will search its database to find all terms that can be folded. You can impose a (rather severe) restriction on this: if you select a term in the term list, Ultra will only try to fold this function. Note that it is possible that the same function can be folded in different ways, if the function is defined over pattern matching (or equivalently over a case) and more than one of the right hand sides match. In all cases, Ultra will construct a list of all possible ways the selected sub-tree can be folded. If the list is empty, you will get a message that a fold cannot be performed (this happens hardly ever because the identity function can always be used). If the list contains one element, the fold is immediately carried out by the system. Otherwise, Ultra opens a dialog that lets you select the right instance.

![Figure 59: Folding x : iterate (const 1) (const 1 x)]
For functions defined over pattern matching or case the number of the equation or branch that can be folded is shown. Ultra numbers the different equations or branches in the order of their definition.

For example, if you fold the term

\[ x : \text{iterate} \ (\text{const} \ 1) \ (\text{const} \ 1 \ x) \]

and select iterate from the list of possible instances, the result is

\[ \text{iterate} \ (\text{const} \ 1) \ x \]

**Define:** to define a function means to give a name to a term so that it can be referred to in later steps of the derivation or in other derivations. The input for the define operation is an arbitrary sub-tree. Ultra then asks you for the name of the function. If the name you enter already exists in the term database you will get an error message. Otherwise a new function with this name is created and stored in the system theory. All free variables in the selected sub-term become parameters to the new function. The definition of the function is just the selected sub-term. In the editor, the selected sub-term is immediately folded to its new definition. Other occurrences of the same sub-term are not affected. The new function is ready for use in the on-going derivation.

As an example, selecting the whole term

\[ x : \text{iterate} \ (\text{const} \ 1) \ (\text{const} \ 1 \ x) \]

for function definition, and entering `func1` as the name will result in a new function

\[ \text{func1 :: Int -> [Int]} \]
\[ \text{func1} \ x = x : \text{iterate} \ (\text{const} \ 1) \ (\text{const} \ 1 \ x) \]

stored in the system theory and the term in the editor replaced by the folded definition

\[ \text{func1} \ x \]

**Let Abstract:** to perform a let-abstraction means to introduce a new local name for a sub-term. Instead of creating a new function as Define does, the name is introduced via the Gofer let expression. The input for the let-abstract operation is an arbitrary sub-tree. Ultra creates a list of the free variables in the selected sub-term. For each free variable, the defining lambda expression is searched. Of all these lambda expressions, the inner-most is the place where Ultra inserts the new let expression. To be precise, it is inserted just after the inner-most lambda expression, but still at a level after the outer-most lambda abstractions. Therefore, no bounded variable leaves its scope accidentally, and the newly created let-expression will be visible. Of course, the name defined by the let abstraction is new.

You can select multiple sub-trees to serve as input for the let-abstract operation. Ultra will then introduce names for all selected sub-terms. The order of selection effects the nesting order of the let-expressions. Structurally equal sub-terms will receive a common name.

For example, if we start a derivation on
\[ \text{id} (\lambda \mathbf{x} \rightarrow \text{map} (\text{map} f) (\text{map} f \mathbf{x})) \]

and select the two occurrences of the term \text{map} f simultaneously and apply \textbf{Let Abstract}. \textit{Ultra} will generate

\[ \text{id} (\lambda f \rightarrow \text{let} \ a = \text{map} f \ \text{in} \ \lambda \mathbf{x} \rightarrow \text{map} \ a (a \mathbf{x})) \]

- \textbf{Lambda Abstract}: similarly to \textbf{Let Abstract}, a new variable is introduced. This time however, not via the \texttt{let} expression but via a lambda expression. The input for the lambda-abstract operation is an arbitrary sub-tree. In contrast to the let-abstraction, \textit{Ultra} does not calculate a place where the abstraction is inserted. Instead, \textit{Ultra} opens a window that reads \texttt{mark scope} and prompts you for the selection of another sub-tree. After you have selected the scope in the editor, press the \texttt{Ok} button. The new variable is then defined at the outer-most level of this sub-term. Of course, the second sub-term must include the sub-term that is to be abstracted. If this is not the case, you will get an error message.

Again, multiple sub-trees can serve as input to the lambda-abstract operation. New lambda expressions are introduced for each of the selected sub-terms. They are nested according to the order of selection. Structurally equal sub-terms will receive a common lambda variable. The scope for the new variable must include all selected sub-trees.

As an example, applying \textbf{Lambda Abstract} to \texttt{iterate g} in

\[ \text{all null} (\text{map} f (\text{iterate} g \mathbf{x})) \]

and selecting \((\text{map} f (\text{iterate} g \mathbf{x}))\) as the scope results in

\[ \text{all null} ((\lambda a \rightarrow \text{map} f (a \mathbf{x})) (\text{iterate} g)) \]

- \textbf{Case Intro}: a case-introduction is used to introduce pattern matching. \textit{Pattern matching} can be done over expressions that have an algebraic type: this includes types defined by the Gofer \texttt{data} declaration and the built-in list type \([a]\). The input for the case-introduction operation is a sub-tree that has an algebraic type. Just as with a lambda-abstraction, \textit{Ultra} opens a window that reads \texttt{mark scope} and prompts you for the selection of another sub-tree. After you have selected the scope in the editor, press the \texttt{Ok} button. At its outer-most level, the \texttt{case} expression is inserted. Again, the scope of the case-introduction must be a super-term of the term that will be decomposed into its cases. An error message results otherwise.

\textit{Ultra} will generate a pattern for each constructor the type of the expression has. If the expression is a simple variable, its occurrences in the scope are automatically replaced by the case patterns.

For example, a case-introduction can be performed on \(\mathbf{x} \in \) in

\[ \text{map} f \ \mathbf{x} \]

with the whole term as the scope. The result is

\[
\text{case} \ \mathbf{x} \ \text{of} \\
\qquad [] \rightarrow \text{map} f \ [] \\
\qquad x : a \rightarrow \text{map} f (x : a)
\]
which could be further simplified by unfolding the calls to the map function to

```
case xs of
  [] -> []
  x : a -> f x : map f a
```

- **Apply**: this task is used to apply user-defined transformation rules. The input to the apply operation is an arbitrary sub-tree. Normally, **Ultra** will search its database to find all rules that can be applied to the selected sub-term. You can impose a (rather severe) restriction on this: if you select a rule in the rule list, **Ultra** will only try to apply this rule.

If none of the rules can be applied (because the selected sub-term does not match the input template of any rule), you will get an error message. If there are two or more rules that can be applied, **Ultra** opens a dialog that lets you select the rule you want to be applied. You can identify a rule by its name and the theory it belongs to.

If you have selected a rule (or there was just one applicable rule anyway), there still may be many different ways to instantiate the formal parameters of the rule with real terms. In this case, **Ultra** opens a dialog that lets you select the appropriate instantiation.

When you have selected the right instance (or there was just one way to instantiate the parameters anyway), the rule is applied. If the rule has applicability conditions they are instantiated as well. If they cannot be resolved they are added to the list of applicability conditions of the current derivation (and then of the generated rule). The issue of resolving applicability conditions is currently not handled in **Ultra**, see (9.2).

As an example, suppose you have loaded the theory `algebra.thy` which specifies rules for algebraic structures and apply the rule `add_neutral_right` to the sub-term `xs` of

```
map f xs
```

**Ultra** will present you several instantiations for the formal parameters of the rule. We want the instance where the function is the concatenation (++) and the right-neutral element is the empty list `[]`. The resulting expression is

```
map f (xs ++ [])
```

- **Apply Backwards**: if a user-defined rule describes an equivalence relationship, it can be applied in two directions. The **Apply** task only tries to apply each rule from left to right. If you need to apply a rule in the other direction use the apply backwards operation. Except for the switched direction, there are no differences between the two apply tasks.

Applying the rule `add_neutral_right` backwards to the sub-term `(xs ++ [])` of the result of the last example

```
map f (xs ++ [])
```

leads (as expected) to

```
map f xs
```
Besides the unfold-fold tasks, there are general tactics reachable through the control bar. We will describe these in detail now:

- **Claim**: you can claim that an expression can be transformed into another expression and defer the proof to a later time. The input for the claim operation is an arbitrary sub-tree. *Ultra* then asks you to provide the term that the selected sub-term should be replaced with. Of course the supplied term must have a type that allows it to be inserted in the selected context. If this is the case, *Ultra* carries out the replacement and extends the applicability conditions of the current derivation by a clause that asserts just the equality of the two terms. You can continue with the new term and leave the proof of this condition to a later stage.

For example, if you start a derivation on

```
null . filter null . filter (not . null)
```

you might know that applying the two `filter` functions always has the empty list as its result. If you are unwilling or unable to prove this now by direct transformation, you can mark the term `filter null . filter (not . null)` and use Claim and enter `const []` to get

```
null . const []
```

Now you can continue the derivation. If you stopped the derivation now, *Ultra* would generate the following rule:

```
[filter null . filter (not . null) <=> const []] |=
  null . filter null . filter (not . null) <=>
  null . const []
```

The applicability conditions reflect the yet unproved equivalence that you used in the transformation. There is currently no means to eliminate applicability conditions by proving them in *Ultra*, see (9.2).

- **Assoc/Comm**: whenever associative or commutative operators occur in a derivation, sub-terms often happen to appear in the wrong order or with parentheses set in the wrong places so that they do not match the input template of a rule that you want to apply. If *Ultra* knows about the associativity or commutativity of these operators (and you can tell these facts via the clauses section of theory files), it will let you re-order the sub-terms.

The input to the Assoc/Comm operation is a multiple selection of sub-terms. In the syntax-tree, all sub-terms must be connected by the same operator that must be at least associative, and may be commutative as well. Here, the multiple selection is treated as a sequence, i.e. the order in which you select the sub-terms does matter.

*Ultra* will first try to bring all selected sub-terms together. If the terms are not already adjacent and the operator is not commutative, the task fails and you will get an error message. Then, *Ultra* uses the associative property of the operator to group all selected sub-terms into a single sub-tree. Afterwards, *Ultra* will try to re-order the sub-terms so that they appear in the same order as you specified in the multiple selection. If they are not already in this order and the operator is not commutative, the task fails, too. Finally, *Ultra* inserts
parentheses according to whether the operator associates to the left or to the right by default. The resulting sub-tree that contains all selected terms will be displayed without any parentheses.

As a simple example, you can re-order the terms \(a\), \(b\), and \(c\) in

\[ a + (b + c) \]

by simultaneously selecting first \(c\), then \(a\), finally \(b\) and using **Assoc/Comm** to yield

\[ c + a + b \]

In the resulting expression the parentheses of the left associative addition are omitted. This transformation only works if you have told *Ultra* that \(+\) is indeed associative and commutative. These facts are stored e.g. in the theory `algebra.thy`, see appendix (D.2).

- **Simplify**: to simplify a term means to try to apply several built-in transformation rules with the hope that the resulting term is less complex than the one at start. The transformations involved are very general and make only small changes and are combined by higher-level combinators. We call them tactics in this context. The simplify operation is itself a (high-level) tactic. As input it takes an arbitrary sub-tree. *Ultra* then tries to apply in a recursive manner small transformations. They include reductions (e.g. \(\beta\)-reduction), evaluations of arithmetic, logical and descriptive operations, and simplifications of language-specific constructs such as `let` or `case`. If no transformations could be performed, you will get an error message. Otherwise, you will immediately see the resulting term after the simplifications.

- **Solve**: the solve tactic is an extended version of the **Simplify** task. In addition to the transformations there, *Ultra* tries to perform unfold operations. Because the term database must be consulted for this kind of tasks, the solve tactic is more time-consuming than the simplify tactic. On the other hand, the results are generally better. Concerning the input to and the output from the solve operation, the same comments apply as with the simplify operation. As an example consider the term \((\lambda x \to \text{fst} \ x) \ (1+2, \ 3+4)\). **Simplify** with the whole term as input will evaluate the arithmetic operations and perform the \(\beta\)-reduction to yield

\[ \text{fst} \ (3, \ 7) \]

**Solve** will further unfold the call to \(\text{fst}\). Thereafter, another \(\beta\)-reduction can be carried out, and the result is

3

By the time the system finishes some task, it will update its status information which can be seen at the bottom of the user interface. You might want the system to perform some simplifying steps automatically after each transformation. This can be toggled with help of the **Automatic Simplification** widget in the bottom right corner of the main window.
You can choose between three items:

- **None**: do not perform any automatic simplifications.
- **Changed part**: try to perform automatic simplifications within the scope selected for the underlying task. The simplifications are carried out after the task is finished. Terms not selected as input for the task are not affected.
- **Whole term**: try to perform automatic simplifications on the whole resulting term.

If any, the number of simplifications is displayed in the status line at the end of the operation. The transformations involved are similar to that of the **Simplify** task. By default, the **Automatic Simplification** option is set to **Changed part** which is a trade-off between clarity and performance considerations.

### 7.4.2 The Tactics Menu

Just like the tasks from the control bar, all tasks in the **Tactics** menu are enabled only during a derivation and most of them require a selected term as input.
The tactics reachable through this menu are not as general as the \texttt{Simplify} or the \texttt{Solve} tactics from the control bar, but offer elaborate transformations for special situations. The input for all tactics is a sub-tree of the displayed expression.

First, there are tactics that aid the transformation of language-specific constructs. Since the source language for programs is Gofer, these tactics address constructs from Gofer, of course. Nevertheless, similar constructs exist in most modern functional languages.

- \textbf{Eta Reduction}: is the application of the transformation rule \((\lambda x \to fx) \equiv f\). The selected sub-term must match the left hand side of the rule.

- \textbf{Inverse Eta Reduction}: is the backward application of the rule \textit{Eta Reduction}. A new lambda-bound variable is created. The selected sub-term must have the type of a non-constant function, since a new parameter is added.

- \textbf{Let in Let}: manipulates \texttt{let} expressions in three different ways. If the body of a \texttt{let} expression contains another \texttt{let} expression, \textit{Ultra} tries to group the declarations together and forms a single \texttt{let} expression. The same happens, if the declaration part of a \texttt{let} expression contains a nested \texttt{let}. If the selected sub-term is not an unnested \texttt{let} expression with a single item in its declaration part, it is converted to a lambda abstraction. The last transformation is intended to be followed by a \(\beta\)-reduction.

- \textbf{Lift Let and Case}: moves \texttt{let} and \texttt{case} expressions from nested scopes of the selected sub-term to the outer-most level.

- \textbf{Lambda to Case}: inserts a \texttt{case} expression right after the lambda abstraction that has a single case.

Next, there are variations of the generic \texttt{Simplify} and \texttt{Solve} tasks from the control bar that might fit special situations.

- \textbf{Simplify with Unfold}: performs part of the simplification rules of the \texttt{Simplify} tactic. Additionally, \textit{Ultra} tries to unfold sub-terms.

- \textbf{Lazy Solve}: performs roughly the same simplification rules as the \texttt{Solve} tactic, but they are combined in different ways. This can lead to other results as with the \texttt{Solve} operation.

- \textbf{Advanced Context Rule}: uses information about the equality of two terms. In expressions such as \((x == y) \&\& f\) where \(f\) may depend on \(x\) and \(y\), the equality can lead to simplifications in \(f\).

Then follow tactics for the introduction and simplification of non-operational (descriptive) expressions. Besides assertions (guard \(\triangleright\)), \textit{Ultra} has several built-in constructs to express non-determinism: selection (\texttt{some}, \texttt{that}), quantification (\texttt{exists}, \texttt{forall}), and a non-deterministic choice operator (\(\|\)). For a short introduction to descriptive specification in \textit{Ultra}, see the appendix (B.2).

- \textbf{Flat}: replaces a \texttt{case} expression by an equivalent \(\lambda\)-abstraction and then eliminates the abstractions in favor of guarded expressions. For a detailed description of the \textit{flat}-tactic see [12], pp. 37-38.
- **Simplify with Flat**: performs simplification steps in addition to the Flat tactic.

- **Import Assertion**: applies the transformation rule

\[
\frac{g \triangleright (l \odot r)}{(g \triangleright l) \odot (g \triangleright r)}
\]

if the selected term fits the input scheme. This distributes a guard over the two arguments of a binary operator. It is a correct transformation only if the operator is strict in at least one of its arguments.

- **Export Assertion**: applies the transformation rule from **Import Assertion** in backward direction. The same restrictions apply.

Finally, there are tactics that perform basic transformation steps that could be counted to the unfold-fold rules from the control bar, as well.

- **Apply Catalog**: carries out the application of the user-defined rules in the currently selected theory. This operation is described in the section about the Rules menu (7.1.3) in detail.

- **Local Fold**: tries to fold the selected sub-term. Instead of searching the database for possible instances as the Fold operation does, Ultra looks for a suitable definition in the derived expression (i.e. in its let declarations).

- **Rename**: is used to rename variables in the sub-term. Ultra searches the selected sub-term for local (bound) variables and displays them in a window. You can then give a new name to each variable you would like to be renamed. Then, all variables are given the new names simultaneously.

### 7.5 Miscellaneous

We conclude the System Reference with the Help menu.

![Manual](Manual.png)

![About](About.png)

Figure 62: The Help Menu

Although there is no online help manual implemented within Ultra, the Manual option opens a window that contains references to the latest versions of documents concerning Ultra.

The About option opens a window that contains information about the developers of Ultra.
8  How To

We now give a few hints for frequently occurring tasks.

8.1  Load Multiple Theory Files

With the Load Theory task, you can only load one theory file. If you want to load another one additionally, use Add Theory instead. If you use the same group of theory files over and over, consider to create a project file with their names.

8.2  Start a Derivation

To start a derivation, you must be in command-line mode (if not, you are already running a derivation — you will have to stop or cancel the running derivation to start a new one). You are in command-line mode, if you can see a blinking cursor next to Ultra’s prompt waiting for some input.

At the cursor prompt, enter a valid Gofer expression. Make sure that all variables are defined either as functions in a loaded theory or locally by a let or lambda abstraction. Press the Enter key on your keyboard.

You cannot enter expressions that make use of the off-side layout rule (as allowed in Gofer scripts). You will have to replace such constructions by their alternative representations with curly braces and semicolons.

8.3  Keep Lambda Abstractions Explicit at the Outer-Most Level

If you start a derivation, all lambda abstractions at the outer-most level are considered to be global parameters of the derivation and removed implicitly. Rarely, you may want to keep the lambda abstractions visible during the derivation. The bad news is that there is no way to do this directly. However, you can always insert an artificial dummy expression (that is no lambda abstraction) at the outer-most level: Ultra then recognizes that there is no lambda abstraction to remove. The best choice is the identity function id. For example, if you want to start a derivation on

\[ \langle f \ g \rightarrow \ \mathrm{map} \ f \ . \ \mathrm{map} \ g \right\]

and keep f and g visible, start a derivation on

\[ \text{id} \ (\langle f \ g \rightarrow \ \mathrm{map} \ f \ . \ \mathrm{map} \ g \right) \]

instead. If you have no identity function available (because you are working without the standard precompiled theory, you can use a case or a let construction instead. After ending the derivation, it might be necessary to remove the dummy expression from the left hand side of the generated rule manually.
9 Future

In this section we will address the question, in which directions Ultra may evolve.

9.1 Automation

With Ultra’s generic tactic combinators it is easy to generate new elaborate tactics that can automatically perform large parts of certain kind of transformations. New tactics and simplification rules can be added to the system with relatively small amount of work.

However, as the level of automation is getting higher, the user loses control about what is really happening. This can lead to the elimination of alternative derivation strategies.

Still, in every greater project, there are design decisions to make. To generate efficient code from a descriptive specification in different situations by pressing a button is unrealistic.

Furthermore, if new data structures are defined (that are unknown to standard Ultra) — and this is almost always the case — there are no built-in automation strategies to handle them. So it’s again up to the user to add transformation rules that apply to the new structures in theories. Of course, large libraries of functions, rules, and properties of commonly used data structures (e.g. graphs) can be written to improve the situation.

9.2 Reduction Mode

At the moment, unresolved premises cannot be proven and removed from rules in Ultra. One way to solve this problem would be to add a further operation mode, called Reduction Mode, in which Ultra is used to prove new facts that are used to resolve premises. Closely related is the support for induction proofs that would improve the usability in situations where recursive data structures or functions are used (which is quite often the case in functional languages).

9.3 Limitations

We start with the limitations which may take a longer time (if ever) to be improved since they would require serious modifications of Ultra or its environment.

- The unification of Ultra is restricted to a first-order pattern matching. In some cases it is thus necessary to rearrange the selected term manually before a rule can be applied (cf. the sample derivation of mss). Higher-order unification is not supported.

- Ultra’s type system is based on the standard Hindley/Milner system. There is no support for type classes and constructor classes. Therefore, e.g. the type of == is a -> a -> Bool.

- The export of theory files to \LaTeX format does not work properly yet and has therefore been removed from the standard system.
• The operations triggered by the mouse short cuts in the rule list and in the term list only work on the currently selected rule or term. You can change the selection only prior to these operations with the left mouse button. The TkGofer interface has currently no safe operation to update the selection immediately when other mouse buttons are pressed.

• Ultra only supports tuples with a maximum of 24 elements (probably you wouldn’t like to write functions operating on larger tuples anyhow).

We continue with limitations that are more likely to be eliminated soon. Those concerning the system core will be eliminated before those affecting the GUI only.

• The built-in parser only accepts a limited set of the Gofer syntax. Not supported are the @ operator, irrefutable patterns (~), modules, the do-notation, and where clauses local to case expressions.

• The editor in command-line mode is derived from a general class of editors and not specially designed to be a command-line. Repositioning the cursor often leads to chaotic behavior. Your input is correctly recognized if you enter text only after the last prompt (i.e. at the very end of the editor contents) and if the cursor is at the end of your input when you press the Enter key.

• Compiled Theories can be inconsistent. If you generate one by using Compile Theory from the File menu, Ultra does not check whether all terms in the theories you compile are also defined in the same theories. Furthermore, Ultra does not perform any checks on the precompiled theories at start-up time. Up to now, it is in your responsibility to ensure that all names are defined.

• Compiling Theories that contain the backslash character \ in strings (e.g. in function identifiers) produces erroneous Gofer scripts, since the character is not displayed the way it should be for the Gofer interpreter (which expects \\).

You can send us comments and bug reports by e-mail. For contact addresses take a look at the Ultra home page:

http://www.informatik.uni-ulm.de/pm/ultra/
A I got an error message and don’t know why

In the following we deal with some common and less common mistakes.

- Errors during Add Theory, Load Theory and Reload Theory

  - repeated definition of "xxx" in theory yyy
    * In most cases you are trying to add a theory that is already loaded in the system. Use Reload Theory instead.
    * If this is not the cause then the theory you want to add contains a repeated (possibly different) definition of a term that is already defined in some other theory you loaded (do not forget about the system theory and the precompiled theory).
    If you do not need the other theory you can wipe out all loaded theories (except the system theory and the precompiled theory) by performing Load Theory instead. Currently there is no way of unloading a single theory (there might be other theories depending on it).
    * If you need the other theory and the new definition of the repeated term is different, you should give them differing names. Currently, there is no module concept implemented in Ultra.
    * If the definitions are equal, discard one of them. If it does not make sense to discard one and leave the other, you will have to factor the definitions out to a common theory.
    * If the other theory is the system theory you can save the contents to a file with Save System Theory and clear it with Clear System Theory.
    * If the other theory is the precompiled theory it usually makes sense to rename the newly defined terms. Only in rare cases you might want to generate a new precompiled theory by Compile Theory.

  - PARSE ERROR (line xxx, column yyy) -- syntax error --
    * If you are sure that the term or type definition that follows this error message is indeed correct, remember that unlike to Gofer files, in theory files the section where function definitions are made must be marked with the special tag --! FUNS. There must be at most one section of each kind (FUNS, RULES and CLAUSES). The sections must occur in this order.

  - undefined identifier "xxx" in theory yyy
    * In most cases you accidentally performed a Load Theory operation on a theory that depends on other theories that were already loaded. Try Add Theory instead (after reloading the necessary theory files — you will notice that all the originally loaded theories are gone now).
    * If you performed Add Theory then the theory you are trying to add depends on other theories that have not been loaded yet. Add these theories first.
* If you are sure that the theory you are trying to add does not depend on other theories, you probably misspelled a function definition or its use.
* Maybe you are trying to use a function which is in the system theory (this is not allowed, for the system theory might be cleared while other loaded theories reference its objects).
* In rare situations, you are trying to use a function from the pre-compiled theory, but the precompiled theory was modified by a new compilation in the mean time.

• Errors during a derivation caused by transformation steps from the control bar

  − no matching definition
  * If you tried to Fold a term, there could be a term selected in the term list. Ultra then only tries to fold the selected function and performs no search in the whole database for functions to fold. You can un-mark the selected term by pressing the left button on the label above the term list.
  * If no term is selected in the term list, there is indeed no function in the database that can be folded. Check if you have loaded all necessary theories and selected the right sub-tree for folding.

  − rule not applicable
  * If you tried to Apply a rule, there could be a rule selected in the rule list. Ultra then only tries to apply the selected rule and performs no search in the whole database for rules that could be applied. You can un-mark the selected rule by pressing the left button on the label above the rule list.
  * If no rule is selected in the rule list, there is indeed no rule in the database that can be applied. Check if you have loaded all necessary theories and selected the right sub-tree to which the rule should be applied.
  * You can also try Apply Backwards if the rule is symmetrical and you are not sure about the direction. The Apply task only applies the rule in forward (left-to-right) direction.

  − not a function application
  * If you tried to Unfold a term, the selected sub-term is not a prefix of a function application (i.e. its left-most leaf is not the name of a function). Lambda abstractions or let or case expressions or numerical or character constants cannot be unfolded.
  * If you tried Assoc/Comm, make sure that the selected sub-terms are all connected by one and only one operator.

  − no definition found
  * You have tried to Unfold a function that is either primitive (such as the (+) operator) or bound locally by a lambda abstraction. In both cases, Ultra has no definition to unfold. Don’t forget about the lambdas at the outer-most level — they are parameters to the derivation and not displayed.
- multiple terms marked
  - The tactic you tried to apply needs only one term as its input. This is the case for almost every tactic, including Apply (there is no way to apply a user-defined rule to multiple sub-terms). Currently, there are three tasks that can handle a multiple selection as the input:
    - Assoc/Comm, which in fact needs multiple terms to perform a meaningful operation.
    - Lambda Abstract introduces multiple variables (one for every selected sub-term) at the same time. Note that the scope of the abstraction must be a single term again.
    - Let Abstract similarly can introduce multiple variables. Here, the scope cannot be explicitly provided by the user, but is calculated by Ultra.

- Errors during a derivation caused by tactics from the menu

- Tactic failed
  - If you tried the Apply Catalog task, it may be the case that there is no applicable rule in the currently selected theory. Check if you have selected the right theory. Check if you have selected the right sub-term as input to the operation. Check if all rules you want Ultra to consider for application are marked with an asterisk (mark them by using Mark/Unmark Rule from the Rules menu). Remember that Ultra only tries to apply the rules in forward (left-to-right) direction and that rules with unresolvable applicability conditions are not considered for application.

  - If you are sure, that a marked rule in the currently selected theory is applicable to the selected sub-term in forward direction it may be the case that this rule or some other marked rule leads to non-termination when applied recursively or in mutual recursion to the selected sub-term. This is the case when a group of rules always inserts sub-trees to which one of the rules in the group can always be applied. Here are some examples (we omit the empty list of applicability conditions):
    - The correct rule id <=> id . id
    - The wrong rule [] <=> [1]
    - Every rule i <=> o together with its backward direction o <=> i
    - The two laws of DeMorgan in combination:
      \[
      \text{not (a && b) <=> not (not a || not b)}
      \]
      \[
      \text{not (a || b) <=> not (not a && not b)}
      \]
A I got an error message and don’t know why
B Predefined Constructs

*Ultra* knows the following types and type constructors:

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool</td>
<td>boolean values</td>
</tr>
<tr>
<td>Char</td>
<td>character values</td>
</tr>
<tr>
<td>Int</td>
<td>integer values</td>
</tr>
<tr>
<td>[]</td>
<td>lists</td>
</tr>
<tr>
<td>-&gt;</td>
<td>functions</td>
</tr>
<tr>
<td>()</td>
<td>tuples</td>
</tr>
</tbody>
</table>

Additionally, there is one type synonym:

```haskell
type String = [Char]
```

B.1 Built-in Functions

Here is a table with all predefined functions, their types, precedence and associativity information. The last entries are constructor functions.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>prec</th>
<th>assoc</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Int -&gt; Int -&gt; Int</td>
<td>6</td>
<td>left</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>Int -&gt; Int -&gt; Int</td>
<td>6</td>
<td>left</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Int -&gt; Int -&gt; Int</td>
<td>7</td>
<td>left</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Int -&gt; Int -&gt; Int</td>
<td>7</td>
<td>none</td>
<td>division</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Bool -&gt; Bool -&gt; Bool</td>
<td>3</td>
<td>right</td>
<td>conjunction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool -&gt; Bool -&gt; Bool</td>
<td>2</td>
</tr>
<tr>
<td>=&gt;&gt;</td>
<td>Bool -&gt; Bool -&gt; Bool</td>
<td>1</td>
<td>right</td>
<td>implication</td>
</tr>
<tr>
<td>not</td>
<td>Bool -&gt; Bool</td>
<td></td>
<td>negation</td>
<td></td>
</tr>
<tr>
<td>ord</td>
<td>Char -&gt; Int</td>
<td></td>
<td>ASCII value</td>
<td></td>
</tr>
<tr>
<td>chr</td>
<td>Int -&gt; Char</td>
<td></td>
<td>ASCII character</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>a -&gt; a -&gt; Bool</td>
<td>4</td>
<td>none</td>
<td>equality</td>
</tr>
<tr>
<td>/=</td>
<td>a -&gt; a -&gt; Bool</td>
<td>4</td>
<td>none</td>
<td>inequality</td>
</tr>
<tr>
<td>&lt;</td>
<td>a -&gt; a -&gt; Bool</td>
<td>4</td>
<td>none</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>a -&gt; a -&gt; Bool</td>
<td>4</td>
<td>none</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>a -&gt; a -&gt; Bool</td>
<td>4</td>
<td>none</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>a -&gt; a -&gt; Bool</td>
<td>4</td>
<td>none</td>
<td>greater than</td>
</tr>
<tr>
<td>undefined</td>
<td>a</td>
<td></td>
<td>⊥</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>String -&gt; a</td>
<td></td>
<td>⊥</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>Bool -&gt; a -&gt; a</td>
<td>1</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a -&gt; a -&gt; a</td>
<td>5</td>
</tr>
<tr>
<td>some</td>
<td>(a -&gt; Bool) -&gt; a</td>
<td></td>
<td>select from many</td>
<td></td>
</tr>
<tr>
<td>that</td>
<td>(a -&gt; Bool) -&gt; a</td>
<td></td>
<td>select from one</td>
<td></td>
</tr>
<tr>
<td>exists</td>
<td>(a -&gt; Bool) -&gt; Bool</td>
<td></td>
<td>existential quantification</td>
<td></td>
</tr>
<tr>
<td>forall</td>
<td>(a -&gt; Bool) -&gt; Bool</td>
<td></td>
<td>universal quantification</td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>()</td>
<td></td>
<td>unit</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>a -&gt; [a] -&gt; [a]</td>
<td>5</td>
<td>right</td>
<td>non-empty list</td>
</tr>
<tr>
<td>[]</td>
<td>[a]</td>
<td></td>
<td>empty list</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>Bool</td>
<td></td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>True</td>
<td>Bool</td>
<td></td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>a -&gt; a</td>
<td></td>
<td>reserved</td>
<td></td>
</tr>
</tbody>
</table>
Of course, character and integer constants are available as well. String constants may be denoted as in Gofer.

For the specification of rules, the following reserved symbols can be used:

<table>
<thead>
<tr>
<th>name</th>
<th>prec</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>=</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>0</td>
</tr>
<tr>
<td>&lt;=&lt;&gt;</td>
<td>1</td>
<td>equivalence clause</td>
</tr>
<tr>
<td>==&gt;</td>
<td>1</td>
<td>descendance clause</td>
</tr>
<tr>
<td>===</td>
<td>2</td>
<td>semantic equivalence</td>
</tr>
<tr>
<td>=&gt;=</td>
<td>2</td>
<td>semantic descendance</td>
</tr>
</tbody>
</table>

**B.2 Descriptive Specification**

We will now give an overview of Ultra's descriptive expressions. For further description of Non-Determinism in Ultra see [12].

Please note that descriptive specification in Ultra is still experimental and should be used with some caution.

- Let's start with the guarded expression \( g \triangleright e \), where \( g \) is of type Boolean. For example, if you want to assert that a number \( d \) is not 0 before dividing \( n \) by \( d \), you can introduce a guard \( (d \neq 0) \triangleright n/d \), which is denoted

\[
d \neq 0 \triangleright n / d
\]

in Ultra. Note the use of \( \triangleright \) as an infix operator with weak precedence. The value of an expression \( g \triangleright e \) is the value of \( e \) if \( g \) can be True, and \( \perp \) (undefined) otherwise.

- Now let's come to the non-deterministic operators. To denote an expression that can have multiple values, we use the choice operator. For the evaluation of \( a \parallel b \), which is written as

\[
a \mid::\mid b
\]

in Ultra, a non-deterministic choice is made between \( a \) and \( b \). Further non-deterministic operators deal with selection and quantification.

- To select an arbitrary value that satisfies a predicate, you can use the some construct. As a simple example, suppose you have a function \( \text{elem} \) that checks for the occurrence in a list (such a function can be found e.g. in the theory file list.thy). Then, the value of

\[
\text{some} (\lambda x \rightarrow x \text{ 'elem' } [77,3,19])
\]

can be either 77 or 3 or 19. Using the choice-operator, this is denoted as

\[
77 \parallel 3 \parallel 19
\]

in Ultra as

\[
77 \mid::\mid 3 \mid::\mid 19
\]

Parentheses can be omitted because the choice-operator is associative. To satisfy type correctness, some must be followed by a predicate \( p :: a \rightarrow \text{Bool} \). The result \( \text{some} \ p \) then has the type \( a \).

If there is no value that satisfies the given predicate \( p \), the value of \( \text{some} \ p \) is \( \perp \).
• To select a uniquely characterized element, use `that`. For example,

\[ \text{that } (\lambda x \to x == 6 \ast 7) \]

has the value 42. The value of `that \ p` is `⊥`, if there is no value or more than one value that satisfies `p`.
Concerning typing, the comments stated for `some` apply to `that` as well.

• If you do not need to select a concrete value, but you are only interested if there is one, use the `exists` expression.

\[ \exists x \to p \ x \]

evaluates to `True`, if the predicate `p` is `True` for at least one argument. Using \( \eta \)-reduction, we can re-write the previous expression to

\[ \exists p \]

and you can now verify that the parameter `p` for `exists` must be of type `p :: a \to \text{Bool}`. The type of the whole expression `exists p` is `Bool`.
If the predicate evaluates to `False` for all arguments, the value of the `∃` quantification is `False`, too.

Concerning typing, the comments stated for `exists` apply to `forall` as well.

• With the `forall` expression, you can state that a predicate must be fulfilled for all possible arguments. For example,

\[ \forall x \to x + (-x) == 0 \]

evaluates to `True`. Note that the type of the bounded variable `x` is restricted to `Int` by the use of the `(+)` function (actually, as well by the use of the `(-)` and the `== 0` functions).
If the predicate evaluates to `False` for at least one argument, the value of the `∀` quantification is `False`, too.
Concerning typing, the comments stated for `exists` apply to `forall` as well.

With the exception of the choice operator, which has `⊥` as a neutral element, all other non-deterministic operators, as well as the guard, are strict.
C  The Ultra System

We present the logical and physical structure of Ultra, and give you hints on how to set up several parameters for your personal use and how to use the mouse for short-cuts to common tasks.

C.1 Structure of the System

The following diagram gives an overview of the components of Ultra:

Terms, rules, and clauses are the basic data that Ultra deals with. In the file system, they are in theory files. That is where the parser reads them from, and the unparsers writes them to. The parsed data is type-checked, name-checked, and comes into the theory database. During a derivation, data from the database is used to manipulate the derived term. The manipulation is carried out with combined tactics, simplifications, and rule applications. Derivations, the theory database, and the operations on theory files are controlled by the system core. The core is separated from the user interface which itself builds upon TkGofer. Events from the user trigger system tasks that have effects on the core and on the GUI.

C.2 Files in the Distribution

We give a description of the directories and files in the Ultra 2.2 distribution. We begin with the directories.

- **Bitmaps** contains small bitmap files that are displayed in some of Ultra’s dialogs.

- **Theories** contains theory files that cover some areas of programming including lists, trees, natural numbers, and algebraic objects. The source code of the standard *precompiled theory* is given here, too.
• **lib** contains various useful libraries for parser, monads, graphs, pretty documents. Tasks, which combine the (otherwise separated) programming of the GUI and of the system core, are implemented here, too.

• **parser** contains the sources of *Ultra*’s parser for theory files. The parser is called when theory files are loaded and whenever you enter a term (e.g. to start a derivation).

• **term** contains the sources for the representation and the manipulation of expressions. Furthermore, term substitutions and the first-order term-matcher are implemented here.

• **typechecker** contains the sources for type-checking theory files. Loaded theories and terms you enter are type-checked after parsing.

• **unparser** contains the sources for pretty-printing theory files. The unparser is used when the *system theory* is saved or whenever *Ultra* displays a term (e.g. when you select show term) or a rule.

We continue now with the source files in the main directory. Files for similar tasks are grouped together, and described in the order of their appearance in the group.

• **about.txt, manual.txt** contain the information that is displayed in the dialogs called by the options from the **Help** menu.

• **basic_definitions.gs, generate_theory.gs** implement basic data types, constructors, and functions together with the builder of theories which includes the name-checker.

• **c_derivation.gs, c_neutral.gs, core.gs**, implement *Ultra*’s core: operations on the internal database in derivation mode and in command-line mode, and the definition of the database structure along with operations on it.

• **catbase.gs, derivation.gs, reduction.gs** implement basic operations on theories, derivations, and reductions.

• **display.gs** contains the definition of the layout of the GUI and the bindings of the menu and control buttons to their tasks.

• **g_derivation.gs, g_neutral.gs, gui.gs** implement the GUI effects of operations in derivation mode and in command-line mode, and the internal representation of the GUI along with access functions.

• **inference.gs, prolog.gs** implement the data structures and the calculus of rules and clauses.

• **main.gs** contains *Ultra*’s main function.

• **precompiled.gs, corestate.gs** are the compiled (ready to load in Gofer) versions of the standard precompiled theory and a snapshot of the *Ultra* core and GUI state. The latter is not used at the moment.
C.3 Customization

You can customize Ultra by modifying the standard settings in the file `setup.gs`. There is a section for font specification, and a section for color specification where you can customize the appearance of the GUI elements. The most important other settings are

<table>
<thead>
<tr>
<th>Function</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>False</td>
<td>Output debug messages.</td>
</tr>
<tr>
<td>expert</td>
<td>False</td>
<td>More, experimental (unsafe) system tasks are available in expert mode.</td>
</tr>
<tr>
<td>prompt_string</td>
<td>&quot;? &quot;</td>
<td>The string displayed as prompt in command-line mode.</td>
</tr>
<tr>
<td>system_thy</td>
<td>&quot;system.thy&quot;</td>
<td>The file name of the system theory (when saved).</td>
</tr>
<tr>
<td>f_fontPixelSize</td>
<td>15</td>
<td>The font size controls the size of the GUI window.</td>
</tr>
</tbody>
</table>

The only customization we recommend is the modification of the font size variable `f_fontPixelSize` to adjust the size of the Ultra window to your screen resolution.

C.4 Mouse Use

The following lists some of the most common tasks you can do using the mouse.

<table>
<thead>
<tr>
<th>Task</th>
<th>Mouse Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply Catalog</td>
<td>select theory and left double-click</td>
</tr>
<tr>
<td>Reload Theory</td>
<td>select theory and middle double-click</td>
</tr>
<tr>
<td>Apply Rule</td>
<td>select rule and left double-click</td>
</tr>
<tr>
<td>Apply Rule Backwards</td>
<td>select rule and middle double-click</td>
</tr>
<tr>
<td>Mark/Unmark Rule</td>
<td>select rule and middle click</td>
</tr>
<tr>
<td>Show Rule</td>
<td>select rule and right click</td>
</tr>
<tr>
<td>Show Term</td>
<td>select term and right click</td>
</tr>
</tbody>
</table>
D  Theory Files

We first describe the syntax of theory files. Afterwards we give an overview on Ultra’s standard theories and present the two theories used in the sample derivations.

D.1 Syntax

Ultra theory files are enriched versions of Gofer scripts. We only present the differences. Here is a short description of the syntax:

Program ::= Infix* [FunSection] [RuleSection] [ClauseSection]

FunSection ::= "--! FUNS" <<new-line>> FunTopDecl*
RuleSection ::= "--! RULES" <<new-line>> RuleTopDecl*
ClauseSection ::= "--! CLAUSES" <<new-line>> ClauseTopDecl*

FunTopDecl ::= TypeDecl | DataDecl | PrimDecl | FunDecl | FunDef
RuleTopDecl ::= TypeDecl | DataDecl | PrimDecl | RuleDecl | RuleDef
ClauseTopDecl ::= ClauseDecl | ClauseDef | Fact

RuleDecl ::= FunDecl "< declaration of the type of rules >>
RuleDef ::= Ident Ident* "=" Inference

Inference ::= "[" [Premises] "]" "|=" Conclusion
Premises ::= Clause | Clause "," Premises
Conclusion ::= Clause

Clause ::= Formula | Equivalence | Descendance
Formula ::= "[" [Antecedents] "]" "|=" Consequent
Antecedents ::= Expr | Expr "," Antecedents
Consequent ::= Expr
Equivalence ::= Expr "<=>" Expr
Descendance ::= Expr "==>" Expr

ClauseDecl ::= FunDecl "< declaration of the type of clauses >>
ClauseDef ::= Ident Ident* ":-" Expr "," Expr
Fact ::= Ident Expr

Infix ::= "< declaration of infix operators >>
TypeDecl ::= "< type declaration >>
DataDecl ::= "< data declaration >>
PrimDecl ::= "< primitive function declaration >>
FunDecl ::= "< declaration of the type of functions >>
FunDef ::= "< definition of functions >>
Expr ::= "< expression >>
Ident ::= "< identifier >"

The differences to standard Gofer files are:

• Class or instance declarations are not recognized yet. Identifiers must not contain the (reserved) character #.
• Declarations of the precedence and the associativity of infix operators must occur at the beginning of a theory file (in Gofer, they can occur everywhere).

• A theory file is separated into three sections. The rule section and the clause section are extensions to Gofer scripts. Each section must be marked with its special tag of the form \(--! \text{TAG}\). Each section is optional, but the sections that actually occur in a theory file must occur in the above specified order.

• Type declarations, data declarations, and primitive function declarations in the rule section have the same meaning as if they were in the function section (which is the usual Gofer meaning).

• Declarations of the type of rules and clauses are ignored at the moment. The same happens to declarations of the type of functions that occur without a function definition.

• Clause definitions have a Prolog-like syntax. The consequent is followed optionally by a \(\text{:-}\) and a comma-separated, non-empty list of antecedents (Gofer expressions). If the antecedents are omitted, the clause is a fact.

• Rule definitions are syntactically very similar to function definitions. The left hand side is composed of the name of the rule and an optional list of parameter variables while the right hand side is an inference.
  The left hand side of an inference are the premises: an optionally empty, otherwise comma-separated list of clauses while the right hand side is the conclusion: a clause.
  The left hand side of a clause are the antecedents: an optionally empty, otherwise comma-separated list of expressions while the right hand side is the consequent: an expression.
  There are two special clauses: equivalences and descendances. These are only abbreviations:
  \[
  \begin{align*}
  \text{lhs} & \leftrightarrow \text{rhs} \quad \text{is the same as} \quad \[\] \vdash \text{lhs} \equiv \text{rhs} \\
  \text{lhs} & \implies \text{rhs} \quad \text{is the same as} \quad \[\] \vdash \text{lhs} \implies \text{rhs}
  \end{align*}
  \]
  where \(\equiv\) and \(\implies\) are the semantic equivalence and descendance relations.

Concerning name spaces, Ultra has adopted following rules:

• The names of loaded theory files (without their paths) must be unique in the whole system.

• The names of rules must be unique in the theory they are in.

• The names of terms must be unique in the whole system.

D.2 Standard Theories

We give a short description of the theory files that are automatically installed with Ultra. You can find all these theories in the Theories subdirectory of the Ultra distribution.
D.3 Theory for the Sum-Squares Derivation

Here are the contents of the theory file `sumsquares.thy` used for the sample derivation of a solution for the Sum-Squares problem (5.3).

```plaintext
--! FUNS

```

\[ \text{sumsquares} :: \[\text{Int}\] \rightarrow \text{Int} \]
\[ \text{sumsquares} \ \text{xs} = \sum (\text{squares} \ \text{xs}) \]

\[ \text{sum} :: \[\text{Int}\] \rightarrow \text{Int} \]
\[ \text{sum} [\] \quad = \quad 0 \]
\[ \text{sum} \ (x:xs) \quad = \quad x + \sum \text{xs} \]

\[ \text{squares} :: \[\text{Int}\] \rightarrow \[\text{Int}\] \]
\[ \text{squares} [\] \quad = \quad [] \]
\[ \text{squares} \ (x:xs) \quad = \quad \text{square} \ x : \text{squares} \ \text{xs} \]

\[ \text{square} :: \text{Int} \rightarrow \text{Int} \]
\[ \text{square} \ x \quad = \quad x \times x \]

D.4 Theory for the Maximum-Segment-Sum Derivation

Here are the contents of the theory file `mss.thy` used for the sample derivation of a solution for the Maximum-Segment-Sum problem (6.2).

```plaintext
--! FUNS

```

\[ \text{sum} :: \[\text{Int}\] \rightarrow \text{Int} \]
\[ \text{sum} \ = \ \text{foldl} \ (+) \ 0 \]

\[ \text{maximum} :: \[\text{Int}\] \rightarrow \text{Int} \]
\[ \text{maximum} \ = \ \text{foldl} \ \text{max} \ 0 \]

\[ \text{inits} :: [\text{a}] \rightarrow [[\text{a}]] \]
inits [] = [[]]
inits (x:xs) = [] : map ((:) x) (inits xs)

tails :: [a] -> [[a]]
tails [] = [[]]
tails (x:xs) = (x:xs) : tails xs

segs :: [a] -> [[a]]
segs = concat . map tails . inits

mss :: [Int] -> Int
mss = maximum . map sum . segs

--! RULES

map_promotion f =
[] |=
    map f . concat
<>
    concat . map (map f)

fold_promotion f a =
[[ ] |- monoid f a ] |=
    foldl f a . concat
<>
    foldl f a . map (foldl f a)

map_distribution f g =
[] |=
    map f . map g
<>
    map (f . g)

horner_scheme f a g b =
[[ ] |- horner f a g b ] |=
    foldl f a . map (foldl g b) . tails
<>
    foldl (\x y -> f (g x y) b) b

scan_lemma f a =
[] |=
    map (foldl f a) . inits
<>
    scanl f a

fold_scan_fusion f a g b =
[] |=
foldl f a . scanl g b
<=>
let h (u,v) x = let w = g v x in (f u w, w)
in fst . foldl h (f a b, b)

--! CLAUSES

associative :: (a -> a -> a) -> Clause
lneutral :: (a -> a -> a) -> a -> Clause
rneutral :: (a -> a -> a) -> a -> Clause
commutative :: (a -> a -> a) -> Clause
distributive :: (a -> a -> a) -> (a -> a -> a) -> Clause

associative (+)
lneutral (+) 0
rneutral (+) 0
commutative (+)

associative (*)
lneutral (*) 1
rneutral (*) 1
commutative (*)

associative max
lneutral max 0
rneutral max 0
commutative max

associative (.)
lneutral (.) id
rneutral (.) id

distributive max (+)
distributive (+) (*)

neutral :: (a -> a -> a) -> a -> Clause
neutral f e :- lneutral f e, rneutral f e

monoid :: (a -> a -> a) -> a -> Clause
monoid f e :- associative f, neutral f e

horner :: (a -> a -> a) -> a -> (a -> a -> a) -> a -> Clause
horner f x g y :- distributive f g, neutral f x, neutral g y


E Glossary

**Applicability Conditions** of a rule tell what must be valid so that it can be applied. Usually, they are semantic predicates that restrict the values of variables in the rule.

**Automatic Simplification** is a mechanism that applies simplify operations after each transformation step. You can select the scope of the application. This often reduces the amount of work you need to do to clean up the mess that other tactics leave behind.

**Clauses** are parts of *Ultra’s* Prolog-like database. They give information about valid predicates for terms and how to deduce new valid predicates from existing ones. The database is used to resolve the applicability conditions of transformation rules.

**Command-Line Mode** is the mode *Ultra* starts with. Like in the Gofer interpreter you can enter commands to the system. By entering a valid expression you can start a derivation.

**Compiled Theory** is a data structure generated from a theory file that can be parsed by Gofer. It can be included into the *Ultra* source code to accelerate the loading process of theories.

**Derivation** is the process by which an expression is sequentially transformed into another expression with the help of transformation rules. The result of a derivation is itself a transformation rule that states some relation between the expressions at the start and at the end of the process.

**Derivation Mode** is the mode *Ultra* is in during a derivation. The derived expression is displayed, and you can select parts of it and apply transformation rules and tactics to them.

**Derivation Step**, also called transformation step, is the building-block of a derivation. It is performed by the application of a transformation rule to selected parts of the derived expression.

**Derived Rule** is the rule that *Ultra* would create when a derivation ended at that moment.

**Descriptive Specification**, also called non-operational specification, is a program that cannot be executed directly, because some of its operations are not yet implemented. These operations may have informal or formal specifications that describe their behavior. Usually, it is not hard to give some (very inefficient) implementation.

**Gofer** is a higher-order, referentially transparent, strongly typed functional language with lazy semantics. It supports polymorphic functions, type and constructor classes. Classes are not yet supported by *Ultra*.

**History**, also called derivation history, is the sequence of terms that resulted from the application of transformation rules during a derivation. *Ultra* uses the history if you want to undo one or more steps of the derivation.
Operational Specification is a program that can be executed directly, because there are implementations for all functions that it uses.

Project is a collection of theories that are used together for large transformation tasks. A project file specifies the theories and the sequence in which they should be loaded by Ultra. The sequence must conform to the following dependence requirement: the definition of a function must be loaded with an earlier or the same theory as any of its uses.

Protocol is a textual document created during a derivation. It shows the sequence of transformation steps that have been carried out. In particular, it shows the transformation rules that have been applied and the intermediate results.

Rule, also called transformation rule, describes how to change a term into another term while preserving an equivalence or descendence relation (for details on these relations see [9]). Technically, it is a special kind of an inference. The premises denote the applicability conditions.

Simplify, or in its stronger form Solve, is a tactic that applies several built-in transformation rules (e.g. β-reduction) to reduce the complexity of a term. The existence of a simplify mechanism allows that other (more complicated) tactics do not have to worry about the form of terms they create.

System Theory is a special theory where Ultra stores transformation rules and function definitions that are created during a derivation.

Tactic is the composition of transformation rules to achieve complex transformation effects.

Term is a synonym for a function (as an object of a theory) or the application of a function to further functions (as a part of the derived expression).

Theory is an entity that contains clauses, transformation rules, data structures, and function definitions. A theory file specifies these objects in a Gofer-like syntax.

TkGofer [14] is an extension of Gofer which is especially suited for the definition of graphical user interfaces in a functional language. Thus, new GUI features can be rapidly integrated in Ultra.

Transformation, see derivation.
References


Index

abstraction, 9, 75
also, see \( \lambda \)-abstraction
Add Theory, 19, 37, 60, 68, 89
Advanced Context Rule, 83
antecedents, 12, 102
applicability conditions, 13, 64, 65, 70, 74, 79, 80, 91
Apply, 44, 79, 99
Apply Backwards, 79, 99
Apply Catalog, 64, 84, 91, 99
assertion, see guard \( \triangleright \)
Assoc/Comm, 42, 73, 80, 90
asterisk, 64, 91
Automatic Simplification, 55, 75, 81
Back, 21, 25, 74
\( \beta \)-reduction, 81
Cancel Derivation, 21, 39, 71
Case Intro, 78
choice \( \parallel \), 83, 94
Claim, 80
classes, 12, 65, 102
Clear System Theory, 61, 89
command-line mode, 17, 39, 67, 85
Compile Theory, 61, 89
collection, 12, 102
confluent, 65
consequent, 12, 102
collection bar, 17, 39, 75, 90
customize, 99
database, 17, 65, 69, 75–77, 79, 81, 84, 90
Define, 9, 77
derivation, 63, 69, 75, 85
derivation mode, 17, 39, 61, 67, 69
derived term, 63
descendance, 13, 102, 108
descriptive, 83, 87
editor, 17, 67, 71
equivalence, 12, 79, 102, 108
error messages, 89
Eta Reduction, 83
exists, 83, 95
Export Assertion, 84
expression, 12, 17, 42, 67, 68, 80, 102
facts, 14, 59, 65, 80, 102
Flat, 83
Fold, 9, 30, 76, 90
forall, 83, 95
Forward, 21, 74
function, see term
future, 74
generated rule, 32, 57, 70, 79, 85
Gofer, 9, 61, 83, 101
guard \( \triangleright \), 83, 94
GUI, 17, 59, 99
history, 74
identity function, 76, 85
Import Assertion, 84
induction, 37, 87
inference, 12, 65, 102, 108
infix, 40, 72, 102
instance, 12, 31, 76, 79, 84, 101
Inverse Eta Reduction, 83
\( \lambda \)-abstraction, 69, 76, 83, 85, 90
Lambda Abstract, 78
Lambda to Case, 83
Lazy Solve, 83
Let Abstract, 57, 77
Let in Let, 83
Lift Let and Case, 24, 25, 83
Load Project, 61, 68
Load Theory, 60, 68, 89
Local Fold, 84
mark scope, 78
Mark/Unmark Rule, 64, 91, 99
Maximum-Segment-Sum, 37, 103
modules, 61, 89
multiple selection, 42, 73, 91
name spaces, 60, 102
navigation, 17, 74
non-determinism, 83, 94
operational, 15, 18, 37
operator, 13, 30, 40, 72, 80, 90, 102
outer-most lambdas, 69, 72, 76, 77, 85, 90
pattern matching, 11, 46, 76, 78
platforms, 9
precedence, 102
precompiled theory, 18, 61, 89
predicate, 12, 107
prefix, 71, 90
prelude, see precompiled theory
premises, 12, 65, 87, 102, 108
project, 61, 68, 85
Prolog, 65, 102, 107
proof, 37, 80, 87
protocol, 71
Quit, 10, 62, 68
re-order, 42, 73, 80
recursive, 64, 87
redo, 21, 74
Reload Theory, 60, 68, 99
Rename, 84
rule, 12, 59, 60, 65, 71, 75, 87, 102
application, 9, 13, 14, 42, 63, 70, 79, 90
horner scheme, 52
list, 17, 44, 63, 79
map distribution, 13
Rule to Term, 34, 64
sample derivation, 17, 37
Save System Theory, 33, 58, 61, 89
scheme, 12, 46
selecting terms, 17, 26, 42, 67, 71, 75
Show Clauses, 18, 65, 69
Show Derived Rule, 32, 57, 70
Show Protocol, 32, 57, 70
Show Rule, 34, 44, 63, 70, 99
Show Term, 35, 66, 68, 99
Show Type, 67, 68
Simplify, 25, 76, 81
Simplify with Flat, 84
Simplify with Unfold, 83
Solve, 56, 81
some, 83, 94
status line, 20, 55, 81
Stop Derivation, 33, 58, 71
Sum-Squares, 18, 103
syntax tree, 26, 42, 71
system theory, 33, 58, 60, 71, 77, 89
tactic, 14, 75, 80, 83, 91
term, 12, 60, 66, 68, 71, 75, 89
list, 18, 35, 66
that, 83, 95
theory, 35, 59, 61, 79, 84, 89
files, 14, 33, 59, 65, 80, 101
list, 17, 35, 62
syntax, 59, 101
TkGofer, 9
transformation, see derivation
transformation step, 17, 21, 32, 57, 74
type checking, 13, 60
un-mark, 90
undo, 21, 74, 107
Unfold, 9, 22, 40, 75, 81, 90
version, 8, 10, 84
view, see show
Zoom In, 27, 74
Zoom Out, 31, 74