A Functional Shell that Dynamically Combines Compiled Code

Arjen van Weelden* and Rinus Plasmeijer

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

Abstract. We present a new shell that provides the full basic functionality of a strongly typed lazy functional language, including overloading. The shell can be used for manipulating files, applications, data and processes at the command line. The shell does type checking and only executes well-typed expressions. Files are typed, and applications are simply files with a function type. The shell executes a command line by combining existing code of functions on disk. We use the hybrid static/dynamic type system of Clean to do type checking/inference. Its dynamic linker is used to store and retrieve any expression (both data and code) with its type on disk. Our 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

Programming languages, especially pure and lazy functional languages like Clean [1] and Haskell [2], 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 [3]. This has shown that (moderate) use of dynamic typing [4], in combination with Clean's dynamic run-time

* Part of this work was supported by InterNLnet.

system and dynamic linker [5, 6], 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.

In this paper we present a shell for a functional language-based operating system that combines the best of both worlds: the interactivity of an interpreter and the efficiency and type safety of a compiler. This shell is used as the user interface for Famke, the above mentioned kernel of a prototype functional operating system in development. The shell can make use of compiled functions/programs, without losing type information. Functions defined in the shell can also be used by compiled applications.

The shell is 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 of Clean in Sect. 2. Section 3 gives a global description of how Esther uses dynamics to type check an expression. It also give examples of the use of dynamics. 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

In addition to its static type system, Clean has recently been extended with a (polymorphic) dynamic type system [4–6]. 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.

```
::[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 [7]. A caret (`^`) behind a variable in a pattern associates it with the type variable with

[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] 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.

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
```

The dynamic run-time system of Clean supports writing dynamics to disk and reading them back again, possibly in another program or during another execution of the same program. The dynamic will be read in lazily after a successful run-time unification (triggered by a pattern match on the dynamic). The amount of data and code that the dynamic linker will link, is therefore determined by the amount of evaluation of the value inside the dynamic. Dynamics written by a program can be safely read by any other program, providing a simple form of persistence and some rudimentary means of communication.

```
writeDynamic :: String Dynamic *[8]World -> (Bool, *World)
readDynamic :: String *World -> (Bool, Dynamic, *World)
```

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` 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)
```

[7] Clean uses `|` to denote overloading. In Haskell this would be written as `(TC t) => Dynamic -> Maybe t`.

[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.

3 An Overview of Esther

The last example of the previous section 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.

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: 1) Clean's dynamics allow us to do type safe and lazy I/O of expressions; 2) we do not need to convert between the (hidden) type representation used by dynamics and the type representation used by our type checking algorithm; 3) it shows whether Clean's current dynamics interface is powerful enough to implement basic type inference and type checking; 4) we get future improvements of Clean's dynamics interface for free (e.g. uniqueness attributes or overloading).

Unlike common command interpreters or shells, our shell Esther does not work on untyped files that consist of executables and streams of characters. Instead, all functions/programs are stored as dynamics, forming a rudimentary typed file system.

Moreover, instead of evaluating the expression by interpretation of the source code, Esther generates a new dynamic that contains a closure that refers to the compiled code of other programs. The shell, therefore, is a hybrid interpreter that generates compiled code. The resulting dynamic can be used by any other compiled Clean program without using an interpreter or the shell. Dynamics can contain closures, which refer to code and data belonging to other compiled Clean programs. When needed for evaluation, the code is automatically linked to the running program by Clean's dynamic linker. This approach results in less overhead during evaluation of the expression than using a conventional source code interpreter.

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.

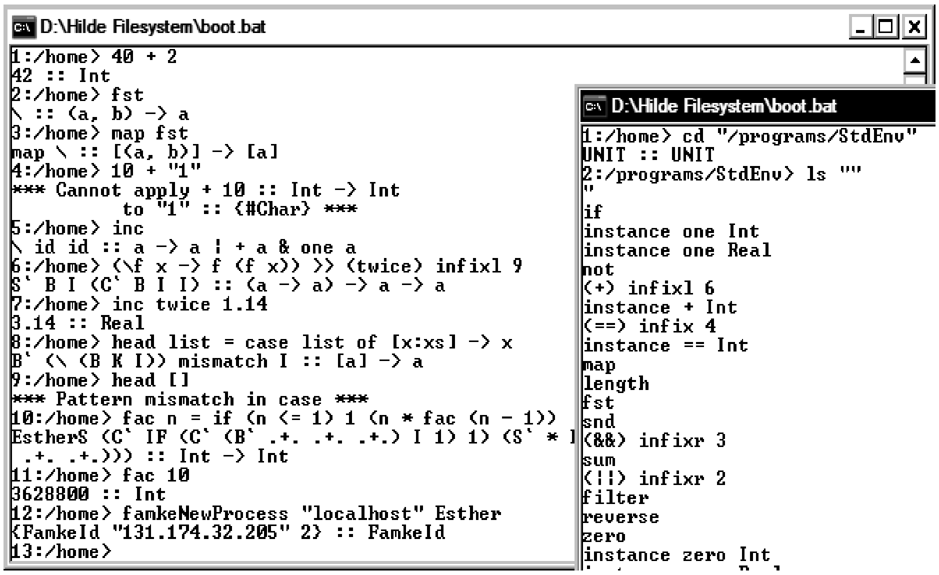

Fig. 1. A combined screenshot of two incarnations of Esther

3.1 Example: a Session with Esther

To illustrate the expressive power of Esther, we show an Esther session in Fig. 1 (the left window with the white title bar) and explain what happens:

1. 'Simple' arithmetic. The shell looks in the current search-path to find the infix function `+`. The `+` is overloaded, and the shell searches again for an instance for `+` for type *Int*. Finally, it responds with the value and inferred type of the result.
2. Typing the name of a dynamic at the prompt shows its contents, which can contain unnamed lambda functions (\), and its type.
3. The dynamic `map` is applied to the dynamic `fst` yielding the expected type.
4. The infix operator `+` cannot be applied to an integer and a string.
5. The overloaded function `inc` is revealed to be overloaded in `+` and `one`. The `\ id id` is caused by the way Esther handles overloading (see Sect. 4.6.).
6. The lambda expression `\f x -> f (f x)` is written to disk, using the `>>` operator, and named `twice`. It is defined as a left associative infix operator with priority 9. Esther shows the internal code and type of the lambda expression, exposing the fact that it uses combinators (see Sect. 4.2).
7. The dynamic `inc` is applied to `1.14` via the previously defined operator `twice`.
8. Defines a function named `head` that selects the first argument of a list using a case expression.
9. Applies `head` to an empty list yielding a pattern mismatch exception.

10. Defines a function named `fac` that yields the factorial of its argument.
11. `fac 10` is evaluated to `3628800`.
12. `famkeNewProcess` is used to start Esther (which is also stored as a dynamic) as new process, on the same computer (right window with black title bar):
    1 Evaluates `cd "/programs/StdEnv"` to 'change directory' to the directory that provides Clean's standard library to Esther, by storing the functions as dynamics in the file system. Because `cd` has type *String* $*World \rightarrow *World$ and therefore no result, Esther shows `UNIT` (i.e. *void*).
    2 Evaluates the application of `ls` to the empty string, showing all files in the current directory: the functions in the standard library.

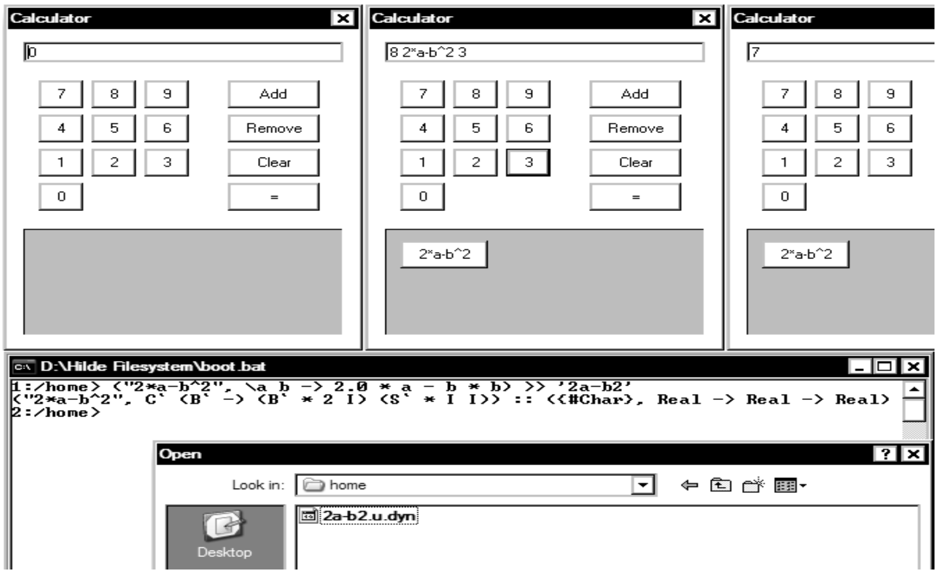

Fig. 2. A combined screenshot of the calculator in action and Esther

3.2 Example: a Calculator that Uses a Shell Function

Figure 2 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*.

The lower half of Fig. 2 shows a command line in the Esther shell that writes such a tuple as a dynamic named "2a-b2.u.dyn" to disk. The extension ".dyn" is added by Clean dynamic linker, the ".u" before the extension is used to store the file fixity attributes ("u" means prefix). Esther pretty prints these attributes, but the Microsoft Windows file selector shows the file name in a raw form.

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. 2. 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 Type Checking with Dynamics

In this section, we show how one can use the type unification of Clean's dynamic run-time system to type check a common syntax tree, and how to construct the corresponding Clean expression. The parsing is trivial and we will assume 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[9] Expr Expr  //[10] Application
        | 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[11]("Cannot apply " +++ typeOf df
                                       +++ " to " +++ typeOf dx)
```

[9] This defines an infix constructor with priority 9 that is left associative.

[10] This a Clean comment to end-of-line, like Haskell's `--`.

[11] For easier error reporting, we implemented imprecise user-defined exceptions à la Haskell [8]. We used dynamics to make the set of exceptions extensible.

```
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: \ .. -> ..
        | 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 (\y -> 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 [9] solves both problems.

Applications and constant values are `compose`d 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 \x -> x
compose K         = dynamic \x y -> x
compose S         = dynamic \f g x -> f x (g x)

ski :: Expr Expr -> Expr  // common bracket abstraction
ski x       (y --> e) = ski x (ski y e)
ski (Var x) (Var y)   |12 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

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

combinators [10]. 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 η-conversion, for example, results in a too general type for `Var "f" --> Var "x" --> f x`: $\forall a.a \rightarrow a$, instead of $\forall ab.(a \rightarrow b) \rightarrow a \rightarrow b$. 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 [11].

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 [12] without η-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

tupleConstr :: Int -> Dynamic
tupleConstr 2 = dynamic \x y -> (x, y)
tupleConstr 3 = dynamic \x y z -> (x, y, z)
tupleConstr ... // and so on...13

ski :: Expr Expr -> Expr
ski (f @ x)   e = ski f (x --> e)
ski (Tuple n) e = Value (matchTuple n) @ e
ski ...        // previous def.
```

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

```
matchTuple :: Int -> Dynamic
matchTuple 2 = dynamic \f t -> f (fst t) (snd t)
matchTuple 3 = dynamic \f 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 `\(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 where y f = f (y f)

letRecToLambda :: [Def] Expr -> Expr
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 [13].

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 \th el y -> if (x == y) th (el y)
                    :: A.b: b (a^ -> b) a^ -> b

mismatch = dynamic raise "Pattern mismatch" :: A.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 $\forall a.a \rightarrow [a] \rightarrow [a]$, 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 \th el y -> if (matchNode x y) (th y) (el y)
                          :: A.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 \f x -> f (arg1of2 x) (arg2of2 x)
                                    :: A.d: (a b -> d) c -> d
applyTo (_ :: a -> b)   = dynamic \f x -> f (arg1of1 x)
                                       :: A.c: (a -> c) b -> c
applyTo (_ :: a)        = dynamic \f x -> f :: A.b: b a -> b

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

argiofn :: a -> b // low-level code; selects ith argument of 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 $(T\ Int) \rightarrow Int$ is inferred).

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

Start world =
    let (_, w1) = writeDynamic "C" (dynamic C :: A.a: a -> T a) world
        (_, w2) = writeDynamic "D" (dynamic D :: A.a: Int -> T a) w1
        (_, w3) = writeDynamic "E" (dynamic E :: A.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 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 $d \rightarrow t$), 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.a: O a (a a -> Bool) (a a -> Bool) String
one = dynamic O id "one" :: A.a: O a a a String

instance_Eq_Int  = dynamic \x 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 $a \rightarrow a \rightarrow Bool$ 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 $\forall a.a$). 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 $O\ \_\ \_\ (a \rightarrow b)\ \_$ 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 (\d_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 $a \rightarrow Bool$, and it is overloaded in `Eq` and `one`. Esther will pretty print this as: `isOne :: a -> Bool | Eq a & one a`.

```
isOne = dynamic O (\(P d_Eq d_one) -> id d_Eq (id d_one)) (P "Eq" "one")
      :: A.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 (\(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/dictionary 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 [14] 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 [15] by Jim Mattson is an interactive program that also launches executables, and provides pipelining and redirections. Tcl [16] 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) [17] 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 [18], 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 [19] 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.

6 Conclusions and Future Work

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 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. The support for infix operators and overloading required the storage of additional information in the file system. We have chosen to use file attributes to store the infix information, and instances for an overloaded function `f` are stored as files named "instance `f` *Type*".

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.

Further research will be done on a more elaborate typed file system, and support for types and type definitions at the command line. Esther will be incorporated into our ongoing research on the development of a strongly typed functional operating system.

References

1. Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean Language Report version 2.1.* University of Nijmegen, November 2002. http://cs.kun.nl/∼clean.
2. Simon Peyton Jones and John Hughes et al. *Report on the programming language Haskell 98.* University of Yale, 1999. http://www.haskell.org/definition/.
3. Arjen van Weelden and Rinus 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.
4. Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.
5. Marco Pil. Dynamic Types and Type Dependent Functions. In K. Hammond, T. Davie, and C. Clack, editors, *10th International Workshop on the Implementation of Functional Languages, IFL '98*, volume 1595 of *LNCS*, pages 169–185, London, 1999. Springer.
6. Martijn Vervoort and Rinus 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.
7. M. Abadi, L. Cardelli, B. Pierce, G. Plotkin, and D. Rèmy. Dynamic Typing in Polymorphic Languages. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, June 1992.
8. Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. A Semantics for Imprecise Exceptions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–36, 1999.
9. M. Schönfinkel. Über die Bausteine der mathematischen Logik. In *Mathematische Annalen*, volume 92, pages 305–316. 1924.
10. Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.
11. J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986. ISBN 0521268966.
12. Antoni Diller. *Compiling Functional Languages.* John Wiley and Feys Sons Ltd, 1988.
13. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
14. 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.
15. Jim Mattson. The Haskell Shell. http://www.informatik.uni−bonn.de/∼ralf/software/examples/Hsh.html.
16. 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.
17. O. Shivers. A Scheme Shell. Technical Report MIT/LCS/TR-635, 1994.
18. Mark P Jones, Alastair 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/.
19. Pat Niemeyer. Beanshell 2.0. http://www.beanshell.org.