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
{martijnv,rinus}@cs.kun.nl

Abstract. In this paper we explain how dynamics can be communicated between independently programmed Clean applications. This is an important new feature of Clean because it allows type safe exchange of both data and code. In this way mobile code and plug-ins can be realized easily. The paper discusses the most important implementation problems and their solutions in the context of a compiled lazy functional language. The implemented solution reflects the lazy semantics of the language in an elegant way and is moreover quite efficient. The resulting rather complex system in which dynamics can depend on other dynamics, is effectively hidden from the user by allowing her to view dynamics as "typed files" that can be manipulated like ordinary files.

1 Introduction

The new release of the Clean system [11] offers a hybrid type system with both static and dynamic typing. Any statically typed expression can in principle be converted into a dynamically typed expression i.e. a dynamic, and backwards.

The type stored in the dynamic, i.e. 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.

In this paper we discuss the storage and the retrieval of dynamics: any application can read a dynamic that has been stored by some other application. Such a dynamic can contain unevaluated function applications, i.e. closures, functions and types that are unknown to the receiving application. The receiving application therefore has to be extended with function definitions. Dynamics can be used to realize plug-ins, mobile code and persistency in a type safe way without loss of efficiency in the resulting code.

The integration of strongly typed lazy dynamic I/O in a compiled environment with minimal changes to the existing components of the system while maintaining efficiency and user-friendliness, requires a sophisticated design and implementation. This paper presents the most interesting problems and their solutions by means of examples.

* This work was supported by STW as part of project NWI.4411

This work is based on earlier work by Cardelli [1] who introduced the theoretical foundations of dynamics. Marco Pil has extended and adapted it for Clean. In contrast to his work [9] and [10], this paper addresses the I/O aspects of dynamics. Dynamic I/O is the input and output of dynamics by appropriate extensions of the compilation environment and its run-time system.

Our contribution is that we have designed and implemented an efficient extension of the Clean compilation environment and its run-time system to support lazy dynamic I/O. The presented solution can also be applied to other lazy functional programming languages such as Haskell [5].

The outline of this paper is as follows. In section 2 we introduce the elementary operations of dynamics: packing and unpacking. In section 3 we introduce I/O of dynamics: dynamic I/O. The requirements and the architecture are presented in section 4. For reasons of efficiency, dynamics are divided into pieces. This is explained in section 5. This splitting up of dynamics can cause sharing problems. These are solved in section 6. In section 7 we explain how we have managed to hide the resulting complexity of the system from the user. The paper concludes with related work, conclusions and future work.

2 Elementary language operations on dynamics

A dynamic basically is a typed container for an ordinary expression. The elementary operations on dynamics are packing and unpacking. In essence these elementary operations convert a statically typed expression into its dynamically typed equivalent and vice versa.

2.1 Packing a typed expression into a dynamic

A dynamic is built using the keyword `dynamic`. Its arguments are the expression to be packed into a dynamic and, optionally, the *static* type **t** of that expression. The actual packing is done lazily. The resulting dynamic is of static type `Dynamic`. A few examples are shown below:

```
(dynamic True    :: Bool             ) :: Dynamic
(dynamic fib 3                       ) :: Dynamic
(dynamic fib     :: Int -> Int       ) :: Dynamic
(dynamic reverse :: A.a: [a] -> [a] ) :: Dynamic
```

A dynamic should at least contain:

- The expression to be packed, which is called the *dynamic expression* for the rest of this paper.
- An encoding of its static type **t** (either explicitly specified or inferred), which is called the *dynamic type* for the rest of this paper.

2.2 Unpacking a typed expression from a dynamic

Before a dynamically typed expression enclosed in a dynamic can be used, it must be converted back into a statically typed expression and made accessible. This can be achieved by a run-time dynamic pattern match.

A dynamic pattern match consists of an ordinary pattern match and a type pattern match. First, the type pattern match, on the dynamic type is executed. Only if the type pattern match succeeds, the ordinary pattern match on the dynamic expression is performed. If the ordinary pattern match succeeds, the right hand side of an alternative is executed. Otherwise, evaluation continues with the next alternative. A small example is shown below:

```
f :: Dynamic -> Int
f (0 :: Int) = 0
f (n :: Int) = n * n + 1
f else       = 1
```

The dynamic pattern match of the first alternative requires the dynamic type to be an integer type and the dynamic expression to be zero. If both conditions are met, zero is returned. The second alternative only requires the dynamic type to be an integer. The third alternative handles all remaining cases.

The example below shows the dynamic version of the standard apply function:

```
dyn_apply :: Dynamic Dynamic -> Dynamic
dyn_apply (f :: a -> b) (x :: a) = dynamic (f x) :: b
dyn_apply else1         else2    = abort "dynamic type error"
```

The function takes two dynamics and tries to apply the dynamic expression of the first dynamic to the dynamic expression of the second. In case of success, the function returns the (lazy) application of the function to its argument in a new dynamic. Otherwise the function aborts.

The multiple occurrence of the type pattern variable `a` effectively forces unification between the dynamic types of the two input dynamics. If the first alternative succeeds, the application of the dynamic expression `f` to the dynamic expression `x` is type-safe.

3 Dynamic I/O: writing and reading typed expressions

Different programs can exchange dynamically typed expressions by using dynamic I/O. In this manner, plug-ins and mobile code can be realized. To achieve this, the system must be able to store and retrieve type definitions and function definitions associated with a dynamic. Among other things, this requires dynamic linking.

3.1 Writing a dynamically typed expression to file

Any dynamic can be written to a file on disk using the `writeDynamic` function of type `String Dynamic *World -> *World`. In the producer example below a dynamic is created which consists of the application of the function `sieve` to an infinite list of integers. This dynamic is then written to file using the `writeDynamic` function.

Evaluation of a dynamic is done lazily. As a consequence, the application of `sieve` to the infinite list, is constructed but not evaluated because its evaluation is not demanded. We will see that the actual computation of a list of prime numbers will be triggered later by the consumer.

```
producer :: *World -> *World
producer world = writeDynamic "primes" (dynamic sieve [2..]) world
where
    sieve :: [Int] -> [Int]
    sieve [prime:rest] = [prime : sieve filter ]
    where
        filter = [ h \\ h <- rest | h mod prime <> 0 ]
```

More information than the dynamic expression and its type have to be stored at run-time, if the dynamic is to be used as a plug-in by applications other than its creating application. We also need:

- The function definitions required for the evaluation of the dynamic expression. A severe complication here is that these function definitions have been compiled to native machine code. When the dynamic is used, these compiled function definitions have to be added to the running application.
- The type definitions required for matching the dynamic type against the type pattern specified in the dynamic pattern. The type definitions are needed because different Clean applications may have different definitions of equally named types. A type definition check is only needed to check that equally named types are indeed equivalent.

In general this information is already known at compile-time, but it should be made accessible at run-time.

3.2 Reading a dynamically typed expression from file

Any dynamic can be read from disk using the `readDynamic` function of type `String *World -> (Dynamic,*World)`. This `readDynamic` function is used in the consumer example below to read the earlier stored dynamic. The dynamic pattern match checks whether the dynamic expression is an integer list. In case of success the first 100 elements are taken. Otherwise the consumer aborts.

```
consumer :: *World -> [Int]
consumer world
    # (dyn, world) = readDynamic "primes" world
```

```
    = take 100 (extract dyn)
where
    extract :: Dynamic -> [Int]
    extract (list :: [Int]) = list
    extract else            = abort "dynamic type check failed"
```

To turn a dynamically typed expression into a statically typed expression, the following steps need to be taken:

1. Unify the dynamic type and the type pattern of the dynamic pattern match. If unification fails, the dynamic pattern match also fails.
2. Check the type definitions of equally named types from possibly different applications for structural equivalence provided that the unification has been successful. If one of the type definition checks fails, the dynamic pattern match also fails. Equally named types are equivalent iff their type definitions are syntactically the same (modulo α-conversion and the order of algebraic data constructors).
3. When evaluation requires the now statically typed expression, construct it and add the needed function definitions to the running application.

The addition of compiled function definitions and type definitions referenced by the dynamic being read is handled by the dynamic linker.

4 Architecture for dynamic I/O

The architecture based on requirements listed in this section, is presented. The context it provides is used by the rest of this paper.

4.1 Requirements

Our requirements are:

– *Correctness.* We want the system to preserve the language semantics of dynamics: storing and retrieving an expression using dynamic I/O should not alter the expression and especially not its evaluation state.
– *Efficiency.* We want dynamic I/O to be efficient.
– *Preservation of efficiency.* We do not want any loss of efficiency compared to ordinary Clean programs not using dynamics, once a running program has been extended with the needed function definitions.
– *View dynamics as typed files.* We want the user to be able to view dynamics on disk as "typed files" that can be used without exposing its internal structure.

4.2 Architecture

For the rest of this paper, figure 1 provides the context in which dynamic I/O takes place.

The Clean source of an application consists of one or more compilation units. The Clean compiler translates each compilation unit into compiled function definitions represented as symbolic machine code and compiled type definitions. The compiled function and type definitions of all compilation units are stored in the application repository.

Application 1 uses the `writeDynamic` function to create a dynamic on disk. The dynamics refers to the application repository.

Application 2 uses the `readDynamic` function to read the dynamic from disk. If the evaluation of the dynamic expression is required after a successful dynamic pattern match, the dynamic expression expressed as a graph is constructed in the heap of the running application. The dynamic linker adds the referenced function and type definitions to the running application. Then the application resumes normal evaluation.

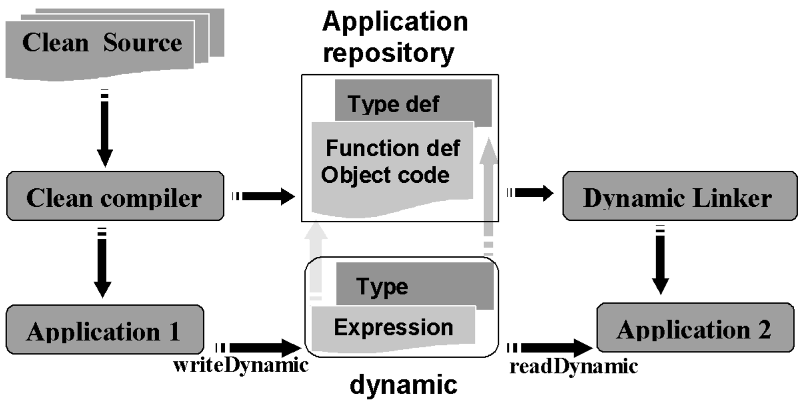

Fig. 1. Architecture of dynamic I/O.

Some of the requirements are already (partially) reflected in the architecture:

– *Efficiency.* The figure shows that the dynamic expression and its dynamic type can be identified separately from each other. Therefore the rather expensive construction of the dynamic expression can be postponed until its dynamic type is successfully pattern matched. The next sections refine this *laziness* even more.

The figure also shows that a dynamic does not contain the compiled function and type definitions. The dynamic merely *refers to* the repository which means that dynamics can share repositories and especially function definitions that have already been compiled. This *sharing* reduces the expensive cost of linking function definitions.

– *Preservation of efficiency.* As the figure shows *compiled* function definitions are used by dynamics. The very same function definitions are also used by ordinary Clean programs not using dynamic I/O. Therefore after dynamic I/O completes, the program is resumed at normal efficiency.

5 Partitions: dynamic in pieces

In this section, we explain that dynamics are not constructed in their entirety but in smaller pieces called *partitions.* This is sensible because often the evaluator does not need all pieces of a dynamic. As a consequence the expensive linking process of function definitions is postponed until required.

5.1 Dynamics are constructed piece by piece

Up until now, we have implicitly assumed that dynamics are constructed in their entirety. But only the following steps need to be taken, to use a dynamic expression (nested dynamics may contain more dynamic expressions):

1. Read a dynamic type from file.
2. Decode the type from its string representation into its graph representation.
3. Do the unifications specified by the dynamic pattern match.

   Only after successful unifications:
4. Read the dynamic expression from file.
5. Decode the expression from its string representation into its graph representation.

We have decided to construct a dynamic piece by piece for reasons of efficiency. In general the construction of a dynamic in its entirety is both unnecessary and expensive. For example when a dynamic pattern match fails, then it is unnecessary to construct its dynamic expression. Moreover, it is even expensive because it would involve the costly process of linking the compiled function definitions.

As a consequence, a dynamic which is represented at run-time as a graph, must be partitioned. The (nested) dynamic expressions and the dynamic types should be constructible from the dynamic by its partitions.

5.2 Partitions

Partitions are pieces of graph encoded as strings on disk which are added in their entirety to a running application. Partitions are:

– (parts of) a dynamic expressions.
– (parts of) a dynamic types.
– subexpressions shared between dynamic expressions.

In this paper we only present the outline of a naïve partitioning algorithm which colours the *graph* representing the dynamic to be encoded:

– A set of colours is associated with each node of the graph. A unique colour is assigned to each dynamic expression and to each dynamic type.
– If a node is reachable from a dynamic expression or from a dynamic type, then the colour assigned to that dynamic expression or that dynamic type is added to the colour set of that node.

We have chosen a partition to be a set of equally coloured graphs: the colour sets of the graph nodes must all be the same. This *maximizes* the size of a partition to reduce linker overhead. Any other definition of a partition would also do, as long as it contains only equally coloured nodes.

For example, consider the `Producer2` example below. After partitioning the `shared_dynamic` expression, the encoded dynamic consists of seven partitions. There are three dynamics involved. For each dynamic two partitions are created: one partition for the dynamic expression and one partition for its type. An additional partition is created for the shared tail expression.

```
Producer2 :: *World -> *World
Producer2 world = writeDynamic "shared_dynamic" shared_dynamic world
where
    shared_dynamic  = dynamic (first, second)
    first           = dynamic [1 : shared_expr ]
    second          = dynamic [2 : shared_expr ]
    shared_expr     = sieve [3..10000]
```

5.3 Entry nodes

In general several different nodes within a partition can be pointed to by nodes of other partitions. A node of a partition is called an *entry* node iff it is being pointed to by a node of another partition. For the purpose of sharing, the addresses of entry nodes of constructed partitions have to be retained. The following example shows that a partition can have multiple entry nodes:

```
:: T = Single T | Double T T
f = dynamic (dynamic s1, dynamic s2)
where
    s1 = Single s2
    s2 = Double s1 s2
```

The nodes `s1` and `s2` form one partition because both nodes are reachable from the (nested) dynamics in the `f` function and from each other. Both nodes therefore have the same colour sets. Both nodes are pointed to by the nested dynamics, which makes them both entry nodes. Apart from cyclic references, multiple entry nodes can also occur when dynamics share at least two nodes without one node referencing the other.

5.4 Linking the function definitions of a partition

The dynamic linker takes care of providing the necessary function definitions when evaluation requires a partition to be decoded. The decoding of a partition consists of:

1. *the linking of its* compiled *function definitions*. The references between the compilation units stored in the repositories are *symbolic*. The dynamic linker for Clean resolves these references into *binary* references. This makes the function definitions executable. The linker optimizes by only linking the needed function definitions and its dependencies.
2. *the construction of the graph from its partition*. The graph consists of a set of nodes and each node references a function definition i.e. a Clean function or a data constructor. The *encoded* references to function definitions of each node are resolved in *run-time* references to the earlier linked function definitions.

The dynamic linker has the following additional tasks:

- It checks the equivalence of equally named type definitions. This is used during unification and to preserve the semantics of ordinary pattern matches. The Clean run-time system identifies data constructors by unique addresses in memory. In case of equivalent type definitions, it must be ensured that equally named constructors are all identified by a single unique address. Therefore the dynamic linker guarantees that:
  - There is only a single implementation for equally named and structural equivalent types.
  - All references to data constructors e.g. in dynamic pattern matches, point to that single implementation.
- It presents dynamics to the user as typed files abstracting from the complex representation of a dynamic. Section 8 discusses this topic in more detail.

6 Sharing of partitions

Partitioned dynamics may lose sharing. Efficiency can be increased by preserving sharing as much as possible. In this section we identify three cases in which sharing needs to be preserved. We conclude by discussing one solution for all cases to prevent loss of sharing.

6.1 Case 1: References between dynamics on disk

In this subsection, we show that sharing between dynamics on disk can be preserved. The example below extends the dynamic apply example by using the `readDynamic` and `writeDynamic` functions to perform I/O. The `fun`-dynamic from the file `application` (e.g. your favourite word-processor) and the `arg`-dynamic from the file `document` (e.g. the paper you are writing) are passed to the dynamic apply function `dyn_apply` which returns a new dynamic. The new dynamic is stored in the file result (e.g. a new version of your paper).

```
Start world
    # (fun, world) = readDynamic "application" world
    # (arg, world) = readDynamic "document" world
    = writeDynamic "result" (dyn_apply fun arg) world
where
    dyn_apply :: Dynamic Dynamic -> Dynamic
    dyn_apply (f :: a -> b) (x :: a) = dynamic (f x) :: b
    dyn_apply else1         else2    = abort "dynamic type error"
```

The function application of `fun` to its argument `arg` *itself* is packed into the dynamic because the dynamic constructor is lazy in its arguments. Since the evaluation of `fun` and `arg` is not required, the system does not read them in at all.

For this example only the first three steps of subsection 5.1 have to be executed to use the dynamic expressions. The reason is that the dynamic expressions `f` and `x` were never required. We preserve the sharing between dynamics on disk by allowing dynamic expressions to be *referenced* from other dynamics on disk.

As figure 2 shows, the dynamic stored in the file `result` contains two references to the `application` and `document` dynamics. To be more precise these references refer to the dynamic expressions of both dynamics. In general a dynamic is distributed over several files. Section 7 abstracts from this internal structure by permitting dynamics to be viewed as typed files.

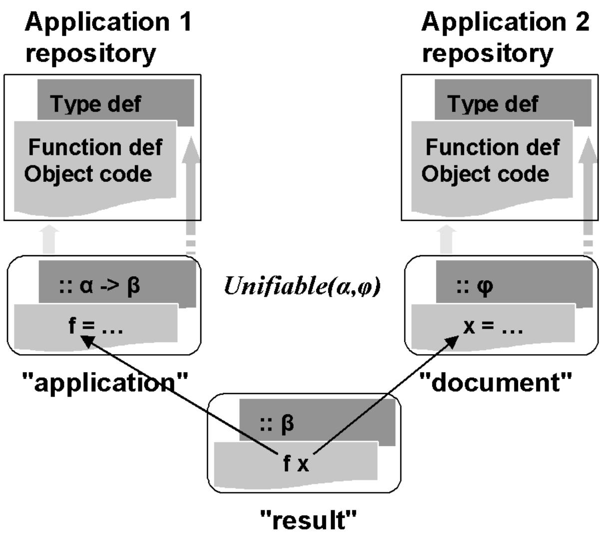

Fig. 2. Sharing between dynamics on disk after the dynamic apply.

6.2 Case 2: Sharing within dynamics at run-time

In this subsection, we show that the sharing of partitions at run-time can also be preserved. For example, the `Producer2` function of subsection 5.2 stores the dynamic `shared_dynamic` on disk. The stored dynamic is a pair of the two other dynamics `first` and `second`. The dynamic expressions of these nested dynamics both share the tail of a list called `shared_expr`.

The consumer application reads the dynamic written by the producer application. If the dynamic pattern matches succeed, the function returns the length of both lists.

```
Consumer2 :: *World -> *(Int, *World)
Consumer2 world
# (dyn,world) = readDynamic "shared_dynamic" world
= (g dyn,world)
where
    g :: Dynamic -> Int
    g ( (list1 :: [Int], list2 :: [Int]) :: (Dynamic,Dynamic) )
     = length list1 + length list2
```

The lists stored in the dynamic `shared_dynamic` are lazy: they are independently constructed from each other when evaluation requires one of the lists. The length of `list1` is computed after constructing its head (i.e. `1`) and its tail (which it shares with `list2`). Then the length of the second list is computed after constructing its head (i.e. `2`) and reusing the tail (shared with `list1`). In this manner sharing at run-time can be preserved.

In general, the order in which partitions are constructed is unpredictable because it depends on the evaluation order. Therefore partitions must be constructible in any order.

6.3 Case 3: Sharing and references within dynamics on disk

In this subsection, we show that sharing can also be preserved across dynamic I/O. It combines the preservation of sharing discussed in subsections 6.1 and 6.2. In contrast to the dynamic stored in the example of subsection 5.2, the first component of the dynamic stored by the `consumer_producer` function shown below has been completely evaluated. From the second tuple component *only* the tail which `list2` shares with the first component has been constructed. Its head is merely a reference to a partition of the dynamic `shared_dynamic`. Thus the newly created dynamic stores the same expression but in a more evaluated state.

```
consumer_producer :: *World -> *World
consumer_producer world
    # (dyn,world) = readDynamic "shared_dynamic" world
    # (list1,list2) = check dyn
    | length list1 <> 0          // force evaluation of list1
        # list_dynamic
```

```
            = dynamic (list1,list2)
        = writeDynamic "partially_evaluated_dynamic" list_dynamic world
where
    check :: Dynamic -> ([Int],[Int])
    check ( (list1 :: [Int], list2 :: [Int]) :: (Dynamic,Dynamic) )
        = (list1,list2)
```

The dynamic named `partially_evaluated_dynamic` is read by an also modified consumer example from the previous subsection. To compute the total length, it should only construct the head of the second list i.e. `2` because the shared tail expression constructed in the slightly modified consumer example of above can be *reused*.

To preserve sharing across dynamic I/O, the `consumer_producer` must also store on disk that the partition for the shared tail has already been constructed. *In this manner sharing within a dynamic can be preserved across dynamic I/O.*

6.4 A solution to preserve sharing during decoding

In this subsection we explain how dynamic expressions and dynamic types are decoded by inserting so-called decode-nodes for each dynamic expression or dynamic type while preserving sharing. A decode-node reconstructs its dynamic expression or its dynamic type when evaluated.

The decoding of a dynamic expression or a dynamic type may require the decoding of several partitions at once. For example, consider the `Consumer2` function of subsection 6.2: the dynamic expression `list1` extends over two partitions: a partition which contains the head and a partition containing its tail.

We have decided to construct a dynamic expression or a dynamic type in its entirety. For example when the function `length` is about to evaluate the list, the dynamic expression is constructed in its entirety by constructing the head and its tail from its two partitions. The other option would be to construct a partition at a time but this is not discussed in this paper.

Decode-nodes are implemented as *closure*-nodes i.e. a node containing an unevaluated function application to postpone evaluation until really required. Decode-nodes which refer to an entry node of the partition it decodes, are put at the lazy argument positions of the `dynamic`-constructor.
A decode node has the following arguments:

1. *An entry node of the* dynamic *partition.* Dynamic partitions are those partitions which are *directly* referenced from the lazy argument positions of the keyword `dynamic`. All other partitions are called *shared* partitions. An example of a shared partition is the partition for `shared_expr` of subsection 5.2.
2. *The list of entry nodes from already decoded partitions.* This list is used at run-time to preserve sharing within dynamics as discussed in subsection 6.2.

Upon decoding a partition via an entry-node, it is first checked whether the dynamic partition has already been decoded. In this case it is sufficient to return the address of the entry node to preserve sharing (see 6.2). Otherwise the dynamic partition must be decoded. After the shared partitions have been decoded in an appropriate order, the dynamic partition itself is decoded. The entry-nodes of decoded partitions are stored in the list of already decoded partitions. The address of the entry node of the dynamic partition is returned.

We now show how the sharing discussed in subsections 6.1 and 6.3 can be solved. To preserve the sharing of the former subsection, it is already sufficient to encode decode-nodes. To preserve the sharing of the latter subsection, the second argument of a decode-node must be encoded in a special way.

The dynamic `partially_evaluated_dynamic` of subsection 6.3 contains the *encoded* decode-node. The second argument of the decode-node only contains the shared tail `shared_expr` because it is shared between `list1` and the not yet decoded `list2`, and both lists are contained in the dynamic `list_dynamic`. In this manner sharing is preserved.

In general after encoding a dynamic d, the encoded second arguments of the decode-nodes of a nested dynamic n should only contain a subset of the already decoded partitions. This subset can be computed by only including those decoded partitions that are reachable from the decode-nodes of a dynamic n and leaving out the partitions which are not *re*-encoded in dynamic d. Therefore the list of encoded decode-nodes only contains those partitions which are already used within that newly created dynamic.

7 User view of dynamics on disk

The complexity of dynamics is hidden from the user by distinguishing between a user level and a hidden system level. Dynamics are managed as typed files by the users. Only for deletion and copying dynamics to another machine additional tool support is required.

7.1 The system level

This layer contains the actual system dynamics with the extension *sysdyn* and the executable application repositories with the extensions *typ* and *lib*. A system dynamic may refer to other system dynamics and repositories. These files are all hidden and managed by the dynamic run-time system. User file access is hazardous and therefore not permitted. For example, deleting the system dynamic `document` renders the system dynamic `result` unusable.

All system dynamics and system repositories are stored and managed in a single system directory. This may quickly lead to name clashes; dynamics need to have unique names within the system directory.

We have chosen to solve the unique naming problem by assigning a unique 128-bit identifier to each system dynamic. The *MD5*-algorithm in [12] is used to compute a unique identifier. The generated unique names of system dynamics and repositories are hidden from the user.

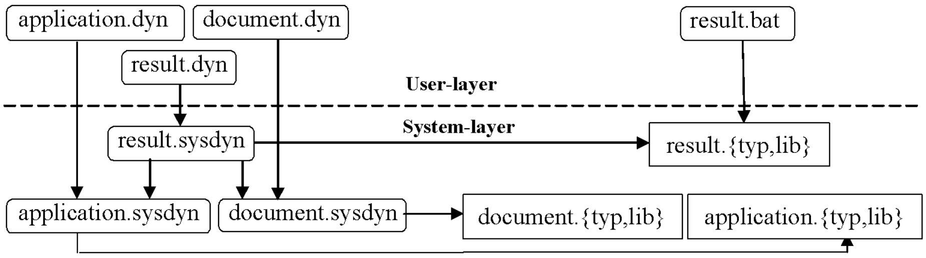

Fig. 3. System organization after executing the dynamic apply.

7.2 The user level

The user level merely contains links to the system layer. Links to system dynamics have the *dyn* extension and links to application repositories have the *bat* extension. These files may freely be manipulated (deleted, copied, renamed) by the user. This does not affect the integrity of the (system) dynamics.

Manipulation of user dynamics may have consequences for system dynamics and repositories, however. The following file operations need tool support:

- *The deletion of a user dynamic.* When the user deletes a user dynamic or a dynamic application, system dynamics and repositories may become garbage. These unreferenced dynamics and repositories can safely be removed from the system by a garbage collector. For example first deleting the user dynamic `document` does not create garbage in the system level but deleting the user dynamic `result` makes its system dynamic `result` garbage and also the system dynamic `document`.
- *The copying of a user dynamic to another machine.* When the user copies a user dynamic to another machine, its system dynamic and the other system dynamics and repositories it refers to, need to be copied too. The copying tool takes care of copying a dynamic and its dependencies. Using a network connection, it only copies dynamics and repositories not already present at the other end. The unique MD5-identification of dynamics makes this possible.

8 Related work

There have been several attempts to implement dynamic I/O in a wide range of programming languages including the strict functional language Objective Caml [6], the lazy functional languages Staple [7] and Persistent Haskell [3], the orthogonal persistent imperative language Napier88 [8], the logic/functional language Mercury [4] and the multiple paradigm language Oz [13]. In standard Haskell only language support for dynamics is provided; it has therefore not been considered. The table below compares the different approaches:

Dynamic	Clean	Ocaml	Staple	Napier88	Mercury	Pers. Haskell	Oz
native code	+	+	−	+	+	+	−
data objects	+	+	+	+	+	+	+
closures/functions	+	−	+	+	−	+	+
application indep.	+	+/−	+	+	+/−	+	+
platform indep.	−	+/−	−	−	+/−	−	+
network	+	+/−	−	−	+/−	−	+
lazy I/O	+	−	−	−	−	−	−

Within the table + and − indicate the presence or the absence of a left column feature for dynamic I/O. A slashed table entry discriminates between data objects on the one hand and closures/functions on the other hand. For reasons of clarity the slashed entries −/− and +/+ are represented by respectively − or +. The table lists the following features:

- *Native code.* The presence of this feature means that dynamics can use compiled function definitions i.e. binary code.
- *Data objects.* The presence of this feature means that data objects i.e. without functions or function applications can be packed into a dynamic and unpacked from a dynamic.
- *Closures/functions.* The presence of this feature means that also function objects and closures i.e. postponed function applications can be packed into a dynamic and unpacked from a dynamic.
- *Application independence.* The presence of this feature means that dynamics can be exchanged between independent applications.
- *Platform independence.* The presence of this feature means that a dynamic has a platform independent representation. This also applies to the representation of (compiled) function definitions it uses.
- *Network.* The presence of this feature means that a dynamic can be exchanged between different machines.
- *Lazy I/O.* The presence of this feature means that a dynamic can be retrieved in pieces when evaluation requires it. Only languages with a run-time mechanism to postpone evaluation can implement this.

Objective Caml restricts dynamic I/O on closures/functions to one particular application provided that it is not recompiled. The Mercury implementation is even more restrictive: it does not support I/O on closures/functions. All other languages support dynamic I/O for closures/functions.

The persistent programming languages Persistent Haskell, Napier88 and Staple do not address the issue of exporting and importing of dynamics between different so called persistent stores. As a consequence the mobility of a dynamic is significantly reduced.

Although the Clean implementation is not yet platform independent, dynamics can be exchanged among different Windows-networked machines. The Mozart programming system offers the language Oz, which supports platform independent dynamics and network dynamics because it runs within a web-browser. However, currently the Oz-language is being interpreted.

Currently only Clean supports lazy dynamic I/O. In non-lazy functional language there is no mechanism to postpone evaluation which makes it impossible to implement lazy I/O in these languages.

9 Conclusions and future work

In this paper we have introduced dynamic I/O in the context of the compiled lazy functional language Clean. We have presented the most interesting aspects of the implementation by means of illustrative examples. Our implementation preserves the semantics of the language and in particular laziness and sharing. Acceptable efficiency is one of the main requirements for the design and implementation work and has indeed been realized. The resulting system described in this paper hides its complexity by offering a user-friendly interface. It allows the user to view dynamics as typed files. The resulting system is as far as we know unique in the world.

Dynamic I/O already has some interesting applications. A few examples: our group is working on visualizing and parsing of dynamics using the generic extension of Clean, extendible user-interfaces are created using dynamics, a typed shell which uses dynamics as executables has been created as a first step towards a functional operating system and a Hungarian research group uses dynamics to implement proof carrying code.

The basic implementation of dynamic I/O is nearly complete. However, a lot of work still needs to be done:

- *Increase of performance.* The administration required to add function definitions to a running application is quite large. By sharing parts of repositories e.g. the standard environment of Clean between dynamics, a considerable increase in performance can be realized.
- *Language support.* Several language features are not yet supported. These features include overloading, uniqueness typing, and abstract data types. Especially interesting from the dynamic I/O perspective are unique dynamics which would permit destructive updates of dynamics on disk.
- *Conversion functions.* The rather restrictive definition of type equivalence may result in problems when the required type definition and the offered type definition only differ slightly. For example, if a demanded `Colour`-type is defined as a subset of the offered `Colour`-type, then it would be useful to have a programmer defined conversion function from the offered type to the demanded type. Generics in [2] could help here.
- *Network dynamics.* In order to realize network dynamics both platform independence and the port of the implementation to other platforms are required.
- *Garbage collection of dynamically added function definitions.* This is a generalization of heap-based garbage-collection as used in functional languages. Traditionally only the heap area varies in size at run-time. Dynamic I/O makes also the code/data-areas grow and shrink. To prevent unnecessary run-time errors due to the memory usage of unneeded function definitions, garbage collection is also needed for function definitions.

10 Acknowledgments

We are grateful to Marco Pil for the more theoretical foundations on which dynamic I/O has been designed and implemented. We would also like to thank the Clean-group for taking part in discussions and for making numerous small changes all over the system, the Hungarian students Mátyás Ivicsics and Zoltan Várnagy for their contributions and the anonymous referees who reviewed this paper.

References

1. M. Abadi, L. Cardelli, B. Pierce, D. Rémy. *Dynamic typing in polymorphic languages*, Journal of Functional Programming 5(1):111-130, Cambridge University Press 1995.
2. A. Alimarine and R. Plasmeijer, *A Generic Programming Extension for Clean*, In Proc. of Implementation of Functional Languages, Älvsjö, Sweden, Arts Th., Mohnen M., Springer-Verlag, LNCS 2312, pages 168-185.
3. T. Davie, K. Hammond, J. Quintela, *Efficient Persistent Haskell*, In: Draft proceedings of the 10th workshop on the implementation of Functional Languages, pp. 183-194, University College London, September 1998.
4. F. Henderson, T. Conway, Z. Somogyi amd D. Jeffery, *The Mercury language reference manual*, Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996.
5. S. P. Jones and J. Hughes (eds.) (1999), *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language.*
6. X. Leroy, D. Doligez, J. Garrigue, D. Rémy and J. Vouillon *The Objective Caml system release 3.04 Documentation and user's manual*, December 10, 2001 Institut National de Recherche en Informatique et en Automatique.
7. D. McNally, *Models for Persistence in Lazy Functional Programming Systems*, PhD Thesis, University of St Andrews Technical Report CS/93/9, 1993.
8. R. Morrison, A. Brown, R. Connor, Q. Cutts, A. Dearle, G. Kirby and D. Munro. *Napier88 Reference Manual (Release 2.2.1)*, University of St.Andrews, July 1996.
9. M.R.C. Pil, (1997) *First Class I/O*, In Proc. of Implementation of Functional Languages, 8th International Workshop, Selected Papers, Bad Godesberg, Germany, Kluge Ed., Springer Verlag, LNCS 1268, pp. 233-246.
10. M.R.C.Pil, (1999), *Dynamic types and type dependent functions*, In Proc. of Implementation of Functional Languages, London, UK, Hammond, Davie and Clack Eds. Springer-Verlag, Berlin, Lecture Notes in Computer Science 1595, pp 169-185.
11. R. Plasmeijer, M.C.J.D. van Eekelen (2001), *Language Report Concurrent Clean, Version 2.0 (Draft)* Faculty of mathematics and Informatics, University of Nijmegen, December 2001. Also available at www.cs.kun.nl/ clean/Manuals/manuals.html
12. R.L. Rivest, RFC 1321: *The MD5 Message-Digest Algorithm*, Internet Activities Board, 1992.
13. P. van Roy and S. Haridi, *Mozart: A Programming System for Agent Applications*, International Workshop on Distributed and Internet Programming with Logic and Constraint Languages, Part of International Conference on Logic Programming.