NAME

pure − the Pure interpreter

SYNOPSIS

pure [*options* ...] [*script* ...] [-- *args* ...]
pure [*options* ...] -x *script* [*args* ...]

OPTIONS

--help, **-h**
Print help message and exit.

-i Force interactive mode (read commands from stdin).

-I*directory*
Add a directory to be searched for included source scripts.

-L*directory*
Add a directory to be searched for dynamic libraries.

--noediting
Do not use readline for command-line editing.

--noprelude, **-n**
Do not load the prelude.

--norc Do not run the interactive startup files.

-q Quiet startup (suppresses sign-on message in interactive mode).

-v[*level*]
Set verbosity level. See below for details.

--version
Print version information and exit.

-x Execute script with given command line arguments.

-- Stop option processing and pass the remaining command line arguments in the **argv** variable.

DESCRIPTION

Pure is a modern-style functional programming language based on term rewriting. Pure programs are basically collections of equational rules used to evaluate expressions in a symbolic fashion by reducing them to normal form. A brief overview of the language can be found in the PURE OVERVIEW section below. (In case you're wondering, the name ''Pure'' actually refers to the adjective. But you can also write it as ''PURE'' and take this as a recursive acronym for the ''Pure Universal Rewriting Engine''.)

pure is the Pure interpreter. The interpreter has an LLVM backend which JIT-compiles Pure programs to machine code, hence programs run blazingly fast and interfacing to C modules is easy, while the interpreter still provides a convenient, fully interactive environment for running Pure scripts and evaluating expressions.

Pure programs (a.k.a. *scripts*) are just ordinary text files containing Pure code. A bunch of syntax highlighting files and programming modes for various popular text editors are included in the Pure sources. There's no difference between the Pure programming language and the input language accepted by the interpreter, except that the interpreter also understands some special commands when running in interactive mode; see the INTERACTIVE USAGE section for details.

If any source scripts are specified on the command line, they are loaded and executed, after which the interpreter exits. Otherwise the interpreter enters the interactive read-eval-print loop. You can also use the **-i** option to enter the interactive loop (continue reading from stdin) even after processing some source scripts. To exit the interpreter, just type the **quit** command or the end-of-file character (^D on Unix) at the beginning of the command line.

Options and source files are processed in the order in which they are given on the command line. Processing of options and source files ends when either the **--** or the **-x** option is encountered. The **-x** option must

be followed by the name of a script to be executed, which becomes the ''main script'' of the application. In either case, any remaining parameters are passed to the executing script by means of the global **argc** and **argv** variables, denoting the number of arguments and the list of the actual parameter strings, respectively. In the case of **-x** this also includes the script name as **argv!0**. The **-x** option is useful, in particular, to turn Pure scripts into executable programs by including a ''shebang'' like

```
#!/usr/local/bin/pure -x
```

as the first line in your main script. (This trick only works with Unix shells, though.)

On startup, the interpreter also defines the **version** variable, which is set to the version string of the Pure interpreter, and the **sysinfo** variable, which provides a string identifying the host system. These are useful if parts of your script depend on the particular version of the interpreter and the system it runs on.

If available, the prelude script **prelude.pure** is loaded by the interpreter prior to any other other definitions, unless the **-n** or **--noprelude** option is specified. The prelude is searched for in the directory specified with the **PURELIB** environment variable. If the **PURELIB** variable is not set, a system-specific default is used. Other source scripts specified on the command line are searched for in the current directory if a relative pathname is given. In addition, the executed program may load other scripts and libraries via a **using** declaration in the source, which are searched for in a number of locations, including the directories named with the **-I** and **-L** options; see the sections DECLARATIONS and C INTERFACE below for details.

If the interpreter runs in interactive mode, it may source a few additional interactive startup files immediately before entering the interactive loop, unless the **--norc** option is specified. First **.purerc** in the user's home directory is read, then **.purerc** in the current working directory. These are ordinary Pure scripts which can be used to provide additional definitions for interactive usage. Finally, a **.pure** file in the current directory (containing a dump from a previous interactive session) is loaded if it is present. See the INTERACTIVE USAGE section for details.

When the interpreter is in interactive mode and reads from a tty, unless the **--noediting** option is specified, commands are read using **readline**(3) (providing completion for all commands listed under INTERACTIVE USAGE, as well as for symbols defined in the running program). When exiting the interpreter, the command history is stored in **~/.pure_history**, from where it is restored the next time you run the interpreter.

The **-v** option is most useful for debugging the interpreter, or if you are interested in the code your program gets compiled to. The *level* argument is optional; it defaults to 1. Six different levels are implemented at this time (two more bits are reserved for future extensions). For most purposes, only the first two levels will be useful for the average Pure programmer; the remaining levels are most likely to be used by the Pure interpreter developers.

1 (0x1) denotes echoing of parsed definitions and expressions;

2 (0x2) adds special annotations concerning local bindings (de Bruijn indices, subterm paths; this can be helpful to debug tricky variable binding issues);

4 (0x4) adds descriptions of the matching automata for the left-hand sides of equations (you probably want to see this only when working on the guts of the interpreter).

8 (0x8) dumps the ''real'' output code (LLVM assembler, which is as close to the native machine code for your program as it gets; you *definitely* don't want to see this unless you have to inspect the generated code for bugs or performance issues).

16 (0x10)
adds debugging messages from the **bison**(1) parser; useful for debugging the parser.

32 (0x20)
adds debugging messages from the **flex**(1) lexer; useful for debugging the lexer.

These values can be or'ed together, and, for convenience, can be specified in either decimal or hexadecimal. Thus 0xff always gives you full debugging output (which isn't most likely be used by anyone but the Pure developers).

Note that the **-v** option is only applied *after* the prelude has been loaded. If you want to debug the prelude, use the **-n** option and specify the **prelude.pure** file explicitly on the command line. Verbose output is also suppressed for modules imported through a **using** clause. As a remedy, you can use the interactive **show** command (see the INTERACTIVE USAGE section below) to list definitions along with additional debugging information.

PURE OVERVIEW

Pure is a fairly simple yet powerful language. Programs are basically collections of rewriting rules and expressions to be evaluated. For convenience, it is also possible to define global variables and constants, and for advanced uses Pure offers macro functions as a kind of preprocessing facility. These are all described below and in the following sections.

Here's a first example which demonstrates how to define a simple recursive function in Pure, entered interactively in the interpreter (note that the ''>'' symbol at the beginning of each input line is the interpreter's default command prompt):

```
> // my first Pure example
> fact 0 = 1;
> fact n::int = n*fact (n-1) if n>0;
> let x = fact 10; x;
3628800
```

The language is free-format (whitespace is insignificant). As indicated, definitions and expressions at the toplevel have to be terminated with a semicolon. Comments have the same syntax as in C++ (using // for line-oriented and /* ... */ for multiline comments; the latter may not be nested). Lines beginning with #! are treated as comments, too; as already discussed above, on Unix-like systems this allows you to add a ''shebang'' to your main script in order to turn it into an executable program.

There are a few reserved keywords which cannot be used as identifiers: case const def else end extern if infix infixl infixr let nullary of otherwise postfix prefix private then using when with.

Pure is a terse language. You won't see many declarations, and often your programs will read more like a collection of algebraic specifications (which in fact they are, only that the specifications are executable). This is intended and keeps the code tidy and clean.

On the surface, Pure is quite similar to other modern functional languages like Haskell and ML. But under the hood it is a much more dynamic language, more akin to Lisp. In particular, Pure is dynamically typed, so functions can be fully polymorphic and you can add to the definition of an existing function at any time:

```
> fact 1.0 = 1.0;
> fact n::double = n*fact (n-1) if n>1;
> fact 10.0;
3628800.0
> fact 10;
3628800
```

Like in Haskell and ML, functions and variables are often defined by *pattern-matching*, i.e., the left-hand side of a definition is compared to the target expression, binding the variables in the pattern to their actual values accordingly:

```
> foo (bar x) = x-1;
> foo (bar 99);
98
```

In fact, due to its term rewriting semantics, Pure goes beyond most other functional languages in that it can do symbolic evaluations just as well as ''normal'' computations:

```
> square x = x*x;
> square 4;
```

```
16
> square (a+b);
(a+b)*(a+b)
```

Leaving aside the built-in support for some common data structures such as numbers and strings, all the Pure interpreter really does is evaluating expressions in a symbolic fashion, rewriting expressions using the equations supplied by the programmer, until no more equations are applicable. The result of this process is called a *normal form* which represents the ‘‘value’’ of the original expression. Keeping with the tradition of term rewriting, there’s no distinction between ‘‘defined’’ and ‘‘constructor’’ function symbols in Pure; any function symbol (or operator) also acts as a constructor if it happens to occur in a normal form term:

```
> (x+y)*z = x*z+y*z; x*(y+z) = x*y+x*z;
> x*(y*z) = (x*y)*z; x+(y+z) = (x+y)+z;
> square (a+b);
a*a+a*b+b*a+b*b
```

Expressions are generally evaluated from left to right, innermost expressions first, i.e., using *call by value* semantics. Pure also has a few built-in special forms (most notably, conditional expressions, the short-circuit logical connectives && and ||, the sequencing operator $$, and the lazy evaluation operator &) which take some or all of their arguments unevaluated, using *call by name*. (User-defined special forms can be created with macros. More about that later.)

The Pure language provides built-in support for machine integers (32 bit), bigints (implemented using GMP), floating point values (double precision IEEE), character strings (UTF-8 encoded) and generic C pointers (these don’t have a syntactic representation in Pure, though, so they need to be created with external C functions). Truth values are encoded as machine integers (as you might expect, zero denotes **false** and any non-zero value **true**). Pure also provides some built-in support for lists and matrices, although most of the corresponding operations are actually defined in the prelude.

Expression Syntax

Expressions consist of the following elements:

Constants: 4711, 4711L, 1.2e-3, "Hello, world!\n"

The usual C’ish notations for integers (decimal, hexadecimal, octal), floating point values and double-quoted strings are all provided, although the Pure syntax differs in some minor ways, as discussed in the following. First, there is a special notation for denoting bigints. Integer constants that are too large to fit into machine integers will be interpreted as bigints automatically. Moreover, integer literals immediately followed by the uppercase letter ‘‘L’’ will always be interpreted as bigint constants, even if they fit into machine integers. This notation is also used when printing bigint constants.

Second, character escapes in Pure strings have a more flexible syntax borrowed from the author’s Q language, which provides notations to specify any Unicode character. In particular, the notation *****n*, where *n* is an integer literal written in decimal (no prefix), hexadecimal (‘0x’ prefix) or octal (‘0’ prefix) notation, denotes the Unicode character (code point) #*n*. Since these escapes may consist of a varying number of digits, parentheses may be used for disambiguation purposes; thus, e.g. **"\(123)4"** denotes character #123 followed by the character ‘4’. The usual C-like escapes for special non-printable characters such as **\n** are also supported. Moreover, you can use symbolic character escapes of the form **\&***name***;**, where *name* is any of the XML single character entity names specified in the ‘‘XML Entity definitions for Characters’’, see *http://www.w3.org/TR/xml-entity-names/*. Thus, e.g., "\©" denotes the copyright character (code point 0x000A9).

Function and variable symbols: foo, foo_bar, BAR, bar2

These consist of the usual sequence of ASCII letters (including the underscore) and digits, starting with a letter. The ‘_’ symbol, when occurring on the left-hand side of an equation, is special; it denotes the *anonymous variable*. The case of identifiers is significant, but it doesn’t carry any meaning (that’s in contrast to languages like Prolog and Q, where variables must be capitalized). Instead, Pure distinguishes function and variable symbols by their position on the left-hand side of

an equation, using the following *head = function* rule: Any symbol (except the anonymous variable) which occurs as the head symbol of a function application is a function symbol, all other symbols are variables (except symbols explicitly declared as ''constant'' a.k.a. **nullary** symbols, see below).

Operator and constant symbols: x+y, x==y, **not** x, []

As indicated, these take the form of an identifier or a sequence of punctuation symbols. As of Pure 0.6, operator and constant symbols may also contain arbitrary extended (non-ASCII) Unicode characters, which makes it possible, e.g., to use symbols from the math and APL symbol sets offered by Unicode.

Operator and constant symbols must always be declared before they can be used, using corresponding **prefix**, **postfix**, **infix** and **nullary** declarations, which are discussed in section DECLARATIONS.

Note that enclosing an operator in parentheses, such as (+) or (**not**), turns it into an ordinary function symbol. Symbols declared as **nullary** denote special constant symbols which simply stand for themselves. Technically, these are just ordinary identifiers; however, the **nullary** attribute tells the compiler that when such an identifier occurs on the left-hand side of an equation, it is to be interpreted as a constant rather than a variable (see above).

Lists and tuples: [x,y,z], x..y, x:xs, x,y,z

The necessary constructors to build lists and tuples are actually defined in the prelude: '[]' and '()' are the empty list and tuple, ':' produces list ''conses'', and ',' produces ''pairs''. As indicated, Pure provides the usual syntactic sugar for list values in brackets, such as [x,y,z], which is exactly the same as x:y:z:[]. Moreover, the prelude also provides an infix '..' operator to denote arithmetic sequences such as 1..10. Sequences with arbitrary stepsizes can be written by denoting the first two sequence elements using the ':' operator, as in 1.0:1.2..3.0.

Pure's tuples are a bit unusual: They are constructed by just ''pairing'' things using the ',' operator, for which the empty tuple acts as a neutral element (i.e., (),x is just x, as is x,()). Pairs always associate to the right, meaning that x,y,z == x,(y,z) == (x,y),z, where x,(y,z) is the normalized representation. This implies that tuples are always flat, i.e., there are no nested tuples (tuples of tuples); if you need such constructs then you should use lists instead. Also note that parentheses are generally only used to group expressions and are *not* part of the tuple syntax in Pure. There's one exception to this rule, however, namely that in order to include a tuple in a bracketed list you have to put it inside parentheses. E.g., [(1,2),3,(4,5)] is a three element list consisting of the tuple 1,2, the integer 3, and another tuple 4,5. Likewise, [(1,2,3)] is list with a single element, the tuple 1,2,3.

Matrices: {1.0,2.0,3.0}, {1,2;3,4}, {1L,y+1;foo,bar}

Pure also offers matrices, a kind of arrays, as a built-in data structure which provides efficient storage and element access. These work more or less like their Octave/MATLAB equivalents, but using curly braces instead of brackets. As indicated, commas are used to separate the columns of a matrix, semicolons for its rows. In fact, the {...} construct is rather general, allowing you to construct new matrices from individual elements and/or submatrices, provided that all dimensions match up. E.g., {{1;3},{2;4}} is another way to write a 2x2 matrix in ''column-major'' form (however, internally all matrices are stored in C's row-major format).

If the interpreter was built with support for the GNU Scientific Library (GSL) then both numeric and symbolic matrices are available. The former are thin wrappers around GSL's homogeneous arrays of double, complex double or (machine) int matrices, while the latter can contain any mixture of Pure expressions. Pure will pick the appropriate type for the data at hand. If a matrix contains values of different types, or Pure values which cannot be stored in a numeric matrix, then a symbolic matrix is created instead (this also includes the case of bigints, which are considered as symbolic values as far as matrix construction is concerned). If the interpreter was built without GSL support then symbolic matrices are the only kind of matrices supported by the interpreter.

More information about matrices and corresponding examples can be found in the EXAMPLES section below.

Comprehensions: [x,y | x=1..n; y=1..m; x<y], {i!=j | i=1..n; j=1..m}

Pure provides the usual comprehension syntax as a convenient means to construct both list and matrix values from a ''template'' expression and one or more ''generator'' and ''filter'' clauses (the former bind a pattern to values drawn from a list or matrix, the latter are just predicates determining which generated elements should actually be added to the result). Both list and matrix comprehensions are in fact syntactic sugar for a combination of nested lambdas, conditional expressions and ''catmaps'' (a collection of operations which combine list or matrix construction and mapping a function over a list or matrix, defined in the prelude), but they are often much easier to write.

Matrix comprehensions work pretty much like list comprehensions, but produce matrices instead of lists. Generator clauses in matrix comprehensions alternate between row and column generation so that most common mathematical abbreviations carry over quite easily. Examples of both kinds of comprehensions can be found in the EXAMPLES section below.

Function applications: foo x y z

As in other modern FPLs, these are written simply as juxtaposition (i.e., in ''curried'' form) and associate to the left. Operator applications are written using prefix, postfix or infix notation, as the declaration of the operator demands, but are just ordinary function applications in disguise. E.g., x+y is exactly the same as (+) x y.

Conditional expressions: if x **then** y **else** z

Evaluates to y or z depending on whether x is ''true'' (i.e., a nonzero integer). An exception is generated if the condition is not an integer.

Lambdas: \x -> y

These work pretty much like in Haskell. More than one variable may be bound (e.g, \x y -> x*y), which is equivalent to a nested lambda (\x -> \y -> x*y). Pure also fully supports pattern-matching lambda abstractions which match a pattern against the lambda argument and bind multiple lambda variables in one go, such as \(x,y) -> x*y.

Case expressions: case x **of** *rule*; ... **end**

Matches an expression, discriminating over a number of different cases, similar to the Haskell **case** construct. The expression x is matched in turn against each left-hand side pattern in the rule list, and the first pattern which matches x gives the value of the entire expression, by evaluating the corresponding right-hand side with the variables in the pattern bound to their corresponding values.

When expressions: x **when** *rule*; ... **end**

An alternative way to bind local variables by matching a collection of subject terms against corresponding patterns. Similar to Aardappel's **when** construct. A single binding such as x **when** u = v **end** is equivalent to **case** v **of** u = x **end**, but the former is often more convenient to write. In difference to Aardappel, Pure also allows multiple definitions in a single **when** clause, which are processed from left to right, so that later definitions may refer to the variables in earlier ones. In fact, a **when** expression with multiple definitions is treated like several nested **when** expressions, with the first binding being the ''outermost'' one.

With expressions: x **with** *rule*; ... **end**

Defines local functions. Like Haskell's **where** construct, but it can be used anywhere inside an expression (just like Aardappel's **where**, but Pure uses the keyword **with** which better lines up with **case** and **when**). Several functions can be defined in a single **with** clause, and the definitions may consist of as many equations as you want.

Operators and Precedence

Expressions are parsed according to the following precedence rules: Lambda binds most weakly, followed by **when**, **with** and **case**, followed by conditional expressions (**if-then-else**), followed by the *simple*

expressions, i.e., all other kinds of expressions involving operators, function applications, constants, symbols and other primary expressions. Precedence and associativity of operator symbols are given by their declarations (in the prelude or the user's program), and function application binds stronger than all operators. Parentheses can be used to override default precedences and associativities as usual.

The common operator symbols like +, -, *, / etc. are all declared at the beginning of the prelude, see the **prelude.pure** script for a list of these. Arithmetic, relational and logical operators usually follow C conventions. However, out of necessity some of Pure's operator symbols deviate from C. Most notably, the '!' symbol is Pure's indexing operator, hence logical negation is denoted **not** instead (named for Haskell compatibility, but Pure's **not** is a real prefix operator instead of an ordinary function symbol). Moreover, the bitwise operators are named **and** and **or** instead of '&' and '|', which are used for other purposes in Pure.

Special Forms

As already mentioned, some operators are actually implemented as special forms. In particular, the conditional expression **if** x **then** y **else** z is a special form with call-by-name arguments y and z; only one of the branches is actually evaluated, depending on the value of x. Similarly, the logical connectives && and || evaluate their operands in *short-circuit* mode just like in C. Thus, e.g., x&&y immediately becomes false if x evaluates to false, without ever evaluating y.

The *sequencing* operator $$ evaluates its left operand, immediately throws the result away and then goes on to evaluate the right operand which gives the result of the entire expression. This operator is useful to write imperative-style code such as the following prompt/input interaction:

```
> using system;
> puts "Enter a number:" $$ scanf "%g";
Enter a number:
21
21.0
```

The & operator does *lazy evaluation*. This is the only postfix operator defined in the standard prelude, written as 'x&', where x is an arbitrary Pure expression. The & operator binds stronger than any other operation except function application. It turns its operand into a kind of parameterless anonymous closure, deferring its evaluation. These kinds of objects are also commonly known as *thunks* or *futures*. When the value of a future is actually needed (during pattern-matching, or when the value becomes an argument of a C call), it is evaluated automagically and gets *memoized*, i.e., the computed result replaces the thunk so that it only has to be computed once. Futures are useful to implement all kinds of lazy data structures in Pure, in particular: lazy lists a.k.a. *streams*. A stream is simply a list with a thunked tail, which allows it to be infinite. The Pure prelude defines many functions for creating and manipulating these kinds of objects; further details and examples can be found in the EXAMPLES section below.

Toplevel

At the toplevel, a Pure program basically consists of rewriting rules (which are used to define functions and macros), constant and variable definitions, and expressions to be evaluated:

Rules: *lhs* = *rhs*;

These rules always combine a left-hand side *pattern* (which must be a simple expression) and a right-hand side (which can be any kind of Pure expression described above). In some cases, this basic form can also be augmented with a condition **if** *guard* tacked on to the end of the rule (which restricts the applicability of the rule to the case that the guard evaluates to a nonzero integer), or the keyword **otherwise** denoting an empty guard which is always true (this is nothing but syntactic sugar to point out the ''default'' case of a definition; the interpreter just treats this as a comment). Pure also provides some abbreviations for factoring out common left-hand or right-hand sides in collections of rules; see section RULE SYNTAX below for details.

Macro rules: def *lhs* = *rhs*;

A rule starting with the keyword **def** defines a *macro* function. No guards or multiple left-hand and right-hand sides are permitted here. Macro rules are used to preprocess expressions on the right-hand side of other definitions at compile time, and are typically employed to implement user-defined special forms and simple kinds of optimization rules. See the MACROS section below for

details and examples.

Global variable bindings: let *lhs = rhs*;
: Binds every variable in the left-hand side pattern to the corresponding subterm of the right-hand side (after evaluating it). This works like a **when** clause, but serves to bind *global* variables occurring free on the right-hand side of other function and variable definitions.

Constant bindings: const *lhs = rhs*;
: An alternative form of **let** which defines constants rather than variables. (These are not to be confused with **nullary** symbols which simply stand for themselves!) Like **let**, this construct binds the variable symbols on the left-hand side to the corresponding values on the right-hand side (after evaluation). The difference is that **const** symbols can only be defined once, after which their values are substituted directly into the right-hand sides of other definitions, rather than being looked up at runtime.

Toplevel expressions: *expr*;
: A singleton expression at the toplevel, terminated with a semicolon, simply causes the given value to be evaluated (and the result to be printed, when running in interactive mode).

Scoping Rules

A few remarks about the scope of identifiers and other symbols are in order here. Like most modern functional languages, Pure uses *lexical* or *static* binding for local functions and variables. What this means is that the binding of a local name is completely determined at compile time by the surrounding program text, and does not change as the program is being executed. In particular, if a function returns another (anonymous or local) function, the returned function captures the environment it was created in, i.e., it becomes a (lexical) *closure*. For instance, the following function, when invoked with a single argument x, returns another function which adds x to its argument:

```
> foo x = bar with bar y = x+y end;
> let f = foo 99; f;
#<closure bar>
> f 10, f 20;
109,119
```

This works the same no matter what other bindings of 'x' may be in effect when the closure is invoked:

```
> let x = 77; f 10, f 20 when x = 88 end;
109,119
```

Global bindings of variable and function symbols work a bit differently, though. Like many languages which are to be used interactively, Pure binds global symbols *dynamically*, so that the bindings can be changed easily at any time during an interactive session. This is mainly a convenience for interactive usage, but works the same no matter whether the source code is entered interactively or being read from a script, in order to ensure consistent behaviour between interactive and batch mode operation.

So, for instance, you can easily bind a global variable to a new value by just entering a corresponding **let** command:

```
> foo x = c*x;
> foo 99;
c*99
> let c = 2; foo 99;
198
> let c = 3; foo 99;
297
```

This works pretty much like global variables in imperative languages, but note that in Pure the value of a global variable can *only* be changed with a **let** command at the toplevel. Thus referential transparency is unimpaired; while the value of a global variable may change between different toplevel expressions, it will

always take the same value in a single evaluation.

Similarly, you can also add new equations to an existing function at any time:

```
> fact 0 = 1;
> fact n::int = n*fact (n-1) if n>0;
> fact 10;
3628800
> fact 10.0;
fact 10.0
> fact 1.0 = 1.0;
> fact n::double = n*fact (n-1) if n>1;
> fact 10.0;
3628800.0
> fact 10;
3628800
```

(In interactive mode, it is even possible to completely erase a definition, see section INTERACTIVE USAGE for details.)

So, while the meaning of a local symbol never changes once its definition has been processed, toplevel definitions may well evolve while the program is being processed, and the interpreter will always use the *latest* definitions at a given point in the source when an expression is evaluated. This means that, even in a script file, you have to define all symbols needed in an evaluation *before* entering the expression to be evaluated.

RULE SYNTAX

Basically, the same rule syntax is used in all kinds of global and local definitions. However, some constructs (specifically, **when**, **let**, **const** and **def**) use a restricted rule syntax where no guards or multiple left-hand and right-hand sides are permitted. When matching against a function or macro call, or the subject term in a **case** expression, the rules are always considered in the order in which they are written, and the first matching rule (whose guard evaluates to a nonzero value, if applicable) is picked. (Again, the **when** construct is treated differently, because each rule is actually a separate definition.)

In any case, the left-hand side pattern (which, as already mentioned, is always a simple expression) must not contain repeated variables (i.e., rules must be ''left-linear''), except for the anonymous variable '_' which matches an arbitrary value without binding a variable symbol.

A left-hand side variable (including the anonymous variable) may be followed by one of the special type tags **::int**, **::bigint**, **::double**, **::string**, **::matrix**, **::pointer**, to indicate that it can only match a constant value of the corresponding built-in type. (This is useful if you want to write rules matching *any* object of one of these types; note that there is no way to write out all ''constructors'' for the built-in types, as there are infinitely many.)

Pure also supports Haskell-style ''as'' patterns of the form *variable***@***pattern* which binds the given variable to the expression matched by the subpattern *pattern* (in addition to the variables bound by *pattern* itself). This is convenient if the value matched by the subpattern is to be used on the right-hand side of an equation. Syntactically, ''as'' patterns are primary expressions; if the subpattern is not a primary expression, it must be parenthesized. For instance, the following function duplicates the head element of a list:

```
foo xs@(x:_) = x:xs;
```

The left-hand side of a rule can be omitted if it is the same as for the previous rule. This provides a convenient means to write out a collection of equations for the same left-hand side which discriminates over different conditions:

lhs = *rhs* **if** *guard*;
= *rhs* **if** *guard*;
...
= *rhs* **otherwise**;

For instance:

```
fact n  = n*fact (n-1) if n>0;
        = 1 otherwise;
```

Pure also allows a collection of rules with different left-hand sides but the same right-hand side(s) to be abbreviated as follows:

```
lhs     |
        ...
lhs     = rhs;
```

This is useful if you need different specializations of the same rule which use different type tags on the left-hand side variables. For instance:

```
fact n::int    |
fact n::double |
fact n         = n*fact(n-1) if n>0;
               = 1 otherwise;
```

In fact, the left-hand sides don't have to be related at all, so that you can also write something like:

```
foo x | bar y = x*y;
```

However, this is most useful when using an ''as'' pattern to bind a common variable to a parameter value *after* checking that it matches one of several possible argument patterns (which is slightly more efficient than using an equivalent type-checking guard). E.g., the following definition binds the xs variable to the parameter of foo, if it is either the empty list or a list starting with an integer:

```
foo xs@[] | foo xs@(_::int:_) = ... xs ...;
```

The same construct also works in **case** expressions, which is convenient if different cases should be mapped to the same value, e.g.:

```
case ans of "y" | "Y" = 1; _ = 0; end;
```

EXAMPLES

Here are a few examples of simple Pure programs.

The factorial:

```
fact n  = n*fact (n-1) if n>0;
        = 1 otherwise;

let facts = map fact (1..10); facts;
```

The Fibonacci numbers:

```
fib n   = a  when a, b   = fibs n end
          with fibs n = 0, 1 if n<=0;
                       = case fibs (n-1) of
                           a, b = b, a+b;
                         end;
          end;

let fibs = map fib (1..30); fibs;
```

It is worth noting here that in most cases Pure performs tail call optimization so that tail-recursive definitions like the following will be executed in constant stack space (see the CAVEATS AND NOTES section for more details on this).

```
// tail-recursive factorial using an "accumulating parameter"
fact n = loop 1 n with
  loop p n = if n>0 then loop (p*n) (n-1) else p;
end;
```

Here is an example showing how constants are defined and used. Constant definitions take pretty much the same form as variable definitions with **let** (see above), but work more like the definition of a parameterless function whose value is precomputed at compile time:

```
> extern double atan(double);
> const pi = 4*atan 1.0;
> pi;
3.14159265358979
> foo x = 2*pi*x;
> show foo
foo x = 2*3.14159265358979*x;
> foo 1;
6.28318530717958
```

List Comprehensions

List comprehensions are Pure's main workhorse for generating and processing all kinds of list values. Here's a well-known example, Erathosthenes' classical prime sieve:

```
primes n        = sieve (2..n) with
  sieve []      = [];
  sieve (p:qs)  = p : sieve [q | q = qs; q mod p];
end;
```

For instance:

```
> primes 100;
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```

If you dare, you can actually have a look at the catmap-lambda-if-then-else expression the comprehension expanded to:

```
> show primes
primes n = sieve (2..n) with sieve [] = []; sieve (p:qs) = p:sieve
(catmap (\q -> if q mod p then [q] else []) qs) end;
```

List comprehensions are also a useful device to organize backtracking searches. For instance, here's an algorithm for the n queens problem, which returns the list of all placements of n queens on an n x n board (encoded as lists of n pairs (i,j) with i = 1..n), so that no two queens hold each other in check.

```
queens n        = search n 1 [] with
  search n i p = [reverse p] if i>n;
               = cat [search n (i+1) ((i,j):p) | j = 1..n; safe (i,j) p];
  safe (i,j) p = not any (check (i,j)) p;
  check (i1,j1) (i2,j2)
               = i1==i2 || j1==j2 || i1+j1==i2+j2 || i1-j1==i2-j2;
end;
```

Lazy Evaluation and Streams

As already mentioned, lists can also be evaluated in a ''lazy'' fashion, by just turning the tail of a list into a *future*. This special kind of list is also called a *stream*. Streams enable you to work with infinite lists (or finite lists which are so huge that you would never want to keep them in memory in their entirety). E.g., here's one way to define the infinite stream of all Fibonacci numbers:

```
> fibs = 0L : 1L : zipwith (+) fibs (tail fibs) &;
> fibs;
0L:1L:#<thunk 0xb5f875e8>
```

Note the '&' on the tail of the list. This turns 'fibs' into a stream, which is required to prevent 'fibs' from recursing into samadhi. Also note that we work with bigints in this example because the Fibonacci numbers grow quite rapidly, so with machine integers the values would soon start wrapping around to negative integers.

Streams like these can be worked with in pretty much the same way as with lists. Of course, care must be taken not to invoke ''eager'' operations such as '#' (which computes the size of a list) on infinite streams, to prevent infinite recursion. However, many list operations work with infinite streams just fine, and return the appropriate stream results. E.g., the 'take' function (which retrieves a given number of elements from the front of a list) works with streams just as well as with ''eager'' lists:

```
> take 10 fibs;
0L:1L:#<thunk 0xb5f87630>
```

Hmm, not much progress there, but that's just how streams work (or rather they don't, they're lazy bums indeed!). Nevertheless, the stream computed with 'take' is in fact finite and we can readily convert it to an ordinary list, forcing its evaluation:

```
> list (take 10 fibs);
[0L,1L,1L,2L,3L,5L,8L,13L,21L,34L]
```

(Conversely, you can also turn a list into a stream value with the 'stream' function. This is done less frequently, but becomes useful if you want to generate a stream from a finite list in a list comprehension.)

For interactive usage it's often convenient to define an eager variation of 'take' which combines 'take' and 'list'. Let's do this now, so that we can use this operation in the following examples.

```
> takel n xs = list (take n xs);
> takel 10 fibs;
[0L,1L,1L,2L,3L,5L,8L,13L,21L,34L]
```

Well, this naive definition of the Fibonacci stream works, but it's awfully slow. In fact, it takes exponential running time to determine the *n*th member of the sequence, because of the two recursive calls to 'fibs' on the right-hand side. This defect soon becomes rather annoying if we access larger members of the sequence. Just for fun, let's measure some evaluation times with the interactive **stats** command:

```
> stats
> fibs!25; fibs!26; fibs!27;
75025L
2.29s
121393L
3.75s
196418L
6.12s
> stats off
```

It's quite apparent that the ratios between successive running times are about the golden ratio (which is of course no accident!). So, assuming a fast computer which can produce the head element of a stream in just a nanosecond, a conservative estimate of the time needed to compute just the 128th Fibonacci number would already exceed the current age of the universe by some 29.6%, if done this way. It goes without saying that this kind of algorithm won't even pass muster in a freshman course.

So let's get back to the drawing board. One nice trick of the trade is to have the *value* of the Fibonacci stream refer to itself in its definition, rather than just the function generating it. For that we need a kind of ''recursive variable definition'' which Pure doesn't have. (Haskellers should note here that the values of

parameterless functions are never memoized in Pure, because that would wreak havoc on functions with side effects.) Fortunately, we can work around this quite easily by employing the so-called *fixed point combinator*, incidentally called **fix** in Pure and defined in the prelude as follows:

```
> show fix
fix f = y y with y x = f (x x&) end;
```

(Functional programming buffs will quickly recognize this as an implementation of the normal order fixed point combinator, decorated with '&' in the right place to make it work with eager evaluation. Aspiring novices may go read Wikipedia or a good book on the lambda calculus now.)

So here's how we can define a ''linear-time'' version of the Fibonacci stream. (Note that we also define the stream as a variable now, to take full advantage of memoization.)

```
> clear fibs
> let fibs = fix (\f -> 0L : 1L : zipwith (+) f (tail f) &);
> fibs;
0L:1L:#<thunk 0xb58d8ae0>
> takel 10 fibs;
[0L,1L,1L,2L,3L,5L,8L,13L,21L,34L]
> fibs;
0L:1L:1L:2L:3L:5L:8L:13L:21L:34L:#<thunk 0xb4ce5d30>
```

As you can see, the invocation of our 'takel' function forced the corresponding prefix of the 'fibs' stream to be computed. The result of the evaluation is memoized, so that this portion of the stream is now readily available in case we need to have another look at it later. By these means, possibly costly reevaluations are avoided, trading memory for execution speed.

Also, the trick we played with the fixed point combinator pays off, and in fact computing large Fibonacci numbers is a piece of cake now:

```
> stats
> fibs!199;
173402521172797813159685037284371942044301L
0.1s
> stats off
```

Let's take a look at some of the other convenience operations for generating stream values. The prelude defines infinite arithmetic sequences, using **inf** or **-inf** to denote an upper (or lower) infinite bound for the sequence, e.g.:

```
> let u = 1..inf; let v = -1.0:-1.2..-inf;
> takel 10 u; takel 10 v;
[1,2,3,4,5,6,7,8,9,10]
[-1.0,-1.2,-1.4,-1.6,-1.8,-2.0,-2.2,-2.4,-2.6,-2.8]
```

Other useful stream generator functions are 'iterate', 'repeat' and 'cycle', which have been adopted from Haskell. In fact, infinite arithmetic progressions are implemented in terms of 'iterate'. The 'repeat' function just repeats its argument, and 'cycle' cycles through the elements of the given list:

```
> takel 10 (repeat 1);
[1,1,1,1,1,1,1,1,1,1]
> takel 10 (cycle [0,1]);
[0,1,0,1,0,1,0,1,0,1]
```

Moreover, list comprehensions can draw values from streams and return the appropriate stream result:

```
> let rats = [m,n-m | n=2..inf; m=1..n-1; gcd m (n-m) == 1]; rats;
(1,1):#<thunk 0xb5fd08b8>
```

```
> takel 10 rats;
[(1,1),(1,2),(2,1),(1,3),(3,1),(1,4),(2,3),(3,2),(4,1),(1,5)]
```

Finally, let's rewrite our prime sieve so that it generates the infinite stream of *all* prime numbers:

```
all_primes      = sieve (2..inf) with
  sieve (p:qs)  = p : sieve [q | q = qs; q mod p] &;
end;
```

Note that we can omit the empty list case of 'sieve' here, since the sieve now never becomes empty. Example:

```
> let P = all_primes;
> takel 20 P;
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
> P!299;
1987
```

You can also just print the entire stream. This will run forever, so hit Ctrl-C when you get bored.

```
> using system;
> do (printf "%d\n") all_primes;
2
3
5
  ...
```

(Make sure that you really use the 'all_primes' function instead of the P variable to print the stream. Otherwise the stream stored in P will grow with the number of elements printed until memory is exhausted. Calling 'do' on a fresh instance of the stream of primes allows 'do' to get rid of each 'cons' cell after having printed the corresponding stream element.)

Matrix Computations

Pure offers a number of basic matrix operations, such as matrix construction, indexing, slicing, as well as getting the size and dimensions of a matrix (these are briefly described in the STANDARD LIBRARY section below). However, it does *not* supply built-in support for matrix arithmetic and other linear algebra algorithms. The idea is that these can and should be provided through separate libraries, such as a GSL interface (which will hopefully be available in the near future).

But Pure's facilities for matrix and list processing also make it easy to roll your own, if desired. First, the prelude provides matrix versions of the common list operations like map, fold, zip etc., which provide a way to implement common matrix operations. E.g., multiplying a matrix x with a scalar a amounts to mapping the function \x->a*x to x, which can be done as follows:

```
> a * x::matrix = map (\x->a*x) x if not matrixp a;
> 2*{1,2,3;4,5,6};
{2,4,6;8,10,12}
```

Likewise, matrix addition and other element-wise operations can be realized using zipwith, which combines corresponding elements of two matrices using a given binary function:

```
> x::matrix + y::matrix = zipwith (+) x y;
> {1,2,3;4,5,6}+{1,2,1;3,2,3};
{2,4,4;7,7,9}
```

Second, matrix comprehensions make it easy to express a variety of algorithms which would typically be implemented using 'for' loops in conventional programming languages. To illustrate the use of matrix comprehensions, here is how we can define an operation to create a square identity matrix of a given dimension:

```
> eye n = {i==j | i = 1..n; j = 1..n};
> eye 3;
{1,0,0;0,1,0;0,0,1}
```

Note that the i==j term is just a Pure idiom for the Kronecker symbol. Another point worth mentioning here is that the generator clauses of matrix comprehensions alternate between row and column generation automatically. (More precisely, the last generator, which varies most quickly, always yields a row, the next-to-last one a column of these row vectors, and so on.) This makes matrix comprehensions resemble customary mathematical notation very closely.

As a slightly more comprehensive example (no pun intended!), here is a definition of matrix multiplication in Pure. The building block here is the ''dot'' product of two vectors which can be defined as follows:

```
> dot x::matrix y::matrix = foldl (+) 0 [x!i*y!i | i=0..#x-1];
> dot {1,2,3} {1,0,1};
4
```

(For the sake of simplicity, this doesn't do much error checking; if the two vectors aren't the same size then you'll get an 'out_of_bounds' exception with the definition above.)

The general matrix product now boils down to a simple matrix comprehension which just computes the dot product of all rows of x with all columns of y (the rows and cols functions are prelude operations found in matrices.pure):

```
> x::matrix * y::matrix = {dot u v | u = rows x; v = cols y};
> {0,1;1,0;1,1}*{1,2,3;4,5,6};
{4,5,6;1,2,3;5,7,9}
```

Well, that was easy. So let's take a look at a more challenging example, Gaussian elimination, which can be used to solve systems of linear equations. The algorithm brings a matrix into ''row echelon'' form, a generalization of triangular matrices. The resulting system can then be solved quite easily using back substitution. Here is a Pure implementation of the algorithm:

```
gauss_elimination x::matrix = p,x
when n,m = dim x; p,_,x = foldl step (0..n-1,0,x) (0..m-1) end;

// One pivoting and elimination step in column j of the matrix:
step (p,i,x) j
= if max_x==0 then p,i,x
  else
    // updated row permutation and index:
    transp i max_i p, i+1,
    {// the top rows of the matrix remain unchanged:
     x!!(0..i-1,0..m-1);
     // the pivot row, divided by the pivot element:
     {x!(i,l)/x!(i,j)               | l=0..m-1};
     // subtract suitable multiples of the pivot row:
     {x!(k,l)-x!(k,j)*x!(i,l)/x!(i,j) | k=i+1..n-1; l=0..m-1}}
when
  n,m = dim x; max_i, max_x = pivot i (col x j);
  x = if max_x>0 then swap x i max_i else x;
end with
  pivot i x       = foldl max (0,0) [j,abs (x!j)|j=i..#x-1];
  max (i,x) (j,y) = if x<y then j,y else i,x;
end;
```

The real meat is in the pivoting and elimination step ('step' function) which is iterated over all columns of the input matrix. In each step, x is the current matrix, i the current row index, j the current column index,

and p keeps track of the current permutation of the row indices performed during pivoting. The algorithm returns the updated matrix x, row index i and row permutation p.

Please refer to any good textbook on numeric mathematics for a closer description of the algorithm. But here is a brief rundown of what happens in each elimination step: First we find the pivot element in column j of the matrix. (We're doing partial pivoting here, i.e., we only look for the element with the largest absolute value in column j, starting at row i. That's usually good enough to achieve numerical stability.) If the pivot is zero then we're done (the rest of the pivot column is already zeroed out). Otherwise, we bring it into the pivot position (swapping row i and the pivot row), divide the pivot row by the pivot, and subtract suitable multiples of the pivot row to eliminate the elements of the pivot column in all subsequent rows. Finally we update i and p accordingly and return the result.

In order to complete the implementation, we still need the following little helper functions to swap two rows of a matrix (this is used in the pivoting step) and to apply a transposition to a permutation (represented as a list):

```
swap x i j = x!!(transp i j (0..n-1),0..m-1) when n,m = dim x end;
transp i j p = [p!tr k | k=0..#p-1]
with tr k = if k==i then j else if k==j then i else k end;
```

Finally, let us define a convenient print representation of double matrices a la Octave (the meaning of the __show__ function is explained in the CAVEATS and NOTES section):

```
using system;
__show__ x::matrix
= strcat [printd j (x!(i,j))|i=0..n-1; j=0..m-1] + "\n"
with printd 0 = sprintf "\n%10.5f"; printd _ = sprintf "%10.5f" end
when n,m = dim x end if dmatrixp x;
```

Example:

```
> let x = dmatrix {2,1,-1,8; -3,-1,2,-11; -2,1,2,-3};
> x; gauss_elimination x;

  2.00000   1.00000  -1.00000   8.00000
 -3.00000  -1.00000   2.00000 -11.00000
 -2.00000   1.00000   2.00000  -3.00000

[1,2,0],
  1.00000   0.33333  -0.66667   3.66667
  0.00000   1.00000   0.40000   2.60000
  0.00000   0.00000   1.00000  -1.00000
```

MACROS

Macros are a special type of functions to be executed as a kind of ''preprocessing stage'' at compile time. In Pure these are typically used to define custom special forms, and to perform inlining of function calls and other simple kinds of source-level optimizations. In the following, these are also referred to as *convenience* and *optimization macros*, respectively.

Whereas the macro facilities of most programming languages simply provide a kind of textual substitution mechanism, Pure macros operate on symbolic expressions and are implemented by the same kind of rewriting rules that are also used to define ordinary functions in Pure. In difference to these, macro rules start out with the keyword **def**, and only simple kinds of rules without any guards or multiple left-hand and right-hand sides are permitted.

Syntactically, a macro definition looks just like a variable or constant definition, using **def** in lieu of **let** or **const**, but they are processed in a different way. Macros are substituted into the right-hand sides of function, constant and variable definitions. All macro substitution happens before constant substitutions and the actual compilation step. Macros can be defined in terms of other macros (also recursively), and will be

expanded using the leftmost-innermost reduction strategy (i.e., macro calls in macro arguments are expanded before the macro gets applied to its parameters).

Optimization Rules

Here is a simple example, showing a rule which expands saturated calls of the **succ** function (defined in the prelude) at compile time:

```
> def succ x = x+1;
> foo x::int = succ (succ x);
> show foo
foo x::int = x+1+1;
```

Rules like these can be useful to help the compiler generate better code. Note that a macro may have the same name as an ordinary Pure function, which is essential if you want to optimize calls to an existing function, as in the previous example.

A somewhat more practical example is the following rule from the prelude, which eliminates saturated instances of the right-associative function application operator:

```
def f $ x = f x;
```

Like in Haskell, this low-priority operator is handy to write cascading function calls. With the above macro rule, these will be ‘‘inlined’’ as ordinary function applications automagically. Example:

```
> foo x = bar $ bar $ 2*x;
> show foo
foo x = bar (bar (2*x));
```

Here is slightly more tricky rule from the prelude, which optimizes the case of ‘‘throwaway’’ list comprehensions. This is useful if a list comprehension is evaluated solely for its side effects.

```
def void (catmap f x) = do f x;
```

Note that the ‘void’ function simply throws away its argument and returns () instead. The ‘do’ function applies a function to every member of a list (like ‘map’), but throws away all intermediate results and just returns (), which is much more efficient if you don’t need those results anyway. These are both defined in the prelude.

Let’s see how this rule transforms a list comprehension if we ‘‘voidify’’ it:

```
> using system;
> f = [printf "%g\n" (2^x+1) | x=1..5; x mod 2];
> g = void [printf "%g\n" (2^x+1) | x=1..5; x mod 2];
> show f g
f = catmap (\x -> if x mod 2 then [printf "%g0 (2^x+1)] else []) (1..5);
g = do (\x -> if x mod 2 then [printf "%g0 (2^x+1)] else []) (1..5);
```

Ok, so the ‘catmap’ got replaced with a ‘do’ which is just what we need to make this code go essentially as fast as a ‘for’ loop in conventional programming languages (up to constant factors, of course). Here’s how it looks like when we run the ‘g’ function:

```
> g;
3
9
33
()
```

It’s not all roses, however, since the above macro rule will only get rid of the outermost ‘catmap’ if the list comprehension binds multiple variables:

```
> u = void [puts $ str (x,y) | x=1..2; y=1..3];
> show u
u = do (\x -> catmap (\y -> [puts (str (x,y))]) (1..3)) (1..2);
```

If you're bothered by this, you'll have to apply 'void' recursively, creating a nested list comprehension which expands to a nested 'do':

```
> v = void [void [puts $ str (x,y) | y=1..3] | x=1..2];
> show v
v = do (\x -> [do (\y -> [puts (str (x,y))]) (1..3)]) (1..2);
```

(It would be nice to have this handled automatically, but the left-hand side of a macro definition must be a simple expression, and thus it's not possible to write a macro which descends recursively into the lambda argument of 'catmap'.)

Recursive Macros

Macros can also be recursive, in which case they usually consist of multiple rules and make use of pattern-matching like ordinary function definitions. Example:

```
> def foo (bar x) = foo x+1;
> def foo x = x;
> baz = foo (bar (bar (bar x)));
> show baz
baz = x+1+1+1;
```

Note that, technically, Pure macros are just as powerful as (unconditional) term rewriting systems and thus they are Turing-complete. This implies that a badly written macro may well send the Pure compiler into an infinite recursion, which results in a stack overflow at compile time. See the CAVEATS AND NOTES section at the end of this manual for information on how to deal with these by setting the **PURE_STACK** environment variable.

Convenience Macros

The following 'timex' macro provides an example of how you can use macros to define your own special forms. This is made possible by the fact that the macro arguments will only be evaluated at runtime and can thus be passed to built-in special forms and other constructs which defer their evaluation. In fact, the right-hand side of a macro rule may be an arbitrary Pure expression involving conditional expressions, lambdas, binding clauses, etc. These are *not* evaluated during macro substitution, they just become part of the macro expansion (after substituting the macro parameters).

Our definition of 'timex' employs the system function 'clock' to report the cpu time in seconds needed to evaluate a given expression, along with the computed result:

```
> using system;
> def timex x = (clock-t0)/CLOCKS_PER_SEC,y when t0 = clock; y = x end;
> sum = foldl (+) 0L;
> timex $ sum (1L..100000L);
0.43,5000050000L
```

(Note that the above definition of 'timex' wouldn't work as an ordinary function definition, since by virtue of Pure's basic eager evaluation strategy the x parameter would have been evaluated already before it is passed to 'timex', making 'timex' always return a zero time value. Try it.)

Macro Hygiene

Pure macros are lexically scoped, i.e., symbols on the right-hand-side of a macro definition can never refer to anything outside the macro definition, and macro parameter substitution also takes into account binding constructs, such as **with** and **when** clauses, in the right-hand side of the definition. Macro facilities with these pleasant properties are also known as *hygienic* macros. They are not susceptible to so-called ''name capture,'' which makes macros in less sophisticated languages bug-ridden and hard to use.

Pure macros also have their limitations. Specifically, the left-hand side of a macro rule must be a simple

expression, just like in ordinary function definitions. This restricts the kinds of expressions which can be rewritten by a macro. But Pure macros are certainly powerful enough for most common preprocessing purposes, while still being robust and easy to use.

DECLARATIONS

Pure is a very terse language by design; you don't declare much stuff, you just define it and be done with it. Usually, all necessary information about the defined symbols is inferred automatically. However, there are a few toplevel constructs which let you declare special symbol attributes and manage programs consisting of several source modules. These are: **private**, fixity (operator) and **nullary** (constant symbol) declarations, **extern** declarations for external C functions (described in the C INTERFACE section), and **using** clauses which let you include other scripts in a Pure script.

Private symbol declarations: private *symbol ...*;

Declares the listed symbols as *private*. Pure programs usually consist of several source scripts (see the description of the **using** clauses below). By default, all global symbols are *public* symbols which are visible throughout the entire Pure program. Symbols explicitly declared as private are only visible in the script which declares them. This must be done before using these symbols. Example:

private foo bar;
foo (bar x) = x+1; // foo and bar symbols are private here

Note that to declare multiple symbols in a single declaration, you just list them all with whitespace in between. The same applies to the other types of symbol declarations discussed below.

Operator declarations: infix *level op ...*;

These may also be prefixed with the keyword **private** to indicate a private operator symbol (see above).

Ten different precedence levels are available for user-defined operators, numbered 0 (lowest) thru 9 (highest). On each precedence level, you can declare (in order of increasing precedence) **infix** (binary non-associative), **infixl** (binary left-associative), **infixr** (binary right-associative), **prefix** (unary prefix) and **postfix** (unary postfix) operators. For instance:

infixl 6 + - ;
infixl 7 * / div mod ;

One thing worth noting here is that unary minus plays a special role in the syntax. Like in Haskell, unary minus is the only prefix operator symbol which is also used as an infix operator, and it always has the same precedence as binary minus (whose precedence may be chosen freely in the prelude). Thus, with the standard prelude, -x+y will be parsed as (-x)+y, whereas -x*y is the same as -(x*y). Also note that the notation '(-)' always denotes the binary minus operator; the unary minus operation can be denoted using the built-in 'neg' function.

Constant symbol declarations: nullary *symbol ...*;

Constant symbols are introduced using a **nullary** declaration (again, a **private** prefix may be used to denote private constant symbols), e.g.:

nullary [] ();
private nullary nil;

As explained in the PURE OVERVIEW section, **nullary** symbols are like ordinary identifiers, but are treated as constants rather than variables when they occur on the left-hand side of an equation.

Examples for all types of symbol declarations can be found in the prelude which declares a bunch of standard (arithmetic, relational, logical) operator symbols as well as the list and pair constructors ':' and ',' and the empty list and tuple constants '[]' and '()'.

Using clause: using *name*, ...;

Causes each given script to be included in the Pure program. Each included script is loaded only *once*, when the first **using** clause for the script is encountered. This kind of clause is discussed in further detail below.

Note that the **using** clause also has an alternative form which allows dynamic libraries to be loaded, this will be discussed in the C INTERFACE section.

The 'using' Declaration

The **using** declaration provides a simple but effective way to assemble a Pure program from several source modules. The Pure program is just the concatenation of all the source modules listed as command line arguments and included through **using** clauses. Public constants, variables, functions and macros defined anywhere in the program share one big happy namespace and are available throughout the entire program (not just the module that contains a **using** clause for the script containing a given symbol). This approach has its drawbacks, but it makes it easy to define polymorphic functions and macros across separate modules.

To facilitate modular development, each script also has a separate namespace for private symbols which are only visible in the script which declares them (see the explanation of the **private** declaration above). This makes it possible to hide away internal operations, prevent name clashes between symbols of different modules, and keep the public namespace tidy and clean. A **private** declaration shadows a public symbol with the same print name, but this takes effect only *after* the **private** declaration of the symbol, so you can easily define yourself an alias for the public symbol before that, e.g.:

```
public_foo = foo;
private foo;
foo x = public_foo (bar x);
```

The script name in a **using** clause can be specified either as a string denoting the proper filename (possibly including path and/or filename extension), or as an identifier. In the latter case, the **.pure** filename extension is added automatically. In both cases, the interpreter performs a search to locate the script, unless an absolute pathname was given. It first searches the directory of the script containing the **using** clause (or the current working directory if the clause was read from standard input, as is the case, e.g., in an interactive session), then the directories named in **-I** options on the command line (in the given order), then the colon-separated list of directories in the **PURE_INCLUDE** environment variable, and finally the directory named by the **PURELIB** environment variable. Note that the current working directory is *not* searched by default (unless the **using** clause is read from standard input), but of course you can force this by adding the option **-I.** to the command line, or by including '.' in the **PURE_INCLUDE** variable.

For the purpose of comparing script names, the interpreter always uses the canonicalized full pathname of the script, following symbolic links to the destination file (albeit only one level). Thus different scripts with the same basename, such as **foo/utils.pure** and **bar/utils.pure** can both be included in the same program (unless they link to the same file). The resolution of symbolic links also makes it possible to have an executable script with a shebang line in its own directory, to be executed via a symbolic link placed on the system **PATH**. In this case the script search performed in **using** clauses will use the real script directory and thus other required scripts can be located in the script directory. This is the recommended practice for installing standalone Pure applications in source form which are to be run directly from the shell.

EXCEPTION HANDLING

Pure also offers a useful exception handling facility. To raise an exception, you just invoke the built-in function **throw** with the value to be thrown as the argument. To catch an exception, you use the built-in special form **catch** with the exception handler (a function to be applied to the exception value) as the first and the expression to be evaluated as the second (call-by-name) argument. For instance:

```
> catch error (throw hello_world);
error hello_world
```

Exceptions are also generated by the runtime system if the program runs out of stack space, when a guard does not evaluate to a truth value, and when the subject term fails to match the pattern in a pattern-matching

lambda abstraction, or a **let**, **case** or **when** construct. These types of exceptions are reported using the symbols **stack_fault**, **failed_cond** and **failed_match**, respectively, which are declared as constant symbols in the standard prelude. You can use **catch** to handle these kinds of exceptions just like any other. For instance:

```
> fact n = if n>0 then n*fact(n-1) else 1;
> catch error (fact foo);
error failed_cond
> catch error (fact 100000);
error stack_fault
```

(You'll only get the latter kind of exception if the interpreter does stack checks, see the discussion of the **PURE_STACK** environment variable in the CAVEATS AND NOTES section.)

Note that unhandled exceptions are reported by the interpreter with a corresponding error message:

```
> fact foo;
<stdin>:2.0-7: unhandled exception 'failed_cond' while evaluating 'fact foo'
```

Exceptions also provide a way to handle asynchronous signals. Most standard termination signals (SIGINT, SIGTERM, etc.) are set up during startup of the interpreter to produce corresponding Pure exceptions of the form **signal SIG** where **SIG** is the signal number. Pure's system module provides symbolic constants for common POSIX signals and also defines the operation **trap** which lets you rebind any signal to a **signal** exception. For instance, the following lets you handle the SIGQUIT signal:

```
> using system;
> trap SIG_TRAP SIGQUIT;
```

You can also use **trap** to just ignore a signal or revert to the system's default handler (which might take different actions depending on the type of signal, see **signal**(7) for details):

```
> trap SIG_IGN SIGQUIT; // signal is ignored
> trap SIG_DFL SIGQUIT; // reinstalls the default signal handler
```

Last but not least, exceptions can also be used to implement non-local value returns. For instance, here's a variation of our n queens algorithm which only returns the first solution. Note the use of **throw** in the recursive search routine to bail out with a solution as soon as we found one. The value thrown there is caught in the main routine. Also note the use of 'void' in the second equation of 'search'. This effectively turns the list comprehension into a simple loop which suppresses the normal list result and just returns () instead. Thus, if no value gets thrown then the function regularly returns with () to indicate that there is no solution.

```
queens1 n      = catch reverse (search n 1 []) with
  search n i p = throw p if i>n;
               = void [search n (i+1) ((i,j):p) | j = 1..n; safe (i,j) p];
  safe (i,j) p = not any (check (i,j)) p;
  check (i1,j1) (i2,j2)
               = i1==i2 || j1==j2 || i1+j1==i2+j2 || i1-j1==i2-j2;
end;
```

E.g., let's compute a solution for a standard 8x8 board:

```
> queens 8;
[(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)]
```

C INTERFACE

Accessing C functions from Pure programs is dead simple. You just need an **extern** declaration of the function, which is a simplified kind of C prototype. The function can then be called in Pure just like any other. For instance, the following commands, entered interactively in the interpreter, let you use the **sin** function from the C library (of course you could just as well put the **extern** declaration into a script):

```
> extern double sin(double);
> sin 0.3;
0.29552020666134
```

Multiple prototypes can be given in one **extern** declaration, separating them with commas:

```
extern double sin(double), double cos(double), double tan(double);
```

For clarity, the parameter types can also be annotated with parameter names, e.g.:

```
extern double sin(double x);
```

Parameter names in prototypes only serve informational purposes and are for the human reader; they are effectively treated as comments by the compiler.

The interpreter makes sure that the parameters in a call match; if not, the call is treated as a normal form expression by default, which gives you the opportunity to extend the external function with your own Pure equations (see below). The range of supported C types is a bit limited right now (void, bool, char, short, int, long, float, double, as well as arbitrary pointer types, i.e.: void*, char*, etc.), but in practice these should cover most kinds of calls that need to be done when interfacing to C libraries.

Single precision float arguments and return values are converted from/to Pure's double precision floating point numbers automatically.

A variety of C integer types (char, short, int, long) are provided which are converted from/to the available Pure integer types in a straightforward way. One important thing to note here is that the 'long' type *always* denotes 64 bit integers, even if the corresponding C type is actually 32 bit (as it usually is on most contemporary systems). All integer parameters take both Pure ints and bigints as actual arguments; truncation or sign extension is performed as needed, so that the C interface behaves as if the argument was ''cast'' to the C target type. Returned integers use the smallest Pure type capable of holding the result (i.e., int for the C char, short and int types, bigint for long a.k.a. 64 bit integers).

Pure considers all integers as signed quantities, but it is possible to pass unsigned integers as well (if necessary, you can use a bigint to pass positive values which are too big to fit into a machine int). Also note that when an unsigned integer is returned by a C routine which is too big to fit into the corresponding signed integer type, it will ''wrap around'' and become negative. In this case, depending on the target type, you can use the ubyte, ushort, uint and ulong functions provided by the prelude to convert the result back to an unsigned quantity.

Concerning the pointer types, char* is for string arguments and return values which need translation between Pure's internal utf-8 representation and the system encoding, while void* is for any generic kind of pointer (including strings, which are *not* translated when passed/returned as void*). Any other kind of pointer (except expr* and the GSL matrix pointer types, which are discussed below) is effectively treated as void* right now, although in a future version the interpreter may keep track of the type names for the purpose of checking parameter types.

The expr* pointer type is special; it indicates a Pure expression parameter or return value which is just passed through unchanged. All other types of values have to be ''unboxed'' when they are passed as arguments (i.e., from Pure to C) and ''boxed'' again when they are returned as function results (from C to Pure). All of this is handled by the runtime system in a transparent way, of course.

The matrix pointer types dmatrix*, cmatrix* and imatrix* can be used to pass double, complex double and int matrices to GSL functions taking pointers to the corresponding GSL types (gsl_matrix, gsl_matrix_complex and gsl_matrix_int) as arguments or returning them as results. Note that there is no marshalling of Pure's symbolic matrix type, as these aren't supported by GSL anyway. Also note that matrices are always passed by reference. If you need to pass a matrix as an output parameter of a GSL matrix routine, you can either create a zero matrix or a copy of an existing matrix. The prelude provides various operations for that purpose (in particular, see the dmatrix, cmatrix, imatrix and pack functions in matrices.pure). For instance, here is how you can quickly wrap up GSL's double matrix addition function in a

way that preserves value semantics:

```
> extern int gsl_matrix_add(dmatrix*, dmatrix*);
> x::matrix + y::matrix = gsl_matrix_add x y $$ x when x = pack x end;
> let x = dmatrix {1,2,3}; let y = dmatrix {2,3,2}; x; y; x+y;
{1.0,2.0,3.0}
{2.0,3.0,2.0}
{3.0,5.0,5.0}
```

Most GSL matrix routines can be wrapped in this fashion quite easily. A ready-made GSL interface providing access to all of GSL's numeric functions will be provided in the future.

As already mentioned, it is possible to augment an external C function with ordinary Pure equations, but in this case you have to make sure that the **extern** declaration of the function comes first. For instance, we might want to extend our imported **sin** function with a rule to handle integers:

```
> extern double sin(double);
> sin 0.3;
0.29552020666134
> sin 0;
sin 0
> sin x::int = sin (double x);
> sin 0;
0.0
```

Sometimes it is preferable to replace a C function with a wrapper function written in Pure. In such a case you can specify an *alias* under which the original C function is known to the Pure program, so that you can still call the C function from the wrapper. An alias is introduced by terminating the **extern** declaration with a clause of the form ''= *alias*''. For instance:

```
> extern double sin(double) = c_sin;
> sin x::double = c_sin x;
> sin x::int = c_sin (double x);
> sin 0.3; sin 0;
0.29552020666134
0.0
```

External C functions are resolved by the LLVM runtime, which first looks for the symbol in the C library and Pure's runtime library (or the interpreter executable, if the interpreter was linked statically). Thus all C library and Pure runtime functions are readily available in Pure programs. Other functions can be provided by adding them to the runtime, or by linking them statically into the runtime or the interpreter executable. Better yet, you can just ''dlopen'' shared libraries at runtime with a special form of the **using** clause:

using "lib:*libname*[.*ext*]";

For instance, if you want to call the GMP functions directly from Pure:

```
using "lib:libgmp";
```

After this declaration the GMP functions will be ready to be imported into your Pure program by means of corresponding **extern** declarations.

Shared libraries opened with **using** clauses are searched for in the same way as source scripts (see section DECLARATIONS above), using the **-L** option and the **PURE_LIBRARY** environment variable in place of **-I** and **PURE_INCLUDE**. If the library isn't found by these means, the interpreter will also consider other platform-specific locations searched by the dynamic linker, such as the system library directories and **LD_LIBRARY_PATH** on Linux. The necessary filename suffix (e.g., **.so** on Linux or **.dll** on Windows)

will be supplied automatically when needed. Of course you can also specify a full pathname for the library if you prefer that. If a library file cannot be found, or if an **extern** declaration names a function symbol which cannot be resolved, an appropriate error message is printed.

STANDARD LIBRARY

Pure comes with a collection of Pure library modules, which includes the standard prelude (loaded automatically at startup time) and some other modules which can be loaded explicitly with a **using** clause. The prelude offers the necessary functions to work with the built-in types (including arithmetic and logical operations) and to do most kind of list processing you can find in ML- and Haskell-like languages. It also provides a collection of basic string and matrix operations. Please refer to the **prelude.pure** file (as well as the modules included there, specifically **primitives.pure**, **matrices.pure** and **strings.pure**) for details on the provided operations. Here is a very brief summary of some of the prelude operations which, besides the usual arithmetic and logical operators, are probably used most frequently:

x+y This is also used to denote list concatenation.

x:y This is the list-consing operation. x becomes the head of the list, y its tail.

x..y Constructs arithmetic sequences. x:y..z can be used to denote sequences with arbitrary stepsize y-x. Infinite sequences can be constructed using an infinite bound (i.e., inf or -inf). E.g., 1:3..inf denotes the stream of all positive odd (machine) integers.

#x The size (number of elements) of the list, tuple or matrix x. In addition, dim x yields the dimensions (number of rows and columns) of a matrix.

x' The transpose of a matrix.

x!y This is the list, tuple and matrix indexing operation. Note that all indices in Pure are zero-based, thus x!0 and x!(#x-1) are the first and last element of a list, tuple or matrix, respectively. In the case of matrices, the subscript may also be a pair of row and column indices, such as x!(1,2).

x!!ys This is Pure's list, tuple and matrix ''slicing'' operation, which returns the list, tuple or matrix of all x!y while y runs through the (list or matrix) ys. Thus, e.g., x!!(i..j) returns all the elements between i and j (inclusive). Indices which fall outside the valid index range are quietly discarded. In fact, the index range ys may contain any number of indices (also duplicates), in any order. Thus x![0|i=1..n] returns the first element of x n times, and, if ys is a permutation of the range 0..#x-1, then x!!ys yields the corresponding permutation of the elements of x. In the case of matrices the index range may also contain two-dimensional subscripts, or the index range itself may be specified as a pair of row/column index lists such as x!!(i..j,k..l).

The prelude also offers support operations for the implementation of list and matrix comprehensions, as well as the customary list operations like head, tail, drop, take, filter, map, foldl, foldr, scanl, scanr, zip, unzip, etc., which make list programming so much fun in modern FPLs. In Pure, these also work on strings as well as matrices, although, for reasons of efficiency, these data structures are internally represented as different kinds of array data structures.

Besides the prelude, Pure's standard library also comprises a growing number of additional library modules which we can only mention in passing here. In particular, the **math.pure** module provides additional mathematical functions as well as Pure's complex and rational number data types. Common container data structures like sets and dictionaries are implemented in the **set.pure** and **dict.pure** modules, among others. Moreover, the (beginnings of a) system interface can be found in the **system.pure** module. In particular, this module also provides operations to do basic C-style I/O, including printf and scanf. More stuff will likely be provided in future releases.

INTERACTIVE USAGE

In interactive mode, the interpreter reads definitions and expressions and processes them as usual. If the **-i** option was used to force interactive mode when invoking the interpreter, the last script specified on the command line determines the visible namespace, i.e., the private symbols of that script are available in addition to the public symbols defined in the prelude and the other scripts specified on the command line. Additional scripts can be loaded interactively using either a **using** declaration or the interactive **run** command (see the description of the **run** command below for the differences between these). Or you can just

start typing away, entering your own definitions and expressions to be evaluated.

The input language is just the same as for source scripts, and hence individual definitions and expressions *must* be terminated with a semicolon before they are processed. For instance, here is a simple interaction which defines the factorial and then uses that definition in some evaluations. Input lines begin with ''>'', which is the interpreter's default command prompt:

```
> fact 1 = 1;
> fact n = n*fact (n-1) if n>1;
> let x = fact 10; x;
3628800
> map fact (1..10);
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

As indicated, in interactive mode the normal forms of toplevel expressions are printed after each expression is entered. We also call this the *read-eval-print* loop. Normal form expressions are usually printed in the same form as you'd enter them. However, there are a few special kinds of objects like closures (anonymous and local functions), thunks (''lazy'' values to be evaluated when needed) and pointers which don't have a textual representation in the Pure syntax and will be printed in the format #<*object description*> by default. It is also possible to override the print representation of any kind of expression by means of the **__show__** function, see the CAVEATS AND NOTES section for details.

When running interactively, the interpreter also accepts a number of special commands useful for interactive purposes. Here is a quick rundown of the currently supported operations:

! *command*
Shell escape.

cd *dir* Change the current working dir.

clear [*symbol* ...]
Purge the definitions of the given symbols (functions, macros, constants or global variables). If no symbols are given, purge *all* definitions (after confirmation) made after the most recent **save** command, or the beginning of the interactive session. (It might be a good idea to first check your current definitions with **show -t** or save them with **dump** before you do this, though.) See the DEFINITION LEVELS section below for details.

dump [*option* ...] [*symbol* ...]
Dump a snapshot of the current function, macro, constant and variable definitions in Pure syntax to a text file. This works similar to the **show** command (see below and the SHOW COMMAND section), but writes the definitions to a file.

This command only supports a subset of the **show** options, type **dump -h** for a description of these. Also note that by default the **dump** command only writes interactive definitions to the output file, which is equivalent to the **-t1** option of the **show** command. Presumably this is the most common usage, but using the **-t** option, you can select any definitions level just as with **show**. In particular, **dump -t** saves only the definitions made after the most recent **save** command, and **dump -t0** can be used to dump the *entire* program (including the definitions of the prelude), which can be useful for debugging purposes.

The default output file is **.pure** in the current directory, which is then reloaded automatically the next time the interpreter starts up in interactive mode in the same directory. This provides a quick-and-dirty means to save an interactive session and have it restored later. Please note that this isn't perfect; in order to properly handle **extern** and **using** declarations and other special cases such as thunks and pointers stored in variables, you'll probably have to prepare a corresponding **.purerc** file yourself, see ''Startup Files'' below.

A different filename can be specified with the **-F** option. You can then edit that file and use it as a starting point for an ordinary script or a **.purerc** file, or you can just run the file with the **run**

command (see below) to restore the definitions in a subsequent interpreter session.

You can also specify a subset of symbols to be saved. Shell glob patterns can be used if the **-g** option is given. Options may be combined; e.g., **dump -Ffg foo.pure foo*** is just the same as **dump -F foo.pure -f -g foo***, and dumps all functions whose names start with 'foo' to the file 'foo.pure'.

help [*args*]
Display the **pure**(1) manpage, or invoke **man**(1) with the given arguments.

ls [*args*]
List files (shell **ls**(1) command).

override
Enter ''override'' mode. This allows you to add equations ''above'' existing definitions in the source script, possibly overriding existing equations. See the DEFINITION LEVELS section below for details.

pwd Print the current working dir (shell **pwd**(1) command).

quit Exits the interpreter.

run *script*
Loads the given script file and adds its definitions to the current environment. This works more or less like a **using** clause, but only searches for the script in the current directory and loads the script ''anonymously.'' That is, **run** puts the definitions into the current namespace, giving you access to all private symbols of the script. Also, the definitions are placed at the current temporary level, so that **clear** can be used to remove them again. In particular, this makes it possible to quickly reload a script without exiting the interpreter, by issuing the **clear** command followed by **run**. (This works best if you start out from a clean environment, with no scripts loaded on the command line.)

save Begin a new level of temporary definitions. A subsequent **clear** command (see above) will purge all definitions made after the most recent **save** (or the beginning of the interactive session). See the DEFINITION LEVELS section below for details.

show [*option* ...] [*symbol* ...]
Show the definitions of symbols in various formats. See the SHOW COMMAND section below for details.

stats [on|off]
Enables (default) or disables ''stats'' mode, in which various statistics are printed after an expression has been evaluated. Currently, this just prints the cpu time in seconds for each evaluation, but in the future additional profiling information may be provided.

underride
Exits ''override'' mode. This returns you to the normal mode of operation, where new equations are added 'below'' previous rules of an existing function. See the DEFINITION LEVELS section below for details.

Note that these special commands are only recognized at the beginning of the interactive command line (they are not reserved keywords of the Pure language). Thus it's possible to ''escape'' identifiers looking like commands by entering a space at the beginning of the line. However, the compiler also warns you about identifiers which might be mistaken as command names, so that you can avoid this kind of problem.

Some commands which are especially important for effective operation of the interpreter are discussed in more detail in the following sections.

Startup Files

In interactive mode, the interpreter also runs some additional scripts at startup, after loading the prelude and the scripts specified on the command line.

The interpreter first looks for a **.purerc** file in the user's home directory (as given by the **HOME** environment variable) and then for a **.purerc** file in the current working directory. These are just ordinary Pure

scripts which may contain any additional definitions that you need. The **.purerc** file in the home directory is for global definitions which should always be available when running interactively, while the **.purerc** file in the current directory can be used for project-specific definitions.

Finally, you can also have a **.pure** initialization file in the current directory, which is created by the **dump** command (see above) and is loaded after the **.purerc** files if it is present.

The interpreter processes all these files in the same way as with the **run** command (see above). When invoking the interpreter, you can specify the **--norc** option on the command line if you wish to skip these initializations.

SHOW COMMAND

In interactive mode, the **show** command can be used to obtain information about defined symbols in various formats. This command recognizes the following options. Options may be combined, thus, e.g., **show** -tvl is the same as **show** -t -v -l.

-a Disassembles pattern matching automata. Works like the **-v4** option of the interpreter.

-c Print information about defined constants.

-d Disassembles LLVM IR, showing the generated LLVM assembler code of a function. Works like the **-v8** option of the interpreter.

-e Annotate printed definitions with lexical environment information (de Bruijn indices, subterm paths). Works like the **-v2** option of the interpreter.

-f Print information about defined functions.

-g Indicates that the following symbols are actually shell glob patterns and that all matching symbols should be listed.

-h Print a short help message.

-l Long format, prints definitions along with the summary symbol information. This implies **-s**.

-m Print information about defined macros.

-p[*flag*] List only private symbols in the current module if *flag* is nonzero (the default), otherwise (*flag* is zero) list only public symbols of all modules. List both private and public symbols if -p is omitted. The *flag* parameter, if given, must immediately follow the option character.

-s Summary format, print just summary information about listed symbols.

-t[*level*]
List only ‘‘temporary’’ symbols and definitions at the given *level* (the current level by default) or above. The *level* parameter, if given, must immediately follow the option character. A *level* of 1 denotes all temporary definitions, whereas 0 indicates *all* definitions (which is the default if **-t** is not specified). See the DEFINITION LEVELS section below for information about the notion of temporary definition levels.

-v Print information about defined variables.

If none of the **-c**, **-f**, **-m** and **-v** options are specified, then all kinds of symbols (constants, functions, macros and variables) are printed, otherwise only the specified categories will be listed.

Note that some of the options (in particular, **-a** and **-d**) may produce excessive amounts of information. By setting the **PURE_MORE** environment variable accordingly, you can specify a shell command to be used for paging, usually **more**(1) or **less**(1).

For instance, to list all definitions in all loaded scripts (including the prelude), simply say:

> **show**

This may produce quite a lot of output, depending on which scripts are loaded. The following command will only show summary information about the variable symbols along with their current values (using the ‘‘long format’’):

```
> show -lv
argc     var  argc = 0;
argv     var  argv = [];
sysinfo  var  sysinfo = "i686-pc-linux-gnu";
version  var  version = "0.8";
4 variables
```

If you're like me then you'll frequently have to look up how some operations are defined. No sweat, with the Pure interpreter there's no need to dive into the sources, the **show** command can easily do it for you. For instance, here's how you can list the definitions of all ''fold-left'' operations from the prelude in one go:

```
> show -g foldl*
foldl f a x::matrix = foldl f a (list x);
foldl f a s::string = foldl f a (chars s);
foldl f a [] = a;
foldl f a (x:xs) = foldl f (f a x) xs;
foldl1 f x::matrix = foldl1 f (list x);
foldl1 f s::string = foldl1 f (chars s);
foldl1 f (x:xs) = foldl f x xs;
```

DEFINITION LEVELS

To help with incremental development, the interpreter also offers some facilities to manipulate the current set of definitions interactively. To these ends, definitions are organized into different subsets called *levels*. The prelude, as well as other source programs specified when invoking the interpreter, are always at level 0, while the interactive environment starts at level 1.

Each **save** command introduces a new temporary level, and each subsequent **clear** command ''pops'' the definitions on the current level (including any definitions read using the **run** command) and returns you to the previous one (if any). This gives you a ''stack'' of up to 255 temporary environments which enables you to ''plug and play'' in a (more or less) safe fashion, without affecting the rest of your program. Example:

```
> foo (x:xs) = x+foo xs;
> foo [] = 0;
> show -t
foo (x:xs) = x+foo xs;
foo [] = 0;
> foo (1..10);
55
> clear
This will clear all temporary definitions at level #1. Continue (y/n)? y
> show foo
> foo (1..10);
foo [1,2,3,4,5,6,7,8,9,10]
```

(Please note that the **clear** command only works in this way when invoked without arguments. Otherwise the symbols given as arguments will be purged unconditionally, at all levels.)

We've seen already that normally, if you enter a sequence of equations, they will be recorded in the order in which they were written. However, it is also possible to override definitions in lower levels with the **override** command:

```
> foo (x:xs) = x+foo xs;
> foo [] = 0;
> show foo
foo (x:xs) = x+foo xs;
foo [] = 0;
> foo (1..10);
```

```
55
> save
save: now at temporary definitions level #2
> override
> foo (x:xs) = x*foo xs;
> show foo
foo (x:xs) = x*foo xs;
foo (x:xs) = x+foo xs;
foo [] = 0;
> foo (1..10);
0
```

Note that the equation ‘foo (x:xs) = x*foo xs;’ was inserted before the previous ‘foo (x:xs) = x+foo xs;’ rule, which is at level #1.

Even in override mode, new definitions will be added *after* other definitions at the *current* level. This allows us to just continue adding more high-priority definitions overriding lower-priority ones:

```
> foo [] = 1;
> show foo
foo (x:xs) = x*foo xs;
foo [] = 1;
foo (x:xs) = x+foo xs;
foo [] = 0;
> foo (1..10);
3628800
```

Again, the new equation was inserted *above* the existing lower-priority rules, but *below* our previous ‘foo (x:xs) = x*foo xs;’ equation entered at the same level. As you can see, we have now effectively replaced our original definition of ‘foo’ with a version that calculates list products instead of sums, but of course we can easily go back one level to restore the previous definition:

```
> clear
This will clear all temporary definitions at level #2. Continue (y/n)? y
clear: now at temporary definitions level #1
clear: override mode is on
> show foo
foo (x:xs) = x+foo xs;
foo [] = 0;
> foo (1..10);
55
```

Note that **clear** reminded us that override mode is still enabled (**save** will do the same if override mode is on while pushing a new definitions level). To turn it off again, use the **underride** command. This will revert to the normal behaviour of adding new equations below existing ones:

```
> underride
```

CAVEATS AND NOTES

This section is a grab bag of casual remarks, useful tips and tricks, and information on common pitfalls, quirks and limitations of the current implementation and how to deal with them.

Purity

People keep asking me what’s so ‘‘pure’’ about Pure. The long and apologetic answer is that at its core, Pure is in fact purely algebraic and purely functional. Pure doesn’t get in your way if you want to call external operations with side effects (it does allow you to call any C function after all), but with a few exceptions the standard library operations are free of those. Just stay away from operations marked ‘‘IMPURE’’ in the library sources (most notably, eval, catch/throw, references, sentries and direct pointer manipulations) and

avoid the system module, then your program will behave according to the semantics of term rewriting.

The short answer is that I simply liked the name, and there wasn't any programming language named ''Pure'' yet (quite a feat nowadays), so there's one now. :)

Backward Compatibility

Pure 0.7 introduced built-in matrix structures, which called for some minor changes in the syntax of comprehensions and arithmetic sequences. Specifically, the template expression and generator/filter clauses of a comprehension are now separated with '|'. (For the time being, the old [x; ...] list comprehension syntax is still supported, but the compiler will warn you about such constructs and flag them as deprecated.) Moreover, arithmetic sequences with arbitrary stepsize are now written x:y..z instead of x,y..z, and the '..' operator now has a higher precedence than the ',' operator. This makes writing matrix slices like x!!(i..j,k..l) much more convenient.

Debugging

There's no symbolic debugger yet. So **printf**(3) (available in the **system** standard library module) should be your friend. ;-)

The __show__ Function

As of Pure 0.6, the interpreter provides a ''hook'' to override the print representations of expressions at runtime by means of the **__show__** function, which works in a fashion similar to Haskell's show function. This feature is still a bit experimental, but seems to work reasonably well for the purposes for which it is intended.

__show__ is just an ordinary Pure function expected to return a string with the desired custom representation of a normal form value given as the function's single argument. This function is not defined by default, so you are free to add any rules that you want. The interpreter prints the strings returned by __show__ just as they are. It will *not* check whether they conform to Pure syntax and/or semantics, or modify them in any way.

Custom print representations are most useful for interactive purposes, if you're not happy with the default print syntax of some kinds of objects. One particularly useful application of __show__ is to change the format of numeric values. Here are some examples:

```
> using system;
> __show__ x::double = sprintf "%0.6f" x;
> 1/7;
0.142857
> __show__ x::int = sprintf "0x%0x" x;
> 1786;
0x6fa
> using math;
> __show__ (x::double+:y::double) = sprintf "%0.6f+%0.6fi" (x,y);
> cis (-pi/2);
0.000000+-1.000000i
```

The prelude function **str**, which returns the print representation of any Pure expression, calls __show__ as well:

```
> str (1/7);
"0.142857"
```

Conversely, you can call the str function from __show__, but in this case it always returns the default representation of an expression. This prevents the expression printer from going recursive, and allows you to define your custom representation in terms of the default one. E.g., the following rule removes the 'L' suffixes from bigint values:

```
> __show__ x::bigint = init (str x);
> fact n = foldl (*) 1L (1..n);
```

```
> fact 30;
265252859812191058636308480000000
```

Of course, your definition of __show__ can also call __show__ itself recursively to determine the custom representation of an object.

One case which needs special consideration are thunks (futures). The printer will never use __show__ for those, to prevent them from being forced inadvertently. In fact, you *can* use __show__ to define custom representations for thunks, but only in the context of a rule for other kinds of objects, such as lists. For instance:

```
> nullary ...;
> __show__ (x:xs) = str (x:...) if thunkp xs;
> 1:2:(3..inf);
1:2:3:...
```

Another case which needs special consideration are numeric matrices. For efficiency, the expression printer will always use the default representation for these, unless you override the representation of the matrix as a whole. E.g., the following rule for double matrices mimics Octave's default output format (for the sake of simplicity, this isn't perfect, but you get the idea):

```
> __show__ x::matrix =
>   strcat [printd j (x!(i,j))|i=0..n-1; j=0..m-1] + "\n"
> with printd 0 = sprintf "\n%10.5f"; printd _ = sprintf "%10.5f" end
> when n,m = dim x end if dmatrixp x;
> {1.0,1/2;1/3,4.0};

   1.00000   0.50000
   0.33333   4.00000
```

Finally, by just purging the definition of the __show__ function you can easily go back to the standard print syntax:

```
> clear __show__
> 1/7; 1786; cis (-pi/2);
0.142857142857143
1786
6.12303176911189e-17+:-1.0
```

Note that if you have a set of definitions for the __show__ function which should always be loaded at startup, you can put them into the interpreter's interactive startup files, see INTERACTIVE USAGE.

"As" Patterns

In the current implementation, ''as'' patterns cannot be placed on the ''spine'' of a function definition. Thus rules like the following, which have the pattern somewhere in the head of the left-hand side, will all provoke an error message from the compiler:

```
a@foo x y   = a,x,y;
a@(foo x) y = a,x,y;
a@(foo x y) = a,x,y;
```

This is because the spine of a function application is not available when the function is called at runtime. ''As'' patterns in pattern bindings (**case**, **when**) are not affected by this restriction since the entire value to be matched is available at runtime. For instance:

```
> case bar 99 of y@(bar x) = y,x+1; end;
bar 99,100
```

Head = Function

''As'' patterns are also a useful device if you need to manipulate function applications in a generic way. Note that the ''head = function'' rule means that the head symbol f of an application f x1 ... xn occurring on (or inside) the left-hand side of an equation, variable binding, or pattern-matching lambda expression, is always interpreted as a literal function symbol (not a variable). This implies that you cannot match the ''function'' component of an application against a variable, at least not directly. An anonymous ''as'' pattern like f@_ does the trick, however, since the anonymous variable is always recognized, even if it occurs as the head symbol of a function application. Here's a little example which demonstrates how you can convert a function application to a list containing the function and all arguments:

```
> foo x = a [] x with a xs (x@_ y) = a (y:xs) x; a xs x = x:xs end;
> foo (a b c d);
[a,b,c,d]
```

This may seem a little awkward, but as a matter of fact the ''head = function'' rule is quite useful since it covers the common cases without forcing the programmer to declare ''constructor'' symbols (except nullary symbols). On the other hand, generic rules operating on arbitrary function applications are not all that common, so having to ''escape'' a variable using the anonymous ''as'' pattern trick is a small price to pay for that convenience.

Sometimes you may also run into the complementary problem, i.e., to match a function argument against a given function. Consider this code fragment:

```
foo x = x+1;
foop f = case f of foo = 1; _ = 0 end;
```

You might expect 'foop' to return true for 'foo', and false on all other values. Better think again, because in reality 'foop' will *always* return true! In fact, the Pure compiler will warn you about the second rule of the **case** expression not being used at all:

```
> foop 99;
warning: rule never reduced: _ = 0;
1
```

This happens because a non-nullary symbol on the left-hand side of a rule, which is not the head symbol of a function application, is always considered to be a variable, even if that symbol is defined as a global function elsewhere. So 'foo' isn't a literal name in the above **case** expression, it's a variable! (As a matter of fact, this is rather useful, since otherwise a rule like 'f g = g+1' would suddenly change meaning if you happen to add a definition like 'g x = x-1' somewhere else in your program, which certainly isn't desirable.)

Fortunately, the syntactic equality operator '===' defined in the prelude comes to the rescue here. Just define 'foop' as follows:

```
> foop f = f===foo;
> foop foo, foop 99;
1,0
```

With or when?

A common source of confusion for Haskellers is that Pure provides two different constructs to bind local function and variable symbols, respectively. This distinction is necessary because Pure does not segregate defined functions and constructors, and thus there is no magic to figure out whether an equation like 'foo x = y' by itself is meant as a definition of a function foo with formal parameter x and return value y, or a definition binding the local variable x by matching the constructor pattern foo x against the value y. The **with** construct does the former, **when** the latter.

Another pitfall is that **with** and **when** clauses are tacked on to the end of the expression they belong to, which mimics mathematical language but may be unfamilar if you're more accustomed to programming languages from the Algol/Pascal/C family. If you want to figure out what is actually going on there, it's usually best to read nested scopes ''in reverse'' (proceeding from the rightmost/outermost to the

leftmost/innermost clause).

Also note that since **with** and **when** are part of the expression, not the rule syntax, these clauses cannot span both the right-hand side and the guard of a rule. Usually it's easy to work around this with conditional and **case** expressions, though.

Numeric Calculations

If possible, you should decorate numeric variables on the left-hand sides of function definitions with the appropriate type tags, like **::int** or **::double**. This often helps the compiler to generate better code and makes your programs run faster. The '|' syntax makes it easy to add the necessary specializations of existing rules to your program. E.g., taking the polymorphic implementation of the factorial as an example, you only have to add a left-hand side with the appropriate type tag to make that definition go as fast as possible for the special case of machine integers:

```
fact n::int   |
fact n        = n*fact(n-1) if n>0;
              = 1 otherwise;
```

(This obviously becomes unwieldy if you have to deal with several numeric arguments of different types, however, so in this case it is usually better to just use a polymorphic rule.)

Also note that **int** (the machine integers) and **bigint** (the GMP ''big'' integers) are really different kinds of objects, and thus if you want to define a function operating on both kinds of integers, you'll also have to provide equations for both. This also applies to equations matching against constant values of these types; in particular, a small integer constant like '0' only matches machine integers, not bigints; for the latter you'll have to use the ''big L'' notation '0L'.

Constant Definitions

When defininining a function in terms of constant values which have to be computed beforehand, it's usually better to use a **const** definition (rather than defining a variable or a parameterless function or macro) for that purpose, since this will often allow the compiler to generate better code using constant folding and similar techniques. Example:

```
> extern double atan(double);
> const pi = 4*atan 1.0;
> foo x = 2*pi*x;
> show foo
foo x = 2*3.14159265358979*x;
```

(If you take a look at the disassembled code for this function, you will find that the value 2*3.14159265358979 = 6.28318530717959 has actually been computed at compile time.)

Note that constant definitions differ from parameterless macros in that the right-hand side of the definition is in fact evaluated at compile time. E.g., compare the above with the following macro definition:

```
> clear pi foo
> def pi = 4*atan 1.0;
> foo x = 2*pi*x;
> show foo
foo x = 2*(4*atan 1.0)*x;
```

The LLVM backend also eliminates dead code automagically, which enables you to employ a constant computed at runtime to configure your code for different environments, without any runtime penalties:

```
const win = index sysinfo "mingw32" >= 0;
check boy = bad boy if win;
          = good boy otherwise;
```

In this case the code for one of the branches of 'check' will be completely eliminated, depending on the outcome of the configuration check.

On the other hand, constant definitions are somewhat limited in scope compared to variable definitions, since the bound value must be usable at compile time, so that it can be substituted into other definitions. Thus, while there is no *a priori* restriction on the computations you can perform to obtain the value of the constant, the value must not be a pointer object (other than the null pointer), or an anonymous closure (which also rules out local functions, because these cannot be referred to by their names at the toplevel), or an aggregate value containing any such values.

Constants also differ from variables in that they cannot be redefined (that's their purpose after all) and will only take effect on subsequent definitions. E.g.:

```
> const c = 2;
> foo x = c*x;
> show foo
foo x = 2*x;
> foo 99;
198
> const c = 3;
<stdin>:5.0-8: symbol 'c' is already defined as a constant
```

Well, in fact this not the full truth because in interactive mode it *is* possible to redefine constants after all, if the old definition is first purged with the **clear** command. However, this won't affect any other existing definitions:

```
> clear c
> const c = 3;
> bar x = c*x;
> show foo bar
foo x = 2*x;
bar x = 3*x;
```

(You'll also have to purge any existing definition of a variable if you want to redefine it as a constant, or vice versa, since Pure won't let you redefine an existing constant or variable as a different kind of symbol. The same also holds if a symbol is currently defined as a function or a macro.)

External C Functions

The interpreter always takes your **extern** declarations of C routines at face value. It will not go and read any C header files to determine whether you actually declared the function correctly! So you have to be careful to give the proper declarations, otherwise your program will probably segfault calling the function.

You also have to be careful when passing generic pointer values to external C routines, since currently there is no type checking for these; any pointer type other than char* and expr* is effectively treated as void*. This considerably simplifies lowlevel programming and interfacing to C libraries, but also makes it very easy to have your program segfault all over the place. Therefore it is highly recommended that you wrap your lowlevel code in Pure routines and data structures which do all the checks necessary to ensure that only the right kind of data is passed to C routines.

Special Forms

Special forms are recognized at compile time only. Thus the catch function as well as the logical connectives && and ||, the sequencing operator $$ and the lazy evaluation operator & are only treated as special forms in direct (saturated) calls. They can still be used if you pass them around as function values or partial applications, but in this case they lose all their special call-by-name argument processing.

Laziness

Pure does lazy evaluation in the same way as Alice ML, providing an explicit operation (&) to defer evaluation and create a ''future'' which is called by need. However, note that like any language with a basically eager evaluation strategy, Pure cannot really support lazy evaluation in a fully automatic way. That is, coding an operation so that it works with infinite data structures always requires additional effort to recognize futures in the input and handle them accordingly. This can be hard, but of course in the case of the prelude operations this work has already been done for you, so as long as you stick to these, you'll never have to

think about these issues.

Specifically, the prelude goes to great lengths to implement all standard list operations in a way that properly deals with streams (a.k.a. list futures). What this all boils down to is that all list operations which can reasonably be expected to operate in a lazy way on streams, will do so. (Exceptions are inherently eager operations such as '#', reverse and foldl.) Only those portions of an input stream will be traversed which are strictly required to produce the result. For most purposes, this works just like in fully lazy FPLs such as Haskell. However, there are some notable differences:

* Since Pure uses dynamic typing, some of the list functions may have to peek ahead one element in input streams to check their arguments for validity, meaning that these functions will be slightly more eager than their Haskell counterparts.

* Pure's list functions never produce truly cyclic list structures such as the ones you get, e.g., with Haskell's 'cycle' operation. (This is actually a good thing, because the current implementation of the interpreter cannot garbage-collect cyclic expression data.) Cyclic streams such as 'cycle [1]' or 'fix (\x -> 1:x)' will of course work as expected, but, depending on the algorithm, memory usage may increase linearly as they are traversed.

* Pattern matching is always refutable (and therefore eager) in Pure. If you need something like Haskell's irrefutable matches, you'll have to code them explicitly using futures. See the definition of the 'unzip' function in the prelude for an example showing how to do this.

Stack Size and Tail Recursion

Pure programs may need a considerable amount of stack space to handle recursive function calls, and the interpreter itself also takes its toll. So you may have to configure your system accordingly (8 MB of stack space is recommended for 32 bit systems, systems with 64 bit pointers probably need more). If the **PURE_STACK** environment variable is defined, the interpreter performs advisory stack checks and raises a Pure exception if the current stack size exceeds the given limit. The value of **PURE_STACK** should be the maximum stack size in kilobytes. Please note that this is only an advisory limit which does *not* change the program's physical stack size. Your operating system should supply you with a command such as **ulimit**(1) to set the real process stack size. Also note that this feature isn't 100% foolproof yet, since for performance reasons the stack will be checked only on certain occasions, such as entry into a global function.

Fortunately, Pure normally does proper tail calls (if LLVM provides that feature on the platform at hand), so most tail-recursive definitions should work fine in limited stack space. For instance, the following little program will loop forever if your platform supports the required optimizations:

```
loop = loop;
```

This also works if your definition involves function parameters, guards and multiple equations, of course. Moreover, conditional expressions (**if**-**then**-**else**) are tail-recursive in both branches, and the sequence operator $$ is tail-recursive in its second operand. Note, however, that the logical operators && and || are *not* tail-recursive in Pure, because they are required to *always* yield a proper truth value (0 or 1), which wouldn't be possible with tail call semantics. (The rationale behind this design decision is that it allows the compiler to generate much better code for logical expressions.)

There is one additional restriction in the current implementation, namely that a tail call will be eliminated *only* if the call is done *directly*, i.e., through an explicit call, not through a (global or local) function variable. Otherwise the call will be handled by the runtime system which is written in C and can't do proper tail calls because C can't (at least not in a portable way). This also affects mutually recursive global function calls, since there the calls are handled in an indirect way, too, through an anonymous global variable. (This is done so that a global function definition can be changed at any time during an interactive session, without having to recompile the entire program.) However, mutual tail recursion does work with *local* functions, so it's easy to work around this limitation.

Handling of Asynchronous Signals

As described in section EXCEPTION HANDLING, signals delivered to the process can be caught and handled with Pure's exception handling facilities. Like stack checks, checks for pending signals are only

performed at certain places, such as entry into a global function. This doesn't include tail calls, however, so a busy loop like 'loop = loop;' will *never* be interrupted. To work around this, just add a call to another global function to your loop to make it interruptible. For instance:

```
loop = check $$ loop;
check = ();
```

To handle signals while the above loop is executing, you can add an exception handler like the following:

```
loop = catch handle check $$ loop
with handle (signal k) = catch handle (...) end;
```

(Note the 'catch handle' around the signal processing code which is needed for safety because another signal may arrive while the signal handler is being executed.)

Of course, in a real application the 'check' function would most likely have to do some actual processing, too. In that case you'd probably want the 'loop' function to carry around some ''state'' argument to be processed by the 'check' routine, which then returns an updated state value for the next iteration. This can be implemented as follows:

```
loop x = loop (catch handle (check x))
with handle (signal k) = catch handle (...) end;
check x = ...;
```

FILES

~/.pure_history
: Interactive command history.

~/.purerc, **.purerc**, **.pure**
: Interactive startup files. The latter is usually a dump from a previous interactive session.

prelude.pure
: Standard prelude. If available, this script is loaded before any other definitions, unless **-n** was specified.

ENVIRONMENT

PURELIB
: Directory to search for library scripts, including the prelude. If **PURELIB** is not set, it defaults to some default location specified at installation time.

PURE_INCLUDE
: Additional directories (in colon-separated format) to be searched for included scripts.

PURE_LIBRARY
: Additional directories (in colon-separated format) to be searched for dynamic libraries.

PURE_MORE
: Shell command to be used for paging through output of the **show** command, when the interpreter runs in interactive mode.

PURE_PS
: Command prompt used in the interactive command loop ("> " by default).

PURE_STACK
: Maximum stack size in kilobytes (default: 0 = unlimited).

LICENSE

GPL V3 or later. See the accompanying COPYING file for details.

AUTHOR

Albert Graef <Dr.Graef@t-online.de>, Dept. of Computer Music, Johannes Gutenberg University of Mainz, Germany.

SEE ALSO

(All software listed here is freely available, usually under the GNU Public License.)

Aardappel
: Another functional programming language based on term rewriting, *http://wouter.fov120.com/aardappel*.

Alice ML
: A version of ML (see below) from which Pure borrows its model of lazy evaluation, *http://www.ps.uni-sb.de/alice*.

GNU Octave
: A popular high-level language for numeric applications and free MATLAB replacement, *http://www.gnu.org/software/octave*.

GNU Scientific Library
: A free software library for numeric applications, required for Pure's numeric matrix support, *http://www.gnu.org/software/gsl*.

Haskell
: A popular non-strict FPL, *http://www.haskell.org*.

LLVM The LLVM code generator framework, *http://llvm.org*.

ML A popular strict FPL. See Robin Milner, Mads Tofte, Robert Harper, D. MacQueen: *The Definition of Standard ML (Revised)*. MIT Press, 1997.

Q Another term rewriting language by yours truly, *http://q-lang.sf.net*. Comment] Local Variables: Comment] mode: nroff Comment] End: