Lazy Dynamic Input/Output in the lazy functional language Clean

Martijn Vervoort and Rinus Plasmeijer

Nijmegen Institute for Information and Computing Sciences,
Toernooiveld 1, 6525 ED, Nijmegen, The Netherlands
{martijn,rinus}@cs.kun.nl

early DRAFT of September 2, 2002

Abstract. The new release of Clean offers a hybrid type system with both static and dynamic typing. Any common static Clean expression can in principle be converted into a dynamic expression (called a "dynamic"), and backwards. The type of the dynamic (an encoding of the original static type) can be checked at run-time via a special pattern match after which the dynamic expression can be evaluated as efficiently as usual. Clean furthermore offers "dynamic I/O": any application can read in a dynamic that is stored by some other application. Such a dynamic can contain unevaluated functions (closures) that are unknown in the receiving application. The receiving application therefore has to be extended with these function definitions. This is not trivial because Clean uses compiled code and is not an interpreted language that uses some byte code. A running Clean application therefore communicates with a dynamic linker that is able to add the missing binary code to the running application. So, dynamics can be used to realize plugins and mobile code in a type safe way without loss of efficiency in the resulting code. In this paper we explain the implementation of dynamic I/O. Dynamics are written in such a way that internal sharing is preserved when a dynamic is read. Dynamics are read in very lazily in phases: first its type is checked, and only if the evaluation is demanded, the dynamic expression is reconstructed and the new code is linked in. Dynamics can become quite complicated: they can contain sharing, they can be cyclic, they can even refer to other dynamics, and they may be distributed over a computer network. We have managed to completely hide the internal storage structure for the user by separating the storage of dynamics in system space and user space. For the user a dynamic on disc is just a Clean expression of some type that can be used in any application.

1 Programming with dynamics

This paper describes how dynamic I/O is realized in Clean. Before we go into more detail, we give two small examples of Clean applications that use dynamics. We explain what these applications do and sketch what is required to store and read dynamics from disc.
1.1 A dynamic producer and its consumer

The producer application shown below creates a dynamic which contains an infinite list of integers. The writeDynamic-function takes this dynamic as its argument and stores it into a file.

```
Start world // Producer application
# dyn = dynamic [1..] :: [Int]
# (ok,world) = writeDynamic "infinite_list" dyn world
    = world
```

When a dynamic is built, its static type is the type Dynamic regardless of its contents. The type Dynamic can be regarded as an existential type defined as:

```
:: Dynamic = E.a: {
    value :: a
    , type :: T_typeObject

    It is important to known that the dynamic constructor is a lazy constructor; it stores the unevaluated infinite list in the record. The infinite list contains closures i.e. unevaluated function applications. These compiled function definitions have to be stored as well.

    A representation of the static type of the infinite list is stored in the type-field of the dynamic. It needs to be stored to check the type consistency between the producer and consumer who are compiled independently of eachother.

    The writeDynamic-function encodes the dynamic dyn into a string. Using a standard I/O write-operation, the string is stored in a file named infinite_list. The consumer application shown below, uses the readDynamic-function to read the stored dynamic from that file, checks that the type matches and then prints the sum of the first 100 elements of the integer list.
```

```
Start world // Consumer application
# (_,x,world) = readDynamic "infinite_list" world
    = (sum (take 100 (extract_or_die x)),world)
where
    extract_or_die :: Dynamic -> [Int]
    extract_or_die (1 :: [Int]) = 1
    extract_or_die _ = abort "type mismatch"
```

The readDynamic-function uses a standard I/O read operation to read the encoded dynamic from file. This string representation is then decoded into a dynamic. The decoding is done lazily. The compiled function definitions required for evaluation are also retrieved from disk and added to the consumer application. This is also part of decoding phase.

The consumer first tries to unify the dynamic type stored in the type-field of the dynamic with the type specified in the dynamic pattern match of the
extract_or_die-function. If it fails, the consumer is aborted. Otherwise the
now statically typed producer list 1 is decoded and returned.

The consumer can now apply its own take and sum to the still lazy producer
list which has been extracted from the dynamic. The result is printed. Note that
the encoding and decoding of a dynamic preserves laziness.

More detailed information about the creation of dynamics and dynamic pattern
matches can be found in [3] and [4]. The standard I/O operations can be any
string I/O operation as provided by interfaces to operating systems. In Clean
values, and therefore also dynamics, are represented as graphs at run-time. This
paper focuses on the encode and decode-operations to be performed on graphs
e.g. dynamics.

1.2 Dynamic version of the well-known apply function

The next example is a dynamic version of the well-known apply function. The
apply function shown below serves to illustrate dynamic pattern variables and
laziness in the decoding of a dynamic.

Start world
  # (_,f,world) = readDynamic "function" world
  # (_,a,world) = readDynamic "argument" world
  # (ok2,world)
    = writeDynamic "result" (apply f a) world
  = world;
where
apply :: Dynamic Dynamic -> Dynamic
apply (func :: a -> b) (arg :: a) = dynamic (func arg) :: b
apply _ _ = abort "type mismatch"

The dynamic apply application reads two dynamics: function and argument.
The apply-function then performs a type safety check: if the former contains
a function which can take the value of the latter as its argument, then the
function can be applied to the argument. This function application is stored
into a dynamic of type b which is stored into a file. Due to the laziness of the
dynamic constructor, the actual application of the function to the argument is
postponed until some (other) application reads the dynamic in again and forces
its evaluation. The second alternative of apply causes the application to abort.

As the producer/consumer example of section 1.1 showed: dynamics are read
and lazily decoded from file. Therefore the readDynamic-functions postpone
decoding until it is really required.

In contrast to the previous example, the dynamic patterns now contain the
dynamic pattern variables a and b. Both dynamic patterns succeed iff τ₁ is
unifiable with a → b and τ₂ is unifiable with a, where τ₁ and τ₂ are the dynamic
types of the dynamics stored in respectively function and argument.

The occurrence of a in both patterns, forces unification between the types
of the two different dynamics. In case of a successful dynamic pattern match, it
guarantees the type-safe application of function func to its argument arg.
If both dynamic patterns of the first apply–alternative succeed, only the required types \( \tau_1 \) and \( \tau_2 \) will be decoded. There is no need to read and decode the complete graphs because their evaluation is not demanded. It is sufficient to store references to the encoded func and arg on disc. So, reading of dynamics is done very lazy.

The writeDynamic-function encodes the dynamic and stores its string representation in the file named result. In this case the encoded dynamic will depend on the encoded dynamics stored in the files function and argument.

The encoded result dynamic would have no decode references, if the value components func and arg would have been evaluated before the encoding of the resulting dynamic took place. So, the degree of evaluation of an application determines the dependencies of a dynamic on parts of other dynamics.

2 The full paper

The full paper will address the following issues:

- the design and implementation of the encode and lazy decode operations.
- user support for dynamics.
- related work.
- conclusion and future work.

References

5. Rinus Plasmeijer, Marko van Eekelen, Marco Pil and Pascal Serrarens (1999), \emph{Parallel and Distributed Programming in Concurrent Clean}, in Research Directions in Parallel Functional Programming, K. Hammond and G. Michaelson (Eds), Springer Verlag, pp. 323-338.


