First Class File I/O

Marco Pil

Computing Science Institute, University of Nijmegen,
Postbus 9010, 6500 GL Nijmegen, The Netherlands
e-mail: marcop@cs.kun.nl

Abstract. In most functional languages little attention has been paid to file I/O. The file system is poorly typed at best and some classes of objects, in particular functions, cannot be stored on disk at all. In this article we present a mature type system for typing files. We also discuss briefly how we plan to implement the storage of functions in files. We make use of the concept of dynamic types, as introduced in Abadi et al., which provide an interface between statically and dynamically typed parts of a program. We have modified the concept of dynamic types to include polymorphic types in a natural way and we are implementing this modified system of dynamic types in the functional language *Clean*. We have developed a simple run-time matching algorithm for the dynamic type checks. We present some of the problems we encountered when the system of dynamic types is implemented to its full extent in a language that already has an elaborate type system. Finally we show that the same concepts that we used for constructing the file system can be used for communication between independently running programs in general.

1 Introduction

In the framework of a functional operating system, an operating system that is specified and implemented in a functional language, one of the main problems is the communication between independent processes (programs). When programs communicate with other programs, or possibly with incarnations of themselves, flexibility is demanded. Programs do not have to know about their mutual existence and can be terminated before others are launched. Good examples of such communication are asynchronous message passing and file I/O.

In this article we will focus mainly on file I/O, but the methods we discuss and the tools we present will be equally applicable to other forms of communication between independent programs.

When processes communicate, both sender and receiver must abide by the same set of mutually known agreements: the communication protocol. The agreement most commonly made for file I/O is that files consist of series of characters. All data that is written to and read from files must be coded in a series of characters.

Obviously, this protocol is very primitive: for all datatypes the user has to define encoding and decoding mechanisms. Furthermore it is hardly possible to store functions in files.

In functional languages functions are viewed as 'first class citizens'. They can be passed to other functions or yielded as a result, just like any other 'first-order' value But, in contrast to their 'first class' status, they cannot be stored on disk like the basic datatypes. Overcoming this imperfection of the language would be a real improvement, not only for reasons of orthogonality. Functions in files occur often in everyday programming. An executable, for instance, can be viewed as just being a function: it would be nice to be able to apply the contents of one file — a compiler — to the contents of another file — the source program — yielding yet another file — a new executable.

In concept a function definition cannot be treated as an object in the program in which it is defined. To simulate function manipulation, we have to settle for a *description* of its function definition. This description can be read from file and must then be interpreted by the program. In fact, this principle holds for functions in memory too: only a description of the function has been stored, which is interpreted by the CPU. In order to make the interpretation by the program efficient, we can choose the representation of functions in files in such way that the interpretation does not have to be done by the program, but can be done by the CPU right away.

The type system might provide the more flexible protocol we are looking for. Types can ensure that an action such as the one described above is safe: the data that is interpreted as a function will at least be a representation of a function of the right type.

Clean has an elaborate static type system, based on a combination of Hindley-Milner and Mycroft typing, extended with uniqueness types, existential types, and type constructor classes. We would like to equip the file system with a comparably rich type system, in which objects can be stored without having to convert them to series of characters explicitly. This leads to the formulation of our goal: *the construction of a strongly typed file system in which any object may be stored that can be expressed in the* Clean *programming language.* The objects in memory should become fully interchangeable with objects in files.

In this article we present an extension of the type system of Clean with a form of dynamic typing such that the file system can be typed properly (section 3). Most of that is due to Abadi, Cardelli & Pierce [ACP89,ACP91] and Leroy & Mauny [LM93]; we simplified deduction rules and the matching algorithm and integrated dynamics in Clean's current type system. We discuss why 'dynamic' types serve our purposes best (section 2.1) and how we are implementing them in Clean (section 4). In section 4.2 we discuss how we plan to implement the storage of functions in files.

2 Typing the file system

Types can be used in a protocol for performing file I/O. For this we must make the types of the objects in the files explicit. In a file exactly one object can be stored. We say that the file has type $\mathsf{File}\ \sigma$ iff the object stored in the file has type σ. This type should be stored together with the file in some way. An object can only be read from a file if its type component matches the type that was expected by the context defined by the program.

Typing file I/O differs from typing ordinary expressions because there is no static link between the name of a file and its contents and thus its type. Due to this fact, types of the parts of a program that perform file I/O can only be checked at run-time, when reading or writing. For this reason a certain amount of dynamic typing is unavoidable.

Typing file I/O is not the only problem area in which dynamic typing is a necessity. In general, programs that communicate data with their environment cannot be typed statically.

But we do not want our entire language to become dynamically typed. Statically typed programming languages allow earlier error checking, and generation of more efficient object code than languages where all type consistency checks have to be performed at run-time.

We are looking for a *hybrid* type system in which both forms of type checking can coexist in a sound way, offering the advantages of both type systems. An interface has to be defined between the statically and dynamically typed parts of the program. The compiler must be able to distinguish between those values whose types may be needed in dynamic type checks during execution and those values whose types can be discarded after compilation.

In [ACP89], Abadi et al. introduce such an interface. It is based on the construction of pairs of an object and its type. Such a pair is called a 'dynamic'. Access to the components of a dynamic is limited to a pattern match, where the type component of the dynamic is compared with a given type pattern. The interface provides a carefully designed but subtle separation of the statically and dynamically typed parts of the program: pattern matches specify the dynamic checks, while the static checks ensure that a program is correctly typed, independent of the results of dynamic type checks in the pattern matches.

A type system that types the file system can also be used to solve other typing problems. In concept, message passing between independently running processes can be seen as a particular instance of file I/O. The messages can be viewed as short-lived files which are created by the sending process and destroyed by the receiving process. In this way, our typing mechanism for typing files can be reused to type message passing.

Although in the actual implementation messages are probably more efficiently transferred than through files, this analogy provides a good theoretical model for message passing and eases its implementation.

In section 3 we present the hybrid type system, which is very powerful but complex. Before going into thecnical details we first discuss why we need such a powerful and hence complex system in the first place.

2.1 Why dynamic types?

In this section we discuss why dynamic checks are unavoidable and why it is convenient to separate file access from dynamic type checking, as can be done with dynamics. We start with a less complex type system, in which file access and type checking are viewed as a single atomic action, and show that its power is insufficient to satisfy our demands.

We aim for a file system in which any file may contain any object that can be expressed in our language. A simple read or write statement should suffice to copy an object from memory to disk or vice versa. One could think of these functions as:

```
class store    a :: Filename a *Disk → (Status,    *Disk)
class retrieve a :: Filename   *Disk → (Status, a, *Disk)
```

The function store has a filename and an object as input and puts the object on disk, under the given filename. The retrieve function copies an object from disk to memory. Both functions return a Status value in with error messages can be found. The $*$ before a type, in this case $*$Disk, expresses that the function demands unique (i.e. single threaded) access to this argument. Clean's type system takes care that at run-time the actual argument is indeed single threaded, i.e. there are no other references to that argument. In [BS93] more information about uniqueness types can be found.

Disk access should be type safe. This means that if a program is specified to read an object of a certain type, then it must be guaranteed that the object which is read during execution has that same type.

If we want to statically check this type requirement, we have to determine at compile-time which filename comes with which type during the execution of the program. As mentioned before, this generally cannot be determined at compile-time, since the filename of a file or its contents may change between compilation and execution of the program. Reliable information is only obtained during the execution of the program, when the file is read. This shows that run-time type checks are necessary when dealing with the file system.

Since Clean programs are compiled rather than interpreted, it is hard, sometimes even impossible, to reconstruct the correct type of an object at run-time. For the purpose of dynamic type checks, it is therefore necessary to preserve some type information together with the object. This should be stored with the object, in the file. The type information must be supplied by the write function and used by the read function to perform a run-time type check.

In such a system, type checking and disk access cannot be seen separately. They are combined in one atomic action. This causes some problems. Consider the following function:

```
copyfile :: Filename Filename *Disk → *Disk
copyfile src_name dest_name disk = newDisk
    where
    (stat1, object, disk1) = retrieve src_name disk
    (stat2, newDisk) = store dest_name object disk1
```

The function `copyfile` specifies that an object (`object`) is copied from one file to another. The object can be of arbitrary type, the program poses no restrictions on the type of '`object`'. This causes trouble for the `store` function. It cannot determine what type it has to write with its object unless some type information remains after transferring the object from disk to memory. Somehow the type information should be passed from the source file to the destination file.

Now consider the following function. It reads two objects from files and applies the first to the second:

```
apply :: Filename Filename *Disk → (Int, *Disk)
apply f_name x_name disk
        | (status1==OK) && (status2==OK) = (f x, disk2)
        | otherwise                      = (0,   disk2)
    where
    (status1, f, disk1)  = retrieve f_name disk
    (status2, x, disk2)  = retrieve x_name disk1
```

In contrast to the previous program, this one does pose restrictions on the types of the objects that are to be read. Static analysis shows that the object f should have a function type: $X \rightarrow Int$, and the object x should have that same type X, or an instance thereof. The type requirements for retrieving x depend on the type of f. Somehow the type information that is stored with f must be administered during the execution of the apply function, so that it can be compared with the type that is stored with x.

Both cases show that type information has to persist beyond the scope of the read and write functions. For some objects type information has to be dragged along during the execution of the program. Since Dynamic types consist of a pair of an object and its type, they are a perfect tool to accomplish this.

3 Introducing a Dynamic type

In this section we present a type called Dynamic, first introduced by Abadi et al. in [ACP89]. We modify its syntax a bit to fit into our language and extend it to cover polymorphism, much like Leroy and Mauny did in CAML [LM93]. Furthermore, we integrate the type in Clean's type system.

We first introduce the notion of dynamic types informally. Then we give a formal definition of its syntax (section 3.1) and we extend Clean's type system with new type rules (section 3.2). These new rules turn out to be very simple. Finally, in section 3.4 we specify the run-time semantics of the dynamic constructs. These too are simple.

The values of Dynamic are, roughly speaking, pairs of a value and a type, such that the value component can be typed with the type component. The access to objects of type Dynamic is limited. A *dynamic* (a value of type Dynamic) can be built from a value and its type using the "dynamic" construct, as shown in the following examples (Note that all four expressions are of type Dynamic, the "::"-symbol is used to separate the value of the dynamic construct from its type argument):

```
dynamic 3 :: Int
dynamic (dynamic 3 :: Int) :: Dynamic
dynamic reverse :: [a] → [a]
dynamic reverse :: [Int] → [Int]
```

Statically it is checked that the specified value (e.g. 3) can indeed be typed with the specified type (e.g. Int). Note that this type may be polymorphic and/or an instance of the most general type of the value.

The value stored in a dynamic can only be retrieved after a successful match against the type component of the dynamic. When the match succeeds, the contents of the dynamic can be used in the expression which is guarded by the match.

The following functions are examples of such type pattern matches:

```
f :: Dynamic → Int
f  (n::Int)    =  n ∗ n
f  dn          =  0

g :: Dynamic → [Int]
g  (f::[a]→[a])   =  f [1,2,3]
g  df             =  [ ]

h :: Dynamic Dynamic → Dynamic
h  (f::X→Y)   (x::X)   =  dynamic (f x) :: Y
h  df         dx       =  dynamic "Error" :: String
```

Each time the first function (f) is evaluated, it checks whether its dynamic argument contains an integer. If this is the case, the function returns the square of the integer value. Otherwise f yields 0.

The type [a]→[a] in the function g expresses polymorphism. In other formalisms this type is sometimes written as $\forall$a.[a]→[a]. The function g only enters the body of the first alternative if its Dynamic argument contains a polymorphic function that maps lists to lists. Otherwise it will return the empty list.

In the last example special variables are used (the uppercase variables X and Y). These are called "type-pattern variables". During the evaluation of the function these variables are bound to parts of the contents of the type field of its Dynamic argument. Type pattern variables have the same scope as the ordinary variables that occur on the left hand side of a rule alternative (i.e. the whole rule alternative).

The function h takes two dynamic values. It demands that the first one contains a function of type X→Y, for some X and Y. Then it checks (at run-time) that the second dynamic contains a value of the same type X. If both requirements are met, the function h builds a new dynamic whose value component is formed by the application of the value components of the arguments of h. The type component of the new dynamic is Y. Note that such a type pattern variable can only be used to form a new dynamic. It can never be used in the type of the function's result.

3.1 Formal syntax of 'dynamic' expressions

In this section we define the constructs formally, starting with their syntax. To link up with the Clean syntax we modified the syntax as specified in [ACP89].

```
<dynamic-expr> = 'dynamic' <expr> [ '::' <type> ]


<type-pattern> = 'case' <expr> 'of'
                        '|' <termvar_1> '::' <type_1> '=' <expr_1>
                         ⋮
                        '|' <termvar_n> '::' <type_n> '=' <expr_n>


<type> = <type-variable>
       | <type-pattern-variable>
       | <type> '→' <type>
       | <type-constructor> {<type>}*
```

Note that in the *<dynamic-expr>* the specification of a type is optional. The type σ in the expression (d*ynamic* $e :: \sigma$) can be omitted, since Clean's type checker is able to derive the most general type for e. This will then be stored as the type component of the dynamic.

3.2 Extending the type system

The type rules for typing Clean expressions have to be extended with the new constructs for packing and unpacking dynamics. Clean's set of deduction rules consist of statements of the form:

$$\mathcal{F}, B \vdash e : \sigma$$

which states that the expression e can be typed with type σ, given a function environment $\mathcal{F}$ and a basis B. As usual, as basis B consists of type declarations. The function environment $\mathcal{F}$ consists of type declarations of function definitions. Contrasting the types appearing in the basis, the latter types are assumed to be universally quantified, which means that a proper instance of the given function type can be used each time the function is applied. (See [BS93] for more details.)

To Clean's set of rules, two new deduction rules are added for introducing and eliminating values of type Dynamic respectively.

An expression (dynamic $e :: \sigma$) is of type Dynamic if the expression e can be typed with σ:

$$\frac{\mathcal{F}, B \vdash e : \sigma \qquad V(B) \cap V(\sigma) = \emptyset}{\mathcal{F}, B \vdash (\mathsf{dynamic}\ e :: \sigma) : \mathsf{Dynamic}} \quad \text{(DynI)}$$

The type rule for eliminating a Dynamic is somewhat more complicated. This is due to the possible presence of type pattern variables. They are used to enable type information to persist beyond the scope of the read and write function.

A type pattern match is typed correctly if the selector-expression is of type Dynamic and each entry is correctly typed for every possible substitution of its type-pattern variables.

$$\frac{\mathcal{F}, B \vdash e : \mathsf{Dynamic} \qquad \forall \widehat{\ } : (\mathbf{X}_{\mathtt{i}} \to \mathbb{T}).\ \ \{\mathcal{F},\ x_i : \widehat{\sigma_i}\}, B\ \vdash\ \widehat{e_i} : \tau}{\mathcal{F}, B\ \vdash \begin{pmatrix} \mathsf{case}\ e\ \mathsf{of} & | \ \{\mathbf{X_1}\}(x_1 :: \sigma_1) = e_1 \\ & \vdots \\ & | \ \{\mathbf{X_n}\}(x_n :: \sigma_n) = e_n \end{pmatrix} : \tau} \quad \text{(DynE)}$$

where i ranges over $\{1 \ldots n\}$. (Note that bold characters denote vectors.)

The expressions $\{\mathbf{X_i}\}$ in the branches of the case-expression do not belong to the syntax of the language. They are used in this type rule to make the type pattern variables that occur in σ_i and e_i visible. The function $\widehat{\ }$ maps the type pattern variables to types.

The assertion $x_i : \widehat{\sigma_i}$ is placed in the function environment rather than in the basis. In this way we handle polymorphism: a polymorphic pattern is treated in the same way as a polymorphic function. By placing the assertion in the function environment, the type becomes implicitly universally quantified. So for each application of x_i in e_i the free variables in

$\widehat{\sigma_i}$ are taken fresh.

3.3 Uniqueness and Dynamic types

Treatment of Dynamic types in Clean's uniqueness type system is a subtle matter. With respect to uniqueness Dynamic types behave in the same way as higher-order functions do. A dynamic containing a unique expression must itself be unique as well. But more: it is *necessarily* unique. We cannot allow a unique dynamic to coerce to its non-unique variant. If we allow a dynamic which contains a unique expression to be copied, we can no longer guarantee that the object remains unique during evaluation.

3.4 Run-time type checks

The type rules above are used in static type checking. In this section we present a mechanism for dynamic type checks.

Each time the run-time system evaluates a type pattern match, it should perform a run-time type check as part of the match. The current contents of the type component of a dynamic value (Tag) are compared with the pattern specified in a type pattern match (Pattern). The run-time matching algorithm is specified by the Clean function below. For this purpose the types are encoded in an algebraic datastructure. The Environment is used to store substitutions for variables. The function sub stores a substitution in the Environment, and the function val retrieves a substitution again. The function unify takes care of the necessary run-time unification.

```
match :: Tag        Pattern      Env → (Bool, Env)
match    INT        INT          env = (True, env)
match    type       (PattVar X)  env = unify type (val X env) env
match    (Var a)    type         env = (True, sub a type env)
match    *A_T       *A_P         env = match A_T A_P env
match    (C A_T)    (C A_P)      env = match A_T A_P env
match    (A_T → B_T) (A_P → B_P) env = (b1 && b2, env2)
                    where
                    (b1,env1) = match A_T A_P env
                    (b2,env2) = match (val B_T env1) (val B_P env1) env
```

The algorithm ensures, amongst other things, that no substitution occurs for type variables in the pattern (which denote real polymorphism):

the type tag must be polymorphic in at least the same positions as the pattern is. So the function

```
f :: Dynamic → [Int]
f  (g::[a]→[a])   =  g [1,2,3]
f  dg             =  [ ]
```

will only enter the first alternative if the actual argument of f is a polymorphic function whose type can be specialized to [a]→[a]. For example the expression f (dynamic I :: a→a) reduces to [1,2,3], but f (dynamic I :: [Int]→[Int]) reduces to [ ].

Type pattern variables can be used to write functions for dynamic values of arbitrary type. The function

```
f :: Dynamic → (Dynamic, Dynamic)
f  (x,y)::(U,V)   =  (dynamic x :: U, dynamic y :: V)
f  else           =  (dynamic "Error", dynamic "Error")
```

will choose the first alternative whenever the dynamic argument is a tuple. It poses no restriction on the types of the separate parts of the tuple.

Type pattern variables extend the expressive power of the language but they have a drawback. They induce, as specified in the match function, a run-time unification. This is the price that has to be paid for a flexible type system.

4 Implementation of Dynamics

In this section we discuss how dynamics can be implemented in Clean. Three things have to be accomplished: types have to be encoded; a mechanism is needed to preserve them during the execution of a program; and the matching algorithm has to be inserted in Clean's run-time system (section 4.1). Finally we discuss how we plan to implement file I/O with dynamics (section 4.2).

In the previous sections, no distinction was made between complex types (algebraic types, record types, list types, etc). They were all represented by the case <*type-constructor*> {<*type*>}*. In this section their difference is made explicitly, since the implementation of the various types differs slightly.

4.1 Encoding the types

The representation of a dynamic value consists of a representation of a value and a representation of its type. Thus for the construction of a dynamic, we have to encode types into a form which is preserved during execution. We should make a distinction between the use of dynamics within the boundaries of a single program and the use that goes beyond that boundary.

When we use dynamics within the boundaries of a single program, where all type definitions are known throughout the entire program, a simple encoding of types suffices. The following example shows such an encoding (not all types are listed, for reasons of brevity).

$$\begin{array}{ll}
\text{Code}(\mathsf{Int}) & = \texttt{Int} \\
\text{Code}(\mathsf{a}) & = \texttt{Var a} \\
\text{Code}(\mathsf{A}\rightarrow\mathsf{B}) & = \texttt{Arrow}\ \text{Code}(\mathsf{A})\ \text{Code}(\mathsf{B}) \\
\text{Code}([\mathsf{A}]) & = \texttt{List}\ \text{Code}(\mathsf{A}) \\
\text{Code}(*\mathsf{A}) & = \texttt{Unique}\ \text{Code}(\mathsf{A}) \\
\text{Code}(\mathsf{T}\ \mathsf{A}_1 \ldots \mathsf{A}_n) & = \texttt{Algebraic "T"}\ \text{Code}(\mathsf{A}_1)\ \ldots\ \text{Code}(\mathsf{A}_n) \\
 & = \texttt{Record "T"}\ \text{Code}(\mathsf{A}_1)\ \ldots\ \text{Code}(\mathsf{A}_n) \\
 & = \texttt{Abstract "T"}\ \text{Code}(\mathsf{A}_1)\ \ldots\ \text{Code}(\mathsf{A}_n)
\end{array}$$

For user-defined types this form of coding is inadequate when the use of dynamics goes beyond the boundary of a single program. The representation of these types no longer contains enough information to decide outside an individual program whether two types should be considered to be equal. An algebraic datatype Colour, for example, can differ in definition from one program to another, it can be defined as, say,

$$\mathsf{Colour} = \mathsf{Red} \mid \mathsf{Blue} \mid \mathsf{Green}$$

in the first program, and

$$\mathsf{Colour} = \mathsf{Red} \mid \mathsf{Yellow} \mid \mathsf{Blue}$$

in the second. We do not want to consider these types equal.

Within one program it is sufficient to test type equality by equality of name. Beyond the boundaries of one program some information about the definition of the type is essential. This difference between *name equivalence* and *structural equivalence* is discussed extensively in [CBC90]. They conclude that both name and structural equivalence checks are needed for checking persistent types both adequately and efficiently.

A new version of the 'Code'-function incorporates both notions of equivalence:

$$
\begin{array}{ll}
\text{Code}(\mathsf{Int}) & = \texttt{Int} \\
\text{Code}(\mathsf{a}) & = \texttt{Var a} \\
\text{Code}(\mathsf{A}{\rightarrow}\mathsf{B}) & = \texttt{Arrow}\ \text{Code}(\mathsf{A})\ \text{Code}(\mathsf{B}) \\
\text{Code}([\mathsf{A}]) & = \texttt{List}\ \text{Code}(\mathsf{A}) \\
\text{Code}(*\mathsf{A}) & = \texttt{Unique}\ \text{Code}(\mathsf{A}) \\
\text{Code}(\mathsf{T}\ \mathsf{A}_1 \ldots \mathsf{A}_n) & = \text{Definition}(\mathsf{T})\ \text{Code}(\mathsf{A}_1)\ \ldots\ \text{Code}(\mathsf{A}_n)
\end{array}
$$

where “Definition(T)” is some representation of the definition of T.

For *abstract datatypes* a mechanism has to be implemented that obtains the actual implementation of the datatype from its defining module. The mechanism of overloading could provide a solution here. One can require that for all types a packing function must be available (i.e. for each datatype its definition must be somehow obtainable). A module that introduces an abstract datatype must provide a function that packs a value of that datatype as well.

The type T may depend on other datatypes. It may even depend (indirectly) on itself. Therefore the representation cannot be obtained by simply replacing the name of the type by its definition.

The functional language Clean is based on graph rewriting. It turns out to be profitable to represent the type component of a dynamic as a graph. Variables that occur more than once in a type can be shared. Representing types as graphs has two advantages.

Firstly, the representation of definitions of datatypes can be constructed easily with the use of graphs. Recursive dependency can be represented by cyclic pointer structures.

The second advantage is that all tools that are available for graph rewriting can now be used for handling the types as well.

The matching algorithm can be programmed directly into Clean's run-time system. Two graphs representing the types are compared with each other. Since the variables in a type are shared, a substitution of a variable during the match can be done in place. This means that in the actual implementation we get rid of the Environment as used in the matching algorithm.

4.2 Disk Access

A feature which we get almost for free now is writing and reading dynamics to and from disk. Since Dynamics are just graphs, it is possible to write and read files simply by copying graphs from memory to disk and vice versa. Copying a graph is a simple, basic operation in a graph rewrite system, which can be performed quickly, and uniformly for all representable objects.

For functions this mechanism does not work, however. This is because of the way functions are represented in the graph. They are again not treated as other types, in this respect. The graph that represents the function contains pointers into the *program store* (the part of Clean's run-time system that contains blocks of executable code).

If a function that is written by a program was only read by the same instantiation of the same program, this difference in representation would not lead to a problem. One could simply store the pointer into the program store as a representation of the function. But files are designed to outlive the instantiation of the program that created them. Moreover they tend to be shared by *more than one program*, possibly over *different platforms*. A solution must be found in more complex forms of representation.

We would like to stress here that we are only interested in efficient solutions. Clean's language and its compiler are focussed on an efficient implementation of the programs written in Clean. The representation of a function by its source code, which can be interpreted at run-time, is not a satisfactory solution. In our opinion, the interpretation of source code would have an unacceptable effect on the efficiency.

If files are known to be used on one kind of platform only, a function can be represented by its blocks of executable code. Mechanisms to copy those blocks of code are not yet available in Clean. A problem is that inside the block arbitrary jumps in the code can be made. It may turn out that the whole contents of the program store is necessary in order to evaluate one function. Furthermore, when blocks of code are to be read again, the executable code has to be linked dynamically to the existing program.

When files are used over different platforms, we have to settle for a less efficient representation. The function description must be sufficiently general to be dynamically translatable into proper executable code on any target machine. A suitable representation might be in the intermediate code of the abstract ABC-machine. This ABC-code can be compiled dynamically to the target machine. It is of course not very efficient, but a lot of flexibility is gained.

Although both solutions are hard to implement, we do not expect fundamental problems. Dynamic linking is daily business in almost every large operating system. Dynamic compilation might slow down the execution a bit, but the slowdown is negligible compared to the time that is spent due to sending the information over a network.

5 Related work and Conclusions

Abadi et al. present in [ACP89] a theoretical framework for Dynamic types, complete with an extensive review of the history of dynamic typing in statically typed languages. They also mention a large number of languages in which restricted use has been made of the dynamic concept. In [ACP91] this theoretical framework is extended to cover polymorphism. Higher order pattern variables are introduced to be able to match against polymorphic types. We decided to deal with polymorphism differently and use ordinary type variables to accomplish this (section 3).

Leroy and Mauny describe in [LM93] how a simple version of Dynamic is implemented in ML. They also specify a more complicated system of dynamics with universally and existentially quantified variables, which has not yet been implemented. It uses the existentially qualified types to specify dynamic matches with incomplete type information. These are basically the same as our type pattern variables. Their type system of "Mixed quantifications", in which they use both polymorphism and type pattern variables, is as powerful as the type system that is presented in this article, but our implementation, in particular the run-time matching algorithm, is much less complicated (section 3.4). Furthermore we showed how the type Dynamic can be integrated in a complex type system.

Morrison et al. present in [MBC89] the type system as implemented in Napier, a language which is specially developed to handle persistent objects. They use the type any to pack and unpack values with their types. The difference between this type any and our type Dynamic is in the fact in Napier, with any, the difference between statically and dynamically checkable values is left more implicit.

A nice feature of Dynamic types and our way of viewing the file system is that we can use the extended type system for typing message passing between independently running processes as well (section 2).

We have not reached our goal yet: *the construction of a strongly typed file system in which any object may be stored that can be expressed in the* Clean *programming language.* But we have made a lot of progress. We have designed a suitable extension of Clean's type system. We are able to type the file system, and we are extending the Clean compiler accordingly. Currently we are developing the mechanisms for storage of functions in files.

References

[ACP89] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin. *Dynamic typing in a statically typed language,* ACM Transactions on Programming Languages and Systems 13(2):237–268, 1991.

[ACP91] M. Abadi, L. Cardelli, B. Pierce, D. Rémy. *Dynamic typing in polymorphic languages,* In: Journal of Functional Programming 5(1):111–130, Cambridge University Press 1995.

[BS93] E. Barendsen and J. Smetsers. *Uniqueness Typing for Functional Languages with Graph Rewriting Semantics,* to appear in MSCS.

[CBC90] R. Connor, A.Brown, Q Cutts, A. Dearle, R. Morrison and J. Rosenberg. *Type Equivalence Checking in Persistent Object Systems,* In: Implementing Persistent Object Bases pp.151–164, Morgan Kaufman 1990

[PE93] M. van Eekelen, M. Plasmeijer. *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley 1993.

[PE97] M. van Eekelen, M. Plasmeijer. *Concurrent Clean 1.0 language report,* Computing Science Institute, University of Nijmegen, http://www.cs.kun.nl/~clean/Clean.Cleanbook.html.

[LM93] X. Leroy, M. Mauny. *Dynamics in ML,* In: Journal of Functional Programming 3(4), Cambridge University Press 1995.

[MBC89] R. Morrison, A.Brown, R. Carrick, R. Connor, A. Dearle, and M. Atkinson. *The Napier Type System,* In: Persistent Object Systems pp.3–18, Springer-Verlag 1989.