Interactive Combining of Typed Object Code

ICIS Technical Report

Rinus Plasmeijer, Sjaak Smetsers Arjen van Weelden

Abstract

The functional language Clean includes a hybrid type system: in addition to the traditional static type system it offers support for dynamically typed values, so called *dynamics*. An expression of arbitrary static type can be packed into a dynamic, and via pattern matching on types a dynamic can be converted back to a value of statically known type. The most important aspect of dynamics is that they can be serialized, written to file, and exchanged between independently compiled applications in a type-safe way. Since dynamics may contain functions and closures, they can be regarded as type-safe plugins. An application can combine plugins in a type safe way to new ones, without having to know what the actual contents and types of the original plugins are. In this paper we explain the underlying implementation and the architecture of the run-time system needed to make this all possible. To show the expressive power of Clean's dynamics we present an interactive shell, called Esther. The command line language of this shell provides the basic functionality of a strongly typed lazy functional language. Esther behaves like an interpreter (direct response) yet its resulting performance is comparable to compiled code. This is achieved by using the dynamic facility of Clean. The shell actually combines compiled code; no help from a compiler is needed. Moreover, type checking is for free using the run-time type unification of dynamics. In this way we have obtained a novel architectural framework in which one can freely mix functionality created by the compiler, functionality stored in dynamics by compiled applications, and functionality interactively created by using the shell command line interpreter.

Contents

1 Introduction

Part of the success of many popular applications, such as web-browsers, spreadsheets, data base management systems, but also more dedicated software such as editors, email clients and media players, is their ability to adapt themselves to specific demands of their users. These applications offer facilities to extend or change their functionality dynamically i.e. without the necessity to rebuild the entire application each time an adjustment takes place. In principal it is sufficient to provide the user with some basic instance of the application and extend it dynamically depending on the facilities or features the user wants.

In essence dynamic extendability can be described as an interprocess communication: adding a new component (a *plugin*) to a running system involves communication between the system and the source (e.g. the internet or code stored on a local disc, or code directly produced by a compiler that is used

to generate a self-written piece of code) of the extension. The built-in checks that are used to determine whether a plugin fits in its new environment are often implemented ad-hoc. In practice, even static data can be difficult to check because the format in which the data is offered to the application is often not well described. This may lead to unpredictable behavior of the application, in the worst case to a crash. Real (dynamic) plugins are even more dangerous because these components themselves are capable of corrupting data sometimes even without being noticed. Indeed, one of the main threats via the internet is the so-called *man in the browser attack*: a malicious extension disguised as harmless plugin is accepted by the ignorant user, extending the browser with undesired, possibly dangerous functionality.

Within a single program, the safety of components is mostly checked statically (i.e. at the moment the application is built). Functional programming languages such as Haskell [PJHJea99] and Clean [PvE01] offer a very flexible and powerful type system to perform these statical checks. Compact, reusable, and readable programs can be written in these languages while the static type system is able to detect many programming errors at compile time. However, software systems that consist of possibly independently developed applications cannot be checked completely in this way. In particular, the communication between components can only be verified dynamically, i.e. while the system is running. Traditional programming languages provide only very limited support for such dynamic type checks. Usually, applications written in these language only communicate simple data types such as streams of characters (e.g. for ASCII or XML texts), or use some ad-hoc defined (binary) format. Moreover, only data can be communicated, no computations. In lazy functional languages the existence of computations (unevaluated expressions) is essential. Besides, functions are treated as 'first class citizens': they can be passed to other functions or yielded as a result, just like any other 'first-order' value.

In this paper we focus on the incorporation of dynamic typing in statically typed functional languages with as goal the type safe dynamic exchange between independently programmed applications of not only of data but also of functions. The main challenge is to provide a hybrid system that, one the hand is completely transparant in the sense that is should not impose any additional constraints on the use of dynamics compared to the statically typed counterparts. Laziness, polymorphism and higher-order functions should be fully supported. On the other hand, the implementation should be flexible and efficient, i.e. the dynamic communication should not be obstructed by application or platform borders nor suffer from (unacceptable) loss of performance. At first, it is difficult to image how, for instance, the communication of computations or functions between applications should occur. However, it should be clear that, in case we want to develop a general mechanism for communicating components (represented as functions), a solution to this problem has to be found.

We will describe the way this problem has been solved in the functional language Clean. Clean equips the environment of a running application (e.g. the file system) with a rich dynamic type system, comparable to the static type system offered by Clean. This dynamic type system is based on the concept of

objects with dynamic types, or *dynamics* for short, as introduced by [ACP+92]. Values of this Dynamic type are, roughly speaking, pairs of an expression and a type, such that the expression component can be typed with the type component. Converting an (arbitrary) object into a dynamic is done by simply wrapping it with a special **dynamic** constructor. Unwrapping these values is done via combined type/value pattern matching. A very special feature of Clean are the powerful operations for reading and writing dynamics from and to files. In this way, expressions (containing data as well as closures) can be communicated between independently programmed applications.

For the programmer, the use of dynamics is easy and intuitive. The implementation, however, is quite complex. Not surprisingly, the ability to communicate computations between separately compiled applications is a fairly uncommon feature of programming languages. It implies that one has to able to add new functionality to a running application. If we would be dealing with a interpreted language, this is relatively easy to achieve. But since Clean uses compiled code, adding a dynamic plugin requires a rather sophisticated run-time architecture which is, as far as we know, unique in its sort. It is able to add code dynamically, to deal with distributed applications, to take care of version management, and so on.

The main properties of our system are:

1. it is completely (i.e. statically and dynamically) type-safe;
2. it preserves laziness;
3. it is not restricted to ordinary first-order data: (polymorphic) function objects can also be (un)wrapped;
4. it permits dynamics to be exchanged between different applications, again in a type-safe way;
5. it enables a running application to type safely combine dynamics to new ones without being aware of the actual types and values involved;
6. it provides a novel architectural framework in which there is no distinction between compiled code, dynamics created at run-time by applications, and new functionality that is interactively defined using a command line interpreter.

In this paper we first explain in Section 2 how dynamics can be used in Clean. We show how a statically typed expression can be packed into a dynamic and how it can be unpacked again in a type safe way. Furthermore, we show how at run-time dynamics can be combined into new dynamics in a type safe way, e.g. by applying dynamics to each other. Then we explain how separately compiled applications can exchange data and compiled functionality via special I/O operations with which an application can write/read a dynamic to/from a file.

The architecture of the implementation is explained in Section 3. On disk, a firm distinction is made between protected *system* space and *user* space.

Repositories for code and types are constructed by the compiler in system space which are used by a special dynamic linker that will serve any running Clean application asking to plug in a dynamic. Dynamics themselves reside in system space too, while the user obtains a placebo dynamic file in user space: for safety and security reasons it does not contain a dynamic but a reference to the real dynamic.

Next, in Section 4, we demonstrate the power of our architecture. We show that an *interpreter* for a *complete* lazy functional language can be defined which translates expressions to combinations of *compiled* code. It uses dynamics on disk as basic operators, but it can actually use *any* function or constant stored as dynamic by any application as basic primitive as well. New functions can be interactively defined in the interpreter as usual which can be stored again in a dynamic on disk. The magic thing is that functions defined in this way are just new combinations of existing code: no compiler is needed, yet the resulting application will run (almost) as fast as when it would have been defined directly in Clean using the compiler. The functions defined in the interpreter are type safe by construction, no type checker is needed: the dynamic type unification system is used to take care of this: new dynamics can only be constructed in a type safe way. New functions interactively defined in this way cannot only be used by the interpreter, but by any other Clean application. They are just dynamics on disk that can be plugged in when their types match the expected types. We show how a *compiled* calculator application is extended at run-time with functionality defined in the *interpreter*.

In Section 5 we explain in more detail how the interpreter is realized. The main trick is that the interpreter converts expressions to SKI-combinators. This conversion is well known [Sch24, CF58], yet practically not often used because it is known to be inefficient. However, in our case only just a very small part will generally be compiled to combinators. The main building blocks of the interpreter will be the dynamics stored on disk by ordinary Clean applications which therefore generally will contain plain compiled Clean code. The interpreter connects this code with (compiled) combinators, so generally almost no efficiency penalty has to be paid.

In Section 6 we discuss related work and we conclude in Section 7.

For people more familiar with Haskell than with Clean, there are a few syntactical differences between these languages. Where needed we explain these differences in a footnote. For a more elaborate comparison of Haskell and Clean we refer to [Ach07].

2 Dynamics in Clean

Clean offers a hybrid type system: in addition to its static type system, it has a (polymorphic) dynamic type system [ACPP91, ACP+92, Pil99, VP03b]. This allows programs to create (statically) untyped values, so called *dynamics*. Actually, a dynamic in Clean is not really an untyped value but a value of static type Dynamic. The dynamic object contains an expression as well as

a representation of the real (static) type of that expression. Dynamics can be formed (i.e., lifted from the static to the dynamic type system) using the special constructor **dynamic** applied to an expression. Unpacking is done via type pattern matching. Type correctness of statically typed expressions is checked by the Clean static type system, type correctness of dynamically typed values can of course only be checked at run-time by Clean's run-time system, containing dynamic type unification.

2.1 Packing Statically Typed Expressions into a Dynamic

Clean expressions can be converted to dynamic objects by wrapping them with the built-in constructor **dynamic**. A value of type Dynamic can be seen as a container in which an expression is stored together with a representation of the its original static type. This static type can be specified explicitly, but may be omitted if it can be inferred by the type system. For example, if the type contains universally or existentially qualified type variables or if one wants to assign a more specific type to the expression than the inferred type, it is required to specify the type explicitly.

```
expression1 :: Dynamic
expression1 = dynamic 42 :: Int[1]

expression2 :: Int → Dynamic
expression2 n = dynamic n + 42

expression3 :: [a] → Dynamic
expression3 list = dynamic list :: [Int]

expression4 :: Int Int → Dynamic[2]
expression4 n m = dynamic (dynamic n, dynamic m)

:: Tree a  = Leaf | Node a (Tree a) (Tree a)[3]

expression5 :: Dynamic
expression5 = dynamic Node 2 (Node 3 Leaf Leaf) Leaf

expression6 :: Dynamic
expression6 = dynamic map fst :: ∀[4].a b: [(a, b)] → [a]
```

Above we see some simple examples of expressions that are turned into a dynamic. In principle, any Clean data type can be packed into a dynamic, including basic types, function types, polymorphic types, user defined algebraic data types, etc. Also the type dynamic itself is no special case: a dynamic can be packed into a dynamic as well.

[1] Numerical denotations are not overloaded in Clean.
[2] Clean separates argument types by whitespace, instead of →.
[3] Defines a new data type in Clean, Haskell uses the `data` keyword.
[4] Clean's syntax for Haskell's `forall`.

However, there are a some exceptions. First of all, like in any other programming language, in Clean there exist objects that have a very special content which conceptually cannot be stored/copied. The values of such objects do not directly correspond to the 'real' objects they represent. Examples of such values are files (in Clean of type `File`) and values of type `World` (incorporating the whole environment in which an application is running). It is very difficult or even impossible to pack objects of these types into a dynamic, because the actual representation may be insufficient to reconstruct the concrete object to which the value refers, at a later stage (i.e. when the dynamic is read).

There are also a few types which cannot be packed into a dynamic due to implementation restrictions. Overloaded functions (due to the context restrictions) cannot be handled. In the current implementation when can deal with uniqueness types, but we cannot deal with uniqueness types inequations. This is not a big practical problem because when writing a dynamic containing uniqueness types one can define a more specific type which obey the inequations, such that these are solved and disappear in the restricted type. Most practical usage of uniqueness types is rather simple. For instance, a complete Clean application stored into a dynamic will have type `*World → *World`, which we can handle.

It is also not allowed to pack abstract data types. The reason is that the test on equality on such abstract types is not easy: it is not enough to test the equality of the definitions of the types involved. One should also test whether the operations defined on these abstract data types are exactly the same. There are solutions for this problem (see Section 6) which we could apply, but our implementation currently does not support abstract types.

Any incorrect usage of dynamics will result in a static compile time error.

```
IllegalCreateDynamic[5]:: t → Dynamic                    // Compile-time type error
IllegalCreateDynamic any = dynamic any
```

It is impossible to pack an expression into a dynamic if its type is unknown. For instance, when a polymorphic function like `IllegalCreateDynamic` is defined it is unknown what the type of the actual argument will be. This is determined by the context in which this function is used, and this context type is not passed. The definition of `IllegalCreateDynamic` is therefore not allowed, and will give a type error. This problem can be solved by using *ad-hoc* polymorphism instead of real polymorphism.

```
CreateDynamic:: t → Dynamic | TC t[6]
CreateDynamic any = dynamic any
```

The type constructor class `TC` (for Type Code) is predefined in Clean. However, unlike ordinary type classes, the programmer cannot define instances for `TC`. Instead, the compiler will automatically generate an appropriate instance of the `TC` class whenever needed. Such an instance consists of a representation of the concrete static argument type. In this way, the caller of `CreateDynamic` will

[5] In Clean it is allowed to start function names with an uppercase character.

[6] This indicates a context restriction for an overloaded function.

supply this function with both ingredients (value and type) that are needed to construct the dynamic on the right-hand side.

```
CreateDynamicTree:: Dynamic
CreateDynamicTree = CreateDynamic (Node 2 (Node 3 Leaf Leaf) Leaf)

IllegalCreateDynamicFile :: File → Dynamic          // Compile-time type error
IllegalCreateDynamicFile file = CreateDynamic file
```

Types which cannot be packed don't have a representation. Any attempt to construct such a dynamic will fail and result in a compile time error.

In the example above the definition of `CreateDynamicTree` is correct: an instance of `TC` for the type `Tree Int` is constructed and (internally) passed to `CreateDynamic`. The definition of `IllegalCreateDynamicFile` raises a compile time error because there exists no instance of `TC` class for type `File`.

2.2 Unpacking of Dynamics

Once a dynamic is created, the (type and value) information about the original object is discarded. This enables the construction of composite values such as

```
DynamicList :: [Dynamic]
DynamicList = [dynamic 1, dynamic 3.14, dynamic 'a']
```

that would be untypable if these values were not wrapped[7].

The only way to determine the actual type of an object is via a special (run-time) pattern match. As with ordinary pattern matches, a type pattern match may fail. However, to determine whether a type pattern match fails is more involved. To maximize flexibility, the type pattern matching is not based on simple type equality, but on resolution. For this reason, Clean offers run-time type unification.

With a pattern match on dynamics a case distinction can be made on the contents. If the actual dynamic matches the type and the value specified in the pattern, the corresponding function alternative is chosen. Since it is known in that case that the dynamic matches the specified type, this knowledge is used by the static type system in the remainder of the function: the decomposed parts of a dynamic now have a known type and can be handled as ordinary expressions in the body of the function. In this way dynamics can be converted back to ordinary statically typed expressions again. The static type checker will use the knowledge to check the type consistency in the body of the corresponding function alternative. We explain the different ways of unpacking with some illustrative examples.

2.2.1 Pattern Matching on Specific Values and Types.

In the example below, `matchInt` returns `Just` the value contained inside the dynamic if it has type `Int`; and `Nothing` if it has any other type.

[7] Indeed, one could use *existential types*, but this would introduce problems as soon as the list object is decomposed

```
::  Maybe a = Nothing | Just a

matchInt :: Dynamic → Maybe Int
matchInt (x :: Int) = Just x
matchInt  other     = Nothing
```

The compiler translates a type pattern match into run-time type unification. The type packed in the dynamic is unified with the type specified in the pattern of the function alternative. If the unification fails, the next alternative is tried, as usual.

```
transform :: Dynamic → [Int]
transform (0 :: Int)                = []
transform (n :: Int)                = [n]
transform (f :: [Int] → [Int])      = f [1..100]
transform ((x,y) :: ([Int],[Int]))  = x ++ y
transform other                     = []
```

The function `transform` illustrates the different kind of checks one can do. In this case the empty list is used to catch the situation that none of the previous values and types are matching.

```
testPoly :: Dynamic a b → (a, b)
testPoly (g :: ∀c: c → c) x y = (g x, g y)
testPoly else x y             = (x  , y  )
```

In the function `testPoly` it is checked whether the dynamic contains the polymorphic identity function. In this example two additional arguments are passed. In case the pattern match succeeds, the extracted function will be applied to both of these arguments. In case of a failure, the arguments are delivered as the result. As is the case with the construction of a dynamic, one has to explicitly state the fact that one wants a polymorphic function by using the universal quantifier. We have chosen for explicit quantification to avoid interference with type pattern variables, explained in the next paragraph.

2.2.2 Unifying Dynamic Types with Dynamic Types.

An important constituent of Clean's dynamic type system are *type pattern variables*. After successful unification, type pattern variables are bound to the corresponding parts of the concrete type. They actually serve as placeholders to exchange type information beween different parts of a function definition. Type pattern variables can have multiple occurrences on both left- and right-hand side of a function alternative. On the left-hand side, multiple occurrences of the same variable lead to unifications: all actual types that are bound to the same variable are unified. If unification fails, the corresponding type pattern match fails. This is a very powerful mechanism for checking the type consistency between different dynamics.

```
dynamicApply :: Dynamic Dynamic → Dynamic
dynamicApply (f :: a → b) (x :: a) = dynamic f x :: b
```
[8]

[8] The type b can also inferred by the compiler.

```
dynamicApply  df          dx      = dynamic "Error: cannot apply"
```

A key example showing the expressive power is the function `dynamicApply`. It has two arguments of type Dynamic and it yields a value of type Dynamic as well. In the first function alternative it is checked whether the first dynamic contains a function: its type has to be unifiable with type `a → b`. The second dynamic now has to be unifiable with the argument type `a` of the function `f`. If the match succeeds, it is type safe to apply `f` to `x`, without *exactly* knowing what the actual types of `f` and `x` are. It is clear that `f` is a function, but the concrete types of `a` and `b` are statically unknown. The result will be some *dynamically* known type `b`. The actual result type is not known at compile time, but, packing the application into a dynamic again, resolves the issue: the static type of `f x` is `Dynamic`, and the dynamic type stored with this application is the type assigned to `b` at run-time.

If the dynamic types cannot be unified, it is not type safe to apply the function to the argument, and the next alternative of `dynamicApply` is chosen. It yields an error message stored into a dynamic.

The `dynamicApply` example shows that we are able to combine dynamically constructed objects in a type safe way. This is a key feature enabling the construction of complex applications like type safe shells, and type safe process communication.

2.2.3 Unifying Dynamic Types with Static Context Types.

Type variables in dynamic patterns can also relate to type variables in the static type signature of a function. Such type variables are always overloaded in the predefined `TC` class. Those functions are called *type dependent functions* [ACPP91]. In the specification of a type pattern match, overloaded variables are distinguished from type pattern variables by marking the former with a caret (`^`): in this way a marked variable is associated with the type variable with the same name in the type signature. The `TC` class is used to 'carry' the type representation. In the example below, the substitution of the static type variable `a` will depend on the (static) context in which `lookup` is used. Consequently, a restriction will be imposed on the actual types that are accepted at run-time.

```
lookup :: [Dynamic] → a  | TC a
lookup [(x :: a^):xs] = x
lookup [x:xs]         = lookup xs
lookup []             = abort "dynamic type error"

Start = (lookup DynamicList + 5, lookup DynamicList + 2.5)   // result will be (6,5.64)

DynamicList = [dynamic 1, dynamic 3.14, dynamic 'a']
```

The function `lookup` searches for a value of a certain type `a` in its list of dynamics. It will yield the first element from the list matching the requested type. In `Start` this `lookup` function is used twice. In the first case an `Int` is demanded (due to `+ 5`), in the second case a `Real` (due to `+ 2.5`).

2.3 Type Safe Exchange of Data and Code between Independent Applications

Until now, we only considered the situation in which dynamics were constructed and examined within a single application. However, the most important application of dynamics is type safe communication of data and code between different, possibly independently developed Clean applications.

To enable the exchange of data between applications, the dynamic run-time system of Clean supports writing dynamics to disk and reading them back again. This is not trivial, particularly if writing and reading is performed by different applications or by different versions of the same application. The problem is caused by the fact that a dynamic may contain unevaluated functions. Note that Clean is not an interpreted language: it uses compiled code. Reading a dynamic implies that compiled code of unevaluated functions has to be added to the code of the running application. This requires a dynamic linker. Furthermore, one needs to be able to retrieve the type definitions and function definitions associated with a stored dynamic. The need for function definitions not only arises from lazy evaluation (which can introduce closures) but also from the presence of functions as values.

A dynamic can be written to a file on disk using the `writeDynamic` function.

```
writeDynamic :: String Dynamic *World → (Bool, *World)
```
[9]

In the `Producer` example below, a dynamic is created which contains the list of all prime numbers. This list is defined as an application of the function `sieve` to an infinite list of integers. This dynamic is written to file using the `writeDynamic` function. It is important to note that the **dynamic** constructor is lazy, i.e. if an expression is wrapped into a **dynamic** it will *not* be evaluated. In our example this implies that the closure containing the application of the function `sieve` is written to file in its unevaluated form. The file named `primes` will therefore contain a calculation that yields a potentially infinite list of prime numbers.

How far a closure has been evaluated before it was written to file generally depends on the way the program happens to be evaluated. Cleanis by default lazy. However, it is possible in Cleanto force evaluation of expressions using strictness annotations. So, if wanted one can explicitly force the evaluation of a dynamic before writing it to disk.

module `Producer`

```
Start :: *World → *World
Start world
    #[10] (ok,world)  = writeDynamic "primes" (dynamic sieve [2..]) world
    | not ok          = abort "could not write primes"
    | otherwise       = world
```

[9] The * is a uniqueness attribute, indicating that the world environment is passed around in a single threaded way. Unique values allow safe destructive updates and are used for I/O in Clean. The value of type `World` corresponds to the hidden state of the `IO` monad in Haskell.

[10] This is a special let construct introducing a new scope which allows us to re-use variable names (e.g. `world`).

```
where
    sieve :: [Int] → [Int]
    sieve [prime:rest] = [prime:sieve filter]
    where
        filter = [h \\ h ← rest | h mod prime ≠ 0]
```

When the dynamic is stored on disk, it is insufficient to include just the expression and its static type. To allow the dynamic to be used by another application additional information is necessary:

- the code of the functions appearing in closures inside the dynamic;
- the definitions of all the types that are needed to check type consistency when matching on the type of the dynamic in another Clean program.

The required code and type information will be generated by the compiler and are stored in a special database; see Section 3 for more details.

Another example of an application that writes a dynamic to a file is

```
module Consumer

Start :: *World → *World
Start world
    # (ok,world)   = writeDynamic "consume10" (dynamic take 10 :: ∀a:[a] → [a]) world
    | not ok       = abort "could not write consume10"
    | otherwise    = world
```

This time a polymorphic function is wrapped in a dynamic and stored. After running both applications, two dynamics are created on disk. These dynamics can be read back using the `readDynamic` function.

```
readDynamic :: String *World → (Bool, Dynamic, *World)
```

This `readDynamic` function is used in the `Combine` application below to read the earlier stored dynamics.

```
module Combine

Start world
    # (ok,pr,world)     = readDynamic "primes" world
    | not ok            = abort "could not read primes"
    # (ok,take10,world) = readDynamic "consume10" world
    | not ok            = abort "could not read consume10"
    | otherwise         = show (dynamicApply take10 pr)
where
    show (v :: [Int]) = prettyPrint v
    show else         = abort "Unexpected result type"
```

When a dynamic is read, this is also done lazily: only when the evaluation of the dynamic is demanded (which can only happen after its type is approved in a pattern match), the content of the file is decoded back to a dynamic which then can be unpacked. If functions are read that are not known in the application, their code will be plugged into the running application automatically; see 3.2.1.

3 Implementation of Dynamic File I/O

The most important practical application of dynamics in Clean is the exchange of data and code between separate applications. This aspect is also the most interesting one from an implementation point of view. In this section we explain how this is realized. Without loss of generality we only look at the communication via files and assume that these files can be accessed by all participating communicating applications.

Our aim is to provide a mechanism for (de)serializing dynamics in such a way that both their values and types can be stored and retrieved efficiently. This means that we have to design an appropriate encoding for both components of a dynamic.

After a dynamic is stored, the type information of all types involved has to be accessible when it is read again, possibly by a different Clean application. The static type encoded in the dynamic is needed for the unification in the type pattern match. However, it is insufficient to store a plain representation of a type. To determine whether two types are unifiable, it is necessary that type constructors can be compared. This comparison cannot be implemented as a simple equality check on the names of the constructors. For example, suppose an application contains the following type for representing binary trees storing information not only in the nodes but also in the leaves:

```
:: Tree a   = Leaf a | Node a (Tree a) (Tree a)
```

Obviously, this type is incompatible with `Tree` type defined Section 2.1 (only storing information in the interior nodes), though the names are identical. A proper comparison has to take the complete type definition into account. Therefore, at run-time access is needed to *all* type definitions of *all* the applications which take part in the communication with dynamics. Still, there is some freedom in the way types are compared. Like in Haskell, the order in which type constructors are specified in a Clean program matters. For this reason, two types are identical if names, types and order of all data constructors are the same.

Moreover, a dynamic expression read by some program might contain calls to functions which are not available in this program. Hence, the running application has to be extended with the code of these functions in order to enable evaluation of these calls.

In an interpreted language one usually has access to *all* sources of *all* applications involved. This would make the exchange of dynamics a lot easier. However, Clean uses compiled code. We cannot assume the availability of source code. The standard run-time environment of Clean applications does not provide run-time access to type definitions nor to compiled code. This has to be changed. Besides, we have to take care of version management since applications can be recompiled possibly changing types and/or code. Finally, when dynamics are used we want to avoid an overhead penalty in the execution time of applications as much as possible.

In the remaining part of this section we first explain the standard compilation architecture of the Clean system. Hereafter, we discuss the changes that have

been made to facilitate the exchange of dynamics.

3.1 Clean's Standard Compilation and Run-Time Architecture

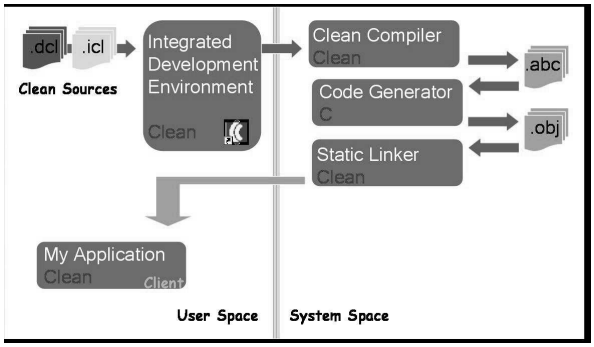

Figure 1: The standard compilation steps for Clean.

Clean is a modular language. Each module consists of an implementation module (stored in a `.icl` file) and a definition module (stored in a `.dcl` file) that specifies what is exported from the corresponding implementation module. The definition module is used by the compiler to check the consistency between the Clean modules and enables separate compilation of the implementation modules. The Clean compiler compiles each implementation module to machine independent ABC-code (stored in a `.abc` file). The ABC-code contains machine instructions for a virtual abstract machine, the ABC-machine (see [Koo90]). The ABC-code is a kind of platform independent byte code specially designed for Clean. As a final step, ABC-code is translated to (platform dependent) object code. Currently, several different platforms are supported, such as Intel (Windows, Linux, Mac), Motorola (Mac) and Sparc (Unix) processors. The static linker combines the generated object modules of one Clean program into an executable application.

3.2 The Compilation and Run-Time Architecture for Dynamics

As long as dynamics are written to and read from a file by one and the same application, nothing has to be changed in the compile-time and run-time system. In this restricted case one can assume that the application has all necessary information at its disposal. However, as soon as a dynamic is communicated to a different application, or to a new version of the same application, the run-time system has to be adapted to support access to information about types and generated function code.

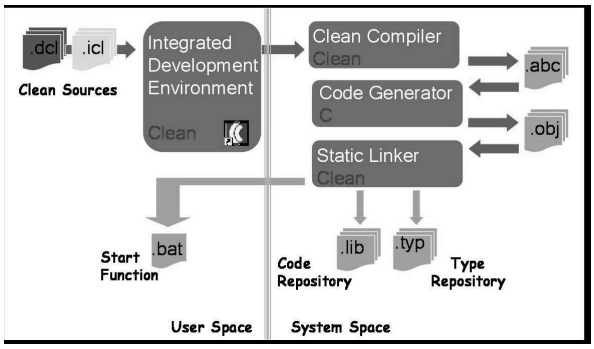

Figure 2: The compilation steps for applications working with dynamic's

In the changed compilation scheme, the static linker no longer generates an executable. Its only task is to generate a script (a `.bat` file) that, when it is executed, will call the dynamic linker to build the main application, which is started subsequently. In addition, the static linker generates two new files: a *code* repository (a `.lib` file) and a *type* repository (a `.typ` file). In the code repository all object codes of the application are collected. The type repository contains the definitions of all types defined in the application.

3.2.1 Plugging in Code at Run-Time Using a Dynamic Linker

When a dynamic is written to a file, an encoding of the expression and its type are written to disk. The type is encoded in an (internally defined) algebraic data type. In a dynamic on disk, functions occurring in the encoded expression are represented by symbolic pointers to the corresponding code repositories, whereas encoded data types refer to type repositories.

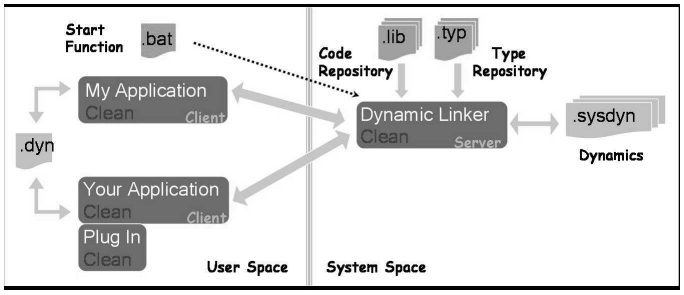

Figure 3: Dynamically extending a running application.

The actual construction of the running application is done by the dynamic linker. We have chosen for a client-server architecture: one dynamic linker is serving all Clean applications that want to read in dynamics. It can determine all the pieces of code which are needed for the execution of a particular function,

and will append these to a running application. In this way we can (re-)construct the executable code not only for any part but also for a whole application itself. To obtain the code, it is sufficient to supply the dynamic with the name of the function to be executed.

For example, when the `.bat` script is executed, the dynamic linker is asked to provide all the pieces of code that belong to Clean's main function `Start`. Actually, the only difference with the static way of linking is that the actual linking together of object code is deferred until the application is started. Starting an application in the new setting takes slightly more time (usually less than a second), but there is no efficiency loss once the application is running.

Whenever a running application wants to execute a function in a dynamic that has been read from file, it will also send a code request to the dynamic linker.

3.2.2 Lazy Construction of Dynamics

A dynamic read from disk is treated lazily: no code will be plugged in *unless* its type is approved *and* the evaluation of the dynamic is needed. At first instance, the Clean program will only receive a pointer directly referring to the stored dynamic. When a type pattern match is performed, the type component is retrieved from disk and decoded, and unified with the pattern type. If unification was successful the type definitions of all type constructors that were assumed to be equal are fetched from the corresponding type repository and compared. The pattern match will still fail if type constructors appear to be different. But even after successful comparison, the expression will not be read unless the value of the expression is needed (e.g. when a combined type/value pattern match was performed). In that case the code that corresponds to function pointers occurring in the expression might also be fetched from disk. Clearly, whether a dynamic value is read completely, and which function code has to be linked in depends on the structure of both the right-hand side of the pattern match and the decoded dynamic expression itself. Consider the following example:

```
Start world
    # (ok1, fun, world) = readDynamic "application" world
    # (ok2, arg, world) = readDynamic "document" world
    = writeDynamic "result" (dynamicApply fun arg) world
```

In this example two dynamics are read (from the file named `application` and the file named `document`) and applied to each other. However, the result of this application is immediately stored in a dynamic again and written to a file named `result`. As explained before, neither the construction of the dynamic in `dynamicApply` itself (see 2.2.2) nor the function `writeDynamic` (see 2.3) will force evaluation. Hence, neither `fun` nor `arg` is evaluated in this example. The storage of the application of `fun arg :: Dynamic` will therefore consist of an application node pointing to the dynamics stored in the files `application` and `document`, respectively.

The example shows that, if one reads a dynamic (like the one stored in `result`) one actually reads a graph expression which can point to other dynamics stored

on disk. As a consequence a dynamic on disk may not only refer to several files, it may also refer to several different code and type repositories. One therefore cannot simply delete dynamics or repositories from disk because other dynamics might still refer to them. This requires an elaborate version management system for dynamics (see 3.2.4).

3.2.3 Maintaining Sharing

When a dynamic containing shared expressions is written to disk, it is encoded in such a way that sharing is preserved, i.e. if the dynamic is read, the expression will be reconstructed such that the original sharing is still present, thus avoiding duplication of work and/or data.

Consider the following example:

```
writeSharedDynamic :: *World → *World
writeSharedDynamic world
    # (ok,world) = writeDynamic "sharedDynamic" (dynamic (dynamic odd, dynamic even)) world
    | not ok     = abort "could not write sharedDynamic"
    | otherwise  = world
where
    (odd,even) = split (sieve [2..])

    split [l,r:ps] = ([l:lps],[r:rps])
    where
        (lps,rps) = split ps

readAndMergeDynamic :: *World → [a] | TC a
readAndMergeDynamic world
    # (ok,pr,world) = readDynamic "sharedDynamic" world
    | not ok        = abort "could not read sharedDynamic"
    = case pr of
        ((list1 :: [a^], list2 :: [a^]) :: (Dynamic,Dynamic)) → merge list1 list2
        else → abort "sharedDynamic of unexpected type"
where
    merge [l:ls] [r:rs] = [l,r:merge ls rs]
```

The functions `writeSharedDynamic` stores a dynamic consisting of a pair of two list also wrapped in a dynamic. The first list contains the prime numbers occurring on odd positions, whereas the second one contains the remaining prime numbers. Both lists are infinite and share the expression `split (sieve [2..])` which produces the numbers on demand. As in the previous example, `writeSharedDynamic` itself does not force the evaluation of any of these expressions.

When the dynamic is written to disk the sharing of `split (sieve [2..])` is encoded. This dynamic is read by the function `readAndMergeDynamic`. As soon as this function merges the incoming lists, the corresponding list expressions `list1` and `list2` will be reconstructed independently while the original sharing is maintained. Hence, the list of prime numbers is generated only once.

3.2.4 Version Management and File Management for Dynamics

We have seen that, due to laziness, dynamics stored on disk might refer to other dynamics stored in different files. As with ordinary references/pointers, this introduces the danger of *dangling pointers*: objects containing pointers that refer to non-existing objects. For instance, such a situation can occur when the user deletes files containing dynamics, but also when code and/or type repositories change. The latter happens each time a Clean program is recompiled. In that case new (code and type) repositories are created. Old repositories cannot be deleted because there might be other dynamics still referring to these old versions.

To deal with this dangerous situation, we designed the following file storage architecture (see also Fig. 3). A distinction is made between files stored in *user space* and files stored in *system space*. The files in user space are owned by the user and can be freely moved, copied, renamed, updated, or deleted. The files in system space are all automatically generated and maintained by the Clean system in a hidden protected area which cannot be directly accessed by users.

When a dynamic is written to disk using the function `writeDynamic`, two files are created: a `.dyn` file stored in user space and a `.sysdyn` file stored in system space. The `.sysdyn` file contains the real information: the serialized encoded dynamic. The user only has access to the `.dyn` file that just contains a reference to the actual dynamic stored in the corresponding `.sysdyn` file. The references have been made unique by means of an *MD5* encoding. Also the code (`.lib` files) and type repositories (`.typ` files) produced by the static linker, are stored in system space. The run-time system has access to all the information it needs for plugging in dynamics.

A disadvantage is that the administration in system space will grow continuously whenever new dynamics are written to file or applications are (re-)compiled. A special garbage collector has been constructed that determines which of the `.lib`, `.typ` and `.sysdyn` files in system space have become obsolete and therefore can be removed safely.

So far we assumed that all information was stored on one machine. If one wants to copy and use a dynamic on another machine, all the `.lib`, `.typ` and `.sysdyn` files it refers to have to be copied as well. There is a special copying tool which takes care of this. It only copies dynamics and repositories not already present at the other end, making use of the unique MD5-identification of dynamics.

Currently, the system has only been implemented on windows based platforms. In principle it is possible to port the system to other main platforms as well: the compiler already produces platform independent ABC-code and there exist static linkers for Linux, Mac and Sun systems. To make a multi platform system one has to shift ABC-code from one machine to another, do just-in-time code generation, and dynamic linking. The latter requires platform specific dynamic versions of the static linkers.

4 Example: A command line interpreter

To illustrate the expressive power of dynamics in Clean we present an interactive command-line shell interpreter, called *Esther*, which is part of a prototype implementation of a strongly-typed operating system written; see [vP03a]. As usual for a shell, Esther can be used for manipulating files, applications, data, and processes at the command line. A special feature of Esther is, however, that it provides the basic functionality of a strongly typed lazy functional language. The shell type-checks each command line and only executes well-typed expressions. Files are also typed: applications and commands are simply files with a function type.

The type-checking/inference performed by the shell uses the hybrid static/-dynamic type system of Clean. The shell behaves like an interpreter, but it actually executes a command line by combining existing compiled code of functions (and programs from disk). Clean's dynamic linker is used to store/retrieve any expression (both data and code) with its type on/from disk. This linker is also used to communicate values of any type, e.g., data, closures, and functions between running applications in a type safe way.

The shell combines the advantages of interpreters (direct response) and compilers (statically typed and fast code). Applications (compiled functions) can be used, in a type safe way, in the shell. And, vise versa, functions defined at the command line can be used by any compiled Clean application.

Like any other shell, Esther enables users to start pre-compiled programs, provides simple ways to combine multiple programs, e.g., pipelining and concurrent execution, and supports execution-flow controls, e.g., if-then-else constructs. It provides a way to interact with the underlying operating system and the file system, using a textual command line/console interface.

Esther also offers a complete typed functional programming language with which programs can be constructed. Traditional shells provide very limited error checking before executing the given command line. This is mainly because the applications mentioned at the command line are practically untyped because they work on, and produce, streams of characters. The intended meaning of these streams of characters varies from one program to the other. Esther type checks a command line before performing any actions. The choice to make our shell language typed also has consequences for the underlying operating system and file system: they should be able to deal with types as well.

In this section, we give a brief overview of the functionality of the Esther shell and the file system it relies on.

4.1 A Typed File System

A shell manipulates applications and data stored on disk. Esther is typed; it can only work if all files it operates on are typed as well. We therefore assume that all files have a proper type. By writing data, functions, and even large complete Clean applications as dynamic to disk, we obtain a rudimentary typed file system for free. In principle, we can also deal with applications not written

in Clean even if they are untyped. Such applications can be incorporated into in our typed file system, by writing properly typed Clean wrapper applications around them, which are then stored again as dynamics on disk.

Esther does not contain any built-in commands. The commands it knows are determined by the files (dynamics) stored on disk. To find a command, the shell searches its directories in a specific order as defined in its search paths, looking for a file with that name.

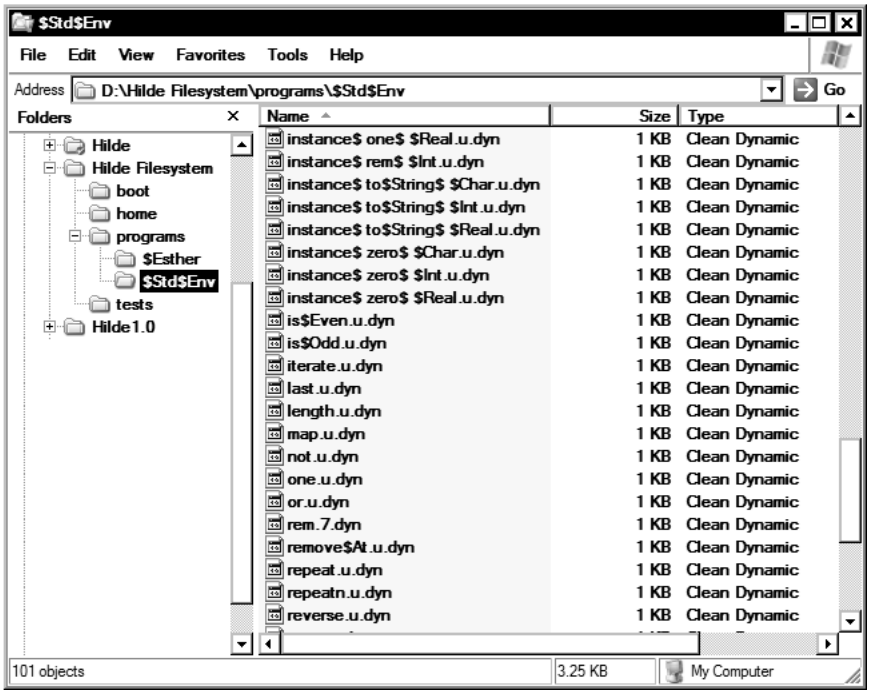

Figure 4: A screenshot of the typed file system; implemented as dynamic on disk (stored in `.dyn` files).

To use Esther, we first need to provide a collection of useful dynamics on disk. When Esther is initialized, the following dynamics are created (see Fig. 4):

- almost all functions from the Clean standard environment[11], are stored such as `+`, `-`, `map`, and `foldr`;
- common commands are provided to manipulated the file system (`mkdir`, `rmdir`, and the like);
- commands are provided to create processes directly based on the functionality offered by the underlying operating system.

[11] Similar to Haskell's Prelude.

All folders are common Window folders, all files contain dynamics created by Clean applications using the `writeDynamic`-function.

Esther performs the following steps in a loop:

1. it reads a string from the console and parses it like a Clean expression. It supports Clean's basic and predefined types, application, infix operators, lambda abstraction, functions, overloading, let(rec), and case expressions;
2. identifiers that are not bound by a lambda abstraction, a let(rec), or a case pattern are assumed to be names of dynamics on disk, and they are read from disk;
3. it type checks the expression using dynamic run-time unification and type pattern matching, which also infers types;
4. if the command expression does not contain type errors, Esther displays the result of the expression and the inferred type. Esther will automatically be extended with any code necessary to display the result (which requires evaluation) by the dynamic linker.

For instance, if the user types in the following expression:

```
> map ((+) 1) [1..10]
```

the shell reacts as follows:

```
[2,3,4,5,6,7,8,9,10,11] :: [Int]
```

If we execute the steps 1-4 above for this expression, the names `map` and `+` appear in unbound positions. The shell therefore assumes that they are the names of dynamics on disk. They are read from disk practically extending the functionality of the shell with these functions. Esther inspects the types of `map` (let us assume that the file `map` contains the type that we expect: $\forall$`a: (a`$\rightarrow$`b) [a`$\rightarrow$`[b]`), `+` (for simplicity, let us assume: `Int Int`$\rightarrow$`Int`) and the list comprehension (which has type: [`Int`]) to type-check the command line. If this succeeds, which it should given the types above, the shell applies the partial application of `+` with the integer one to the list of integers from one to ten, using the `map` function. The application of one dynamic to another is done using the `dynamicApply` function from Section 2.2.2, extended with better error reporting. How this is done exactly, is explained in more detail in Section 5. With the help of the `dynamicApply` function, the shell constructs a new function that performs the computation `map ((+) 1) [1..10]`. This function uses the compiled code of `map`, `+`, and `[ .. ]`. The latter is implemented by calling predefined generator function called `_from_to`.

Esther can therefore be regarded as a hybrid interpreter/compiler, where the command line is interpreted/compiled to a function that is almost as efficient as the same function written directly in Clean and compiled to native code. If functions, such as `map` and `+`, are used in other commands later on, the dynamic linker will notice that they have already been used and linked in, and it will reuse their code. Consequently, the shell will react even quicker, because dynamic linking is not required anymore in such a case.

4.2 The Esther Command Language

Here follow some command line examples with an explanation of how they are handled by the shell. Figure 5 shows two example sessions with Esther. The right Esther window in Fig. 5 shows the result of an `ls` command, listing the same directory as the Windows Explorer window in Fig. 4. We explain Esther's syntax by example below. Like a common UNIX shell, the Esther shell prompts the user with something like `1:/home>` for typing in a new command. For readability we use only `>` in the examples below.

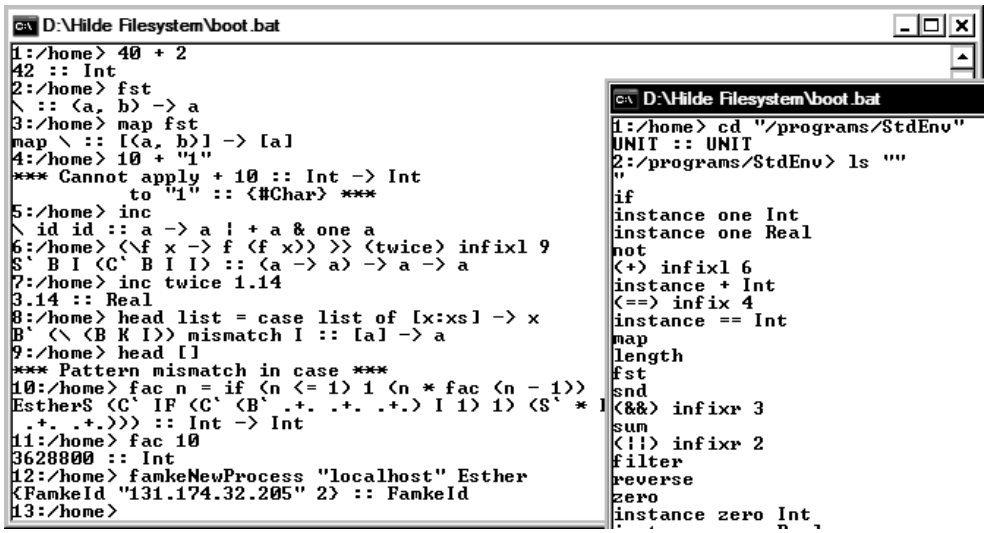

Figure 5: A combined screenshot of the two concurrent sessions with Esther.

4.2.1 Expressions

Here are some more examples of expressions that speak for themselves. Application:

```
> map
map :: (a -> b) [a] -> [b]
```

Expressions that contain type errors:

```
> 40 + "5"
*** cannot apply + 40 :: Int -> Int to  "5" :: {#Char} ***
```

4.2.2 Saving Expressions to Disk

Expressions can be stored as dynamics on disk using ≫:

```
> 2 >> two
2 :: Int
> two
2 :: Int

> (+) 1 >> inc
+ 1 :: Int -> Int
> inc two
3 :: Int
```

4.2.3 Overloading

Esther resolves overloading in almost the same way as Clean. It is currently not possible to define new classes at the command line, but they can be introduced using a simple Clean program storing a class and its instances as dynamics on disc.

```
> +
+ :: a a -> a | + a

> one
one :: a | one a

> (+) one
(+) one :: a -> a | + a & one a
```

4.2.4 Function Definitions and Lambda Expressions

One can define new functions at the command line:

```
> dec x = x - 1
dec :: Int -> Int
```

This defines a new function with the name dec. Functions are implicitly written to disk. In this case the function is written in a file with the same name such that from now on it can be used in other expressions.

```
> fac n = if (n < 2) 1 (n * fac (dec n))
S (C' IF (C' < I 2) 1) (S' * I (B (S .+. .+.) (C' .+. .+. .+.))
) :: Int -> Int
```

The factorial function is constructed by Esther using combinators (see Section 5), which explains why Esther responds in this way.

Since functions are store as dynamics, they cannot only be reused within the shell itself, but also used by any other Clean program.

By means of lambda expressions one can define 'anonymous' functions .

```
> (\f x -> f (f x)) ((+) 1) 0
2 :: Int
```

4.2.5 Let Expressions

Sharing is introduced via let expressions.

```
> let x = 4 * 11 in x + x
88 :: Int

> let ones = [1:ones] in take 10 ones
[1,1,1,1,1,1,1,1,1,1] :: [Int]
```

4.2.6 Case Expressions

It is possible to do a simple pattern match using case expressions. Nested patterns are not yet supported, but one can always nest case expressions by hand. An exception '`Pattern mismatch in case`' is raised if a case fails.

```
> hd list = case list of [x:xs] -> x
B‘ (\ (B K I)) mismatch I :: [a] -> a

> hd [1..]
1 :: Int

> hd []
*** Pattern mismatch in case ***

> sum l = case l of [x:xs] -> x + sum xs; [] -> 0
B‘ (\ (C‘ (B‘ .+.) I (B .+. .+.))) (\ 0 mismatch) I
:: [Int] -> Int
```

The interpreter understands Clean denotations for basic types like `Int`, `Real`, `Char`, `String`, `Bool`, tuples, and lists. How can one perform a pattern match on a user-defined constructor defined in some application? It is not (yet) possible to define new types in the shell itself. However, one can define the types in any Clean module, and construct an application that writes the constructors as dynamic to disk. For example:

```
module IntroduceNewType

import ...

:: Tree a = Node (Tree a) (Tree a) | Leaf a

Start world
    # (ok, world) = writeDynamic "Node"
                    (dynamic Node :: ∀a: (Tree a) (Tree a) → Tree a) world
    # (ok, world) = writeDynamic "Leaf"
                    (dynamic Leaf :: ∀a: a → Tree a) world
    # (ok, world) = writeDynamic "myTree"
                    (dynamic Node (Leaf 1) (Leaf 2)) world
    = world
```

These constructors can then be used by the shell to pattern match on a value of that type.

```
> leftmost tree = case tree of Leaf x -> x; Node l r -> leftmost l
leftmost :: (Tree a) -> a

> leftmost (Node (Node myTree myTree) myTree)
1 :: Int
```

4.3 Example: A dynamic calculator

In the previous section we have presented the command line interpreter Esther, that allows us to interactively define new functions. In essence this interpreter provides us with the same expressive power as a standard compiler does. But the main difference with common interpreters is that Esther enables the user to combine compiled code. Consequently, the constructed applications are (almost) as efficient as applications developed with a compiler. An additional advantage is that these newly defined functions are stored as dynamics on disk and can be reused not only by the interpreter itself, but also by any other application. In this section we show an example of a relatively simple application using these stored dynamics: a dynamic calculator.

This calculator is a separately compiled Clean executable that in principle runs without using the shell. Initially, the calculator has buttons for typing in `Real` numbers, but it lacks buttons for doing calculations with these numbers (see the first picture in Fig. 6). Instead, it has additional buttons for extending the calculator with new operations or for removing superfluous buttons. So, the functionality of the calculator can be tailor made by the user.

Pressing the `Add` button on the calculator opens a file selection dialog, shown at the bottom of Fig. 6. The application expects that the user selects a file containing a dynamic. This dynamic should be a tuple: either of type (`String`, `Real` → `Real`) for unary operators, or of type (`String`, `Real Real` → `Real`) for binary operators. If the selected file is not of appropriate type, an error message is displayed. If the dynamic is of correct type, a button is added dynamically. The `String` component of the tuple is used to label the button and, of course, the corresponding function will be applied by the calculator when the button is pressed. These new operations can be constructed by any other Clean application. For instance, in Esther we can type:

```
> ("2*a-b^2", \a b -> 2.0 * a - b * b) >> '2a-b2'
```

which will write a tuple wrapped in a dynamic to a file named `2a-b2`. The name to label the button is `2*a-b^2` and the function is `λa b → 2.0 * a - b * b`. The lower half of Fig. 6 shows this command line typed in Esther. After selecting this dynamic (appearing as a file (named `2a-2b.u.dyn` on disk), it becomes available in the calculator. The screenshot shows that the new operation is applied to 8 and 3 yielding 7.

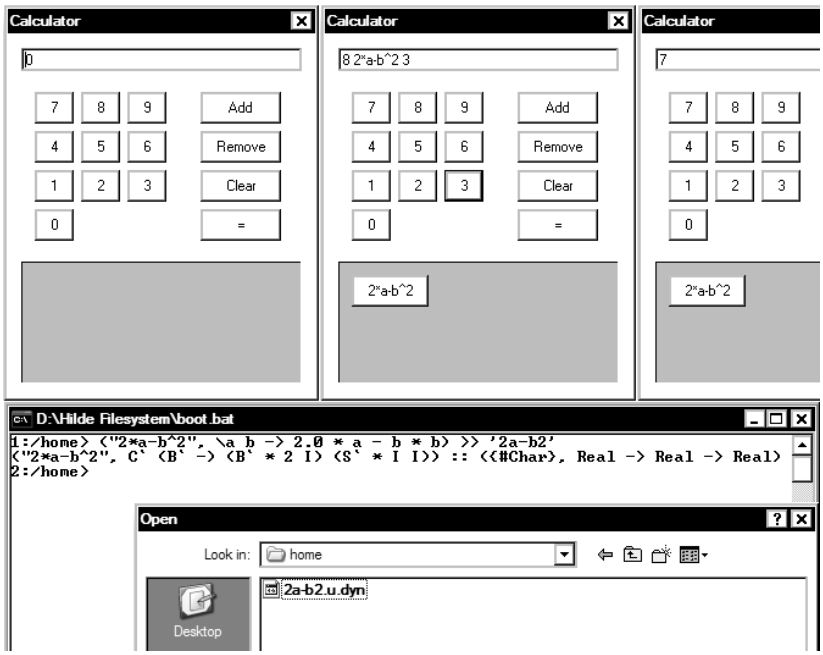

Figure 6: A combined screenshot of the calculator and Esther in action.

It can be convenient to offer a set of basic buttons. This can also be done without use of the shell, as the following Clean application shows. Here, the auxiliary function `toCalc2` is necessary to enforce the `Real` instances of the corresponding overloaded operations.

```
Start :: *World → (String,*World)
Start world
# (ok,world)    = writeDynamic "+Calc"   (dynamic (toCalc2 "+" (+))) world
# (ok,world)    = writeDynamic "-Calc"   (dynamic (toCalc2 "-" (-))) world
# (ok,world)    = writeDynamic "xCalc"   (dynamic (toCalc2 "*" (*))) world
# (ok,world)    = writeDynamic "divCalc" (dynamic (toCalc2 "/" (/))) world
= ("Writing dynamic succeeded.λn",world)

toCalc2:: String (Real Real → Real) → (String,Real Real → Real)
toCalc2 s f = (s,f))
```

This calculator is restricted to dynamics with a very specific type. In principle, any functionality can be stored on disk as a dynamic. For instance, the calculator itself, which has type `*World → *World`[12], can be written to disk as a dynamic. The calculator can then be started from the shell or by any other application.

[12] In Section 2.1 we mentioned that object of certain types, including the built-in `World` type, cannot be packed. However, here we are not dealing with an object of this type, but with a function manipulating such an object. Storing such a function is permitted.

5 Implementation of the Shell Interpreter

Esther's command line language offers a simple yet complete (higher order, typed, pure and lazy) functional language. After parsing and performing the usual statical checks the main tasks to do are type checking and code generation.

In Section 2.2.2 we have shown that with a dynamic apply two unknown dynamics can be applied type safely on each other. As a matter of fact, Clean's run-time unification for dynamics using pattern matching on types is powerful enough to do the type checking for a complete functional language. To execute a shell command we furthermore don't need help from a special interpreter nor do we need help from the compiler. If we are able to define all new functionality in terms of existing ones, we can obtain the required code by just combining the corresponding existing code and store it for later use in a dynamic again.

An old technique serves very well in this context: we perform bracket abstraction (e.g. see [Dil88]) to compile a shell command into a semantically equivalent expression of pre-defined combinators (SKI-code). Since only a small fixed number of such combinators is used, we do not need to generate new code at all: every function definition and every expression can in this way be converted to a (new) combination of existing code that are glued together with combinators. The combinator expressions are known not to be very efficient, but our intention is to use the interpreter to compose existing functions and applications stored as dynamic, not to define large new functions (although there is no limitation on the complexity a function may have). In most cases, the inefficiency caused by the few combinators which appear as glue in the translation will be negligible.

In Section 5.1 we first explain how a shell command line is translated to combinators using the dynamic type check facility for verifying type consistency. In particular we pay attention to the transformation of case expressions. In Section 5.2 the combinator expression thus obtained is packed into a dynamic again such that it can be written to disk for further use in the shell or use by other applications.

Almost the entire implementation of Esther has been be written in plain Clean. The exception is formed by pattern matches on user defined data types, which can only be realized with help of a very few low-level routines.

5.1 Converting a Shell command line to combinators

The parsing of a shell command line is trivial and we will assume here that the string has already been successfully parsed. In order to provide a basic, but complete, functional language in our shell we need to support function definitions, lambda, let(rec), and case expressions. This leads to the following inductive syntax for expressions. Below, x ranges over variables, $\mathbf{b}$ over basic values, and d over dynamic values (actually references to dynamic values residing on disk).

$$\begin{array}{lcl} E & ::= & x \mid \mathbf{b} \mid d \mid E\,E' \mid \lambda x.E \mid \\ & & \textsc{let}\ v = E\ \textsc{in}\ E' \mid \textsc{case}\ E\ \textsc{of}\ P_1 \rightarrow E_1 \cdots P_n \rightarrow E_n \\ P & ::= & d\ y_1 \cdots y_k | \mathbf{b} \end{array}$$

In Clean, we represent this syntax by the following algebraic type definition. In the grammar specification (not explicitly mentioned in this paper) there is no denotation for dynamic values. The identifiers occurring free in an expression (i.e. not bound by a surrounding let, case of lambda) are interpreted as names of dynamics and are read from disk during code generation as explained in Section 4.1.

```
:: VId :== String
:: DId :== String

:: Exp  = Var VId
        | Bas BVal
        | Dyn DId
        | App Exp Exp
        | Lam VId Exp
        | Let VId Exp Exp
        | Cas Exp [(Pat,Exp)]

:: Pat  = DP DId [VId] | BP BVal

:: BVal = BInt Int | BBool Bool | BChar Char | BReal Real | BStirng String
```

For reasons of clarity, the transformation of expressions into dynamics is split up into two steps using the following intermediate *combinator* language. This time, d ranges over real dynamic values, i.e. expressions of type `Dynamic`, and x once again over variables.

$$\begin{array}{lcl} \mathrm{C} & ::= & x \mid d \mid \mathrm{C}\,\mathrm{C}' \mid \mathrm{IKSY} \\ \mathrm{IKSY} & ::= & I \mid K \mid S \mid Y \end{array}$$

Basic values are packed in dynamics. The variables in the combinator expressions are auxiliary and will not appear in the final code. The representation of this syntax in Clean is as follows.

```
:: Comb = CVar VId | CDyn Dynamic | CApp Comb Comb | I | K | S | Y
```

Except for CASE expressions, the translation of expressions into combinators is straightforward as is shown by the function `Exp2Comb`. In `Exp` dynamics are represented by names (referring to dynamic values on disk) whereas `Comb` expects them as real values. This means that they have to be read during the translation, which requires the unique `World` object to be passed as an environment. The LET expressions can be recursive which explains the presence of the fixed point combinator `Y`.

```
Exp2Comb :: Exp *World → (Comb, *World)
Exp2Comb (Var v) world  = (CVar v, world)
Exp2Comb (Bas b) world  = (CDyn (BVal2Dyn b), world)
Exp2Comb (Dyn d) world
    # (ok, dyn, world) = readDynamic d world
    | ok               = (CDyn dyn, world)
```

```
    | otherwise        = (raise [13]("file " +++ d +++ " not found"), world)
Exp2Comb (App f a) world
    # (fc, world) =  Exp2Comb f world
      (ac, world) =  Exp2Comb a world
    = (CApp fc ac, world)
Exp2Comb (Lam v e) world
    # (ec, world) =  Exp2Comb e world
    = (abstr v ec, world)
Exp2Comb (Let v d e) world
    # (dc, world) =  Exp2Comb d world
      (ec, world) =  Exp2Comb e world
    = (CApp (abstr v ec) (CApp Y (abstr v dc)), world)
Exp2Comb (Cas e alts) world
    # (ec, world) =  Exp2Comb e world
    # (ac, world) =  Cas2Comb alts world
    = (CApp ac ec, world)

BVal2Dyn:: BVal → Dynamic
BVal2Dyn (BInt i)   = dynamic i
BVal2Dyn (BBool b)  = dynamic b
...                                          // for all basic types
```

The function `Exp2Comb` makes use of the function `BVal2Dyn` which packs basic values into a dynamic, and the function `abstr` which is explained below.

Bracket abstraction translates lambda expressions into combinator (S,K and I) applications. Usually, this transformation is described with the aid of an auxiliary function `abstr` that eliminates a given variable from an (combinator) expression. Here `fvs` is a predicate indicating whether a variable occurs free in an expression.

```
abstr :: VId Comb → Comb
abstr v c
    | fvs v c
        = case c of
            CVar w      → I
            CApp c1 c2  → CApp (CApp S (abstr v c1)) (abstr v c2)
        = CApp K c
```

Special combinators and combinator optimization rules are often used to improve the speed of the generated combinator code by reducing the number of combinators [CF58]. One has to be careful not to optimize the generated combinator expressions in such a way that the resulting type becomes too general. In an untyped world this is allowed because they preserve the intended semantics when generating untyped (abstract) code. However, our generated code is contained within a dynamic and is therefore typed. This makes it essential that we preserve the *principal type* of the expression during bracket abstraction. Consider adding eta-reduction to `abstr` as follows:

[13] For easier error reporting, we implemented imprecise user-defined exceptions à la Haskell [MPMR01]. We used dynamics to make the set of exceptions extensible.

```
abstr v c
    | fvs v c
        = case c of
            CVar w      → I
            CApp c1 c2
                | c2 == CVar v
                    = c1
                    = CApp (CApp S (abstr v c1)) (abstr v c2)
        = CApp K c
```

With this version of `abstr`, the expression $\lambda f.\lambda x.fx$ is transformed to the (eta)-equivalent expression I. However, the first one has type (`∀a b: (a → b) → a → b`) which is less general than the type (`∀a : a → a`) of I. Such optimizations might prevent us from getting the principal type for an expression.

The transformation of cases is much more involved, and explained in more detail in the next section.

This combinator expression has the same type as the original expression would have. Obviously, this will only work correctly if the transformation from expressions to combinators preserves types. This has been proven in [SW06].

5.1.1 Case Expressions

Case expressions are transformed (by `Cas2Comb`) into a cascade of if-then-else constructs. For each alternative `Alt2Comb` creates a dynamic containing a function that expects three arguments: the then part (executed if the pattern match succeeds), the else part (executed if the pattern match fails), and the expression on which the pattern match is performed. The latter has already been converted to a combinator expression (see the definition of `Exp2Comb` in previous section).

In essence, the then part will be the (translated) right-hand side of the corresponding alternative, and the else part is a cascade of the remaining alternatives.

Matching basic values is done using `MatchBasic` that uses Clean's built-in equalities for each basic type. We always add a default alternative, using the `mismatch` function, that informs the user that none of the patterns matched the expression.

```
Cas2Comb :: [(Pat,Exp)] *World → (Comb, *World)
Cas2Comb [(pat,rhs) : alts] world
    # (rhs',world)  = Exp2Comb rhs world
    # (alts',world) = Cas2Comb alts world
    = (Alt2Comb pat rhs' alts', world)
Cas2Comb [] world
    = (CDyn mismatch, world)

Alt2Comb :: Pat Comb Comb → Comb
Alt2Comb (BP b) th el
    = case b of
        BInt i  → CApp (CApp (CDyn (MatchBasic i)) th) el
        BBool b → CApp (CApp (CDyn (MatchBasic b)) th) el
        ...                                          // for all basic types
```

```
Alt2Comb (DP dn vars) th el
    = CApp (CApp (CDyn (MatchDyn dn)) (CApp (CDyn applyTo dn) (foldr abstract vars th))) el

MatchBasic :: a → Dynamic | TC, Eq a
MatchBasic bv = dynamic λth el ie → if (bv == ie) th (el ie) :: ∀b: b (a^ → b) a^ → b
```

Note the optical asymmetry in `MatchBasic` between the then (argument `th`) and else (argument `el`) part: the then part already expects that the pattern has succeeded and therefore does not require the inspected expression (argument `ie`) anymore. Conversely, the else part might still contain further tests on this expression in order to determine the matching alternative.

The treatment of algebraic constructors is much more difficult. First of all, patterns in which these constructors appear might contain variables. These variables have to be bound to the arguments of the actual matching constructor. And secondly, constructors are contained in dynamics. For example, if we use Clean's list constructor `[:]` in a pattern, Esther will assume that the definition is stored in a dynamic. However, such a dynamic will actually contain an expression that consists of a plain constructor, i.e. with no arguments. Hence, in case of the list constructor we find that it has type `a → [a] → [a]`. The offered expression on which the pattern match is performed will contain a fully applied constructor (of type `[T]`, for some concrete type `T`).

To solve the first problem, we provide the following low-level function.

```
selectArg  :: Int Int a → b
```

In an application `selectArg i n e`, the $\mathtt{i}^{\text{th}}$ argument from node `e` with arity `n` is selected. This function has to be hand-coded, because selecting an arbitrary argument from a node is not expressible in Clean.

The second problem is solved using Clean's library function

```
compareConstructors  :: a a → Bool
```

that compares two nodes delivering true if these nodes contain identical constructors, and false otherwise. However, if two constructors have a different number of arguments they are considered as being not equal. Therefore, we first complete the node containing the plain pattern constructor by adding dummy arguments until it is saturated. We use the fact that such a saturated node has no function type anymore, to stop this process.

```
MatchAny :: Dynamic → Dynamic
MatchAny (f :: a → b)  = MatchAny (dynamic f undef :: b)
MatchAny (x :: a)      = dynamic λth el y → if (compareConstructors x y) (th y) (el y)
                                :: ∀b: (a → b) (a → b) a → b
```

If a pattern match succeeds, we need to extract the arguments from the expression and pass them to the right-hand side of the corresponding alternative. This is done by the `applyTo` function, which decides how many arguments need to be extracted (and what their types are) also by inspection of the type of the curried constructor. A little unsatisfactory is the fact that we cannot define this function recursively, but have to treat all different arities separately. This induces a maximum on the arity of the constructor allowed in a pattern.

```
applyTo :: Dynamic → Dynamic

applyTo ...              // and so on, most specific type first...

applyTo (_ :: a b → c)  = dynamic λf x → f (selectArg 1 2 x) (selectArg 2 2 x)
                                :: ∀d: (a b → d) c → d
applyTo (_ :: a → b)    = dynamic λf x → f (selectArg 1 1 x)
                                :: ∀c: (a → c) b → c
applyTo (_ :: a)        = dynamic λf x → f :: ∀b: b a → b
```

As explained in Section 4.2.6, pattern matching against user defined constructors requires that the constructors are available, i.e. stored in the file system. For this reason a large part of the Clean standard environment is stored in dynamics. This makes it possible to use standard functions like `+`, `map`, `foldr`, etc. in the shell and allows case distinction on list constructors and the like.

5.2 Translating combinators into dynamics

We introduce the function `comb2Dyn`, that converts a combinator expression into a dynamic. It is based on the dynamic apply function as introduced in Section 2.2.2

```
comb2Dyn :: Comb → Dynamic
comb2Dyn (CDyn d) = d
comb2Dyn (CApp cf cx)   = case (comb2Dyn cf, comb2Dyn cx) of
    (f :: a → b, x :: a) → dynamic f x :: b
    (df, dx)             → raise ("Cannot apply " +++ typeOf df
                                  +++ " to " +++ typeOf dx)
where
    typeOf :: Dynamic → String
    typeOf dyn = toString (typecodeOfDynamic dyn) // pretty print type

comb2Dyn I = dynamic λx → x :: ∀a: a → a
comb2Dyn K = dynamic λx y → x :: ∀a b: a b → a
comb2Dyn S = dynamic λf g x → f x (g x) :: ∀a b c: (a b → c) (a → b) a → c
comb2Dyn Y = dynamic λf → let x = f x in x :: ∀a: (a → a) → a
```

Note that we do not perform any type inference explicitly, but solely use the built-in dynamic unification to reconstruct the type. As the combinators themselves serve as glue for combining existing pieces code, the types of these combinators serve as glue for combining the types associated with these pieces, ensuring type consistency.

5.3 Performance Evaluation

In this section we briefly discuss some efficiency issues of dynamics. In section **??** we mentioned that any application that uses dynamics will be dynamically linked during start-up. Compared to launching a statically linked application, the delay is mainly caused by the fact that several object files have to be read

from disc. It appears that the linking time itself is negligible. For instance, building and launching the complete shell interpreter from the individual object files takes about 0.6 seconds. The size of resulting executable (constructed out of 20 object files) is approximately 2 MB. Just reading the same amount of data from 20 ordinary data files will probably take the same amount of time. Furthermore, we have wrapped the shell in a dynamic, and written this dynamic to disc. Then, in separate application, we have read the wrapped shell from disc, and started its execution. This does not produce any measurable overhead.

We have also performed some test with the shell itself, particularly concerning the translating of shell commands into combinator expressions. As said before, it is known that this translation produces inefficient code. Therefore it does not make much sense to compare these expressions with complied code. Instead, we ran some benchmark tests and compared our implementation with different Haskell interpreters, more specifically, with Helium, Hugs and GHCi. The following benchmark programs were used.

1. **Prime Sieve**, calculating the 5000^{th} prime number;
2. **Fibonacci**, calculating the 35^{th} fibonacci number;
3. **Hamming**, generating an infinite list of hamming numbers from which the 1000^{th} element is taken, repeated 4000 times.

The run-time results[14] are shown in Table 1. For the sake of completeness, we also included the figures for Haskell (GHC 6.4 compiler) and Clean (the V2.1 compiler).

Table 1: Run-times are in seconds, and include garbage collection

	Sieve	Fib	Hamming
Esther	51	14.8	43.6
Helium	14.7	11.4	11.0
GHCi	19.5	36.0	20.2
Hugs	47,6	111.0	30.9
Haskell	0.97	0.19	1.2
Clean	0.97	0.19	1.2

The performance of Esther are better than we had expected. In the fibonacci programma, that only involves calculations on numbers, Esther is even faster than GHCi and Hugs. The inefficient treatment of pattern matching probably causes Esthers (relatively) low execution speed for the other two programs.

[14]The programs were executed on a AMD Athlon^{TM} 64 processor (3500+) under Windows XP.

But again, the intention was not to write an efficient interpreter, but to illustrate our claim that dynamics in Clean provide a solid basis for the implementation of various applications communicating with their environment in a type safe way.

Standard shells combine applications via special built-in commands, like pipe, filter, and tee. In our case, applications are dynamics stored on disk which can be combined by our shell via (type safe) function applications. Obviously, this increases both flexibility and reliability. However, the shell is not intended to specify complete, standalone applications. In that case one is supposed to use the compiler. The use of the shell should be restricted to building ad-hoc combinations and scripting only. In this way a shell script will contain only very few combinators and run fast due to the efficient code produced by the Clean compiler.

6 Related Work

To the best of our knowledge there does not exist any other system which can dynamically combine, store, load compiled code in a type safe way comparable to ours. There are many systems though, even systems which are more recent then ours (our first publications stem from 2002), which offer part of the described functionality.

The idea to have a hybrid type system with both static and dynamic types is not new. See, for instance, [ACPP91], [ACP+92], [LM91], [GKT+06], and [Dug99]. One motivation for having an hybrid type system has been the handling of types which cannot be handled by the static type system alone. Our main motivation for having an hybrid type system is another one and given by the observation that one simply needs dynamic types for the type safe exchange of information (data and functions) between separately compiled applications. In this paper we show how this can be realized, an aspect which is not addressed by the papers above.

Unsafe dynamic linking of code is a standard technology which is available for almost any programming language. Type safe linking of plugins is nowadays becoming more and more available for modern programming languages. A standard technique is that the compiler can generate such a typed plugin from source code. These plugins not only contain dynamically linkable code but are accompanied by a type signature which can be checked for type consistence when the plugin is read. Compared to our approach this technique is rather rigid. Special about our approach is that at run-time any application can write any expression as dynamic, and such dynamics can be dynamically read and combined to form new dynamics. The code can be combined without any need for a compiler.

In [PSSC04] a library enabling type safe plugins for Haskell is described. Like us, they make use of dynamic types and they can dynamically extend a running application in a type safe way. Haskell lacks the special language and run-time support for dynamics as we have in Clean, and therefore a lightweight

approach is used instead. To create a Haskell plugin an interface description has to be made that is used by the Haskell compiler to promote the generated code to a plugin. The application reading such a plugin uses the same interface description for verification. So, the types of the plugins that one can create and the type an application can accept is determined at compile time. In their system no separation is made between user and system space for the storage of dynamics, like we do. Hence, it cannot be guaranteed that the type stored in the dynamic is a correct one. There are some additional limitations which are due to the fact that dynamics in Haskell are more limited than dynamics in Clean, e.g. one cannot deal with polymorphic types.

Recently, dynamic creation of type safe plugins have become possible for other (impure) functional languages such as ML and OCaml. For instance, in Alice ML [AR06] a module can be dynamically packed and unpacked. Packed modules can be stored to disk and read by another Alice application. When, during unpacking, a package does not match the expected type signature, an exception is raised. In our approach, we can make a case distinction using pattern matching on types and values. The types of different dynamics can be compared with each other; the application itself does not need to be aware of the types involved. New dynamics can be composed dynamically from existing ones, and type checking and unpacking of code is done lazily. In a stored Alice package all the corresponding (eager) code is stored, while we share the (lazy) code via the references to the code and type repositories. An important difference with our system is that Alice does not offer the dynamic creation of new packages from existing ones, so interactive combining of plugins is not possible.

An approach quite similar to Alice ML is taken by Acute [SLW+05] which is an extension of OCaml. Acute focusses on distributed development, deployment, and execution of distributed applications. In this setting they have to deal with different communication scenarios (E.g. do sender and receiver originate from the same program or are they separately built programs, do they share or no share source code). Our system is much simpler: every application once compiled has it's own private type and object code repository to which dynamics refer to. No version conflict can occur, and it is always clear which types and code are meant. The down side of our approach is that we currently might duplicate code unnecessary when two dynamics refer to different applications that share some source code. It is relatively simple to improve this by refining the internal structure of the code repositories such that code shared by applications are also shared in the object code repositories, and hence dynamically linked in only once.

We currently don't allow the storage of abstract types in a dynamic. The problem is that one somehow has to guarantee that the same intended implementation is used which cannot be determined by checking the involved types only. A solution for this problem is given by Rossberg [Ros06] who presents a calculus for dynamic type generation. In [LPSW03], a practical solution is given for testing the type equality of abstract types by using a hashing technique. If a module is hashed during builds of two different programs at different sites, the same hash will be obtained. Also in Alice and Acute one has chosen for a

similar kind of hash encoding. Such an encoding solution could be applied in our system as well which would enable us to test the type equality for abstract types.

Es [HR93] is a shell that supports higher-order functions and allows the user to construct new functions at the command line. A UNIX shell in Haskell [Mat] by Jim Mattson is an interactive program that also launches executables, and provides pipelining and redirections. Tcl [Ous90] is a popular tool to combine programs, and to provide communications between them. None of these programs provide a way to read and write typed objects, other than strings, from and to disk. Therefore, they cannot provide our level of type safety.

A functional interpreter with a file system manipulation library can also provide functional expressiveness and either static or dynamic type checking of part of the command line. For example, the Scheme Shell (ScSh, [Shi94]) integrates common shell operations with the Scheme language to enable the user to use the full expressiveness of Scheme at the command line. Interpreters for statically typed functional languages, such as Hugs [JR02], even provide static type checking in advance. Although they do type check source code, they cannot type check the application of binary executables to documents/data structures because they work on untyped executables.

Erlang [AVWW96] is a functional language specifically designed for the development of concurrent processes. It is completely dynamically typed and primarily uses interpreted byte-code, while our system is mostly statically typed and executes native code generated by the Clean compiler. A simple spelling error in a token used during communication between two processes is often not detected by Erlang's dynamic type system, sometimes causing deadlock.

The BeanShell [Nie] is an embeddable Java source interpreter with object scripting language features, written in Java. It is capable of inferring types for variables and to combine shell scripts with existing Java programs. While we generate compiled code via dynamics, the BeanShell interpreter is invoked each time a script is called from a normal Java program.

Back et al. [BTS+98] built two prototypes of a Java operating system. Although they show that Java's extensibility, portable byte code and static/dynamic type system provides a way to build an operating system where multiple Java programs can safely run concurrently, Java does not support dynamic type unification, higher-order functions, and closures as we do.

7 Conclusions

In this paper, we presented a general and flexible framework for extending the functionality of Clean applications at run-time. This framework is based on the concept of *dynamics*: an object of type Dynamic that consists of an expression and (a representation) of the expression's static type. These objects cannot only be manipulated by one and the same application but also exchanged between possibly completely different applications even running on different computers. By extending the run-time system of Clean with *dynamic unification* the ex-

change of dynamics is guaranteed to be type-safe.

We have demonstrated the expressive power of dynamics by building a shell that provides a simple, but powerful strongly typed functional programming language. We were able to do this using solely Clean's support for run-time type unification and dynamic linking. The Esther shell supports type checking and type inference for command-line expressions before they are executed. It offers application, lambda abstraction, recursive let, pattern matching, and function definitions: the basics of any functional language. Additionally, infix operators and support for overloading make the shell easy to use.

By combining code from compiled functions/programs, Esther allows the use of any pre-compiled program (plugins) as a function in the shell. Because Esther stores functions/expressions constructed at the command line as a Clean dynamic, it supports writing compiled programs at the command line. Furthermore, these expressions written at the command line can be used in any pre-compiled Clean program. An example of such a pre-compiled Clean program is a calculator, of which the functionality is completely built at run-time, by interactively using the shell. The evaluation of expressions using recombined compiled code is in general almost as fast as using the native code produced by the Clean compiler due to the fact that most of the code will be compiled in a standard way.

One of the main differences of dynamics in Clean compared to other implementations is that they are not just a proof of concept. Dynamics form a transparant, flexible, and efficient extension of the programming language.

Acknowledgements

We are grateful to Peter Achten for his valuable comments on this paper.

References

[Ach07] Peter Achten. Clean for Haskell98 Programmers – A Quick Reference Guide –. Available at: `http://www.st.cs.ru.nl/papers/-2007/CleanHaskellQuickGuide.pdf`, July 13 2007.

[ACP+92] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, Gordon Plotkin, and Didier Rèmy. Dynamic typing in polymorphic languages. In *Proceedings of the ACM SIGPLAN Workshop on MAL and its Applications*, 1992.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *TOPLAS'91: ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[AR06] Guido Tack Gert Smolka Andreas Rossberg, Didier Le Botlan. Alice through the looking glass. In *Trends in Functional Programming*, volume 5, Bristol, UK, 2006. Intellect Books.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[BTS+98] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Java operating systems: design and implementation. Technical Report UUCS-98-015, University of Utah, 1998.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*. North Holland, 1958.

[Dil88] Antoni Diller. *Compiling Functional Languages*. John Wiley & Sons, 1988.

[Dug99] Dominic Duggan. Dynamic typing for distributed programming in polymorphic languages. *ACM Trans. Program. Lang. Syst.*, 21(1):11–45, 1999.

[GKT+06] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *In Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[HR93] Paul Haahr and Byron Rakitzis. Es: a shell with higher-order functions. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 51–60, 1993.

[JR02] Mark P. Jones and Alastair Reid. *The Hugs 98 User Manual*. The Yale Haskell Group and the OGI School of Science and Engineering at OHSU, 2002. `http://www.haskell.org/hugs/`The Hugs web site.

[Koo90] Pieter Koopman. *Functional Programs as Executable Specifications*. PhD thesis, University of Nijmegen, The Netherlands, 1990.

[LM91] Xavier Leroy and Michel Mauny. Dynamics in ml. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 406–426, London, UK, 1991. Springer-Verlag.

[LPSW03] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 87–98, New York, NY, USA, 2003. ACM.

[Mat] Jim Mattson. The Haskell shell. Online source code. `http://www.informatik.uni-bonn.de/řalf/software/examples/Hsh.html`Ralf Hinze's web site.

[MPMR01] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *PLDI'01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 274–285. ACM Press, 2001.

[Nie] Pat Niemeyer. Beanshell. Online web site. `http://www.beanshell.org/`The Beanshell web site.

[Ous90] John K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 133–146. USENIX Association, 1990.

[Pil99] M.R.C. Pil. Dynamic types and type dependent functions. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL '98)*, volume 1595 of *LNCS*, pages 169–185. Springer Verlag, 1999.

[PJHJea99] S. Peyton Jones and Hughes J. et al. *Report on the programming language Haskell 98.* University of Yale, 1999. http://www.haskell.org/definition/.

[PSSC04] André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging haskell in. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 10–21. ACM Press, 2004.

[PvE01] Rinus Plasmeijer and Marko van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/∼clean/.

[Ros06] Andreas Rossberg. The missing link: dynamic components for ml. *SIGPLAN Not.*, 41(9):99–110, 2006.

[Sch24] Moses Schönfinkel. Über die bausteine der mathematischen logik. In *Mathematische Annalen*, volume 92, pages 305–316. Springer, 1924.

[Shi94] Olin Shivers. A Scheme shell - the design paper on the Scheme shell scsh. Technical Report MIT/LCS/TR-635, MIT Laboratory for Computer Science, 1994. `http://www.faqs.org/faqs/unix-faq/shell/scsh-faq/`The Scheme Shell FAQ.

[SLW+05] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: high-level programming language design for

distributed computation. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 15–26, New York, NY, USA, 2005. ACM.

[SW06] Sjaak Smetsers and Arjen Weelden, van. Bracket Abstraction Preserves Typability - A formal proof of Diller-algorithm-C in PVS. In J. Levy, editor, *Proceedings 20th International Workshop on Unification (UNIF'06)*, pages 29–43, Seattle, Washington, August 11 2006.

[vP03a] Arjen van Weelden and Rinus Plasmeijer. Towards a strongly typed functional operating system. In Ricardo Peña and Thomas Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 215–231. Springer, September 2003.

[VP03b] Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In Ricardo Peña and Thomas Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 101–117. Springer, September 2003.