A Functional Shell that Operates on Typed and Compiled Applications

Rinus Plasmeijer and Arjen van Weelden

Computer Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
rinus@cs.kun.nl, arjenw@cs.kun.nl

Abstract. Esther is the interactive shell of Famke, a prototype implementation of a strongly typed operating system written in the functional programming language Clean. As usual, the shell can be used for manipulating files, applications, data and processes at the command line. Special about Esther is that the shell language provides the full basic functionality of a strongly typed lazy functional language. The shell type checks each command line and only executes well-typed expressions. Files are typed as well, and applications are simply files with a function type.
The implementation of the shell has some unusual aspects. The type checking/inferencing performed by the shell is actually performed by the hybrid static/dynamic type system of Clean. The shell behaves like an interpreter, but it actually executes a command line by combining existing code of functions on disk. Cleans dynamic linker is used to store (and retrieve) any expression (both data and code) with its type on disk. This linker is also used to communicate values of any type, e.g., data, closures, and functions (i.e.compiled code), between running applications in a type safe way.
The shell combines the advantages of interpreters (direct response) and compilers (statically typed, fast code). Applications (compiled functions) can be used, in a type safe way, in the shell, and functions defined in the shell can be used by any compiled application.

1 Introduction

Functional programming languages like Haskell [1] and Clean [2, 14] offer a very flexible and powerful static type system. 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, this works only within a single application.

Independently developed applications often need to communicate with each other. One would like the communication of objects to take place in a type safe manner as well. And not only simple objects, but objects of any type, including functions. In practice, this is not easy to realize: the compile time type information is generally available to the compiled executable at run-time. In real life therefore, applications often only communicate simple data types like

streams of characters, ASCII text, or use some ad-hoc defined (binary) format. Although more and more applications use XML to communicate data together with the definitions of the data types used, most programs do not support run-time type unification, cannot use previously unknown data types, or cannot exchange functions (i.e. code) between different programs in a type safe way. This is mainly because the used programming language has no support for such things.

Programming languages, especially pure and lazy functional languages like Clean and Haskell, provide good support for abstraction (e.g. subroutines, overloading, polymorphic functions), composition (e.g. application, higher-order functions, module systems), and verification (e.g. strong type checking and inference).

In contrast, command line languages used by operating system shells usually have little support for abstraction, composition, and especially verification. They do not provide higher-order subroutines, complex data structures, type inference, or even type checking at all before evaluation. Given their limited set of types and their specific area of application, this has not been recognized as a serious problem in the past.

We think that command line languages can benefit from some of the programming language facilities, as this will increase their flexibility, reusability and security. We have previously done research on reducing run-time errors (e.g. memory access violations, type errors) in operating systems by implementing a micro kernel in Clean that provides type safe communication of any value of any type between functional processes, called Famke (*F*unction*A*l *M*icro *K*ernel *E*xperiment) [15]. This has shown that (moderate) use of dynamic typing [3], in combination with Clean's dynamic run-time system and dynamic linker [4, 16], enables processes to communicate any data (and even code) of any type in a type safe way.

During the development of a shell/command line interface for our prototype functional operating system it became clear that a normal shell cannot really make use (at run-time) of the type information derived by the compiler (at compile-time). To reduce the possibility of run-time errors during execution of scripts or command lines, we need a shell that supports abstraction and verification (i.e. type checking) in the same way as the Clean compiler does. In order to do this, we need a better integration of compile-time (i.e. static typing) and run-time (i.e. interactivity) concepts.

Both the shell and micro kernel are built on top of Clean's hybrid static/dynamic type system and its dynamic I/O run-time support. It allows programmers to save any Clean expression, i.e., a graph that can contain data, references to functions, and closures, to disk. Clean expressions can be written to disk as a *dynamic*, which contains a representation of their (polymorphic) static type, while preserving sharing. Clean programs can load dynamics from disk and use run-time type pattern matching to reintegrate it into the statically typed program. In this way, new functionality (e.g., plug-ins) can be added to a running program in a type safe way.

The shell is called Esther (*E*xtensible *S*hell with *T*ype c*H*ecking *E*xpe*R*iment), and is capable of:

- reading an expression from the console, using Clean's syntax for a basic, but complete, functional language. It offers application, lambda abstraction, recursive let, pattern matching, function definitions, and even overloading;
- using compiled Clean programs as typed functions at the command line;
- defining new functions, which can be used by other compiled Clean programs (without using the shell or an interpreter);
- extracting type information (and indirectly, code) from dynamics on disk;
- type checking the expression, and solving overloading, before evaluation;
- constructing a new dynamic containing the correct type and code of the expression.

First, we introduce the static/dynamic hybrid type system and dynamic I/O of Clean in Sect. 2. In Sect. 3 we give an overview of the expressive power of the shell command language using tiny examples of commands that can be given. In Sect. 4 we show how to construct a dynamic for each kind of subexpression such that it has the correct semantics and type, and how to compose them in a type checked way. Related work is discussed in Sect. 5 and we conclude and mention future research in Sect. 6.

2 Dynamics in Clean

Clean offers a hybrid type system: in addition to its static type system it also has a (polymorphic) dynamic type system [3, 4, 16]. A dynamic in Clean is a value of static type *Dynamic*, which contains an expression as well as a representation of the (static) type of that expression. Dynamics can be formed (i.e. lifted from the static to the dynamic type system) using the keyword **dynamic** in combination with the value and an optional type. The compiler will infer the type if it is omitted[1].

dynamic 42 `::` `Int`[2]
dynamic `map fst ::` `A`[3]`.a b:` `[(a, b)]` → `[a]`

Function alternatives and case patterns can pattern match on values of type *Dynamic* (i.e. bring them from the dynamic back into the static type system). Such a pattern match consist of a value pattern and a type pattern. 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. The compiler translates a pattern match on a type into run-time type unification. If the unification fails, the next alternative is tried, as in a common (value) pattern match.

[1] Types containing universally quantified variables are currently not inferred by the compiler. We will not always write these types for ease of presentation.
[2] Numerical denotations are not overloaded in Clean.
[3] Clean's syntax for Haskell's `forall`.

```
::[4]Maybe a = Nothing | Just a

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

A type pattern can contain type variables which, provided that run-time unification is successful, are bound to the offered type. In the example below, `dynamicApply` tests if the argument type of the function `f` inside its first argument can be unified with the type of the value `x` inside the second argument. If this is the case then `dynamicApply` can safely apply `f` to `x`. The type variables a and b will be instantiated by the run-time unification. At compile time it is generally unknown what type a and b will be, but if the type pattern match succeeds, the compiler can safely apply `f` to `x`. This yields a value with the type that is bound to b by unification, which is wrapped in a dynamic.

```
dynamicApply :: Dynamic Dynamic → Dynamic
[5]

dynamicApply (f :: a → b) (x :: a) = dynamic f x :: b[6]
dynamicApply     df           dx   = dynamic "Error: cannot apply"
```

Type variables in dynamic patterns can also relate to a type variable in the static type of a function. Such functions are called type dependent functions [3]. A caret (^) behind a variable in a pattern associates it with the type variable with the same name in the static type of the function. The static type variable then becomes overloaded in the predefined `TC` (or type code) class. The `TC` class is used to 'carry' the type representation. In the example below, the static type variable t will be determined by the (static) context in which it is used, and will impose a restriction on the actual type that is accepted at run-time by `matchDynamic`. It yields `Just` the value inside the dynamic (if the dynamic contains a value of the required context dependent type) or `Nothing` (if it does not).

```
matchDynamic :: Dynamic → Maybe t | TC t[7]
matchDynamic (x :: t^) = Just x
matchDynamic  other    = Nothing
```

2.1 Reading and Writing of Dynamics

The dynamic run-time system of Clean supports writing dynamics to disk and reading them back again, possibly in another application or during another execution of the same application. This is not a trivial feature, since Clean is not an interpreted language: it uses compiled code. Since a dynamic may contain unevaluated functions, reading a dynamic implies that the corresponding code produced by the compiler has to be added to the code of the running application. To make this possible one needs a dynamic linker. Furthermore, one needs

[4] Defines a new data type in Clean, Haskell uses the `data` keyword.
[5] Clean separates argument types by whitespace, instead of →.
[6] The type b is also inferred by the compiler.
[7] Clean uses `|` to denote overloading. In Haskell this would be written as `(TC t)` ⇒ `Dynamic` → `Maybe t`.

to to be able to retrieve the type definitions and function definitions that are associated with a stored dynamic. With the ability to read and write dynamics, type safe plug-ins and mobile code can relatively easy be realized in Clean.

Writing a dynamically typed expression to file A dynamic of any value can be written to a file on disk using the `writeDynamic` function.

```
writeDynamic :: String Dynamic *[8]World → (Bool, *World)
```

In the `producer` example below a dynamic is created which consists of the application of the function `sieve` to an infinite list of integers. This dynamic is then written to file using the `writeDynamic` function. Evaluation of a dynamic is done lazily. The `producer` does not demand the result of the application of `sieve` to the infinite list. As a consequence, the application in its unevaluated form is written to file. The file therefore contains a calculation that will yield a potential infinite integer list of prime numbers.

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

When the dynamic is stored to disk, not only the dynamic expression and its type has to be stored somewhere. To allow the dynamic to be used as a plug-in by any other application additional information has to be stored as well. One also has to store:

- the code corresponding to the function definitions needed for the evaluation of the dynamic expression;
- the definitions of all the types involved needed to check type consistency when matching the type of the dynamic against the type specified in the dynamic pattern match.

The required code and type information will be generated by the compiler and is stored in a special data base when an application is created. The code and type information is created and stored once at compile-time, while the dynamic value and dynamic type are created and stored may be several times at run-time. The run-time system has to be able to find both type of information when a dynamic is read in.

[8] This 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 with the hidden state of the `IO` monad in Haskell.

Reading a dynamically typed expression from file A dynamic can be read from disk using the `readDynamic` function.

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

This `readDynamic` function is used in the `consumer` example below to read the earlier stored dynamic. The dynamic pattern match checks whether the dynamic expression is an integer list. In case of success the first 100 elements are taken, in this case of the potential infinite list of sieve numbers. In case that the read in dynamic is not of the indicated type, the consumer aborts. Actually, it is not possible to do something with a read-in dynamic (besides passing it around to other functions or saving it to disk again), unless the dynamic matches some type or type scheme specified in the pattern of the receiving application.

```
consumer :: *World → [Int]
consumer world
♯[9] (dyn, world) = readDynamic "primes" world
= take 100 (extract dyn)
where
    extract :: Dynamic → [Int]
    extract (list :: [Int]) = list
    extract else = abort "dynamic type check failed"
```

To turn a dynamically typed expression into a statically typed expression, the following steps are performed by the run-time system of Clean:

- The type of the dynamic and the type specified in the pattern are unified with each other. If the unification fails, the dynamic pattern match also fails.
- If the unification is successful, it is checked that the type definitions of equally named types coming from different applications are equal as well. If one of the involved type definitions differs, the dynamic pattern match fails. Equally named types are equivalent iff their type definitions are syntactically the same (modulo alpha-conversion and the order of algebraic data constructors).
- If all patterns match, the corresponding function alternative is chosen and evaluated.
- It is possible that the evaluation of the now statically typed expression dynamic expression is required. In that case, the expression is reconstructed out of the information stored in the dynamic on disk, the corresponding code needed for the evaluation of the functions is added to the running application, after which the expression can be evaluated.

Running `prog1` and `prog2` in the example below will write a function and a value to dynamics on disk. Running `prog3` will create a new dynamic on disk that contains the result of 'applying' (using the `dynamicApply` function) the dynamic with the name "function" to the dynamic with the name "value". The closure `40 + 2` will not be evaluated until the `*` operator needs it. In this case, because the 'dynamic application' of `df` to `dx` is lazy, the closure will not be evaluated until the value of the dynamic on disk named "result" is needed. Running `prog4`

[9] Clean's 'do-notation' for environment passing.

tries to match the dynamic `dr`, from the file named "result", with the type *Int*. After this succeeds, it displays the value by evaluating the expression, which is semantically equal to `let x = 40 + 2 in x * x`, yielding `1764`.

```
prog1 world = writeDynamic "function" (dynamic * :: Int Int -> Int) world

prog2 world = writeDynamic "value" (dynamic 40 + 2) world

prog3 world = let (ok1, df, world1) = readDynamic "function" world
                  (ok2, dx, world2) = readDynamic "value" world1
              in  writeDynamic "result" (dynamicApply df dx) world2

prog4 world = let (ok, dr, world1) = readDynamic "result" world
              in  (case dr of (x :: Int) -> x, world1)
```

A dynamic will be read in lazily after a successful run-time unification (triggered by a pattern match on the dynamic). The dynamic linker will take care of the actual linking of the code to the running application and the checking of the type definitions referenced by the dynamic being read. The dynamic linker canis able to find the code and type definitions in the data base in which they are stored by the compiler. The amount of data and code that the dynamic linker will link depends on how far the dynamic expression is evaluated.

Dynamics written by one application program can safely be read by any other application. Only when the types match, it can be plugged in such and the application can do something with it. In this way two Clean applications can communicate values of any type they like, including function types, in a type safe manner.

3 Overview of the Shell

Like any other shell, our Esther shell enables users to start pre-compiled programs and provide simple ways to combine multiple programs, e.g., pipelining and concurrent execution, and it 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.

A special feature of the Esther shell is that it offers a complete typed functional programming language with which programs can be constructed. The shell type checks a command line before performing any actions. 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. 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 underlying operating system and file system it relies on.

3.1 Famke; a Type Safe Micro Kernel

A shell has to be able to start applications and to provide a way to connect applications (e.g. by creating a pipe-line) such that they can communicate. Since our shell is typed, process communication should be type safe as well. The Windows Operating System that we use does not provide such a facility. We therefore have created a micro kernel on top of Windows. Our micro-kernel, Famke, provides Clean programs with ways to start new (possibly distributed running) processes, and the ability to communicate any value in a type safe way. It should be no surprise that Famke uses dynamics for this purpose. Dynamics can be send between applications as strings, which makes it possible to use conventional interprocess communication media such as TCP/IP for the actual communication (see [16]).

3.2 A Typed File System

A shell works on applications and data stored on disk. Our shell 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.

For applications written in Clean this can be easily realized. Any data, function, or even any large complete Clean application (which is a function as well) can be written as dynamic to disk, thus forming a rudimentary typed file system.

Applications written in other languages are usually untyped. We can in principle incorporate such an application into in our typed file system, by writing a properly typed Clean wrapper application around it, which is then stored again as dynamic on disk.

We assume that all documents and compiled applications are stored in a dynamic of appropriate type. Applications in our file system are just dynamics that contain a function type. This typed file system makes it possible for the shell to ensure, for example, that it is type safe to apply a printing application (`print :: WordDocument` → `PostScript`) to a document (`myDocument :: WordDocument`). The Clean dynamic type system will ensure that the types will indeed fit.

Normal directory manipulation operations still apply, but one no longer reads bytes from a file. Instead, one reads whole files (only conceptually, the dynamic linker reads it lazily), and one can pattern match on the dynamic to check the type. This removes the need for parsers and pretty printers, as data structures are stored directly.

The shell contains no 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.

The shell is therefore pretty useless unless a collection of useful dynamics has been stored. When the system is initialized, a standard file system is created (see Fig. 1) in a Windows folder. It contains:

- almost all functions from the Clean standard environment[10], such as `+`, `-`, `map`, and `foldr` (stored as dynamic on disk);

[10] Similar to Haskell's Prelude.

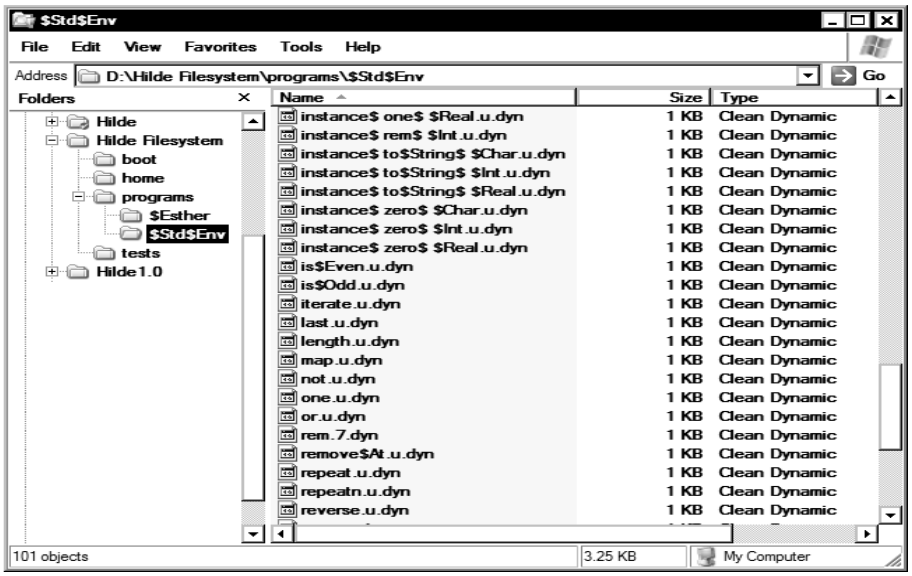

Fig. 1. A screenshot of the typed file system; implemented as dynamic on disk.

- common commands to manipulated the file system (`mkdir`, `rmdir`, and the like);
- commands to create processes directly based on the functionality offered by Famke (`famkeNewProcess`, and the like).

All folders are common Window folders, all files contain dynamics created by Clean applications using the `writeDynamic`-function. The implementation of dynamics on disk is organized in such a way ([16]]) that a user can safely rename, copy or delete files, either using the Esther shell or directly using Windows.

3.3 Esther; a Type Checking Shell

The last example of Sect. 2. shows how one can store and retrieve values, expressions, and functions of any type to and from the file system. It also shows that the `dynamicApply` function can be used to type check an application at run-time using the static types stored in dynamics. Combining both in an interactive 'read expression – apply dynamics – evaluate and show result' loop gives a very simple shell that already supports the type checked run-time application of programs to documents.

Esther performs the following steps in a loop:

- it reads a string from the console and parses it like a Clean expression. It supports denotations of Clean's basic and predefined types, application, infix operators, lambda abstraction, overloading, let(rec), and case expressions;

- 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;
- type checks the expression using dynamic run-time unification and type pattern matching, which also infers types;
- 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
>
```

Roughly the following happens. The shell parses the expression. The expression consists of typical Clean-like syntactical constructs (like (,), and [..]), constants (like 1 and 10), and identifiers (like `map` and `+`).

The names `map` and `+` are unbound (do not appear in the left hand side of a let, case, lambda expression, or function definition) in this example, and the shell therefore assumes that they are names of dynamics on disk. They are read from disk (with help of `readDynamic`), practically extending its functionality with these functions, and inspects the types of the dynamics. It uses the types of `map` (let us assume that the file `map` contains the type that we expect: $\forall$`a b: (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, extended with better error reporting. How this is done exactly, is explained in more detail in Sect.0 4. 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 dotdot expression.

Our shell 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 like `map` and `+` are used in other commands later on, the dynamic linker will notice that they are already have been used and linked in, and it will reuse their code. As a consequence, the shell will react even quicker, because no dynamic linking is required anymore in such a case.

3.4 The Esther Command Language

Here follow some command line examples with an explanation of how they are handled by the shell. Figure 2 show two example sessions with Esther. The right Esther window in Fig. 2 shows the same directory as the Windows Explorer screenshot in Fig. 1. We explain Esther's syntax by example below. Like a common UNIX shell, the Esther shell prompts the user with a ">" for typing in a new command.

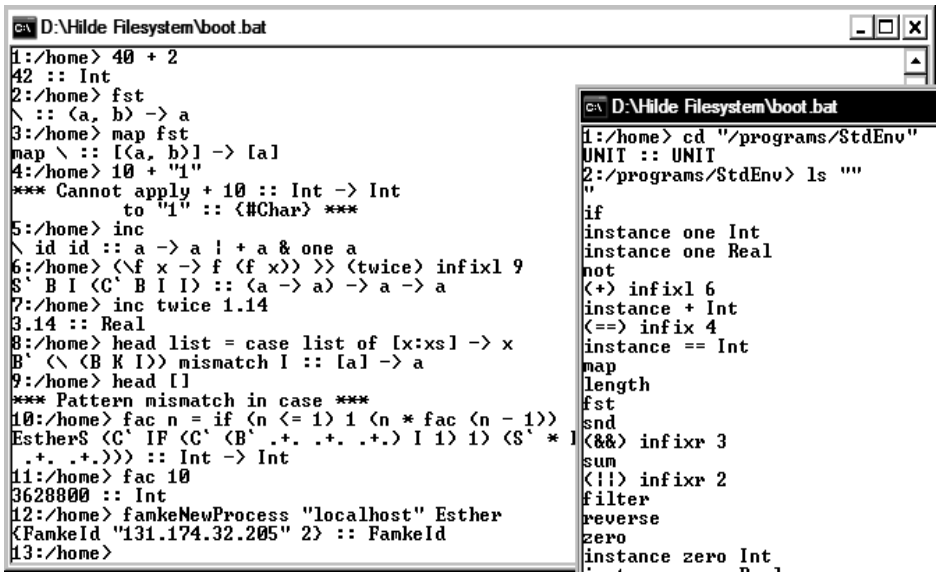

Fig. 2. A combined screenshot of the two concurrent sessions with Esther

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} ***
>
```

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 41
42 :: Int
```

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 that stores the class as an overloaded function. It is also not possible to save overloaded command-line expressions using the >> described above. Arithmetic operations are overloading in Esther, just as they are in Clean:

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

> one
one :: a | one a

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

Function Definitions One can define new functions at the command line:

```
> dec = (-) 1
dec :: Int -> Int
```

This defines a new function with the name `dec` as the partial application of the `-` function to the integer one. This function is written to disk in a file with the same name (`dec`) 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 Sect. 4), which explains why Esther responds in this way.

Not only is it possible to reuse such functions in the shell itself. Any function defined in the shell can be used by any other Clean application. Such a function

is a dynamic and can be used (read in, dynamically linked, copied, renamed, communicated across a network) as usual.

Notice that dynamics are read in before executing the command line, so it is not possible to change the meaning of a part of the command line by overwriting a dynamic.

Lambda Expressions It is possible to define lambda expressions, just as in Clean.

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

> (\x x -> x) "first-x" "second-x"
"second-x" :: String
```

Let Expressions To introduce sharing and cycles (infinite data structures), one can use let expressions.

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

> let ones = [1:ones] in take 100 ones
[1,2,3,4,5,6,  . . .  ,100] :: [Int]
```

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. But 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. But one can define the types in any Clean module, and construct an application writing the constructors as dynamic to disk.

```
module NewType

:: 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
```

Typical Shell Commands Esther's search path also contains a directory with common shell commands, such a file system operations:

```
> mkdir "foo"
UNIT :: UNIT
```

Esther displays `UNIT` because `mkdir` has type CleanInlineWorld -¿ World, i.e., has a side effect, but no result. Functions that operate on the Clean's `World` state are applied to the world by Esther.

More operations on the file system:

```
> cd "foo"
UNIT :: UNIT

> 42 >> bar
42 :: Int

> ls ""
"
bar
" :: {#Char}
```

Processes Examples of process handling commands:

```
> famkeNewProcess "localhost" Esther
{FamkeId "131.174.32.197" 2} :: FamkeId
```

This starts a new, concurrent, incarnation of Esther at the same computer. IP addresses can be used to start processes on other computers. `famkeNewProcess` yields a new process id (of type `FamkeId`). It is necessary to have the Famke running on the other computer, e.g., by starting a shell there, to be able to start a process on another machine. Starting Esther on another machine does no give remote access, the console of the new incarnation of Esther is displayed on the other machine.

3.5 Example: an Application that Uses a Shell Function

Figure 3 shows a sequence of screenshots of a calculator program written in Clean. Initially, the calculator has no function buttons. Instead, it has buttons to add and remove function buttons. These will be loaded dynamically after adding dynamics that contain tuples of *String* and *Real Real* → *Real*.

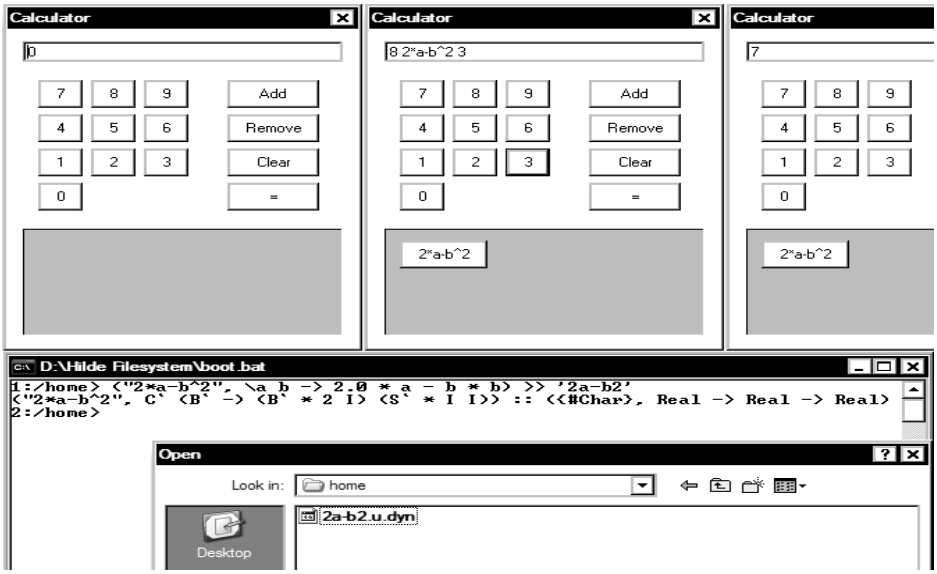

Fig. 3. A combined screenshot of the calculator in action and Esther

The lower half of Fig. 3 shows a command line in the Esther shell that writes such a tuple as a dynamic named "2a-b2.u.dyn" to disk.

Its button name is `2*a-b^2` and the function is `\a b` → `2.0 * a - b * b`. Pressing the `Add` button on the calculator opens a file selection dialog, shown at the

bottom of Fig. 3. After selecting the dynamic named "2a-2b.u.dyn", it becomes available in the calculator as the button `2*a-b^2`, and it is applied to 8 and 3 yielding 7.

The calculator itself is a separately compiled Clean executable that runs without using Esther. Alternatively, one can write the calculator, which has type $[(String, Real\ Real \rightarrow Real)]\ *World \rightarrow *World$, to disk as a dynamic. The calculator can then be started from Esther, either in the current shell or as a separate process.

4 Implementation of Esther using Clean Dynamic Type Checking

In this section, we explain how one can use the type unification of Clean's dynamic run-time system to type check a shell command, and we show how the corresponding Clean expression is translated effectively using combinations of already existing compiled code.

Obviously, we could have implemented type checking ourselves using one of the common algorithms involving building and solving a list of type equations. Instead, we decided to use Clean's dynamic run-time unification, for this has several advantages:

- Clean's dynamics allow us to do type safe and lazy I/O of expressions;
- we do not need to convert between the (hidden) type representation used by dynamics and the type representation used by our type checking algorithm;
- it shows whether Clean's current dynamics interface is powerful enough to implement basic type inference and type checking;
- we get future improvements of Clean's dynamics interface for free (e.g. uniqueness attributes or overloading).

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 support a basic, but complete, functional language in our shell we need to support function definitions, lambda, let(rec), and case expressions.

We will introduce the syntax tree piecewise and show for each kind of expression how to construct a dynamic that contains the corresponding Clean expression and the type for that expression. Names occurring free in the command line are read from disk as dynamics before type checking. The expression can contain references to other dynamics, and therefore to the compiled code of functions, which will be automatically linked by Clean's run-time system.

4.1 Application

Suppose we have a syntax tree for constant values and function applications that looks like:

```
:: Expr = (@) infixl 9[11] Expr Expr   //[12] Application
```

[11] This defines an infix constructor with priority 9 that is left associative.
[12] This a Clean comment to end-of-line, like Haskell's `--`.

```
        | Value Dynamic              // Constant or dynamic value from disk
```

We introduce a function `compose`, which constructs the dynamic containing a value with the correct type that, when evaluated, will yield the result of the given expression.

```
compose :: Expr → Dynamic
compose (Value d) = d
compose (f @ x)   = case (compose f, compose x) of
    (f :: a → b, x :: a) → dynamic f x :: b
    (df, dx)             → raise[13] ("Cannot apply " +++ typeOf df
                                      +++ " to " +++ typeOf dx)
typeOf :: Dynamic → String
typeOf dyn = toString (typecodeOfDynamic dyn) // pretty print type
```

Composing a constant value, contained in a dynamic, is trivial. Composing an application of one expression to another is a lot like the `dynamicApply` function of Sect. 2. Most importantly, we added error reporting using the `typeOf` function for pretty printing the type of a value inside a dynamic.

4.2 Lambda Expressions

Next, we extend the syntax tree with lambda expressions and variables.

```
:: Expr = ···                          // Previous def.
        | (⟶) infixr 0 Expr Expr       // Lambda abstraction: λ .. → ..
        | Var String                   // Variable
        | S | K | I                    // Combinators
```

At first sight, it looks as if we could simply replace a `Lambda` constructor in the syntax tree with a dynamic containing a lambda expression in Clean:

```
compose (Var x ⟶ e) = dynamic (λy → composeLambda x y e :: ?)
```

The problem with this approach is that we have to specify the type of the lambda expression before the evaluation of `composeLambda`. Furthermore, `composeLambda` will not be evaluated until the lambda expression is applied to an argument. This problem is unavoidable because we cannot get 'around' the lambda. Fortunately, bracket abstraction [17] solves both problems.

Applications and constant values are `composed` to dynamics in the usual way. We translate each lambda expression (⟶) to a sequence of combinators (`S`, `K`, and `I`) and applications, with the help of the function `ski`.

```
compose ···          // Previous def.
compose (x ⟶ e) = compose (ski x e)
compose I        = dynamic λx → x
compose K        = dynamic λx y → x
compose S        = dynamic λf g x → f x (g x)
```

[13] For easier error reporting, we implemented imprecise user-defined exceptions à la Haskell [26]. We used dynamics to make the set of exceptions extensible.

```
ski :: Expr Expr → Expr   // common bracket abstraction
ski x       (y ⟶ e) = ski x (ski y e)
ski (Var x) (Var y)   |[14] x == y = I
ski x       (f @ y)   = S @ ski x f @ ski x y
ski x       e         = K @ e
```

Composing lambda expressions uses `ski` to eliminate the `Lambda` and `Var`iable syntax constructors, leaving only applications, dynamic values, and combinators. Composing a combinator simply wraps its corresponding definition and type as a lambda expression into a dynamic.

Special combinators and combinator optimization rules are often used to improve the speed of the generated combinator code by reducing the number of combinators [18]. 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. Adding common eta-conversion, for example, results in a too general type for `Var "f" ⟶ Var "x" ⟶ f x`: `∀a: a → a`, instead of: `∀a b: (a → b) → a → b`. Such optimizations might prevent us from getting the principal type for an expression. Simple bracket abstraction using `S`, `K`, and `I`, as performed by `ski`, does preserves the principal type [19].

Code combined by Esther in this way is not as fast as code generated by the Clean compiler. Combinators introduced by bracket abstraction are the main reason for this slowdown. Additionally, all applications are lazy and not specialized for basic types. However, these disadvantages only hold for the small (lambda) functions written at the command line, which are mostly used for plumbing. If faster execution is required, one can always copy-paste the command line into a Clean module that writes a dynamic to disk and running the compiler.

In order to reduce the number of combinators in the generated expression, our current implementation uses Diller's algorithm C [20] without eta-conversion in order to preserve the principal type, while reducing the number of generated combinators from exponential to quadratic. Our current implementation seems to be fast enough, so we did not explore further optimizations by other bracket abstraction algorithms.

4.3 Irrefutable Patterns

Here we introduce irrefutable patterns, e.g. (nested) tuples, in lambda expressions. This is a preparation for the upcoming let(rec) expressions.

```
:: Expr = ···                    // Previous def.
        | Tuple Int              // Tuple constructor

compose ···        // Previous def.
compose (Tuple n) = tupleConstr n
```

[14] If this guard fails, we end up in the last function alternative.

```
tupleConstr :: Int → Dynamic
tupleConstr 2 = dynamic λx y → (x, y)
tupleConstr 3 = dynamic λx y z → (x, y, z)
tupleConstr ··· // and so on...[15]

ski :: Expr Expr → Expr
ski (f @ x)   e = ski f (x ⟶ e)
ski (Tuple n) e = Value (matchTuple n) @ e
ski ···         // previous def.

matchTuple :: Int → Dynamic
matchTuple 2 = dynamic λf t → f (fst t) (snd t)
matchTuple 3 = dynamic λf t → f (fst3 t) (snd3 t) (thd3 t)
matchTuple ··· // and so on...
```

We extend the syntax tree with `Tuple` n constructors (where n is the number of elements in the tuple). This makes expressions like `Tuple 3 @ Var "x" @ Var "y" @ Var "z" ⟶ Tuple 2 @ Var "x" @ Var "z"` valid expressions. This example corresponds with the Clean lambda expression `λ(x, y, z) → (x, z)`.

When the `ski` function reaches an application in the left-hand side of the lambda abstraction, it processes both sub-patterns recursively. When the `ski` function reaches a `Tuple` constructor it replaces it with a call to the `matchTuple` function. Note that the right-hand side of the lambda expression has already been transformed into lambda abstractions, which expect each component of the tuple as a separate argument. We then use the `matchTuple` function to extract each component of the tuple separately. It uses lazy tuple selections (using `fst` and `snd`, because Clean tuple patterns are always eager) to prevent non-termination of recursive let(rec)s in the next section.

4.4 Let(rec) Expressions

Now we are ready to add irrefutable let(rec) expressions. Refutable let(rec) expressions must be written as cases, which will be introduced in next section.

```
:: Expr = ···                        // Previous def.
        | Letrec [Def] Expr          // let(rec) .. in ..
        | Y                          // Combinator

:: Def = (⊨) infix 0 Expr Expr     // .. = ..

compose ···              // Previous def.
compose (Letrec ds e) = compose (letRecToLambda ds e)
compose Y             = dynamic y :: ∀a: (a → a) → a
                        where y f = f (y f)

letRecToLambda :: [Def] Expr → Expr
```

[15] ...until 32. Clean does not support functions or data types with arity above 32.

```
letRecToLambda ds e = let (p ⊨ d) = combine ds
                      in  ski p e @ (Y @ ski p d)

combine :: [Def] → Def
combine [p ⊨ e]       = p ⊨ e
combine [p1 ⊨ e1:ds] = let (p2 ⊨ e2) = combine ds
                         in  Tuple 2 @ p1 @ p2 ⊨ Tuple 2 @ e1 @ e2
```

When `compose` encounters a let(rec) expression it uses `letRecToLambda` to convert it into a lambda expression. The `letRecToLambda` function `combine`s all (possibly mutually recursive) definitions by pairing definitions into a single (possibly recursive) irrefutable tuple pattern. This leaves us with just a single definition that `letRecToLambda` converts to a lambda expression in the usual way [21].

4.5 Case Expressions

Composing a case expression is done by transforming each alternative into a lambda expression that takes the expression to match as an argument. If the expression matches the pattern, the right-hand side of the alternative is taken. When it does not match, the lambda expression corresponding to the next alternative is applied to the expression, forming a cascade of **if**-then-else constructs. This results in a single lambda expression that implements the case construct, and we apply it to the expression that we wanted to match against.

```
:: Expr = ···                          // Previous def.
        | Case Expr [Alt]              // case .. of ..

:: Alt = (⟹) infix 0 Expr Expr        // .. → ..

compose ···          // Previous def.
compose (Case e as) = compose (altsToLambda as @ e)
```

We translate the alternatives into lambda expressions below using the following rules. If the pattern consists of an application we do bracket abstraction for each argument, just as we did for lambda expressions, in order to deal with each subpattern recursively. Matching against an irrefutable pattern, such as variables of tuples, always succeeds and we reuse the code of `ski` that does the matching for lambda expressions. Matching basic values is done using `ifEqual` 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.

```
altsToLambda :: [Alt] → Expr
altsToLambda []                      = Value mismatch
altsToLambda [f @ x ⟹ e:as]         = altsToLambda [f ⟹ ski x e:as]
altsToLambda [Var x ⟹ e:_]          = Var x ⟶ e
altsToLambda [Tuple n ⟹ e:_]        = Tuple n ⟶ e
altsToLambda [Value dyn ⟹ th:as] = let el = altsToLambda as
    in case dyn of
```

```
        (i :: Int)  → Value (ifEqual i) @ th @ el
        (c :: Char) → Value (ifEqual c) @ th @ el
        ··· // for all basic types

ifEqual :: a → Dynamic | TC a & Eq a
ifEqual x = dynamic λth el y → if (x == y) th (el y)
                    :: ∀b: b (a^ → b) a^ → b

mismatch = dynamic raise "Pattern mismatch" :: ∀a: a
```

Matching against a constructor contained in a dynamic takes more work. For example, if we put Clean's list constructor `[:]` in a dynamic we find that it has type `∀a:a → [a] → [a]`, which is a function type. In Clean, one cannot match closures or functions against constructors. Therefore, using the function `makeNode` below, we construct a node that contains the right constructor by adding dummy arguments until it has no function type anymore. The function `ifMatch` uses some low-level code to match two nodes to see if the constructor of the pattern matches the outermost constructor of the expression. If it matches, we need to extract the arguments from the node. This is done by the `applyTo` function, which decides how many arguments need to be extracted (and what their types are) by inspection of the type of the curried constructor. Again, we use some low-level auxiliary code to extract each argument while preserving laziness.

```
altsToLambda [Value dyn ⟹ th:as] = let el = altsToLambda as
    in case dyn of
        ··· // previous definition for basic types
        constr → Value (ifMatch (makeNode constr))
                                    @ (Value (applyTo dyn) @ th) @ el

ifMatch :: Dynamic → Dynamic
ifMatch (x :: a) = dynamic λth el y → if (matchNode x y) (th y) (el y)
                           :: ∀b: (a → b) (a → b) a → b

makeNode :: Dynamic → Dynamic
makeNode (f :: a → b) = makeNode (dynamic f undef :: b)
makeNode (x :: a)     = dynamic x :: a

applyTo :: Dynamic → Dynamic
applyTo ···              // and so on, most specific type first...
applyTo (_ :: a b → c) = dynamic λf x → f (arg1of2 x) (arg2of2 x)
                                  :: ∀d: (a b → d) c → d
applyTo (_ :: a → b)   = dynamic λf x → f (arg1of1 x)
                                    :: ∀c: (a → c) b → c
applyTo (_ :: a)       = dynamic λf x → f :: ∀b: b a → b

matchNode :: a a → Bool // low-level code; compares two nodes.

argiofn :: a → b // low-level code; selects i^th argument of an n-ary node
```

Pattern matching against user defined constructors requires that the constructors are available from (i.e. stored in) the file system. Esther currently does not support type definitions at the command line, and the Clean compiler must be used to introduce new types and constructors into the file system. The example below shows how one can write the constructors `C`, `D`, and `E` of the type T to the file system. Once the constructors are available in the file system, one can write command lines like `\x -> case x of C y -> y; D z -> z; E -> 0` (for which type `(T Int)` → `Int` is inferred).

```
:: T a = C a | D Int | E

Start world =
    let (_, w1) = writeDynamic "C" (dynamic C :: ∀a: a → T a) world
        (_, w2) = writeDynamic "D" (dynamic D :: ∀a: Int → T a) w1
        (_, w3) = writeDynamic "E" (dynamic E :: ∀a: T a) w2
    in  w3
```

4.6 Overloading

Support for overloaded expressions within dynamics in Clean is not yet implemented (e.g. one cannot write **dynamic** (==) `:: ∀a: a a → Bool | Eq a`). Even when a future dynamics implementation supports overloading, it cannot be used in a way that suits Esther. We want to solve overloading using instances/dictionaries from the file system, which may change over time, and which is something we cannot expect from Clean's dynamic run-time system out of the box.

Below is the Clean version of the overloaded functions == and `one`. We will use these two functions as a running example.

```
class Eq  a where (==) infix 4 :: a a → Bool
class one a where one :: a

instance Eq  Int where (==) x y = // low-level code to compare integers
instance one Int where one      = 1
```

To mimic Clean's overloading, we introduce the type O to differentiate between 'overloaded' dynamics and 'normal' dynamics. The type O, shown below, has four type variables which represent: the variable the expression is overloaded in (`v`), the dictionary type (`d`), the 'original' type of the expression (`t`), and the type of the name of the overloaded function (`n`). Values of the type O consists of a constructor `O` followed by the overloaded expression (of type `d → t`), and the name of the overloaded function (of type `n`). We motivate the design of this type later on in this section.

```
:: O v d t n = O (d → t) n      // Overloaded expression

== = dynamic O id "Eq"  :: ∀a: O a (a a → Bool) (a a → Bool) String
one = dynamic O id "one" :: ∀a: O a a a String

instance_Eq_Int  = dynamic λx y → x == y :: Int Int → Bool
instance_one_Int = dynamic 1             :: Int
```

The dynamic ==, in the example above, is Esther's representation of Clean's overloaded function ==. The overloaded expression itself is the identity function because the result of the expression *is* the dictionary of ==. The name of the class is `Eq`. The dynamic == is overloaded in a single variable a, the type of the dictionary is `a → a → Bool` as expected, the 'original' type is the same, and the type of the name is *String*. Likewise, the dynamic `one` is Esther's representation of Clean's overloaded function `one`.

By separating the different parts of the overloaded type, we obtain direct access to the variable in which the expression is overloaded. This makes it easy to detect if the overloading has been resolved (i.e. the variable no longer unifies with `∀a:a.a`). By separating the dictionary type and the 'original' type of the expression, it becomes easier to check if the application of one overloaded dynamic to another is allowed (i.e. can a value of type `O _ _ (a → b) _` be applied to a value of type `O _ _ a _`).

To apply one overloaded dynamic to another, we combine the overloading information using the P (pair) type as shown below in the function `applyO`.

```
:: P a b = P a b                    // Just a pair

applyO :: Dynamic Dynamic → Dynamic
applyO ((O f nf) :: O vf df (a → b) sf) ((O x nx) :: O vx dx a sx)
    = dynamic O (λd_f d_x → f d_f (x d_x)) (P nf nx)
                                        :: O (P vf vx) (P df dx) b (P sf sx)
```

We use the (private) data type P instead of tuples because this allows us to differentiate between a pair of two variables and a single variable that has been unified with a tuple. Applying `applyO` to == and `one` yields an expression semantically equal to `isOne` below. `isOne` is overloaded in a pair of two variables, which are the same. The overloaded expression needs a pair of dictionaries to build the expression (==) `one`. The 'original' type is `a → Bool`, and it is overloaded in `Eq` and `one`. Esther will pretty print this as: `isOne :: a → Bool | Eq a & one a`.

```
isOne = dynamic O (λ(P d_Eq d_one) → id d_Eq (id d_one)) (P "Eq" "one")
      :: ∀a: O (P a a) (P (a a → Bool) a) (a → Bool) (P String String)
```

Applying `isOne` to the integer 42 will bind the variable a to *Int*. Esther is now able to choose the right instance for both `Eq` and `one`. It searches the file system for the files named "instance Eq Int" and "instance one Int", and applies the code of `isOne` to the dictionaries after applying the overloaded expression to 42. The result will look like `isOne10` in the example below, where all overloading has been removed.

```
isOne42 = dynamic (λ(P d_Eq d_one) → id d_Eq (id d_one) 42)
                                        (P d_Eq_Int d_one_Int) :: Bool
```

Although overloading is resolved in the example above, the plumbing/dict-ionary passing code is still present. This will increase evaluation time, and it is not clear yet how this can be prevented.

5 Related Work

We have not yet seen an interpreter or shell that equals Esther's ability to use pre-compiled code, and to store expressions as compiled code, which can be used in other already compiled programs, in a type safe way.

Es [13] 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 [22] by Jim Mattson is an interactive program that also launches executables, and provides pipelining and redirections. Tcl [23] is a popular tool to combine programs, and to provide communications between them. None of these programs provides 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) [12] 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 [24], 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.

The BeanShell [25] is an embeddable Java source interpreter with object scripting language features, written in Java. It is able of type inference for variables and to combine shell scripts with existing Java programs. While Esther generates compiled code via dynamics, the BeanShell interpreter is invoked each time a script is called from a normal Java program.

Run-time code generation in order to specialize code at run-time to certain parameters is not related to Esther, which only combines existing code.

There are concurrent versions of both Haskell and Clean. Concurrent Haskell [6] offers lightweight threads in a single UNIX process and provides M-Vars as the means of communication between threads. Concurrent Clean [5] is only available on multiprocessor Transputers and on a network of single-processor Apple Macintosh computers. Concurrent Clean provides support for native threads on Transputer systems. On a network of Apple computers, it ran the same Clean program on each processor, providing a virtual multiprocessor system. Concurrent Clean provided lazy graph copying as the primary communication mechanism. Both concurrent systems cannot easily provide type safety between different programs or between multiple incarnations of a single program.

Both Cooper [8] and Lin [9] have extended Standard ML with threads (implemented as continuations using call/CC) to form a small functional operating system. Both systems implement the basics needed for a stand-alone operating system. However, none of them support the type-safe communication of any value between different computers.

Erlang [10] is a functional language specifically designed for the development of concurrent processes. It is completely dynamically typed and primarily uses interpreted byte-code, while Famke 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.

Back et al. [11] 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 lacks the power of polymorphic and higher-order functions and closures (to allow laziness) that our functional approach offers.

The Scheme Shell [12] integrates a shell into the programming language in order to enable the user to use the full expressiveness of Scheme. Es [13] is a shell that supports higher-order functions and allows the user to construct new functions at the command line. Neither shell provides a way to read and write typed objects from and to disk, and they cannot provide type safety because they operate on untyped executables.

6 Conclusions

We have shown how to build a shell that provides a simple, but powerful strongly typed functional programming language. We were able to do this using only Clean's support for run-time type unification and dynamic linking, albeit syntax transformations and a few low-level functions were necessary. The shell named Esther supports type checking and type inference before evaluation. 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 compiled code, Esther allows the use of any pre-compiled program as a function in the shell. Because Esther stores functions/expressions constructed at the command line as a 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. The evaluation of expressions using recombined compiled code is not as fast as using the Clean compiler. Speed can be improved by introducing less combinators during bracket abstraction, but it seams unfeasible to make Esther perform the same optimizations as the Clean compiler. In practice, we find Esther responsive enough, and more optimizations do not appear worth the effort at this stage. One can always construct a Clean module using the same syntax and use the compiler to generate a dynamic that contains more efficient code.

References

1. S. Peyton Jones and J. Hughes et al. *Report on the programming language Haskell 98*. University of Yale, 1999. http://www.haskell.org/definition/
2. M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.

3. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.
4. M. Pil. Dynamic Types and Type Dependent Functions. In T. Davie K. Hammond and C. Clack, editors, *Proceedings of the 10th International Workshop on the Implementation of Functional Languages*, volume 1595 of *Lecture Notes in Computer Science*, pages 171–188. Springer-Verlag, 1998.
5. E.G.J.M.H. Nocker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE '91: Parallel Architectures and Languages Europe, Volume II*, volume 506 of *Lecture Notes in Computer Science*, pages 202–219. Springer, 1991.
6. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* $23^{mathrmrd}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.
7. V. Stolz and F. Huch. Implementation of Port-based Distributed Haskell, 2001. http://www-i2.informatik.rwth-aachen.de/Research/distributedHaskell/ifl2001.ps.gz.
8. E.C. Cooper and J.G. Morrisett. Adding Threads to Standard ML. Technical Report CMU-CS-90-186, Pittsburgh, PA, 1990.
9. A.C. Lin. *Implementing Concurrency For An ML-based Operating System*. PhD thesis, Massachusetts Institute of Technology, February 1998.
10. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
11. G. Back, P. Wullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java Operating Systems: Design and Implementation. Technical Report UUCS-98-015, 6, 1998.
12. O. Shivers. A Scheme Shell. Technical Report MIT/LCS/TR-635, 1994.
13. P. Haahr and B. Rakitzis. Es: A shell with higher-order functions. In *USENIX Winter*, pages 51–60, 1993.
14. R. Plasmeijer and M. van Eekelen. *Concurrent Clean Language Report version 2.1*. University of Nijmegen, November 2002. http://cs.kun.nl/ clean.
15. A. van Weelden and R. Plasmeijer. Towards a Strongly Typed Functional Operating System. In R. Peña and T. Arts, editors, *14th International Workshop on the Implementation of Functional Languages, IFL'02*, pages 215–231. Springer, September 2002. LNCS 2670.
16. M. Vervoort and R. Plasmeijer. Lazy Dynamic Input/Output in the Lazy Functional Language Clean. In R. Peña and T. Arts, editors, *14th International Workshop on the Implementation of Functional Languages, IFL'02*, pages 101–117. Springer, September 2002. LNCS 2670.
17. M. Schönfinkel. Über die Bausteine der mathematischen Logik. In *Mathematische Annalen*, volume 92, pages 305–316. 1924.
18. H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.
19. J. Roger Hindley and J. P. Seldin. *Introduction to Combinators and lambda-Calculus*. Cambridge University Press, 1986. ISBN 0521268966.
20. A. Diller. *Compiling Functional Languages*. John Wiley and Feys Sons Ltd, 1988.
21. S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
22. J. Mattson. The Haskell Shell. http://www.informatik.uni−bonn.de/ ralf/software/examples/Hsh.html.

23. J. K. Ousterhout. Tcl: An Embeddable Command Language. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 133–146, Berkeley, CA, 1990. USENIX Association.
24. M. P. Jones, A. Reid, the Yale Haskell Group, the OGI School of Science, and Engineering at OHSU. *The Hugs 98 User Manual*, 1994–2002. http://cvs.haskell.org/Hugs/.
25. P. Niemeyer. Beanshell 2.0. http://www.beanshell.org.
26. S. L. Peyton Jones, A. Reid, F. Henderson, C. A. R. Hoare, and S. Marlow. A Semantics for Imprecise Exceptions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–36, 1999.