Seminar Report

Pure

Sebastian Mayr
sebastian.mayr@student.uibk.ac.at

14 February 2013

Supervisor: Dr. Harald Zankl

Abstract

This seminar report is about Pure, which is a recent functional programming language based on term rewriting. After an introduction to Pure and its interpreter we will explain the basic concepts of term rewriting and how it affects the language. In the following sections the language itself is introduced and interesting features of Pure are highlighted and compared to those of other languages. Finally we show how Pure can be combined with other computing environments to make it suitable for mathematical computations.

Contents

1 Introduction

Pure is a fairly young functional programming language. It was created by Albert Gräf as a successor of his previous programming language Q. He was dissatisfied with the syntax and the execution speed of Q and thus started designing a new programming language which he called Pure. He released the first version in 2008 and is still actively developing and enhancing it.

Pure is mainly used in combination with other languages and computing environments such as *Faust* and *TeXmacs*. Even though Pure's user base is quite small, exhaustive documentation is freely available online [4]. All components of the language (interpreter, standard library, runtime library, etc.) are free software, licensed under the GPLv3. The source code and precompiled packages for *Linux*, *Windows*, *BSD* and *Mac OS X* are available on Pure's website.[1]

Pure is a functional programming language and is based on term rewriting. While it is in principle a typeless language, it provides support for dynamic typing. It has features that one would expect from a functional language but has also some very unique features. In this report we highlight the less common features and compare Pure to other programming languages.

2 Pure Interpreter

Pure comes with an interpreter that can be used to run code interactively. The Pure interpreter however does not really interpret the source code but compiles it to native machine code before execution. This approach is also known as *JIT* (just in time) compilation. To generate efficient code Pure uses the LLVM [11] compiler framework. Beyond JIT compilation the interpreter is able to batch compile a Pure program to a native executable or an object file. The generated object file can then be linked into a C or C++ program which makes it possible to access the Pure runtime from other programming languages.

3 Term Rewriting

Since Pure uses term rewriting as underlying computational model it is necessary to understand its basic concepts. In this section we give a short introduction to term rewriting and explain its implications on the Pure programming language.

Term rewriting is a *Turing-complete* computational model. This means that every Turing machine and hence every program can be simulated by a term rewrite system. A term rewrite system is a set of rules of the form `l` $\rightarrow$ `r`, where `l` and `r` are terms consisting of variables and function symbols. Rewrite rules can be applied to terms by replacing instances of the left-hand side by the right-hand side of the rule. This application is called a *rewrite step*.

The following rewrite system can be used to calculate the maximum of two natural numbers. The constant `0` and the successor function `s` are used to encode natural numbers in unary notation.

[1] `http://purelang.bitbucket.org`

```
max(0,x) → x
max(x,0) → x
max(s(x), s(y)) → s(max(x,y))
```

We can calculate the maximum by repeatedly applying these rewrite rules until the resulting term cannot be rewritten any further. A term where no rules are applicable is called a *normal form*. In the following example we determine the maximum of `3` and `2` by rewriting the term `max(s(s(s(0))), s(s(0)))` to its normal form `s(s(s(0)))`.

```
max(s(s(s(0))), s(s(0))) → s(max(s(s(0)), s(0)))
                         → s(s(max(s(0), 0)))
                         → s(s(s(0)))
```

Term rewriting is at the core of the Pure programming language. A Pure program is basically just a collection of rewrite rules and the Pure interpreter performs rewrite steps until a normal form is reached. If multiple rules are applicable, the Pure interpreter selects the first rule in the order in which they are written. (For details on the evaluation strategy see Section 7.2.) By changing the syntax we can translate the above rewrite system directly into a Pure program:

```
> max 0 x = x;
> max x 0 = x;
> max (s x)  (s y) = s (max x y);
> max (s (s (s 0))) (s (s 0));
s (s (s 0))
```

Since it is not very practical to write such programs, Pure extends the basic concept of term rewriting. For instance, data types for integers and floating point numbers and common arithmetic operations are already predefined. Another extension are *guards* that can be added to a rule to determine if the rule is applicable. With these extensions we can write the max function in a more straightforward way.

```
max x y = x if x > y;  /* this rule is only applicable
                          if x is greater than y */
        = y otherwise; /* the left-hand side of a rule
                          can be omitted if it is the
                          same as in the previous rule */
```

The pure interpreter considers the rules in the order in which they are written and hence applies the second rule in this example only if the first rule is not applicable. Therefore the `otherwise` clause in the second rule is just syntactic sugar and can be omitted.

Pure is due to its term rewriting core a very flexible language. Besides „normal“ calculations Pure can be used to do symbolic evaluations.

```
> (x+y)*z = x*z+y*z; /* distributivity */
> x*(y+z) = x*y+x*z;
> x*(y*z) = (x*y)*z; /* associativity */
> x+(y+z) = (x+y)+z;
```

```
> square x = x*x;
> square 4;
16
> square (a+b);
a*a+a*b+b*a+b*b
```

4 Rules and Patterns

Rules in Pure are similar to function definitions in functional languages like OCaml [12] and Haskell [10]. The left-hand side of a rule is a pattern that serves as a template to be matched against a term. A pattern is composed of function symbols and variables. Identifiers at the head position are interpreted as function symbols whereas any other identifiers are either variables or constants. For example the term `max (f x) (3+y)` contains the function symbols `max`, `f` and `+`, the constant `3` and the variables `x` and `y`. If the literal part (function symbols and constants) of a pattern matches a term, the variables in the pattern are bound to the corresponding values in the term. These variables can then be used in the right-hand side of a rule.

```
> fact 0 = 1;
> fact n = n*fact (n-1) if n > 0;
> fact 5;   /* this term matches the second rule */
120
> fact 0.0; /* this term does not match the first rule,
               because 0 and 0.0 are different constants */
fact 0.0    /* and hence is a normal form */
```

Function applications are – like in most functional programming languages – *curried*. This means that a function that takes multiple arguments is transformed into a series of functions that take a single argument. Each function is a *partial application* of the original function. For example `f x y` is transformed into a function `f` that takes the argument `x` and returns a function that takes the argument `y`. Currying allows us to derive new functions by partially applying existing ones. We can, for instance, implement the successor function by partially applying the `+` function to `1`.

```
> s = (+) 1;
> s 2;
3
```

4.1 Non linear patterns

In contrast to most other functional programming languages Pure supports patterns where the same variable occurs more than once. These, so-called *non linear* patterns only match if all instances of a variable are bound to the same value.

```
> f x x = x*x;
> f 5 5;
```

```
25
> f 5 6;
f 5 6
```

Most functional languages that are based on lambda calculus only support linear pattern, because otherwise they would not be confluent, i.e. different evaluation strategies could lead to different results.

5 Special Expressions

Pure has a number of *special expressions* that extend the term rewriting semantic. These expressions are *special*, because they cannot occur in normal form terms. Most special expressions are known from other (functional) programming languages and hence are only shortly covered.

Conditional expressions: `if x then y else z`

Evaluates to `y` if `x` is true or to `z` otherwise.

```
> max x y = if x > y the x else y;
```

Lambdas: `\x -> y`

A lambda is an *anonymous function* that evaluates to the right-hand side `y` if the argument term matches the pattern `x`.

```
> foldl (\x y -> y:x) [] (1..5);
[5,4,3,2,1]
```

Case expressions: `case x of s = t; u = v; ... end`

The case expression is similar to Haskell's *case* and OCaml's *match* expression. If the term `x` matches a pattern (`s`, `u`, etc.) in the rules list, the expression is evaluated to the corresponding right-hand side (`t`, `v`, etc.) of the matching rule.

```
> case (a*b+c) of
>    (x*y) = "multiplication";
>    (x+y) = "addition";
> end;
"addition"
```

When expressions: `x when s = t; u = v; ... end`

The when expression binds *local variables*. The term `x` is evaluated with the variables in the patterns `s`, `u`, etc. bound to their corresponding values in the right-hand side terms `t`, `v`, etc.

```
> s = (+) 1;
> s (a+b) when a=1; b=2; end;
4
```

With expressions: `x with s = t; u = v; ... end`

The with expression is used to define local functions. The term `x` is evaluated considering the additional rules `s = t`, `u = v`, etc.

```
> square 4 with square x = x*x; end;
16
```

Patterns in special expressions behave slightly different than patterns in rules. A `match_failed` expression is thrown if the subject term does not match the pattern whereas a rule is simply not applied.

```
> (\(x:y) -> x:y) 5;
<stdin>, line 14: unhandled exception 'failed_match' ...
> f x:y = x:y;
> f 5; /* the rule is not applied, this term is a normal form */
f 5
```

6 Container Types

6.1 Lists

In Pure, like in many other functional programming languages, lists are built using the *cons* (`:`) operator and the *empty list* constant (`[]`). We can also use the bracket notation to declare lists, for instance `[1,2,3]` denotes the list `1:2:3:[]`. Lists can contain different types of data and can be nested.

Symbols like cons and `[]` are so-called constructors. Unlike OCaml and Haskell, Pure allows us to define rules for constructors, i.e. they can appear as the root symbol on the left-hand side of a rule. There is no distinction between function symbols and constructors in Pure. For instance the following rules for the cons operator cause lists to be always sorted and deduplicated.

```
> x:y:z = y:x:z if y < x;
> [12,1,7,2,3,3];
[1,2,3,3,7,12]
>
> x:x:z = x:z;
> [12,1,7,2,3,3];
[1,2,3,7,12]
```

Pure offers a number of functions and operators for lists. The following examples cover some of them.

```
> 1..10; /* a list from 1 to 10 */
[1,2,3,4,5,6,7,8,9,10]
> let x = 1:3..10; /* all odd numbers from 1 to 10 */
> x
[1,3,5,7,9]
> #x /* compute the length */
5
> x!2 /* access the element at index 2 */
```

```
5
> x!![0,2,3]  /* create a list with elements
                 at the specified indices */
[1,5,7]
```

A convenient way to generate lists are *list comprehensions*. A list comprehensions consists of an expression, a generator clause and an optional filter clause. The resulting list is built by repeatedly evaluating the expression with its variables bound according to the generator and filter clauses. For example, in the comprehension `[2*x | x=1..10]` the expression `2*x` is evaluated with `x` bound to each element of the list `1..10`. By adding the filter clause `x % 2 == 0`, the variable `x` is only bound to even numbers in that range.

```
> [2*x | x=1..10; x mod 2 == 0];
[4,8,12,16,20]
```

6.2 Tuples

Another data structure that is similar to lists are *tuples*. Tuples are constructed with the right-associative comma (`,`) operator and can, in contrast to lists, not be nested. For instance `1,2,3`, `(1,2),3` and `1,(2,3)` are one and the same tuple. To achieve this the following rules for the tuple constructor are defined in the standard library.

```
> x,() = x;
> (),y = y;
> (x,y),z = x,y,z;
```

With these rules every tuple is flattened and the empty tuple becomes a neutral element in respect to the comma operator.

7 Language Features

7.1 Types

Pure is in principle a typeless language, but supports, as an alternative to classic data types, *type rules*. Type rules are similar to normal rules, but are prefixed with the keyword `type`, take only one argument and evaluate to a truth-value. They are used as predicates to determine if a given term belongs to a type. A type consists of those terms for which the type predicate evaluates to true. We can, for example, define a data type for triples as follows.

```
> type triple [x:y:z] = true;
```

We can use the typep function to check if the term `[1,2,3]` belongs to the triple type.

```
> typep triple [1,2,3];
1
> typep triple [1,2,3,4];
```

```
0
```

The previous definition of triple is called *partial*, because we can extend it with additional rules.

```
type triple x = #x == 3; /* all terms for which the
                            length function returns 3 */
> typep triple (1,2,3);
1
```

This type rule applies to all terms `x` and hence the definition of triple is now *complete* and cannot be extended further.

We can use these type predicates in rules as so-called *type-tags*. Type-tags can be added to the arguments of a rule and determine to which data types the rule is applicable. In the following example the `sum` rule is only applied to terms that satisfy the `triple` predicate.

```
> sum x::triple = x!0 + x!1 + x!2;
> sum (1,2,3);
6
> sum [1,2,3];
6
> sum 3;
sum 3
```

The fact that type-tags are only evaluable at runtime makes Pure a *dynamically typed* language.

7.1.1 Interface Types

Pure also offers a way to define a type through a set of operations it has to support. This concept is also known as *duck typing* and is best described with its name giving quote: „When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.“ This style of typing is supported by Pure through *interface types*. An interface definition consists of a set of patterns that indicate which operations are required. These patterns are then matched against the left-hand side of each rule and a list of types is derived. Every type in that list supports the stated operations and is part of the interface. For instance the addable interface consists of all types that support the `+` operator for arguments of equal type.

```
> interface addable with
>       x::addable + y::addable;
> end;
>
> show interface addable
type addable x::int;
type addable x::double;
type addable x::bigint;
type addable s::string;
type addable [];
type addable xs@(_:_);
```

It is also possible to specify an interface with multiple operations. The interface in the following example consists of all types that support typical list operations.

```
> interface listlike with
>       #x::listlike;
>       x::listlike!n::int;
>       head x::listlike;
>       tail x::listlike;
> end;
> show interface listlike
type listlike x::matrix;
type listlike s::string;
type listlike (x:xs);
```

Unlike in the previous example, the empty list is not part of the interface, because neither the head nor the tail functions are defined for the empty list. Interface types can be used just like normal types, e.g. as type-tags or as type predicates using the typep function.

7.2 Lazy evaluation

An expression can comprise multiple sub-terms that are reducible by a rule (so-called *redexes*). The Pure interpreter evaluates such an expression by reducing the leftmost innermost redex first. This evaluation strategy is also known as *eager evaluation*. In some cases, however, it is desirable to delay the evaluation of an expression until its value is needed. In Pure this can be achieved through the postfix operator `&`, that turns its argument into a parameterless anonymous function (*thunk*). If the value of the expression is needed, the thunk is replaced by its computed value.

```
> let x = 5 &;
> x;
#<thunk 0x7f1f8f5575d8>
> x + 2; /* force the evaluation of x */
7
> x; /* x is no longer a thunk */
5
```

Lazy evaluation allows us to build infinite lists, also called *streams*, by turning the tail of a list into a thunk. In the following example we apply the `&` operator to the recursive call of the `from` function to build such an infinite list.

```
> from x = x:from (x+1) &;
> let x = from 5;
> x;
5:#<thunk 0x7f1f8f557448>
> x!!(0..9);
[5,6,7,8,9,10,11,12,13,14]
```

The same result can be achieved by using the built-in list generator with the `inf` constants.

```
> 5..inf;
5:#<thunk 0x7f1f8f557448>
```

7.3 The Quote

Like the `&` operator, *the quote* influences the evaluation of an expression. More precisely, the quote inhibits the evaluation of its argument. We can manipulate quoted (unevaluated) expressions, for instance by using pattern matching or by binding variables to a value.

```
> let x+y = '(23*a+5*1*b) with b=6;
> x;
23*a
> y;
5*1*6
```

The `eval` function evaluates quoted expressions.

```
> eval y;
30
```

7.4 Macros

Pure offers besides ordinary rules and type rules a third class of rules, namely *macros*. They are, similar to macros in other programming languages (e.g. C), evaluated at compile time. Like ordinary rules, macros are evaluated by performing rewrite steps until a normal form is reached. After the evaluation, the compiler replaces every instance of a macro by the resulting normal form. This transformation happens before the actual compilation of the program.

Macros are defined by prefixing rules with the `def` keyword. In the following example we define the list length function as macro. This optimizes the regular length function insofar that it is already evaluated at compile time for list constants.

```
> def #[] = 0;
> def #(x:xs) = #xs+1;
> lstlen = #[1,2,3,4]; /* evaluated during the JIT compilation */
> tpllen = #(1,2,3,4);
> show lstlen /* print the definition of lst */
lstlen = 4;
> show tpllen
tpllen = #(1,2,3,4);
```

As shown in the example above, macros can be recursive. In fact macros can be arbitrary rewrite rules and are hence *Turing-complete*. This implies in particular that the execution of macros and therefore the compilation of a program does not necessarily terminate.

7.5 C Interface

Pure provides an interface to the C programming language that makes it possible to call C functions. To use a C function it has to be declared using the `extern` keyword and can then be called like a normal function. When calling

such a function the interpreter converts all supplied arguments to corresponding C data types.

```
> extern void puts (char *str); //declare the C function puts
> puts "Hello␣world!";
Hello world!
```

Since Pure provides no facilities for input and output, calling C functions is the only way to perform these operations. The C interface can be used to load and access any C library and hence extends the capabilities of the Pure platform vastly. As an alternative to calling external functions Pure makes it possible to embed the C code directly into a Pure program. The inlined code is compiled by the Pure interpreter in cooperation with an LLVM enabled C compiler. This feature is mainly used to implement performance critical parts of an algorithm.

```
> %<
> int fact(int n) { //embedded C function
>       int res=1;
>       for(; n > 1; n--)
>               res *= n;
>       return res;
> }
> %>
> fact 10;
3628800
```

Pure makes it very easy to interface with imperative-style code with arbitrary side effects. Therefore Pure is, despite its name, not a *purely functional* language.

8 Utilization

Besides Pure's usage as standalone language, it is primarily used in combination with other computing environments and languages. Pure integrates with the following computing platforms:

Octave a language intended for numerical computations [2]

Pure Data a graphical programming environment for audio, video, and graphical processing [6]

Faust a functional programming language designed for real-time signal processing [1]

REDUCE a computer algebra system [7]

TeXmacs a scientific text editing platform [9]

Gnumeric a spreadsheet program [3]

8.1 Mathematical computations with TeXmacs and REDUCE

TeXmacs is a graphical text editor with typesetting facilities similar to those of LaTeX. A plugin is available that enables the usage of Pure inside a TeXmacs document, i.e. TeXmacs becomes a graphical interface to the Pure interpreter. This allows us to enter mathematical expressions in a more convenient way so that it becomes easier to use Pure's symbolic evaluation capabilities to perform mathematical computations. In addition we can use the interface to the REDUCE library which provides a rich set of functions for algebraic computations.

The following excerpt of a TeXmacs document contains a Pure shell which is used to differentiate a list of terms by applying the `df` function of the REDUCE library.

```
> using reduce;
```

$$> \left[\frac{\sin(x)}{x^i} \mid i = 1..3\right];$$

$$\left[\frac{\sin(x)}{x}, \frac{\sin(x)}{x^2}, \frac{\sin(x)}{x^3}\right]$$

```
> map (\y -> df y x) ans;
```

$$\left[\frac{\cos(x)\,x - \sin(x)}{x^2}, \frac{\cos(x)\,x - 2\sin(x)}{x^3}, \frac{\cos(x)\,x - 3\sin(x)}{x^4}\right]$$

By combining TeXmacs, REDUCE and Pure we obtain a powerful mathematical computing platform, roughly comparable to Mathematica or Matlab.

9 Conclusion

As we have seen Pure combines many different programming concepts into a language. It combines functional features known from OCaml and Haskell with the concepts of macros and quoting similar to Lisp. In addition it provides a dynamic type system with support for interfaces akin to those of the Go language. These features on top of the term rewriting core make Pure a very flexible and dynamic language. Therefore Pure can cover a wide range of applications: from scientific calculations with focus on symbolic evaluations to digital signal processing with strict time constraints. To achieve this Pure tries to balance functional purity and practicality, for instance by making it easy to call external C functions.

Pure is still a very young language and hence lacks some features, most prominently support for concurrent programming. Despite its rather small user base Pure is very actively developed, especially by its creator Albert Gräf. He puts a lot of effort into making Pure a useful language by implementing new features, writing exhaustive documentation and by integrating Pure into multiple computing platforms. All in all Pure has the potential to become a more widely known and used language.

References

[1] *Faust – Functional Audio Stream*. URL: http://faust.grame.fr (visited on 02/13/2013).

[2] *GNU Octave*. URL: http://www.gnu.org/software/octave/ (visited on 02/13/2013).

[3] *Gnumeric spreadsheet*. URL: http://projects.gnome.org/gnumeric/ (visited on 02/13/2013).

[4] Albert Gräf. *Pure Language and Library Documentation 0.56*. Nov. 2012. URL: http://purelang.bitbucket.org/docs/puredoc.pdf (visited on 11/19/2012).

[5] Aart Middeldorp. *Term Rewriting*. Lecture Notes. University of Innsbruck, 2012.

[6] *Pure Data*. URL: http://puredata.info (visited on 02/13/2013).

[7] *REDUCE*. URL: http://www.reduce-algebra.com (visited on 02/13/2013).

[8] Christian Sternagel and Harald Zankl. *Functional Programming*. Lecture Notes. University of Innsbruck, 2011.

[9] *TeXmacs*. URL: http://www.texmacs.org (visited on 02/13/2013).

[10] *The Haskell Programming Language*. URL: http://www.haskell.org (visited on 02/13/2013).

[11] *The LLVM Compiler Infrastructure*. URL: http://llvm.org (visited on 02/13/2013).

[12] *The OCaml Programming Language*. URL: http://ocaml.org (visited on 02/13/2013).