Combining Generics and Dynamics

Peter Achten[1] and Ralf Hinze[2]
(`peter88@cs.kun.nl`, `ralf@informatik.uni-bonn.de`)

[1]Nijmegen Institute for Information and Computing, University of Nijmegen,
1 Toernooiveld, 6525 ED, Nijmegen, The Netherlands
[2]Institut für Informatik III, Universität Bonn,
Römerstraße 164, 53117 Bonn, Germany

Abstract. In this paper we discuss the surprisingly elegant interaction between two lines of rather independent research, namely *generic* (or *polytypic*) programming and *dynamic types.* We show how the two disciplines meet and emphasize their characteristics in the area of interactive programming as implemented in the functional language Clean. In this language, an extensive library for creating Graphical User Interfaces is provided. We demonstrate how dynamic types can be used to store arbitrary GUI objects as *resources* and how generic functions can be used to manipulate these values. This results in a flexible system of *strongly typed* and *higher-order* resources.

1 Introduction

In this paper we discuss the interaction between two recent additions to the pure, lazy, functional programming language Clean [17]: *dynamic types* [15, 16] and *generic functions* [8, 11, 5].

The main purpose of dynamic types is to *serialise* values, in particular functional values. Serialised values can be freely exchanged between applications, processes, incarnations of the same application, and between functions inside the same program. They can be sent over a network and stored to and retrieved from secondary memory. Reading a dynamic value is as simple as telling its type: there is no need to write parsers and do error-checking. (This work has, of course, to be done by the compiler writer and the runtime-system, but it is done only once.)

Generic programming is all about succinctness. Generic programming allows us to write one single function for tasks that are similar for different types. This is accomplished by defining these functions on both the *type structure* of their argument *and* their *value.* In this way one can write parsers, pretty printers, equality operations, ordering operations etc in one go.

The Clean language has an elaborate library to allow programmers to create interactive applications using Graphical User Interfaces, the *Object I/O library* [3, 4]. A key design decision in this library is to let programmers construct compositions of algebraic data values that describe individual GUI components. Each sort of GUI component has a dedicated algebraic type constructor that describes its *look and feel*. The functionality of GUI components is defined by means of higher-order function arguments. In a sense, the Object I/O library provides an extensible set of types that form a *domain specific language* for constructing GUIs. Perhaps surprisingly, the approach taken in the library is an instance of a more general approach that is typical of generic programming.

One particular advantage of the design is that it provides a language for GUI *resources*. Usually, GUI resources in programming environments are defined by simple special purpose languages that can be parsed by the environment and compiled into code that creates the intended GUI objects. The connection with dynamic types should be obvious: they allow us to store and retrieve GUI definitions, which are (compositions of) algebraic values. The connection with generic programming is as follows: to retrieve a dynamic value one needs to know the type of the extracted value. Due to the design of the Object I/O library, the type of a GUI value closely reflects its structure (see Section 2). A program that reads a GUI resource must therefore either know the exact type or leave it almost undetermined. Neither option is satisfactory: the first alternative makes it impossible to change the resource afterwards (because that almost certainly changes its type), while the second alternative doesn't give programmers the means to control the type of resource they want to use. One goal of this paper is to see whether generic programming can add the required flexibility.

The contributions of this paper to the domain of dynamics and generics are:

- A general framework for generic programming is presented in [9]. The principle idea of this paper is to assign a polytypic value a type that depends on the kind of its type index. Unfortunately, this technique is not immediately applicable here since Object I/O types are parameterised by the types of states (see Section 2). The solution to this problem is to 'lift' kinds and types to a higher realm. Using this lifting we demonstrate that polytypic functions can be written for Object I/O values.
- The standard way of overloading in functional languages, such as Haskell and Clean, fixes a finite set of function instances, thus limiting the application domains of these functions. When all types are statically known, this does not pose any problems because the compiler can detect completeness. However, when working with dynamic values, the domain of applicable types is unknown. We argue that generic functions are perfectly suited in this case: they provide a universal recipe to construct function instances if a type is known. Because dynamic values carry the type of their value around, it is possible to generate the proper instance function *at run-time*.

We have tested simplified versions of the examples presented in this paper in order to work around limitations of the experimental extensions in the current Clean 2.0 compiler. The examples are presented, however, using Haskell 98 syntax, augmented by the features presented in [8], and Clean uniqueness attributes. In addition, we assume that we are allowed to define custom instances to overrule generic definitions, as in [11].

The rest of the paper is structured as follows. We first give a brief introduction to the essential features of the Object I/O library (Section 2), to dynamic types (Section 3), and generic programming (Section 4). We will not go into too much detail since these concepts have been discussed elsewhere. In order to abstract away from the many details of the Object I/O library, we will study the applicability of dynamics and generics by means of three simplified systems in Section 5. We review related work in Section 6 and conclude in Section 7.

2 The Object I/O library

As stated in the introduction, one of the major design decisions of the Object I/O library is to model each sort of GUI object by a separate type constructor. This has the advantage that one can extend the library without recompilation and recoding. GUIs are specified by means of compositions of primitive type constructors. For instance, to define windows, button- and textcontrols one uses:

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Window}\ c\ \mathit{ls}\ \mathit{pst} & = \mathit{Window}\ \mathit{String}\ (c\ \mathit{ls}\ \mathit{pst}) \\
 & \qquad\qquad [\,\mathit{WindowAttribute}\ (\mathit{ls}, \mathit{pst})] \\
\textbf{data}\ \mathit{Button}\ \mathit{ls}\ \mathit{pst} & = \mathit{Button}\ \mathit{String}\ [\,\mathit{ControlAttribute}\ (\mathit{ls}, \mathit{pst})] \\
\textbf{data}\ \mathit{Text}\ \mathit{ls}\ \mathit{pst} & = \mathit{Text}\ \mathit{String}\ [\,\mathit{ControlAttribute}\ (\mathit{ls}, \mathit{pst})]
\end{array}
$$

In order to glue separate GUI objects, we require suitable combinators. The most frequently used combinator is pairing (for historic reasons written as ':+:'):

$$
\begin{array}{l}
\textbf{infixr}\ {:}{+}{:}\ 9 \\
\textbf{data}\ {:}{+}{:}\ t_1\ t_2\ \mathit{ls}\ \mathit{pst} = (t_1\ \mathit{ls}\ \mathit{pst})\ {:}{+}{:}\ (t_2\ \mathit{ls}\ \mathit{pst})
\end{array}
$$

This is basically a lifted pair type that is parameterised by the *state* of its GUI components (we will say more about state as we go along).

Windows and controls are instances of the *Windows* and *Controls* type constructor classes. Type constructor classes fix the 'glue rules'. For instance, the rule that windows shall only contain controls is expressed as:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Windows}\ \mathit{wdef}\ \textbf{where} \\
\quad \mathit{openWindow} :: \mathit{ls} \to \mathit{wdef}\ \mathit{ls}\ (\mathit{PSt}\ \mathit{ps}) \to \mathit{PSt}\ \mathit{ps} \to \mathit{PSt}\ \mathit{ps} \\
\textbf{class}\ \mathit{Controls}\ \mathit{cdef}\ \textbf{where} \\
\quad \mathit{openControl}\ :: \mathit{cdef}\ \mathit{ls}\ (\mathit{PSt}\ \mathit{ps}) \to \mathit{PSt}\ \mathit{ps} \to \mathit{PSt}\ \mathit{ps} \\
\textbf{instance}\ \mathit{Controls}\ \mathit{Button} \\
\textbf{instance}\ \mathit{Controls}\ \mathit{Text} \\
\textbf{instance}\ (\mathit{Controls}\ t_1, \mathit{Controls}\ t_2) \Rightarrow \mathit{Controls}\ ({:}{+}{:}\ t_1\ t_2) \\
\textbf{instance}\ (\mathit{Controls}\ c) \Rightarrow \mathit{Windows}\ (\mathit{Window}\ c)
\end{array}
$$

The actions of the GUI elements are higher-order function arguments that typically appear as an attribute (such as *WindowAttribute* or *ControlAttribute*). For instance, *WindowAttribute* is defined as:

$$
\textbf{data}\ \mathit{WindowAttribute}\ \mathit{st} = \cdots \mid \mathit{WindowClose}\ (\mathit{st} \to \mathit{st}) \mid \cdots
$$

The *WindowClose f* attribute determines the behaviour of the window whenever the user wants to close the window.

Actions are *state transformers*. The initial state of any program is the $*World$. A GUI program, in general, consists of several *interactive processes*. Each process has a *public state ps* that is shared by all of its GUI components. This state can be chosen freely by the programmer. The $*World$ is transformed into a special value called the 'I/O state' and has abstract type $IOSt\ ps$. It contains all the required information needed for doing GUI I/O. The pair of public state and I/O state is collected in a record of type $PSt\ ps$ (the *process state*):

$$\mathbf{data} * PSt\ ps = \{ps :: ps, io :: *IOSt\ ps\}$$

Finally, every GUI component can encapsulate a *local state*. Local state is shared amongst its children, unless a GUI component decides to ignore it.

To summarize, Clean Object I/O programs are tagged collections of (local) state transition functions that are capable of modifying the world. The tags are type constructors that reflect the sort of GUI elements that have to be created by the system.

3 Dynamic types

A dynamic type [15, 16] is basically a pair, consisting of the static value and its type, wrapped into a new value of type *Dynamic*. Examples include:

$$\begin{array}{lll}
\mathbf{dynamic}\ 50 & & :: Int \\
\mathbf{dynamic}\ reverse & & :: [a] \to [a] \\
\mathbf{dynamic}\ reverse\ [1..10] & & :: [Int] \\
\mathbf{dynamic}\ [\mathbf{dynamic}\ 50 & & :: Int \\
\quad\quad ,\mathbf{dynamic}\ reverse & & :: [a] \to [a] \\
\quad\quad] & & :: [Dynamic] \\
\mathbf{dynamic}\ Button\ \texttt{"Hello"}\ [\,] & & \\
\quad\quad :+: & & \\
\quad\quad Text\ \texttt{"World"}\ [\,] & & :: :+: Button\ Text\ ls\ pst
\end{array}$$

As an aside, note that in Clean numeric literals have type *Int*, rather than, $(Num\ a) \Rightarrow a$ as in Haskell. Values of type *Dynamic* can be matched in function alternatives and case expressions. A 'dynamic pattern' matches the value (in the same way as for statically typed values) and additionally its type. It is important to note that type variables in a dynamic pattern do not indicate polymorphism. Rather, they are bound to the offered type. The following may serve as an example:

$$\begin{array}{ll}
f & :: Dynamic \to Int \\
f\ (x :: Int) & = x \\
f\ (g :: Int \to Int) & = g\ 5 \\
f\ ((x :: Int, y) :: (Dynamic, Int)) & = x + y \\
apply & :: Dynamic \to Dynamic \to Dynamic \\
apply\ (f :: a \to b)\ (x :: a) & = \mathbf{dynamic}\ f\ x :: b \\
apply\ _\ _ & = \mathbf{dynamic}\ \texttt{"Error"} :: String
\end{array}$$

Clean 2.0 offers two library functions to store and retrieve dynamic values, identified by its file name:

$$
\begin{aligned}
\mathit{writeDynamic} &:: (\mathit{FileSystem}\ \mathit{env}) \\
&\Rightarrow \mathit{String} \rightarrow \mathit{Dynamic} \rightarrow *\mathit{env} \rightarrow (\mathit{Bool}, *\mathit{env}) \\
\mathit{readDynamic} &:: (\mathit{FileSystem}\ \mathit{env}) \\
&\Rightarrow \mathit{String} \rightarrow *\mathit{env} \rightarrow (\mathit{Bool}, \mathit{Dynamic}, *\mathit{env})
\end{aligned}
$$

Dynamic types are at the foundation of creating GUI resources. However, in contrast to 'standard' GUI resources, these resources are strongly typed and may additionally contain code.

4 Generic programming

This section briefly reviews the concept of polytypic programming. For a more thorough treatment the interested reader is referred to [8, 9].

Consider writing an equality function that checks whether two elements of some given type are equal. As an example, the equality test for strings of type

$$\mathbf{data}\ \mathit{String} = \mathit{Nil} \mid \mathit{Cons}\ \mathit{Char}\ \mathit{String}$$

is given by

$$
\begin{aligned}
&\mathit{equalString} :: \mathit{String} \rightarrow \mathit{String} \rightarrow \mathit{Bool} \\
&\mathit{equalString}\ \mathit{Nil}\ \mathit{Nil} = \mathit{True} \\
&\mathit{equalString}\ (\mathit{Cons}\ c_1\ s_1)\ (\mathit{Cons}\ c_2\ s_2) \\
&\qquad = \mathit{equalChar}\ c_1\ c_2 \wedge \mathit{equalString}\ s_1\ s_2
\end{aligned}
$$

Abstracting over the list element type, we obtain the ubiquitous, parametric list type (which enjoys special syntax, but this is not important here).

$$\mathbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

The list type embraces strings as lists of characters: $\mathit{List}\ \mathit{Char}$.

How can we define an equality test on a list of something? Or, to put it differently, how can we suitably generalize the *equalString* function? Since the element type is not known in general, we must abstract away from the *equalChar* function.

$$
\begin{aligned}
&\mathit{equalList} :: \forall a\,.\,(a \rightarrow a \rightarrow \mathit{Bool}) \rightarrow (\mathit{List}\ a \rightarrow \mathit{List}\ a \rightarrow \mathit{Bool}) \\
&\mathit{equalList}\ \mathit{equala}\ \mathit{Nil}\ \mathit{Nil} = \mathit{True} \\
&\mathit{equalList}\ \mathit{equala}\ (\mathit{Cons}\ a_1\ x_1)\ (\mathit{Cons}\ a_2\ x_2) \\
&\qquad = \mathit{equala}\ a_1\ a_2 \wedge \mathit{equalList}\ \mathit{equala}\ x_1\ x_2
\end{aligned}
$$

Thus, *equalList* takes as an argument an equality function for the element type a and yields an equality function for $\mathit{List}\ a$.[1]

[1] Of course, the type signature cannot ensure that the argument is a genuine equality function.

The types in Clean and in Haskell are assigned so-called kinds, which can be seen as the 'type of types'. Manifest types such as *Bool* or *Int* have kind '$\star$'. The kind $\kappa_1 \to \kappa_2$ represents type constructors that map type constructors of kind κ_1 to those of kind κ_2. Now, the message of the exercise above is that a polytypic function such as taking equality has a type that depends on the kind of the particular instance at hand: for a type T of kind '$\star$', *equal* is assigned type $T \to T \to Bool$; for a type of F of kind $\star \to \star$, equality possesses the type $\forall a\,.\,(a \to a \to Bool) \to (F\ a \to F\ a \to Bool)$. To coin a catchy phrase: a polytypic value possesses a polykinded type. In general, we have (kind and type arguments are written in angle brackets for emphasis):

$$
\begin{aligned}
&Equal\langle\star\rangle\ t &&= t \to t \to Bool\\
&Equal\langle\kappa_1 \to \kappa_2\rangle\ t &&= \forall a\,.\,Equal\langle\kappa_1\rangle\ a \to Equal\langle\kappa_2\rangle\ (t\ a)\\
&equal\langle t :: \kappa\rangle &&:: Equal\langle\kappa\rangle\ t
\end{aligned}
$$

It remains to provide a generic definition of *equal*. Now, one can show that a polytypic function is uniquely defined by giving cases for primitive types. Let us assume that the following type constructors are primitive: the unit type $1 :: \star$, the sum type $+ :: \star \to \star \to \star$, and the product type $\times :: \star \to \star \to \star$ defined:

$$
\begin{aligned}
&\mathbf{data}\ 1 &&= ()\\
&\mathbf{data}\ +\ a\ b &&= Inl\ a \mid Inr\ b\\
&\mathbf{data}\ \times\ a\ b &&= (a, b).
\end{aligned}
$$

Then *equal* is given by:

$$
\begin{aligned}
&equal\langle 1\rangle\ ()\ () &&= True\\
&equal\langle +\rangle\ equala\ equalb\ (Inl\ a_1)\ (Inl\ a_2) &&= equala\ a_1\ a_2\\
&equal\langle +\rangle\ equala\ equalb\ (Inl\ a_1)\ (Inr\ b_2) &&= False\\
&equal\langle +\rangle\ equala\ equalb\ (Inr\ b_1)\ (Inl\ a_2) &&= False\\
&equal\langle +\rangle\ equala\ equalb\ (Inr\ b_1)\ (Inr\ b_2) &&= equalb\ b_1\ b_2\\
&equal\langle \times\rangle\ equala\ equalb\ (a_1, b_1)\ (a_2, b_2) &&= equala\ a_1\ a_2 \wedge equalb\ b_1\ b_2.
\end{aligned}
$$

This definition contains all ingredients needed to specialize *equal* for arbitrary types. Types introduced by **data** declarations are also subsumed since they introduce 'sum of products' types.

Perhaps surprisingly, we can even slightly generalize the type of *equal* (without affecting the code): the element types need not be identical.

$$
\begin{aligned}
&Equal\langle\star\rangle\ t_1\ t_2 &&= t_1 \to t_2 \to Bool\\
&Equal\langle\kappa_1 \to \kappa_2\rangle\ t_1\ t_2 &&= \forall a_1\ a_2\,.\,Equal\langle\kappa_1\rangle\ a_1\ a_2 \to Equal\langle\kappa_2\rangle\ (t_1\ a_1)\ (t_2\ a_2)\\
&equal\langle t :: \kappa\rangle &&:: Equal\langle\kappa\rangle\ t\ t
\end{aligned}
$$

As a final remark, note that the second equation has the same structure for all polytypic values. Thus, a polytypic function is uniquely described by its type on '$\star$' instances and by its action on predefined types.

In Clean, a polytypic function can be captured in a straightforward way using *generic classes* [5]. Here is how *equal* is implemented.

$$
\begin{array}{ll}
\textbf{generic}\ \mathit{equal}\ t_1\ t_2 & :: t_1 \rightarrow t_2 \rightarrow \mathit{Bool} \\
\textbf{instance}\ \mathit{equal}\ 1\ \textbf{where} & \\
\quad \mathit{equal}\ ()\ () & = \mathit{True} \\
\textbf{instance}\ \mathit{equal}\ (+)\ \textbf{where} & \\
\quad \mathit{equal}\ \mathit{equala}\ \mathit{equalb}\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = \mathit{equala}\ a_1\ a_2 \\
\quad \mathit{equal}\ \mathit{equala}\ \mathit{equalb}\ (\mathit{Inl}\ a_1)\ (\mathit{Inr}\ b_2) & = \mathit{False} \\
\quad \mathit{equal}\ \mathit{equala}\ \mathit{equalb}\ (\mathit{Inr}\ b_1)\ (\mathit{Inl}\ a_2) & = \mathit{False} \\
\quad \mathit{equal}\ \mathit{equala}\ \mathit{equalb}\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = \mathit{equalb}\ b_1\ b_2 \\
\textbf{instance}\ \mathit{equal}\ (\times)\ \textbf{where} & \\
\quad \mathit{equal}\ \mathit{equala}\ \mathit{equalb}\ (a_1, b_1)\ (a_2, b_2) & = \mathit{equala}\ a_1\ a_2 \wedge \mathit{equalb}\ b_1\ b_2
\end{array}
$$

The correspondence to the above definition of *equal* should be immediate. Now, if we define an 'ordinary' function that depends on *equal* such as

$$
\begin{array}{ll}
\mathit{member} & :: (\mathit{equal}\ t) \Rightarrow t \rightarrow \mathit{List}\ t \rightarrow \mathit{Bool} \\
\mathit{member}\ a\ \mathit{Nil} & = \mathit{False} \\
\mathit{member}\ a\ (\mathit{Cons}\ b\ x) & = \mathit{equal}\ a\ b \vee \mathit{member}\ a\ x,
\end{array}
$$

then this dependence is recorded in the context of the type signature (like for overloaded functions).

5 Strongly typed GUI resources

In this paper, a GUI resource is a purely functional value that is an instance of one of the GUI type constructor classes, which are provided in the API of the Object I/O library. In order to keep the following discussion reasonably short, we will study three simplified versions of the Object I/O library. Each of these systems identifies a type constructor class whose instances we are interested in. (We do not care about the actual member functions, though.) The systems differ in the increasingly fine-grained access to state. The first system only allows access to global state, the second introduces a state that is shared by all GUI components within the same process, and the third introduces truly local state within GUI components.

In each of these systems we develop two representative *generic* functions: *eval* and *param*. The function *eval* takes a resource value and tries to find a function that can be applied to a given state argument. This function models the Object I/O interpretation function that, given a certain event, needs to find the corresponding callback function. The function *param* takes a resource value and replaces specified callback functions. This function is representative of applications that use GUI resources: a GUI resource contains the *look* of a GUI object, and it is the responsibility of the application to add the *feel*.

5.1 GUI objects with persistent state

The system *Global* is equivalent to the Haskell version of the Object I/O library using *mutable variables* for (local) state, and monadic actions [1]. We have *Obj*ects that are parameterised by *actions* and that can contain other *Obj*ects. Objects are identified by an *Int* value.

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Obj}\ \mathit{obj} & = \mathit{Obj}\ \mathit{Int}\ (\mathit{Transf}\ {*}\mathit{World})\ \mathit{obj} \\
\textbf{type}\ \mathit{Transf}\ x & = x \to x \\
\textbf{instance}\ (\mathit{Global}\ s) & \Rightarrow \mathit{Global}\ (\mathit{Obj}\ \ s) \\
\textbf{instance}\ \mathit{Global}\ () & \\
\textbf{instance}\ (\mathit{Global}\ s_1, \mathit{Global}\ s_2) & \Rightarrow \mathit{Global}\ (+ s_1\ s_2) \\
\textbf{instance}\ (\mathit{Global}\ s_1, \mathit{Global}\ s_2) & \Rightarrow \mathit{Global}\ (\times\ s_1\ s_2)
\end{array}
$$

Note that, in the spirit of the Object I/O library, *Obj* is parameterised by a type constructor variable which has kind $\star$, so *Obj* has kind $\star \to \star$.

Using a bit of imagination, we can think of *Obj* values as interactive objects. We interpret an (*Obj nr io children*) value as an interactive object that is identified by *nr*, which has behaviour *io*, and perhaps sub-objects *children*. Of course, instead of using ($\mathit{Transf}\ {*}\mathit{World}$) for actions, we can also use a monadic style, with (*IO* ()) action types, but this is not essential. The unit type comprises the simplest interactive object possible, one without interaction or sub-objects. The pair type glues together two objects of different types. Sum types model choice of objects.

Evaluation of objects Our first example of a generic function is *eval*, that, given a value s_1, the number *nr* of an object in s_1, and a $*\mathit{World}$ value w, looks up the object (*Obj nr io* _) in s_1, and applies the action *io* to w. Its signature for types t of kind $\star$ is $t \to \mathit{Int} \to \mathit{Transf}\ {*}\mathit{World}$.

$$
\begin{array}{ll}
\mathit{Eval}\langle \kappa :: \Box \rangle & :: \kappa \to \star \\
\mathit{Eval}\langle \star \rangle\ t & = t \to \mathit{Int} \to \mathit{Transf}\ {*}\mathit{World} \\
\mathit{Eval}\langle \kappa_1 \to \kappa_2 \rangle\ t & = \forall x\, .\, \mathit{Eval}\langle \kappa_1 \rangle\ x \to \mathit{Eval}\langle \kappa_2 \rangle\ (t\ x) \\
\mathit{eval}\langle t :: \kappa \rangle & :: \mathit{Eval}\langle \kappa \rangle\ t \\
\mathit{eval}\langle 1 \rangle\ ()\ \_ & = \mathit{id} \\
\mathit{eval}\langle \times \rangle\ \mathit{ea}\ \mathit{eb}\ (a, b)\ \mathit{nr} & = \mathit{eb}\ b\ \mathit{nr} \cdot \mathit{ea}\ a\ \mathit{nr} \\
\mathit{eval}\langle + \rangle\ \mathit{ea}\ \mathit{eb}\ (\mathit{Inl}\ a)\ \mathit{nr} & = \mathit{ea}\ a\ \mathit{nr} \\
\mathit{eval}\langle + \rangle\ \mathit{ea}\ \mathit{eb}\ (\mathit{Inr}\ b)\ \mathit{nr} & = \mathit{eb}\ b\ \mathit{nr} \\
\mathit{eval}\langle \mathit{Obj} \rangle\ \mathit{ea}\ (\mathit{Obj}\ \mathit{nr}'\ \mathit{io}\ a)\ \mathit{nr} & = \textbf{if}\ \mathit{nr}' = \mathit{nr}\ \textbf{then}\ \mathit{io}\ \textbf{else}\ \mathit{ea}\ a\ \mathit{nr}
\end{array}
$$

Here is an example use of *eval*:

$$
\begin{array}{l}
\mathit{Start} :: {*}\mathit{World} \to {*}\mathit{World} \\
\mathit{Start}\ w \\
\quad = \textbf{case}\ \mathit{readDynamic}\ \texttt{"obj.dyn"}\ w\ \textbf{of} \\
\qquad (\mathit{True}, \mathit{expr} :: (\mathit{eval}\ t) \Rightarrow t, w) \\
\qquad\qquad \to \mathit{eval}_\star\ \mathit{expr}\ 1\ w \\
\qquad (\_, \_, w) \to w
\end{array}
$$

In Clean, the main function is called *Start*. In order to indicate that it needs access to the external world, it requires a $*World$ argument and returns a possibly modified $*World$. The type *World* is an instance of the *FileSystem* class. If we assume that the file *obj.dyn* contains the value

$$\begin{array}{l} \textbf{dynamic}\ (\mathit{Obj}\ 0\ (\mathit{print}\ \texttt{"Hi␣there."})\ () \\ \quad\quad\quad\ ,\mathit{Obj}\ 2\ (\mathit{print}\ \texttt{"Goodbye"}) \\ \quad\quad\quad\quad\quad (\mathit{Obj}\ 1\ (\mathit{print}\ \texttt{"Hello␣world"})\ ()) \\ \quad\quad\quad\ ) :: (\mathit{Obj}\ (), \mathit{Obj}\ (\mathit{Obj}\ ())) \end{array}$$

then the executable prints "Hello world", which is the action associated with the 'first' object that is identified by 1 (in depth-first search order). The action $\mathit{print} :: \mathit{String} \to \mathit{Transf}\ *\mathit{World}$ prints its *String* argument to the console.

As an aside, note that *eval* is essentially a *reduction* or a *crush* except it treats one type instance, namely *Obj*, in a special way.

Generic parameterisation of actions The function *param* is given a list of $(\mathit{Int}, \mathit{Transf}\ *\mathit{World})$ pairs and an object structure. It looks up all *Obj* values and replaces each action by the given action entry in the list. The function *lookup* is a variation of the *lookup* function from the Haskell standard prelude. Instead of returning a *Maybe value*, a default result is passed as the second argument, which is returned if the key element was not found.

$$\begin{array}{ll} \mathit{Param}\langle\kappa :: \Box\rangle & :: \kappa \to \star \\ \mathit{Param}\langle\star\rangle\ t & = t \to [(\mathit{Int}, \mathit{Transf}\ *\mathit{World})] \to t \\ \mathit{Param}\langle\kappa_1 \to \kappa_2\rangle\ t & = \forall x\,.\,\mathit{Param}\langle\kappa_1\rangle\ x \to \mathit{Param}\langle\kappa_2\rangle\ (t\ x) \\ \mathit{param}\langle t :: \kappa\rangle & :: \mathit{Param}\langle\kappa\rangle\ t \\ \mathit{param}\langle 1\rangle\ ()\ \_ & = () \\ \mathit{param}\langle\times\rangle\ \mathit{pa}\ \mathit{pb}\ (a,b)\ \mathit{fs} & = (\mathit{pa}\ a\ \mathit{fs}, \mathit{pb}\ b\ \mathit{fs}) \\ \mathit{param}\langle+\rangle\ \mathit{pa}\ \mathit{pb}\ (\mathit{Inl}\ a)\ \mathit{fs} & = \mathit{Inl}\ (\mathit{pa}\ a\ \mathit{fs}) \\ \mathit{param}\langle+\rangle\ \mathit{pa}\ \mathit{pb}\ (\mathit{Inr}\ b)\ \mathit{fs} & = \mathit{Inr}\ (\mathit{pb}\ b\ \mathit{fs}) \\ \mathit{param}\langle\mathit{Obj}\rangle\ \mathit{pa}\ (\mathit{Obj}\ \mathit{nr}\ \mathit{io}\ a)\ \mathit{fs} & = \mathit{Obj}\ \mathit{nr}\ (\mathit{lookup}\ \mathit{nr}\ \mathit{io}\ \mathit{fs})\ (\mathit{pa}\ a\ \mathit{fs}) \end{array}$$

The following program illustrates the use of *param*.

$$\begin{array}{l} \mathit{Start} :: *\mathit{World} \to *\mathit{World} \\ \mathit{Start}\ w \\ \quad = \textbf{case}\ \mathit{readDynamic}\ \texttt{"obj.dyn"}\ w\ \textbf{of} \\ \quad\quad (\mathit{True}, \mathit{expr} :: (\mathit{eval}\ t\ , \mathit{param}\ t) \Rightarrow t, w) \\ \quad\quad\quad\quad\quad\quad\quad\quad \to \mathit{eval}_\star\ (\mathit{param}_\star\ \mathit{expr}\ \mathit{fs})\ 1\ w \\ \quad\quad (\_, \_, w) \to w \\ \quad \textbf{where} \\ \quad\quad \mathit{fs} = [(2, \mathit{print}\ \texttt{"Hi␣there."}) \\ \quad\quad\quad\ \ ,(1, \mathit{print}\ \texttt{"Goodbye"}) \\ \quad\quad\quad\ \ ,(0, \mathit{print}\ \texttt{"Hello␣world"}) \\ \quad\quad\quad\ \ ] \end{array}$$

This time the application nicely prints "Goodbye".

As an aside, note that *param* is similar to a mapping function: the *lookup* function is mapped across an object structure. In fact, *param* could be written as a *map*, if we made the object types parametric in the *Int* component (the integer key). For obvious reasons, however, refactoring the type structure of the Object I/O Library is not a viable option.

5.2 GUI objects with public state

Let us now extend the previous system by a public state that is being passed to all object actions. Instead of having actions of type $(Transf * World)$, all actions are now of type $(Transf\ (ps, * World))$, where ps is the type of the *p*ublic *s*tate. Because in this and in the next system we 'enhance' *World* by additional state, we introduce a convenient synonym type $W\ st = (st, * World)$. Consequently, actions are of type $(Transf\ (W\ ps))$. The object structures we are interested in are constructed as follows:

$$
\begin{array}{ll}
\textbf{data}\ Obj\ obj\ ps & = Obj\ Int\ (Transf\ (W\ ps))\ (obj\ ps) \\
\textbf{type}\ W\ st & = (st, * World) \\
\textbf{type}\ Transf\ x & = x \rightarrow x \\
\textbf{data}\ NilCS\ ps & = NilCS \\
\textbf{data}\ SumCS\ a\ b\ ps & = InlCS\ a\ ps \mid InrCS\ b\ ps \\
\textbf{infixr}\ 9\ :\sim: & \\
\textbf{data}\ :\sim:\ a\ b\ ps & = (a\ ps)\ :\sim:\ (b\ ps) \\
\textbf{instance}\ (Share\ s) & \Rightarrow Share\ (Obj\ s) \\
\textbf{instance}\ Share\ NilCS & \\
\textbf{instance}\ (Share\ s_1, Share\ s_2) & \Rightarrow Share\ (SumCS\ s_1\ s_2) \\
\textbf{instance}\ (Share\ s_1, Share\ s_2) & \Rightarrow Share\ (:\sim:\ s_1\ s_2)
\end{array}
$$

The unit type, the sum and the pair type have been lifted to higher kinds: the type *NilCS* has kind $\star \rightarrow \star$ and *SumCS* and $:\sim:$ have kind $(\star \rightarrow \star) \rightarrow (\star \rightarrow \star) \rightarrow (\star \rightarrow \star)$. Below we will introduce the *eval* and *param* functions for this system.

Evaluation of public state objects The purpose of the evaluation function *eval* is to look up a particular object and apply its action to the proper arguments. In this case, the argument is not only the $* World$, but also the public state. Consequently, for a type t of kind $\star \rightarrow \star$ the type of *eval* must be: $t\ ps \rightarrow Int \rightarrow Transf\ (W\ ps)$. The return value of type ps is the 'new' public state of the system.

Assigning *eval* a polykinded type is not as straightforward as before: we want to enforce that the generic argument is a data structure that contains objects that have actions that work on ps. Hence, the types to generalise from are not just t, but rather $t\ ps$.

Consider the desired type of *eval* for the ':∼:' type constructor combinator. We require that the argument function types are parameterised by *ps*:

$$\begin{aligned}
&eval\langle :\sim: :: (\star \to \star) \to (\star \to \star) \to (\star \to \star)\rangle \\
&\quad :: \forall ps\,.\,(\forall a\,.\,a\ ps \to Int \to Transf\ (W\ ps)) \\
&\qquad\qquad \to (\forall b\,.\,b\ ps \to Int \to Transf\ (W\ ps)) \\
&\qquad\qquad \to :\sim:\ a\ b\ ps \to Int \to Transf\ (W\ ps)
\end{aligned}$$

For comparison, the expanded type of the '×' instance of *eval* in *Global* is[2]:

$$\begin{aligned}
&eval\langle \times :: \star \to \star \to \star\rangle \\
&\quad :: (\forall a\,.\,a \to Int \to Transf\ (W\ ())) \\
&\quad\ \to (\forall b\,.\,b \to Int \to Transf\ (W\ ())) \\
&\quad\ \to \times\ a\ b \to Int \to Transf\ (W\ ())
\end{aligned}$$

The point is that the data structures that we are interested in are *never* basic types of kind $\star$, but have always at least kind $\star \to \star$. To accommodate for this we introduce an operation $\uparrow\kappa$ that does this 'lifting' of kinds:

$$\begin{aligned}
&\uparrow\star &&= \star \to \star \\
&\uparrow(\kappa_1 \to \kappa_2) &&= \uparrow(\kappa_1) \to \uparrow(\kappa_2)
\end{aligned}$$

Using the kind lifter we can specify the polykinded type of *eval*.

$$\begin{aligned}
&Eval_{ps}\langle \kappa :: \Box\rangle &&:: \kappa \to \star \\
&Eval_{ps}\langle \uparrow\star\rangle\ t &&= t\ ps \to Int \to Transf\ (W\ ps) \\
&Eval_{ps}\langle \uparrow(\kappa_1 \to \kappa_2)\rangle\ t &&= \forall x\,.\,Eval_{ps}\langle\uparrow\kappa_1\rangle\ x \to Eval_{ps}\langle\uparrow\kappa_2\rangle\ (t\ x) \\
&eval\langle \uparrow t :: \uparrow\kappa\rangle &&:: \forall ps\,.\,Eval_{ps}\langle\uparrow\kappa\rangle\ \uparrow t
\end{aligned}$$

In the polytypic function body of *eval* we need to lift the type constructors, as well (the unit type 1 is lifted to *NilCS*, '+' to *SumCS*, and '×' to ':∼:').

$$\begin{aligned}
&eval\langle \uparrow t :: \uparrow\kappa\rangle &&:: \forall ps\,.\,Eval_{ps}\langle\uparrow\kappa\rangle\ \uparrow t \\
&eval\langle NilCS\rangle\ NilCS\ _ &&= id \\
&eval\langle :\sim:\rangle\ ea\ eb\ (a\ :\sim:\ b)\ nr &&= eb\ b\ nr \cdot ea\ a\ nr \\
&eval\langle SumCS\rangle\ ea\ eb\ (InlCS\ a)\ nr &&= ea\ a\ nr \\
&eval\langle SumCS\rangle\ ea\ eb\ (InrCS\ b)\ nr &&= eb\ b\ nr \\
&eval\langle Obj\rangle\ ea\ (Obj\ nr'\ io\ a)\ nr &&= \mathbf{if}\ nr' = nr\ \mathbf{then}\ io\ \mathbf{else}\ ea\ a\ nr
\end{aligned}$$

Observe the similarity of this definition to the one in Section 5.1.

Now, assume that the file *obj.dyn* contains objects that share an integer state with initial value 1000:

$$\begin{aligned}
&\mathbf{dynamic}\ (1000, Obj\ 0\ (\lambda\ (c, w) \to (c+1, w))\ () \\
&\qquad\qquad :\sim:\ Obj\ 2\ (\lambda\ (c, w) \to (c-1, w)) \\
&\qquad\qquad\qquad (Obj\ 1\ (\lambda(c, w) \to (c, print\ (show\ c)\ w))\ ()) \\
&\qquad)\ :: (Int, :\sim:\ (Obj\ ())\ (Obj\ (Obj\ ()))\ Int)
\end{aligned}$$

---

[2] We are cheating a little here, as we use the $(Transf\ (W\ ()))$ action type, instead of the $(Transf * World)$ type in order to emphasize the similarity.

The objects with identification value 0 and 2 increment and decrement the public state, respectively. The object with identification value 1 prints the current value of the public state without modification. As an example,

$$
\begin{array}{l}
\mathit{Start} :: *\mathit{World} \rightarrow *\mathit{World} \\
\mathit{Start}\ w \\
\quad = \textbf{case}\ \mathit{readDynamic}\ \texttt{"obj.dyn"}\ w\ \textbf{of} \\
\qquad (\mathit{True}, (\mathit{state}, \mathit{expr}) :: (\mathit{eval}\ t) \Rightarrow (\mathit{ps}, t\ \mathit{ps}), w) \\
\qquad\quad \rightarrow \mathit{snd}\ \ (\ (\mathit{eval}_{(\star\rightarrow\star)}\ \mathit{expr}\ 1 \\
\qquad\qquad\qquad \cdot\mathit{eval}_{(\star\rightarrow\star)}\ \mathit{expr}\ 0 \\
\qquad\qquad\qquad \cdot\mathit{eval}_{(\star\rightarrow\star)}\ \mathit{expr}\ 1 \\
\qquad\qquad\qquad)\ (\mathit{state}, w)) \\
\qquad (\_, \_, w) \rightarrow w
\end{array}
$$

will print the numbers 1000 and 1001.

Generic parameterisation of public state actions The *param* function now replaces actions of objects with a public state. We need to define a generic *param* function on $(s_2\ ps)$ values, that receives a list of $(\mathit{Int}, \mathit{Transf}\ \ (W\ \ ps))$ pairs.

$$
\begin{array}{ll}
\mathit{Param}_{ps}\langle \kappa :: \Box \rangle & :: \ \kappa \rightarrow \star \\
\mathit{Param}_{ps}\langle \uparrow\star \rangle\ t & = t\ ps \rightarrow [(\mathit{Int}, \mathit{Transf}\ \ (W\ \ ps))] \rightarrow t\ ps \\
\mathit{Param}_{ps}\langle \uparrow(\kappa_1 \rightarrow \kappa_2) \rangle\ t & = \forall x\,.\,\mathit{Param}_{ps}\langle \uparrow\kappa_1 \rangle\ x \rightarrow \mathit{Param}_{ps}\langle \uparrow\kappa_2 \rangle\ (t\ x) \\
\mathit{param}\langle \uparrow t :: \uparrow\kappa \rangle & :: \ \forall ps\,.\,\mathit{Param}_{ps}\langle \uparrow\kappa \rangle\ \uparrow t \\
\mathit{param}\langle \mathit{NilCS} \rangle\ \mathit{NilCS}\ \_ & = \mathit{NilCS} \\
\mathit{param}\langle :\sim: \rangle\ \mathit{pa}\ \mathit{pb}\ (a\ \ :\sim:\ \ b)\ \mathit{fs} & = \mathit{pa}\ \ a\ \ \mathit{fs}\ \ :\sim:\ \ \mathit{pb}\ \ b\ \ \mathit{fs} \\
\mathit{param}\langle \mathit{SumCS} \rangle\ \mathit{pa}\ \mathit{pb}\ (\mathit{InlCS}\ \ a)\ \mathit{fs} & = \mathit{InlCS}\ \ (\mathit{pa}\ \ a\ \ \mathit{fs}) \\
\mathit{param}\langle \mathit{SumCS} \rangle\ \mathit{pa}\ \mathit{pb}\ (\mathit{InrCS}\ \ b)\ \mathit{fs} & = \mathit{InrCS}\ \ (\mathit{pb}\ \ b\ \ \mathit{fs}) \\
\mathit{param}\langle \mathit{Obj} \rangle\ \mathit{pa}\ (\mathit{Obj}\ \ \mathit{nr}\ \ \mathit{io}\ \ a)\ \mathit{fs} & = \mathit{Obj}\ \ \mathit{nr}\ \ (\mathit{lookup}\ \ \mathit{nr}\ \ \mathit{io}\ \ \mathit{fs})\ (\mathit{pa}\ \ a\ \ \mathit{fs})
\end{array}
$$

If we use the same file content for *obj.dyn* as above, then the following program will print the numbers 1000 and 999.

$$
\begin{array}{l}
\mathit{Start} :: *\mathit{World} \rightarrow *\mathit{World} \\
\mathit{Start}\ w \\
\quad = \textbf{case}\ \mathit{readDynamic}\ \texttt{"obj.dyn"}\ w\ \textbf{of} \\
\qquad (\mathit{True}, (\mathit{state}, \mathit{expr}) :: (\mathit{eval}\ t, \mathit{param}\ t) \Rightarrow (\mathit{ps}, t\ \mathit{ps}), w) \\
\qquad\quad \rightarrow \mathit{snd}\ \ (\ (\mathit{eval}_{(\star\rightarrow\star)}\ \mathit{expr}\ 1 \\
\qquad\qquad\qquad \cdot\mathit{eval}_{(\star\rightarrow\star)}\ \mathit{expr}\ 0 \\
\qquad\qquad\qquad \cdot\mathit{eval}_{(\star\rightarrow\star)}\ (\mathit{param}_{(\star\rightarrow\star)}\ \mathit{expr}\ \mathit{fs})\ 1 \\
\qquad\qquad\qquad)\ (\mathit{state}, w)) \\
\qquad (\_, \_, w) \rightarrow w \\
\quad \textbf{where} \\
\qquad \mathit{fs} = [(0, \lambda(c, w) \rightarrow (c - 1, w)), (2, \lambda(c, w) \rightarrow (c + 1, w))]
\end{array}
$$

5.3 GUI objects with public and local state

In the system *Share* all objects manipulate the $*World$ and a public state of type ps. Passing around a single public state has, of course, all the disadvantages and advantages of global state. To allow for a more modular design of objects, local state is a prerequisite. To this end the Object I/O library allows objects to encapsulate their own *local* state, or share and extend local state among child objects. As with the public state, the local state in scope is passed along as a distinct type constructor parameter. Our final system is a pretty good model of the Object I/O library.

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Obj}\ \mathit{obj}\ \mathit{ls}\ \mathit{ps} & = \mathit{Obj}\ \mathit{Int}\ (\mathit{Transf}\ (W\ (\mathit{ls}, \mathit{ps})))\ (\mathit{obj}\ \mathit{ls}\ \mathit{ps}) \\
\textbf{type}\ W\ \mathit{st} & = (\mathit{st}, *\mathit{World}) \\
\textbf{type}\ \mathit{Transf}\ x & = x \to x \\
\textbf{data}\ \mathit{NilLS}\ \mathit{ls}\ \mathit{ps} & = \mathit{NilLS} \\
\textbf{data}\ \mathit{SumLS}\ a\ b\ \mathit{ls}\ \mathit{ps} & = \mathit{InlLS}\ a\ \mathit{ls}\ \mathit{ps} \mid \mathit{InrLS}\ b\ \mathit{ls}\ \mathit{ps} \\
\textbf{infixr}\ 9\ {:}{+}{:} & \\
\textbf{data}\ {:}{+}{:}\ a\ b\ \mathit{ls}\ \mathit{ps} & = (a\ \mathit{ls}\ \mathit{ps})\ {:}{+}{:}\ (b\ \mathit{ls}\ \mathit{ps}) \\
\textbf{instance}\ (\mathit{Local}\ s) & \Rightarrow \mathit{Local}\ (\mathit{Obj}\ s) \\
\textbf{instance}\ \mathit{Local}\ \mathit{NilLS} & \\
\textbf{instance}\ (\mathit{Local}\ s_1, \mathit{Local}\ s_2) & \Rightarrow \mathit{Local}\ (\mathit{SumLS}\ s_1\ s_2) \\
\textbf{instance}\ (\mathit{Local}\ s_1, \mathit{Local}\ s_2) & \Rightarrow \mathit{Local}\ ({:}{+}{:}\ s_1\ s_2)
\end{array}
$$

Again, the type constructors *NilLS*, *SumLS* and ':+:' are the lifted versions of 1, '+' and '×': $\mathit{NilCS} : \star \to \star \to \star$ and $\mathit{SumLS}, {:}{+}{:}\ :(\star \to \star \to \star) \to (\star \to \star \to \star) \to (\star \to \star \to \star)$. We introduce the kind lifter $\Uparrow\kappa$ for this purpose:

$$
\begin{array}{ll}
\Uparrow\star & = \star \to \star \to \star \\
\Uparrow(\kappa_1 \to \kappa_2) & = \Uparrow(\kappa_1) \to \Uparrow(\kappa_2)
\end{array}
$$

Passing around the local state type argument allows us to introduce a *new* local state or to *add* a local state. This is accomplished by the following two type constructor combinators that can be found in the Object I/O library:

$$
\begin{array}{ll}
\textbf{data}\ \mathit{NewLS}\ t\ \mathit{ls}\ \mathit{ps} & = \exists \mathit{new}\,.\,\mathit{NewLS}\ \mathit{new}\ (t\ \mathit{new}\ \mathit{ps}) \\
\textbf{data}\ \mathit{AddLS}\ t\ \mathit{ls}\ \mathit{ps} & = \exists \mathit{add}\,.\,\mathit{AddLS}\ \mathit{add}\ (t\ (\mathit{add}, \mathit{ls})\ \mathit{ps}) \\
\textbf{instance}\ (\mathit{Local}\ s) & \Rightarrow \mathit{Local}\ (\mathit{NewLS}\ s) \\
\textbf{instance}\ (\mathit{Local}\ s) & \Rightarrow \mathit{Local}\ (\mathit{AddLS}\ s)
\end{array}
$$

Evaluation of stateful objects Naturally, the *eval* function for *Local* objects has a more involved type than the *eval* function of *Share* objects. The reason is that the local state values are stored in the structure itself, and need to be restored after evaluation of the action.

$$
\begin{array}{l}
\mathit{eval} :: (\mathit{Local}\ s_3) \Rightarrow \mathit{Int} \to \mathit{Transf}\ (W\ (\mathit{ps}, s_3\ ()\ \mathit{ps})) \\
\mathit{eval}\ \mathit{nr}\ ((\mathit{ps}, s_3), w) \\
\quad = \textbf{let}\ (s_3', ((\_, \mathit{ps}'), w')) = \mathit{eval}'_{(\star\to\star\to\star)}\ s_3\ \mathit{nr}\ (((), \mathit{ps}), w)\ \textbf{in}\ ((\mathit{ps}', s_3'), w')
\end{array}
$$

The function *eval* makes use of a generic helper function called *eval'*. The type of *eval'* is:

$$Eval_{(ls,ps)}\langle\Uparrow\star\rangle\ s_3 = s_3\ ls\ ps \rightarrow Int \rightarrow W\ (ls, ps) \rightarrow (s_3\ ls\ ps,\ W\ (ls, ps))$$

The new polykinded type and implementation of *eval'* shouldn't be a surprise by now. The function definition becomes more complicated because now the argument structure must be rebuilt as part of the function result.

$$
\begin{array}{ll}
Eval_{(ls,ps)}\langle\kappa :: \Box\rangle & :: \kappa \rightarrow \star \\
Eval_{(ls,ps)}\langle\Uparrow\star\rangle\ t & = t\ ls\ ps \rightarrow Int \rightarrow W\ (ls, ps) \rightarrow (t\ ls\ ps, W\ (ls, ps)) \\
Eval_{(ls,ps)}\langle\Uparrow(\kappa_1 \rightarrow \kappa_2)\rangle\ t & = \forall x\,.\,Eval_{(ls,ps)}\langle\Uparrow\kappa_1\rangle\ x \rightarrow Eval_{(ls,ps)}\langle\Uparrow\kappa_2\rangle\ (t\ x)
\end{array}
$$

$$
\begin{array}{l}
eval'\langle\Uparrow t :: \Uparrow\kappa\rangle \quad :: \forall ls\,.\,\forall ps\,.\,Eval_{(ls,ps)}\langle\Uparrow\kappa\rangle\ \Uparrow t \\
eval'\langle NilLS\rangle\ NilLS\ \_\ wst = (NilLS, wst) \\
eval'\langle{:+:}\rangle\ ea\ eb\ (a :+: b)\ nr \\
\qquad = \mathbf{let}\ (a', wst') \qquad = ea\ a\ nr\ wst \\
\qquad\qquad\quad (b', wst'') \qquad = eb\ b\ nr\ wst' \\
\qquad\quad \mathbf{in}\ (a' :+: b', wst'') \\
eval'\langle+\rangle\ ea\ eb\ (InlLS\ a)\ nr\ wst \\
\qquad = \mathbf{let}\ (a', wst') \qquad = ea\ a\ nr\ wst\ \mathbf{in}\ (InlLS\ a', wst') \\
eval'\langle+\rangle\ ea\ eb\ (InrLS\ b)\ nr\ wst \\
\qquad = \mathbf{let}\ (b', wst') \qquad = eb\ b\ nr\ wst\ \mathbf{in}\ (InrLS\ b', wst') \\
eval'\langle Obj\rangle\ ea\ (Obj\ nr'\ io\ a)\ nr\ ((ls, ps), w) \\
\quad = \mathbf{if}\ nr' = nr \\
\qquad \mathbf{then\ let}\ ((ls', ps'), w') = io\ ((ls, ps), w) \\
\qquad\qquad \mathbf{in}\ (((ls', ps'), Obj\ nr'\ io\ a), w') \\
\qquad \mathbf{else\ let}\ (a', ((ls', ps'), w')) = ea\ a\ nr\ ((ls, ps), w) \\
\qquad\qquad \mathbf{in}\ (((ls', ps'), Obj\ nr'\ io\ a'), w') \\
eval'\langle NewLS\rangle\ ea\ (NewLS\ new\ a)\ nr\ ((ls, ps), w) \\
\qquad = \mathbf{let}\ (a', ((new', ps'), w')) = ea\ a\ nr\ ((new, ps), w) \\
\qquad\quad \mathbf{in}\ (((ls, ps'), NewLS\ new'\ a'), w') \\
eval'\langle AddLS\rangle\ ea\ (AddLS\ add\ a)\ nr\ ((ls, ps), w) \\
\qquad = \mathbf{let}\ (a', (((add', ls'), ps'), w')) = ea\ a\ nr\ (((add, ls), ps), w) \\
\qquad\quad \mathbf{in}\ (((ls', ps'), AddLS\ add'\ a'), w')
\end{array}
$$

As an example, we will turn the 'counting' object of Section 5.2 into a reusable object that encapsulates the integer state locally. We now assume that the file *obj.dyn* now contains the following object:

$$
\begin{array}{l}
\mathbf{dynamic}\ (NewLS\ 1000 \\
\qquad\qquad\qquad (Obj\ 0\ (\lambda(c, w) \quad \rightarrow (c + 1, w))\ () \\
\qquad\qquad\qquad :+:\ Obj\ 2\ (\lambda(c, w) \rightarrow (c - 1, w)) \\
\qquad\qquad\qquad\qquad (Obj\ 1\ (\lambda\ \ (\ \ c, w) \rightarrow (c, print\ (show\ c)\ w))\ ()) \\
\qquad\qquad\qquad) \\
\qquad\qquad) :: NewLS\ ({:+:}\ (Obj\ ())\ (Obj\ (Obj\ ())))\ ls\ ps
\end{array}
$$

The only difference to the example in Section 5.2 is that we have moved the public integer state inside the object (first argument of *NewLS*). The type of this object clearly hides the fact that the object internally operates on integers. So the following program still produces output 1000 and 1001:

$$
\begin{array}{l}
\mathit{Start} :: *\mathit{World} \to *\mathit{World} \\
\mathit{Start}\ w \\
\quad = \textbf{case}\ \mathit{readDynamic}\ \texttt{"obj.dyn"}\ w\ \textbf{of} \\
\qquad (\mathit{True}, \mathit{expr} :: (\mathit{eval}'\ t) \Rightarrow t\ \mathit{ls}\ \mathit{ps}, w) \\
\qquad\quad \to \mathit{snd}\ ((\mathit{eval}\ 1 \cdot \mathit{eval}\ 0 \cdot \mathit{eval}\ 1)\ (((), \mathit{expr}), w)) \\
\qquad (\_, \_, w) \to w
\end{array}
$$

Generic parameterisation of stateful actions Because local states are existentially quantified, the parameterisation function *param* can't be given a list of actions that work on the intended local state. The only state whose type is known is the public state and, of course, the $*\mathit{World}$. There are basically two ways to pass the action list to *param*:

- make the list of actions polymorphic in *ls* (this requires rank-2 polymorphism), so the type of *param* is:

$$\mathit{Param}\langle\Uparrow\star\rangle\ t = t\ \mathit{ls}\ \mathit{ps} \to [(\mathit{Int}, \forall \mathit{ls}'\,.\,\mathit{Transf}\ (W\ (\mathit{ls}', \mathit{ps})))] \to t\ \mathit{ls}\ \mathit{ps}$$

- Pass $[(\mathit{Int}, \mathit{Transf}\ (W\ \mathit{ps}))]$ identification-action pairs and lift the actions locally to the proper type using the following lifting function:

$$
\begin{array}{ll}
\mathit{noLS} & :: \mathit{Transf}\ (W\ \mathit{ps}) \to \mathit{Transf}\ (W\ (\mathit{ls}, \mathit{ps})) \\
\mathit{noLS}\ f\ ((\mathit{ls}, \mathit{ps}), w) & = \textbf{let}\ (\mathit{ps}', w') = f\ (\mathit{ps}, w)\ \textbf{in}\ ((\mathit{ls}, \mathit{ps}'), w')
\end{array}
$$

Here we adopt the first option. Note that in either case, the new actions do not obtain access to the local state of the object with which they become associated, as this is forbidden by the existential quantification.

The definition of *param* is quite similar to the one in Section 5.2 (the code is omitted).

$$
\begin{array}{ll}
\mathit{Param}_{(ls,ps)}\langle\kappa :: \Box\rangle & :: \kappa \to \star \\
\mathit{Param}_{(ls,ps)}\langle\Uparrow\star\rangle\ t & = t\ \mathit{ls}\ \mathit{ps} \to [(\mathit{Int}, \forall \mathit{ls}'\,.\,\mathit{Transf}\ (W\ (\mathit{ls}', \mathit{ps})))] \to t\ \mathit{ls}\ \mathit{ps} \\
\mathit{Param}_{(ls,ps)}\langle\Uparrow(\kappa_1 \to \kappa_2)\rangle\ t & = \forall x\,.\,\mathit{Param}_{(ls,ps)}\langle\kappa_1\rangle\ x \to \mathit{Param}_{(ls,ps)}\langle\kappa_2\rangle\ (t\ x) \\
\mathit{param}\langle\Uparrow t :: \Uparrow\kappa\rangle & :: \forall \mathit{ls}\,.\forall \mathit{ps}\,.\,\mathit{Param}_{(ls,ps)}\langle\Uparrow\kappa\rangle\ \Uparrow t
\end{array}
$$

6 Related work

To the best knowledge of the authors, the concepts of generic programming and dynamic types have not been brought together in other work. The theme of this collaboration, the creation of strongly typed, higher-order resources (or rather, arbitrary – functional – values) and their flexible manipulation has, of course, occurred many times before. In this section we discuss relevant programming languages and concepts.

Persistent languages: Any persistent programming language has to deal with the same issues as described in this paper: how to handle (persistent) data. In Napier88 [13], a similar way of checking persistent data with a static type is provided as the dynamic type pattern match in Clean. One can also store and retrieve first-order procedures. Napier88 has a polymorphic type system, but does not provide overloading or polytypic techniques to manipulate data.

Polytypic languages: The basic research concerns of PolyP [12] and Generic H∀SKELL[6] are polytypic programming. Both languages are 'front-ends' to Haskell [14] and can therefore use the Haskell features to deal with persistent data, such as the `Dynamic` library of the GHC compiler. This library essentially uses overloading to wrap and unwrap dynamic values:

$$\begin{aligned} &\mathit{toDyn} &&:: \mathit{Typeable}\ a \Rightarrow a \rightarrow \mathit{Dynamic} \\ &\mathit{fromDyn} &&:: \mathit{Typeable}\ a \Rightarrow \mathit{Dynamic} \rightarrow a \rightarrow a \end{aligned}$$

Here overloading is used to check the appropriate types. To resolve potential overloading ambiguity, Haskell allows *expression type-signatures*.
The disadvantage of this approach is that the power of the polytypic language can not be exploited because dynamic values are handled via overloading, with fixed finite type domains.

Serialisation languages: The object-oriented programming language Java 1.1 [7] supports *(de)serialisation* via the classes ObjectOutputStream and ObjectInputStream. An object can only be (de)serialised if it implements the required class. Objects can customise (de)serialisation by defining their own implementation. Serialised values are *opaque*. The type of a deserialised value is determined by the class which readObject method is called, so there is no generic way of deserialising a value.

Resource languages: Or rather, development environments in which application programmers can create (GUI) resources that are used by applications. In these systems, a resource is defined by a dedicated (simple) language, that can be parsed by the development environment, and included in the executable of the application. Reading in a resource usually happens via statically defined identifiers (such as numbers or file names). On the program source level there is no connection between identifier and type. These programming environments usually come with a visual editing tool, which takes care that resources are at least internally consistent. Resources typically do not contain code.

7 Conclusions and future work

Do generics provide the required flexibility to programmers when manipulating resources? We have seen in Section 5 that we can write generic functions for data structures that are equivalent to the Object I/O data structures. Because the open-ended approach of the Object I/O library uses type constructor classes, it is inevitable that dynamic type patterns are overloaded. We have shown that generic functions can strengthen dynamics in this respect by being able to create at run-time an instance of any offered dynamic value. This in contrast to standard situations in which all instances are required to be created at compile-time.

With respect to the way resources are handled in common programming development environments, the 'genamics' approach offers many advantages:

- Resources are *strongly typed*, so the application program can be certain of some of the static properties of the resources that it relies on.
- Resources are *higher-order*, so we can attach default behaviour:

$$\mathbf{dynamic}\ (\mathit{MenuItem}\ \texttt{"Quit"}\ [\mathit{MenuFunction}\ \mathit{closeProcess}]) \\ \quad :: \mathit{MenuItem}\ \mathit{ls}\ (\mathit{PSt}\ \mathit{ps}\ )$$

  This example lets the parent process terminate whenever the associated menu item is selected. We can also parameterise the resource by adding any behaviour f to the default behaviour:

$$\mathbf{dynamic}\ (\lambda f \to \mathit{MenuItem}\ \texttt{"Quit"}\ [\mathit{MenuFunction}\ (\mathit{closeProcess} \cdot f)]) \\ \quad :: (\mathit{Transf}\ (\mathit{ls}, \mathit{PSt}\ \mathit{ps})) \to \mathit{MenuItem}\ \mathit{ls}\ (\mathit{PSt}\ \mathit{ps})$$

  Analogously, one could parameterise a dialog resource with a label-list that should be turned into a column of label-input field pairs, and a function associated with a button that gets the current input of these labels and performs an action.

These advantages shouldn't be a surprise to anyone using statically typed languages, because they result from proper modular programming. The point we would like to make is that this discipline also extends to the realm of resources (and so for dynamically typed languages) due to the combined power of dynamics and generics.

The Clean language incorporates both dynamics and generics, but still in an experimental stage and, most unfortunately, not within a single compiler. The generics language extension is able to handle the *Global* objects and functions in Section 5.1. The dynamics language extension is able to store and retrieve all objects mentioned in Section 5.

The kind lifting operation has been introduced in a rather ad-hoc fashion. We expect that it can be rephrased in the recent framework of type-indexed types [10].

Acknowledgements

Peter Achten would like to thank the department of Software Technology at the University of Utrecht for their hospitality while visiting Ralf Hinze during his sabattical.

References

1. Achten, P. and Peyton Jones, S. Porting the Clean Object I/O Library to Haskell. In Mohnen, M. and Koopman, P. eds., *Proceedings of the 20th International Workshop Implementation of Functional Languages (IFL 2000)*, Aachen, Germany, September 2000, Selected Papers, Springer, LNCS 2011, pp.194-213.
2. Achten, P. and Plasmeijer, M. The ins and outs of Clean I/O. In *Journal of Functional Programming*, **5**(1), January 1995, Cambridge University Press, pp.81-110.
3. Achten, P. and Plasmeijer, M. Interactive Functional Objects in Clean. In Clack, C. Davie, T. *Proceedings of the 9th International Workshop Implementation of Functional Languages (IFL'97)*, St. Andrews, Scotland, UK, September 1997, Selected Papers, Springer, LNCS 1467, pp.304-321.
4. Achten, P. and Wierich, M. *A Tutorial to the Clean Object I/O Library - Version 1.2*, Technical Report CSI-R0003, February 2000, Computing Science Institute, Faculty of Mathematics and Informatics, University of Nijmegen.
5. Alimarine, A. and Plasmeijer, M. A Generic Extension to Clean. In *Proceedings of 21st International Workshop Implementation of Functional Languages (IFL2001)*, Stockholm, Sweden, September 2001. *To appear.*
6. Clarke, D., Hinze, R., Jeuring, J., Löh, A., de Wit, J. Generic H∀SKELL, version 0.99 (Amber), November 1 2001, info@generic-haskell.org.
7. Flanagan, D. *Java in a nutshell*, 2nd edition, O'Reilly & Associates, Inc., 1997.
8. Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and J.N. Oliveira, editors, *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000), July 3-5, 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer-Verlag, July 2000.
9. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programmming*, 2002. To appear.
10. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types, 2002. In submission.
11. Hinze, R. and Peyton Jones, S. Derivable Type Classes. In Graham Hutton, ed., *Proceedings of the Fourth Haskell Workshop*, Montreal, Canada, September 17, 2000.
12. Jansson, P. and Jeuring, J. PolyP – a polytypic programming language extension. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, ACM Press, pp. 470-482.
13. Morrison, R., Connor, R.C.H., Kirby, G.N.C., Munro, D.S., Atkinson, M.P., Cutts, Q.I., Brown, A.L., Dearle, A. The Napier88 Persistent Programming Language and Environment. In Atkinson, M.P., Welland, R. eds. *Fully Integrated Data Environments*, Esprit Basic Research Series, Springer, 1999, pp. 98-154.
14. Peyton Jones, S. and Hughes, J. eds. *Report on the Programming Language Haskell 98 – A Non-strict, Purely Functional Language*, 1 February 1999.
15. Pil, M.R.C., Dynamic types and type dependent functions. In Hammond, Davie, Clack, eds., *Proc. of Implementation of Functional Languages (IFL '98)*, London, U.K., Springer-Verlag, Berlin, LNCS **1595**, pp.169-185.
16. Pil, M. *First Class File I/O*, PhD Thesis, *in preparation.*
17. Plasmeijer, M.J. and van Eekelen, M.C.J.D. *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley Publishing Company, 1993.