Under consideration for publication in J. Functional Programming

Interactive Programming in Agda – Objects and Graphical User Interfaces

ANDREAS ABEL
Department of Computer Science and Engineering, Gothenburg University, Sweden
STEPHAN ADELSBERGER
Department of Information Systems and Operations,
Vienna University of Economics, Austria
ANTON SETZER
Department of Computer Science, Swansea University, Swansea SA2 8PP, UK
(*e-mail:* `andreas.abel@gu.se, sadelsbe@wu.ac.at, a.g.setzer@swan.ac.uk`)

Abstract

We develop a methodology for writing interactive and object-based programs (in the sense of Wegner) in dependently-typed functional programming languages. The methodology is implemented in the ooAgda library. ooAgda provides a syntax similar to the one used in object-oriented programming languages, thanks to Agda's copattern matching facility. The library allows for the development of graphical user interfaces (GUIs), including the use of action listeners.

Our notion of interactive programs is based on the IO monad defined by Hancock and Setzer, which is a coinductive data type. We use a *sized* coinductive type which allows us to write corecursive programs in a modular way. Objects are server-side interactive programs that respond to method calls by giving answers and changing their state. We introduce two kinds of objects: simple objects and IO objects. Methods in simple objects are pure, while method calls in IO objects allow for interactions before returning their result. Our approach also allows us to extend interfaces and objects by additional methods.

We refine our approach to state-dependent interactive programs and objects through which we can avoid exceptions. For example, with a state-dependent stack object, we can statically disable the pop method for empty stacks. As an example, we develop the implementation of recursive functions using a safe stack. Using a coinductive notion of object bisimilarity, we verify basic correctness properties of stack objects and show the equivalence of different stack implementations. Finally, we give a proof of concept that our interaction model allows to write GUI programs in a natural way: we present a simple drawing program, and a program which allows to move a small spaceship using a button.

Note. We recommend printing this paper in color.

1 Introduction

Functional programming is based on the idea of reduction of expressions. This is a good notion for writing batch programs which take a fixed number of inputs to compute a fixed number of outputs. Interactive programs, however, do not fit directly into this paradigm, since they are programs which over time accept a possibly infinite number of inputs and respond in sequence with a possibly infinite number of outputs. There are several ways to overcome this. In functional programming, the main method currently used is Moggi's

IO monad (Moggi, 1991). The IO monad is a type (`IO a`) of computations depending on a return type `a`. Its elements are interactive programs which possibly terminate with a result of type `a`. It is an open-ended type: programming languages such as Haskell provide various functions constructing atomic elements of (`IO a`) for various types `a`, and the bindings `>>= :: IO a -> (a -> IO b) -> IO b` and `return :: a -> IO a` are used to construct programs from these atomic operations.

In a series of articles (Hancock & Setzer, 2000b,a, 2005; Setzer & Hancock, 2004) Peter Hancock and the third author of this article have developed a representation of interactive programs in dependent type theory. This approach is based on the notion of a coalgebra. The idea is that a (client-side) interactive program is represented by a possibly non-well-founded tree. The nodes are labeled with commands being issued to the real world, and the subtrees of a tree are labeled with responses from the real world to the respective command. For instance, if a node is labeled with the command *input a string*, its subtrees would be indexed over strings the user has entered; if the command is *write a character*, the response would be an element of the singleton type Unit, so there is only one subtree.

Execution of an interactive program thus is no longer the simple reduction of an expression. Instead, it is performed as follows: One computes a label from the root of the tree. A corresponding program is executed in the real world. The real world returns a corresponding response. Then, the subtree labeled with this response is chosen, and one repeats the same procedure for the root of that subtree. Additionally, there are special nodes called leaves, labeled by an element of the result type of the interactive program. If we reach such a leaf, the program terminates returning the label. The monadic operations *bind* and *return* can now be defined in a straightforward way as operations on such trees.

If we define trees by inductive data types (Agda keyword data), we obtain only well-founded trees, which means trees which have no infinitely deep branches. Interactive programs correspond to non-well-founded trees because they may run forever if never terminated. A non-well-founded tree can be represented in Agda by a record which is coinductive.

The programs discussed above were client-side interactive programs: they send a command to the real world and then receive a response and continue. In contrast, *graphical user interfaces* are server-side programs; they wait for an event—such as a click on a button—which means they wait for a command from the real world, answer with a result and then wait for the next command. Similarly, *objects* are ready to accept any call of a method. In response, they return a result, and the object changes its state. Based on this idea, the third author has developed (Setzer, 2006) the theory of defining object-based programs (in the sense of Wegner (1987)) in dependent type theory. The interaction trees of server-side programs and objects are functors of the form $\Pi\Sigma$ (meaning *for all requests, return some response*), rather than $\Sigma\Pi$ (*send some request and react to any response*) as for client-side programs.

The goal of this article is to substantially extend this theory and develop a methodology for actually implementing interactive and object-based programs in Agda in a user-friendly way. This will include object-based graphical user interfaces. We have developed the library ooAgda (Abel *et al.*, 2016), which allows writing objects and interactive programs in a way that is very close to how it would be done in an object-based programming language. At this stage, inheritance and subtyping are not available in ooAgda, so ooAgda is currently an object-based library. Using illustrative examples, we will show how ooAgda can be used for writing interactive programs which make use of objects. The simplest example will be a program which interacts with a cell containing a string via its methods *put* and *get*. Then, we will look at how to extend an object by adding more methods and extending its implementation. Furthermore, we will look at state-dependent interactive programs and objects. This allows us to write a safe stack, where popping is only allowed if the stack is non-empty; by safe we mean we can avoid exceptions. We will introduce bisimulation as equality, and show that the operations of *put* and *get* are inverse to each other w.r.t. bisimulation. We also show the equivalence of different stack implementations.

So far, in dependent type theory, not many interactive programs have been written. We prove that it is possible to write graphical interfaces by presenting two examples. The first one is a simple drawing program. The second example will be a graphical user interface having one button. In this example, we will make use of an object, which has action listeners as methods, which in turn will be added to a button event and a repaint event. Note that the focus here is not on developing advanced user interfaces, but to demonstrate that one can use objects and action listeners to develop graphical user interfaces in dependently typed programming.

The content of this article is as follows: In Sect. 2 we give a brief introduction into Agda. In Sect. 3 we recapitulate the theory of coalgebras and their representation in Agda. Then we review (Sect. 4) the theory of interactive programs in Agda. In Sect. 5 we introduce objects in Agda and write a small interactive program which makes use of an object representing a cell.

Guarded recursion (Coquand, 1994) allows only recursive calls of the following three forms: direct recursive calls to the function being defined, an expression which was defined before the function was defined, or constructors applied to the previous two possibilities. In particular, we cannot use functions to combine elements of the coalgebra to form new elements of it. This restricts modularity of programs since one cannot use an auxiliary function in a corecursive call. Instead, one needs to define a new function simultaneously with the function to be defined which computes the result of applying the auxiliary function to its arguments. The function is defined exactly like the auxiliary function, repeating essentially its definition. This makes programming tedious. In Sect. 6 we discuss how sized types allow for corecursive programs to be written more naturally. With sized types, such auxiliary functions are allowed in corecursive definitions provided they are size-preserving.

In Sect. 7 we give an example of how to extend an object by adding a new method. In Subsect. 8.1 and 8.2 we introduce state-dependent objects and show how a stack can be implemented which statically prevents pop-operations when empty. In 8.2.1 we develop a small example of how to implement recursive functions using a safe stack. In Subsect. 8.3 we demonstrate how to define bisimilarity as equality on objects. This equality is used to prove that the push and pop operations are inverse of each other. In Subsect. 8.4 we look at how to define state-dependent interactive programs which will be used later to define a more complex graphical user interface. In the last two sections, we will give examples of how to define graphical user interfaces in Agda; in Sect. 9 we introduce a simple drawing program; in Sect. 10 we introduce a graphical user interface in which we assign an action listener to a button. There are 3 versions of this interface. In the most complex one, action listeners are defined as in ordinary object-oriented programming: by creating an object which contains all the required action listeners as methods and associating them with the button and the repaint event.

We finish with a review of related work (Sect. 11), which in includes a comparison of our approach with Brady's work in Idris, and a conclusion with an outlook on future steps (Sect. 12).

Every line of Agda code provided in this paper has been type-checked by Agda and rendered by the Agda LaTeX-backend. However, we mostly omit administrative parts of the code such as modules and namespace handling; thus, the code as printed in this article will not be accepted by Agda as-is. Relative to the correctness of Agda itself, our code is type-safe. However, we see the need for a more solid theoretical foundation for Agda's sized types (Sect. 6). The complete code, including advanced examples, can be found in Abel *et al.* (2016). These examples compile to executable binaries using Agda 2.5 and GHC 7.8.

2 Introduction to Agda

Agda (AgdaTeam, 2016; Stump, 2016) is a theorem prover based on intensional Martin-Löf type theory (Martin-Löf, 1984). Code can be compiled using the MAlonzo compiler Agda Wiki (2011), which is a *Monadic* form of the Alonzo compiler (Benke, 2007); therefore, Agda can also be seen as a dependently typed programming language. It is closely related to the theorem prover Coq (2015). Furthermore, Agda is a total language, which is guaranteed by its termination and coverage checker without which Agda would be inconsistent. The current version of Agda is Agda 2, which was originally designed and implemented by Ulf Norell in his PhD thesis (2007).

In Agda, there are infinitely many levels of types: the lowest one is called Set. The next type level is called Set_1, which has the same closure properties as Set but also contains Set as an element. The next type level is called Set_2, etc. Furthermore, we can quantify over type levels, and obtain types (Set σ) depending on levels σ.

The main type constructors in Agda are dependent function types, inductive types, and coinductive types. In addition, there exist record types, which are used for defining coinductive types by their observations or elimination rules. Furthermore, there exists a highly generalised version of inductive-recursive and inductive-inductive definitions.

Inductive data types are dependent versions of algebraic data types as they occur in functional programming. They are given as sets A together with constructors which are strictly positive in A. For instance, the even and odd numbers are given by the simultaneous — as denoted by the keyword mutual — indexed inductive data types:

```
mutual
  data Even : ℕ → Set where
    0p   : Even 0
    sucp : {n : ℕ} → Odd n  → Even (suc n)

  data Odd : ℕ → Set where
    sucp : {n : ℕ} → Even n → Odd  (suc n)
```

The expression $(n : \mathbb{N}) \to A$ denotes a dependent function type, which is similar to a function type, but A can depend on n. The expression $\{n : \mathbb{N}\} \to A$ is an implicit version of the previous construct. Implicit arguments can be omitted, provided they can be inferred by the type checker. We can make a hidden argument explicit by writing, e.g., $(\mathsf{sucp}\ \{n\}\ p)$. If there are several explicit or implicit dependent arguments in a type, one can omit "$\to$", as illustrated in the following example: $(a : A)(b : B) \to C$ instead of $(a : A) \to (b : B) \to C$. The elements of $(\mathsf{Even}\ n)$ and $(\mathsf{Odd}\ n)$ are those that result from applying the respective constructors. Therefore, we can define functions by case distinction on these constructors using pattern matching, e.g.

```
mutual
  _+e_ : ∀ {n m} → Even n  → Even m  → Even  (n + m)
  0p      +e  p  =      p
  sucp p  +e  q  =      sucp  (p +o q)

  _+o_ : ∀ {n m}  → Odd n  → Even m  → Odd   (n + m)
  sucp p  +o  q  =      sucp  (p +e q)
```

Here, $\forall a \to B$ is an abbreviation for $(a : A) \to B$, where A can be inferred by Agda. $\forall\{a\} \to B$ is the same but for a hidden argument, while $\forall\{n\ \ m\} \to B$ abbreviates $\forall\{n\} \to \forall\{m\} \to B$. Agda supports mixfix operators, where "_" denotes the position of the arguments. For instance, $(\mathsf{0p} +\mathsf{e}\ p)$ stands for $(\_+\mathsf{e}\_\ \mathsf{0p}\ p)$. The combination of mixfix symbols together with the availability of Unicode symbols makes it possible to define Agda code which is very close to standard mathematical notation.

Nested patterns are allowed in pattern matching. The coverage checker verifies completeness and the termination checker ensures that the recursive calls follow a schema of extended primitive recursion.

An important indexed data type is propositional equality $x \equiv y$ (for $x, y : A$) which has as constructor a proof of reflexivity:

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x
```

This definition says that propositional equality is the least reflexive relation (modulo the built-in definitional equality of Agda).

3 Coalgebras in Agda

The approach to interactive programs we employ in this article is based on *(weakly) terminal coalgebras*. In this section, we recapitulate coalgebras and their definition in Agda, first by example, then in the general case.

3.1 Coalgebra by Example: Colists

Coalgebras are a versatile mathematical tool; for instance, they can model various classes of transition systems. Here, we consider *output automatons*, which consist of a (not necessarily) finite state set S and a transition function $t : S \to (1 + A \times S)$. Given a state $s : S$, the transition $t\,s$ can either lead us to termination (type alternative 1) or emit some output $a : A$ and lead us into a successor state $s' : S$ (type alternative $A \times S$). Defining the functor $F\,S = 1 + A \times S$, we say the pair (S,t) is an *F-coalgebra*. We may sometimes refer to this coalgebra by simply S or t, when the other component is clear from the context of discourse.

Let us call this functor ListF, for reasons that are apparent to the reader or will become so in a short while, and define it in Agda as a disjoint sum type ListF with two constructors nil and cons.

```
data ListF A S : Set where
  nil   : ListF A S
  cons  : (a : A) (s : S) → ListF A S
```

It should be clear that (ListF $A\,S$) is a faithful implementation of $1 + A \times S$, with nil corresponding to the left injection of the empty tuple, and (cons $a\,s$) to the right injection of the pair (a,s).

A ListF-coalgebra is now a pair (S,t) of a type S and a function $t : S \to$ ListF $A\,S$ for a fixed type A, and a transition will take us either to nil, meaning the automaton terminates, or cons $a\,s'$, meaning the automaton outputs a and enters the new state s'.

A ListF-coalgebra *morphism* from automaton $t : S \to$ ListF $A\,S$ to automaton $t' : S' \to$ ListF $A\,S'$ is a state map $f : S \to S'$ with two conditions:

1. Terminal states of t are mapped to terminal states of t', meaning that $t'\,(f\,s) =$ nil whenever $t\,s =$ nil.
2. Non-terminal states of t are mapped to corresponding non-terminal states of t' with the same output, meaning that $t'\,(f\,s_1) =$ cons $a\,(f\,s_2)$ whenever $t\,s_1 =$ cons $a\,s_2$.

These two conditions can be summarized as $t'\,(f\,s) =$ mapF $f\,(t\,s)$ for all $s : S$, using the functoriality witness mapF of ListF A.

```
mapF : ∀{A S S′} (f : S → S′) → (ListF A S → ListF A S′)
mapF f nil          = nil
mapF f (cons a s)   = cons a (f s)
```

Or, for the category-theory enthusiast, we can display this condition in form of a commutative diagram:

$$\begin{tikzcd}
S \arrow[r, "t"] \arrow[d, "f"'] & \mathsf{ListF}\ A\ S \arrow[d, "\mathsf{mapF}\ f"] \\
S' \arrow[r, "t'"] & \mathsf{ListF}\ A\ S'
\end{tikzcd}$$

If we run an output automaton t, starting in state s, to completion, we obtain a possibly terminating sequence, aka *colist*, of outputs $a_0, a_1, \ldots$. We call this colist unfold $t\ s$: Colist A. In Agda, the type of colists over A is defined as a recursive record type:

```
record Colist A : Set where
  force : ListF A (Colist A)
```

An element l : Colist A is a lazily computed record with a single field force l : ListF A (Colist A); one could also view it as an object with a single method force. Invocation of this method via (force l) will yield either nil or (cons $a\ l'$) for an output $a : A$ and a new colist l'.

In this sense, the pair (Colist A, force) can be seen as a ListF-coalgebra; any colist l is the state of an output automaton with force as the transition function. Colist A is even the *weakly terminal* or *weakly final* coalgebra, as every coalgebra (S,t) can be mapped into it via morphism (unfold t), so there exists a function, unfold $t : S \to$ Colist A, which makes the diagram commute. If we take bisimulation on Colist as equality, then Colist is actually *terminal* or *final*. This means that (unfold t) is the only function which makes this diagram commute:

$$\begin{tikzcd}
S \arrow[r, "t"] \arrow[d, "\mathsf{unfold}\ t"'] & \mathsf{ListF}\ A\ S \arrow[d, "\mathsf{mapF}\ (\mathsf{unfold}\ t)"] \\
\mathsf{Colist}\ A \arrow[r, "\mathsf{force}"'] & \mathsf{ListF}\ A\ (\mathsf{Colist}\ A)
\end{tikzcd}$$

The (weak) finality witness unfold can be implemented in Agda as follows. Herein, read the "with $t\,s$ . . ." followed by pattern matching as an additional case distinction over $(t\,s)$: ListF A S. The three dots "..." indicate that the pattern from the previous line is repeated, and "|" starts a pattern related to the term of the with construct:

```
unfold              :  ∀{A S} (t : S → ListF A S) → (S → Colist A)
force (unfold t s)  with t s
... | nil           =  nil
... | cons a s′     =  cons a (unfold t s′)
```

This definition is an instance of a function defined by *copattern matching* (Abel *et al.*, 2013). By itself, (unfold $t\ s$) does not reduce. Only when we subject it to projection force, it reduces as given by the right hand side of the definition; in this case, to a case distinction over $(t\,s)$.

Agda's termination checker accepts the recursively defined unfold: Since each recursive call removes one use of force, the reduction cannot continue forever. In fact, this definition follows the rules of guarded recursion (Coquand, 1994). Guarded recursion means in this setting that we can define a function recursively as long on the left hand side we apply at least one observation (here force) to the function applied to its arguments (of course we need to cover all copatterns). On the right hand side of the recursive definition, one can have either an element of the coalgebra defined before, a recursive call of the function to be defined, or constructors applied to such a recursive call. An example, which demonstrates that we cannot allow arbitrary functions to be applied to the recursive call, would be the black hole recursive definition f : $A \rightarrow$ Colist A, force $(f\ a) =$ force $(f\ a)$.

The above definition of unfold is equivalent to the generic force (unfold $t\ s$) $=$ mapF (unfold t) $(t\ s)$ obtained from the commutative diagram. However, the latter falls out of the scheme of guarded recursion and termination is less obvious. We will further discuss this issue in Section 3.2.

As an application of unfold, we generate the Collatz sequence. It starts with some number n. If $n = 1$, the sequence terminates. Otherwise, if n is even, it continues with $n/2$, and if n is odd, then it continues with $3n+1$.[1] In the following code, "_" is the Agda notation for an unused argument. The application (n divMod m) returns (result $q\ r\ s$), with quotient $q = n$ div m, remainder $r = n$ mod m, and a proof s of $n \equiv q * m + r$. Note also that pattern matching is executed in sequence: The pattern (collatzStep n) is only reached if $n \neq 1$.

```
collatzStep           :  ℕ → ListF ℕ ℕ
collatzStep 1         =  nil
collatzStep n            with n divMod 2
... | result q zero _ =  cons n q
... | _               =  cons n (1 + 3 * n)

collatzSequence :  ℕ → Colist ℕ
collatzSequence =  unfold collatzStep
```

The collatzSequence is obtained as the output of an automaton with transition function collatzStep, which directly implements the rules given before.

3.2 Coalgebras in General

We work in the category of types A : Set and functions $f : A \rightarrow B$. Assume a functor F, whose functoriality is witnessed by mapF, in Agda written as

```
F     :  Set → Set
mapF  :  ∀{A B} (f : A → B) → (F A → F B)
```

Of course, mapF has to fulfill the functor laws to qualify as a functoriality witness, namely mapF id $=$ id and mapF $(f \circ g) =$ mapF $f \circ$ mapF g.

An F-*coalgebra* consists of a pair (S, t) of a type S of states and a transition function t from a state s : S to t s : F S which typically may be some input or output with a (collection of) successor state(s).

[1] It is conjectured that (except for $n = 0$, which creates an infinite sequence of 0s), the resulting sequence is always finite. But as of today, this conjecture has resisted all proof attempts.

```
S : Set
t : S → F S
```

The (weakly) *terminal* F-coalgebra νF or the *coinductive type* obtained as the *greatest fixed point* of F is represented using a coinductive record type in Agda:

```
record νF : Set where
  force : F νF
```

Here, Agda requires F to be *strictly positive*. Projection force : νF → F νF is the *eliminator* of the coalgebra νF. It defines the *observations* one can make on νF. Weak terminality is witnessed by the function unfoldF $t : S \to$ νF for any coalgebra (S,t) which makes the following diagram commute:

$$\begin{tikzcd} S \arrow[r, "t"] \arrow[d, "\mathsf{unfoldF}\ t"'] & \mathsf{F}\ S \arrow[d, "\mathsf{mapF}\ (\mathsf{unfoldF}\ t)"] \\ \nu\mathsf{F} \arrow[r, "\mathsf{force}"'] & \mathsf{F}\ \nu\mathsf{F} \end{tikzcd}$$

Commutation means that the equation of morphisms

$$\mathsf{force} \circ \mathsf{unfoldF}\ t = \mathsf{mapF}\ (\mathsf{unfoldF}\ t) \circ t \tag{1}$$

holds. In Agda, we can *implement* unfoldF by taking the pointwise version of equation (1) as the definition of unfoldF. [2]

```
{-# TERMINATING #-}
unfoldF : ∀{S} (t : S → F S) → (S → νF)
force (unfoldF t s) = mapF (unfoldF t) (t s)
```

Taking the above equation as a rewrite rule preserves strong normalization of rewriting in Agda, as unfoldF is only reduced under projection force and thus not its recursive occurrence on the right hand side of this definition. However, Agda's termination checker (Abel & Altenkirch, 2002; Altenkirch & Danielsson, 2012) does not see that at this

[2] Digression: In System F with products and (impredicative) existential types, the weakly terminal coalgebra νF is definable (Matthes, 2002) requiring *only* the monotonicity witness mapF:

$$\begin{array}{lcl} \nu\mathsf{F} & = & \exists S.\ (S \to \mathsf{F}\ S) \times S \\ \mathsf{unfoldF}\ t & = & \lambda s.\ (t,s) \\ \mathsf{force} & = & \lambda(t,s).\ \mathsf{mapF}\ (\mathsf{unfoldF}\ t)\ (t\ s) \end{array}$$

In contrast to Agda, Matthes' monotone coinductive types do not need strict positivity of F but only the monotonicity witness mapF. Whether this carries over to Agda's predicative type theory needs to be explored. We do not rely on monotone coinductive types, but instead use sized coinductive types to justify the generic unfoldF.

point, so we override its verdict by screaming TERMINATING! To the defense of Agda's termination checker we have to say that a specific implementation of mapF for $B = \nu F$ of the form mapF $f\ x =$ force $(f\ something)$ would lead to the non-terminating reduction rule force (unfoldF $t\ s$) $\longrightarrow$ force (unfoldF $t\ something$). However, such an implementation of mapF is ruled out by its polymorphic type. Indeed, unfoldF passes a *type-based* termination check using sized types (Abel & Pientka, 2013), which we present in Section 6.

4 Interactive Programs in Agda

4.1 Interaction interfaces

Interaction of a program with, e.g., an operating system (OS), can be conceived as a sequence of *commands* (elements of Command), given by the program to the OS, for each of which the OS sends back a *response* (elements of Response). The type $(R\ c)$ of the response is dependent on the command (c : Command) that was given; thus, in Agda we model Response as a type family of kind Command → Set. The set Command and the indexed set Response form an interface for the interaction (Hancock & Setzer, 2000a). In Agda, this is modeled as a record of sets, and its type IOInterface itself inhabits the next type level Set_1 above Set. Note that IOInterface : Set would require Set : Set, but the latter is inconsistent by Girard's paradox (Girard, 1972; Hurkens, 1995).

```
record IOInterface : Set₁ where
    Command   :  Set
    Response  :  (c : Command) → Set
```

As an example, we define an interface ConsoleInterface of simple console programs. It has only two commands:

```
data ConsoleCommand : Set where
  getLine    :  ConsoleCommand
  putStrLn   :  String → ConsoleCommand
```

The first command, getLine, has no arguments; putStrLn is invoked with one argument of type String.

```
ConsoleResponse : ConsoleCommand → Set
ConsoleResponse  getLine        =  Maybe String
ConsoleResponse (putStrLn s)    =  Unit
```

Upon command getLine, the OS responds with a Maybe String, meaning nothing if the end of input has been reached, and just s when String s could be read from the console. Command (putStrLn s) is always answered with the trivial response Unit, which could be interpreted as *success*. Together, command and response types form a simple interaction interface:

```
ConsoleInterface : IOInterface
Command  ConsoleInterface  =  ConsoleCommand
Response  ConsoleInterface  =  ConsoleResponse
```

4.2 Interaction trees

From now on, we assume an arbitrary IOInterface

$I =$ record { Command $= C$; Response $= R$ }

Let (IO I A) be the type of programs which interact with the interface I and which, in case of termination, return an element of type A. The operations of (IO I A) are given below. Note that do follows the notation of earlier papers (Hancock & Setzer, 2000a) and is different from Haskell's *do notation*:

```
do      : ∀{A}   (c : C)       (f : R c → IO I A)  → IO I A
return  : ∀{A}   (a : A)                           → IO I A
_≫=_    : ∀{A B} (m : IO I A)  (k : A → IO I B)    → IO I B
```

The first operation, used in the form do c $\lambda r \to f r$, allows to issue a command c, and continue with $f\,r$ after receiving the response $r : R\,c$. Note Agda's precedence for λ: We do not have to parenthesize a trailing lambda-abstraction, i.e., do not need to write do c $(\lambda r \to f r)$.

The other two operations are desirable so that (IO I) is a monad, i.e., interactive programs can return a result or bind the result $a : A$ of an interactive computation m and continue as another interactive program $(k\,a)$ via $(m \gg= \lambda a \to k\,a)$. One can also show that (IO I) fulfils the standard monad laws up to bisimulation.

In principle, an interactive program can issue infinitely many commands. Consider for instance, the program cat which echoes any input through the standard output:

```
cat :  IO ConsoleInterface Unit
cat =  do getLine            λ{ nothing → return unit ; (just line) →
       do (putStrLn line)  λ _ →
       cat                   }
```

In this code snippet, the *pattern matching* λ expression

λ{nothing → return unit; (just *line*) → ⋯}

denotes a function which makes a case distinction on whether the argument is nothing or (just *line*).

The cat program issues the command getLine and terminates when it receives as response nothing, because the end of input has been reached. When it receives (just *line*), it issues the command (putStrLn *line*) and starts over. Potentially, it runs infinitely long, and statically it unfolds into an infinitely deep IO-tree. Thus, we model IO as a coinductive type Setzer (2006):

```
record IO I A : Set where
  constructor delay
  force : IO′ I A

data IO′ I A : Set where
  do′     : (c : Command I) (f : Response I c → IO I A)  → IO′ I A
  return′ : (a : A)                                       → IO′ I A
```

The declaration constructor delay is a just convenience which defines a lazy constructor for IO, behaving like the following function:

```
delay′ : ∀{I A} → IO′ I A → IO I A
force (delay′ x) = x
```

In particular, we cannot match on coinductive constructors (in the same way as we cannot match on defined functions).

With a little force, we define do and return in IO from do′ and return′ in IO' by copattern matching:

```
do               :  ∀{A} (c : C) (f : R c → IO I A) → IO I A
force (do c f)   =  do′ c f

return           :  ∀{A} (a : A) → IO I A
force (return a) =  return′ a
```

The monadic bind operation is definable by corecursion, making IO I a monad for each interface I, in the form of Kleisli triple (IO I, return, _≫=_):

```
_≫=_           :  ∀{A B} (m : IO I A) (k : A → IO I B) → IO I B
force (m ≫= k) with force m
... | do′ c f    =  do′ c λ x → f x ≫= k
... | return′ a  =  force (k a)
```

The recursive call to $(f\,x \gg= k)$ is justified, as one use of force has been consumed in comparison to the left hand side (force $(m \gg= k)$), and there are only applications of the constructor do′ to the right hand side. Therefore, the right hand side requires more applications of force before we need to make a recursive call.

However, the cat program is not strongly normalizing in its present form, since we can unfold its definition recursively arbitrary many times. We need to redefine it using copattern matching, so that at least one application of force is required before having the recursive call. Furthermore, to get it through the termination checker, we need to replace do by do′, so that the termination checkers sees that the recursive call is guarded by constructors:

```
cat : IO ConsoleInterface Unit
force cat =
  do′ getLine              λ{ nothing → return unit ; (just line) → delay (
  do′ (putStrLn line)  λ _  →
  cat                      )}
```

In the latter version, the recursive call to cat in the function body cannot be further rewritten, as only (force cat) reduces. Compare this to the previous version where cat alone already expands, leading to divergence under full reduction.

4.3 Running interactive programs

To run an IO-computation, which unfolds into a potentially infinite command-response tree, we translate it into a NativeIO monad, which executes the commands. From the perspective of Agda, the NativeIO monad is only axiomatically given by nativeReturn and native≫=. If, further, we have a function $tr : (c : C) \to$ NativeIO $(R\, c)$ which translates the commands c of a specific interface C into NativeIO-computations of the appropriate response type $R\, c$, we can apply translateIO recursively.

```
{-# NON_TERMINATING #-}
translateIO : ∀ {A} (tr : (c : C) → NativeIO (R c)) → IO I A → NativeIO A
translateIO tr m = case (force m) of λ
  { (do′ c f)     → (tr c) native≫= λ r → translateIO tr (f r)
  ; (return′ a)  → nativeReturn a
  }
```

This function is properly NON_TERMINATING, as the translated IO-tree might be infinite. However, this will lead to an infinitely running NativeIO-program, which is the intention.

An example program (using an obvious function translateIOConsole translating console commands into native ones) is as follows:

```
main : NativeIO Unit
main = translateIOConsole cat
```

Note that (translateIOConsole cat) is an element of NativeIOUnit and therefore already an executable program. One can think of translateIOConsole as a compiler or an interpreter. To some extent we get something which is in between. If the function *tr* is concretely given, the Haskell compiler can inline it and optimize the resulting instance of translateIO. Therefore, what we get is more than interpretation. However, translateIO cannot optimize the given IO-program; therefore, we get less than compilation.

Programs in Agda are translated into Haskell programs using the MAlonzo compiler (Benke, 2007; Agda Wiki, 2011), which are then compiled into executable code.

Data types and functions for processing native IO in Agda, including the type NativeIO itself, are represented in Agda as postulated types and functions. We use the COMPILED directive of Agda in order to associate corresponding Haskell types and functions with the postulated ones. Especially, NativeIO is associated with the Haskell type IO. MAlonzo will then translate those postulated types and functions into the corresponding Haskell ones. Operations on NativeIO are therefore translated into corresponding Haskell operations on IO.

One can consider translateIO as part of the compilation process. Therefore, programs in Agda can be developed without using NON_TERMINATING programs. The use of NON_TERMINATING appears only as an intermediate step during the compilation process.

5 Objects in Agda

As explained in the introduction, the idea of objects in dependent type theory (Setzer, 2006) is that they are server-side interactive programs: an object waits for method calls, then in response to them, returns an answer and changes its state. Changing the state is represented by returning an object with the modified state. Therefore, the interface of an object is given by a set of methods (parametrized over the method arguments) and a set of responses for each method.[3] In Agda, this is written as

```
record Interface : Set₁ where
    Method    : Set
    Result    : (m : Method) → Set
```

A (simple) object for interface I is a coalgebra that has one eliminator objectMethod. For each method of I, objectMethod returns an element of the response type and the new object after the method invocation:

```
record Object (I : Interface) : Set where
  objectMethod : (m : Method I) → Result I m × Object I
```

An IO object is like a simple object, but the method returns IO applied to the result type of a simple object. In other words, the method returns an IO program for a given IO interface I_{io}, which, if terminating, returns a result of the same type as the corresponding simple object.

```
record IOObject (Iio : IOInterface) (I : Interface) : Set where
  method : (m : Method I) → IO Iio (Result I m × IOObject Iio I)
```

A constructor for an object with interface I and instance variables of type $A_1, \ldots, A_n$ will be defined coinductively as a function

[3] This data structure is also known as the type of *containers* (Abbott *et al.*, 2003).

```
f : A₁ → ··· → Aₙ → Object I
objectMethod (f a₁ ··· aₙ) (m b₁ ··· bₘ) = (result , f a'₁ ··· a'ₙ)
```

where *result* is the value returned by the method invocation $(m\ b_1 \cdots b_m)$, and $a'_1 \cdots a'_n$ are the instance variables of the updated object after the method has been executed. A constructor for an IO object is defined in the same way as for simple objects, except that on the right-hand side, we have an IO program that returns a value upon termination.

An example is a simple cell of elements of type A. It has two methods: get and (put a) depending on $a : A$. Method get is intended to return the content of the cell, and has the return type A, and (put a) sets the cell content to a and returns an element of the one element type Unit, which corresponds to void in Java, meaning that no information is returned. The interface in Java is given in Figure 1.

```
interface Cell<A> {
    void put (A s);
    A    get ();
}
```

Fig. 1. Cell interface in Java

In Agda, the cell interface is coded as follows:

```
data CellMethod A : Set where
  get  :  CellMethod A
  put  :  A → CellMethod A

CellResult             :  ∀{A} → CellMethod A → Set
CellResult  {A} get    =  A
CellResult  (put _)    =  Unit

cellJ                  :  (A : Set) → Interface
Method  (cellJ A)      =  CellMethod A
Result  (cellJ A) m    =  CellResult m
```

The cell class is of type IOObject, which has the previously defined ConsoleInterface as an IOInterface and the interface of a cell, w.r.t. String, as object interface.

```
CellC : Set
CellC = IOObject ConsoleInterface (cellJ String)
```

A basic implementation of the cell interface in Java is displayed in Figure 2; the methods log on standard output what is happening—for the sole purpose of demonstrating the IO interface.

In Agda, simple cell objects are constructed by simpleCell, which implements the methods.

```
class SimpleCell<A> implements Cell<A> {

  A content;

  SimpleCell (A s) { content = s; }

  public void put (A s) {
    System.out.println("putting(" + s + ")");
    content = s;
  }

  public A get () {
    System.out.println("getting(" + content + ")");
    return content;
  }

  public static void program () {
    SimpleCell<String> c = new SimpleCell<String>("Start");
    String s = System.console().readLine();
    if (s == null) return; else {
      c.put(s);
      s = c.get();
      System.out.println(s);
      program();
    }
  }

  public static void main (String[] args) {
    program();
  }
}
```

Fig. 2. Simple cell implementation in Java

```
simpleCell : (s : String) → CellC
force (method (simpleCell s) get) =
  do′ (putStrLn ("getting (" ++ s ++ ")")) λ _ →
  delay (return′ (s , simpleCell s))
force (method (simpleCell s) (put x)) =
  do′ (putStrLn ("putting (" ++ x ++ ")")) λ _ →
  delay (return′ (unit , simpleCell x))
```

A test program using simpleCell is defined as follows in Agda. It is very similar to the original Java program, presenting an almost line-to-line translation. The main difference is that in Agda, we have no mutable state; hence, we rely on continuation-passing style with explicit state threading.

```
{-# TERMINATING #-}

program : IOConsole Unit
force program =
  let c₁ = simpleCell "Start" in
  do′ getLine          λ{ nothing → return unit; (just s) →
  method c₁ (put s)  >>= λ{ (_ , c₂) →
  method c₂ get       >>= λ{ (s′ , c₃) →
  do (putStrLn s′)     λ _ →
  program }}}

main : NativeIO Unit
main = translateIOConsole program
```

The pragma {-# TERMINATING #-} declares program as terminating, overriding the answer from the termination checker. The termination checker says no because on the right hand side of the coinductive definition there is an occurrence of a defined function _>>=_ whereas guarded recursion allows only constructors.

In the next section, we will revisit this example using sized typing, and see that with sized types it passes the termination check.

6 Sized Coinductive Types

In this section, we show how to use sized types to overcome major limitations of the termination checker and enable the user to write modular IO-programs.

Sized types have been used for type-based productivity checking of corecursive programs (Hughes *et al.*, 1996; Barthe *et al.*, 2004; Sacchini, 2013; Abel & Pientka, 2013). Sect. 3 of Igried & Setzer (2016) contains a brief explanation of sized types for coinductive types in Agda. For coinductive types like IO, we should rather speak of *depth* than of size.[4] The depth is how often one can safely apply force, and the depth of a fully defined coinductive object is ∞. However, during the (co)recursive definition of an object, we want to speak of depths less than infinity, to verify that on the way to the recursive calls, the depth has increased by at least one. This ensures that the depth grows for each unfolding of recursion, becoming ∞ in the limit.

First, let us define a sized version of the generic weakly terminal coalgebra νF from Sect. 3. We can then fulfill our promise and justify the generic coiteration operation unfoldF from the monotonicity witness mapF.

[4] The term *Size* is more suitable for inductive types which are inhabited by trees. There, the size tracked in the type is an upper bound on the height of the tree. In recursive calls, the height should go down, guaranteeing termination. Note that for infinitely branching trees, the height might be transfinite, so semantically sizes correspond to ordinals rather than to natural numbers. For the use with coinductive types, sizes up to ω suffice, which we call ∞ in our syntax.

```
record νF (i : Size) : Set where
  constructor delay
  force : ∀(j : Size< i) → F (νF j)
```

The quantification over j : Size< i is reminiscent of the approximation of the greatest fixpoint by deflationary iteration (Sprenger & Dam, 2003; Abel, 2012; Abel & Pientka, 2013). The approximant $\nu^i F$ is defined by induction on ordinal i as follows:

$$\nu^i F = \bigcap_{j<i} F\,(\nu^j F)$$

The fact that $\nu^i F$ is monotonically decreasing in i follows directly from the use of intersection $\bigcap_{j<i}$ and is independent of the monotonicity of F. Agda still asks for strict positivity of F, which anyway holds for the common coinductive types. Furthermore, the monotonicity of F gives us the isomorphisms $\nu^{i+1} F = F\,(\nu^i F)$ and $\nu^\infty F = F\,(\nu^\infty F)$, mediated by force and delay.[5]

The coiterator unfoldF can likewise be defined by induction on i, again by copattern matching:

```
unfoldF : ∀{S} (t : S → F S) → ∀ i → (S → νF i)
force (unfoldF t i s) j = mapF (unfoldF t j) (t s)
```

The type of force guarantees $j < i$, thus, the recursive call (unfoldF $t\ j$) is justified. It gives us a function of type $S \to$ νF j which we map over the application $(t\,s)$: $F\ S$ to obtain the right hand side of the required type (F (νF j)). Note how the type of mapF ensures that a result having the required depth j is returned. In particular, mapF cannot involve uses of force, which would necessarily tamper with the depth annotation of νF.

In the sized version of IO, applying force to an IO-tree yields a function that expects a size $j < i$ and then yields an IO′-node, which can be either a do′ or a return′. The latter is a leaf, and the former a node consisting of a command c and a (Response c)-indexed collection f of subtrees of that depth.

```
record IO (I_io : IOInterface) (i : Size)  (A : Set) : Set where
  constructor delay
  force : {j : Size< i} → IO′ I_io j A

data IO′ (I_io : IOInterface) (i : Size)  (A : Set) : Set where
  do′      :  (c : Command I_io) (f : Response I_io c → IO I_io i A)  → IO′ I_io i A
  return′  :  (a : A)                                               → IO′ I_io i A
```

Again, this sized coinductive type is justified by deflationary iteration $\nu^i F = \bigcap_{j<i} F\,(\nu^j F)$. In this case, the ith approximant $\nu^i F$ of the greatest fixed point of F would be (IO $I_{io}\ i\,A$) for

[5] force (delay t) $= t$ holds definitionally in Agda; delay (force t) $\cong t$ holds up to bisimilarity.

some fixed I_{io} and A. The type transformation F would be IO′ in dependence of (IO I_{io} i A). We can make the correspondence obvious by using nested recursion instead of mutual recursion:

```
data F (X : Set) : Set where
  do′     : (c : Command Iio) (f : Response Iio c → X)  → F X
  return′ : (a : A)                                      → F X

record νF (i : Size) : Set where
  constructor delay
  force : {j : Size< i} → F (νF j)
```

Sizes in types allow us to track the guardedness level of expressions independent of their exact formulation. In particular, we can express the guardedness level of a function applied to arguments in terms of the guardedness level of the arguments, rather than having to assume that the function application is unguarded. With the same implementations as for the unsized versions, we obtain the following sized typings for do, return, and _≫=_.

```
do     : ∀ {i A}   (c : C) (f : R c → IO Iio i A) → IO Iio i A
return : ∀ {i A}   (a : A) → IO Iio i A
_≫=_   : ∀ {i A B} (m : IO Iio i A) (k : A → IO Iio i B) → IO Iio i B
```

The typings of do and _≫=_ express that these functions are *guardedness-preserving*, meaning that the output is (at least) as guarded as the least guarded input. The type of return simply expresses that we can assume any guardedness for (return a). With subtyping, an equivalent type would be $\forall\{i\,A\}(a : A) \to$ IO I_{io} ∞ A, using the covariance IO I_{io} ∞ $A \leq$ IO I_{io} i A of the coinductive type IO in its size argument $i \leq \infty$.

To understand why the above typing is valid for _≫=_, we cast another glance at its implementation. We have made the sizes explicit to see what is going on; however, they can be inserted by Agda automatically. Unfortunately, to supply hidden arguments to an infix operator like _≫=_, we have to fall back to prefix notation:

```
force (_≫=_ {i} m k) {j} with force m {j}
... | do′ c f    = do′ c λ r → _≫=_ {j} (f r) k
... | return′ a  = force (k a) {j}
```

The call _≫=_ $\{i\}$ m k constitutes an IO-tree of depth i which is defined by the effect of its only elimination form force. Assuming we force it, obtaining a size $j < i$, we are obliged to produce an IO′-node of size j. We do this by forcing the first given tree, m, of depth i, by virtue of our size $j < i$.[6] We proceed by case-distinction on the resulting IO′-node. If

[6] At this point, it is important to note that if we had no size $j < i$, we could not force it, or at least not discriminate on the results of forcing it. In particular, if $i = 0$ then there is no size $< i$. However, when we have successfully forced m≫=k, meaning that the latter actually evaluated to a delayed node, we know its depth is not 0, and thus there exists a size $j < i$.

it is (do′ c f), we execute command c and, after binding the response to r, continue with a recursive call $(f\,r) \gg\!= k$ at depth j, which is strictly smaller than the depth we started with. Thus, the recursive call is justified. If it is (return′ a), we continue with IO-tree $(k\,a)$ of size i, which we have to force to produce the desired IO′-node of size j.

For an IOObject, the notion of depth is how often we can apply one of its methods. The result of applying a method to an IOObject of depth i is an unbounded IO-tree (depth ∞). Its leaves contain the result of the method call and an IOObject of depth $j < i$ resembling the new state of this object after the method call (and the IO-actions).

```
record IOObject (I_io : IOInterface) (I : Interface) (i : Size) : Set where
  method  :  ∀{j : Size< i} (m : Method I)
                 → IO I_io ∞ (Result I m × IOObject I_io I j)
```

Sized types already allow us to write the simpleCell constructor slightly more elegantly, using the defined return instead of the combination of delay and return′. Putting the recursive call to simpleCell under function return is possible due to the polymorphic typing of return : $A \to$ IO I_{io} ∞ A which we use with type $A =$ String × CellC j.

```
CellC : (i : Size) → Set
CellC = IOObject ConsoleInterface (cellJ String)

simpleCell : ∀{i} (s : String) → CellC i
force (method (simpleCell {i} s) {j} get) =
  do′ (putStrLn ("getting (" ++ s ++ ")")) λ _ →
  return (s , simpleCell {j} s)
force (method (simpleCell _) (put s)) =
  do′ (putStrLn ("putting (" ++ s ++ ")")) λ _ →
  return (unit , simpleCell s)
```

The program using simpleCell from Sect. 5 now passes the termination check without modification to its definition. Only its type needs to be refined to exhibit the depth i that will grow with each unfolding of the recursion.

```
program : ∀{i} → IO ConsoleInterface i Unit
force program =
  let c₁ = simpleCell "Start" in
  do′ getLine          λ{ nothing → return unit; (just s) →
  method c₁ (put s)  ≫= λ{ (_ , c₂) →
  method c₂ get        ≫= λ{ (s′ , c₃) →
  do (putStrLn s′)     λ _ →
  program }}}
```

Both do and _≫=_ preserve the guardedness of the recursive call to program, and this is communicated through the type system.

7 Interface Extension and Delegation

So far we have shown the implementation in Agda for a single object. In this section we will show the facets of having several objects and we implement reuse mechanisms based on delegation.

For the purpose of illustration, we introduce the CounterCell class. It extends the functionality of a SimpleCell and counts the number of times an element is stored and retrieved. Further, it includes a method stats to print these statistics. Figure 3 depicts the extended interface in Java.

```java
interface StatsCell<A> extends Cell<A> {
    void stats();
}
```

Fig. 3. StatsCell interface in Java

In Agda, CounterMethod extends the method definition of a SimpleCell, where super lifts a CellMethod of SimpleCell; further, a constructor for the stats method is added:

```
data CounterMethod A : Set where
  super  :  (m : CellMethod A) → CounterMethod A
  stats  :  CounterMethod A
```

Instead of embedding the superclass interface CellMethod into CounterMethod, we could have reused get and put as constructors for CounterMethod, as Agda supports constructor overloading. However, embedding with super gives us benefits later. We can still get nice names for the inherited methods (^{c} indicates a CellMethod) by using Agda's pattern synonym facility:

```
pattern getᶜ     = super get
pattern putᶜ x   = super (put x)
```

The full object Interface in Agda is given by statsCellI, in which the result types for put and get refer to SimpleCell and Unit represents `void`.

```
statsCellI : (A : Set) → Interface
Method  (statsCellI A)             =  CounterMethod A
Result  (statsCellI A) (super m)   =  Result (cellJ A) m
Result  (statsCellI A) stats       =  Unit
```

Figure 4 shows the CounterCell in Java that is equivalent to our implementation in Agda. Notably, we restrict the implementation to delegation as reuse mechanism as we cannot fully express the subtype relationship between CounterCell and SimpleCell. In particular, the Agda code explicitly states that put and get are defined in the interface of SimpleCell; if we moved the methods to a super class of SimpleCell, we have to adapt our code. In Java, we can override any method without the need to specify in which particular superclass the method is defined in.

```
class CounterCell<A> implements StatsCell<A> {

  Cell<A> cell;
  int ngets, nputs;

  CounterCell (Cell<A> c, int g, int p) {
    cell  = c;
    ngets = g;
    nputs = p;
  }

  public A get() {
    ngets++;
    return cell.get();
  }

  public void put (A s) {
    nputs++;
    cell.put(s);
  }

  public void stats() {
    System.out.println ("Counted "
      + ngets + " calls to get and "
      + nputs + " calls to put.");
  }

  public static void program (String arg) {
    CounterCell<String> c = new CounterCell(new SimpleCell("Start"), 0, 0);
    String s = c.get();
    System.out.println(s);
    c.put(arg);
    s = c.get();
    System.out.println(s);
    c.put("Over!");
    c.stats();
    return;
  }

  public static void main (String[] args) {
    program ("Hello");
  }
}
```

Fig. 4. CounterCell implementation in Java

In Agda, the class CounterC is defined as a console object

```
CounterC : (i : Size) → Set
CounterC = IOObject ConsoleInterface (statsCellI String)
```

The constructor counterCell specifies the functionality of the CounterCell class. The local state includes an object of class CounterC (the class of a SimpleCell) and two natural numbers for the get and put statistics. Each method may issue IO commands or call methods of other objects; get^c and put^c delegate to the respective methods in SimpleCell and return an object with the increased counter variable, whereas stats issues printing of the statistics as IO command.

```
counterCell : ∀{i} (c : CellC i) (ngets nputs : ℕ) → CounterC i

method (counterCell c ngets nputs) getᶜ =
  method c get                              ≫= λ { (s , c′) →
  return (s , counterCell c′ (1 + ngets) nputs) }

method (counterCell c ngets nputs) (putᶜ x) =
  method c (put x)                          ≫= λ { (_ , c′) →
  return (unit , counterCell c′ ngets (1 + nputs)) }

method (counterCell c ngets nputs) stats =
  do (putStrLn ("Counted "
    ++ show ngets ++ " calls to get and "
    ++ show nputs ++ " calls to put.")) λ _ →
  return (unit , counterCell c ngets nputs)
```

Finally, the test program is a one-to-one translation from the Java original. This time, it is not recursive, so we do not have to worry about termination.

```
program : String → IO ConsoleInterface ∞ Unit
program arg =
  let c₀ = counterCell (simpleCell "Start") 0 0 in
  method c₀ getᶜ              ≫= λ{ (s  , c₁) →
  do (putStrLn s)             λ _ →
  method c₁ (putᶜ arg)        ≫= λ{ (_  , c₂) →
  method c₂ getᶜ              ≫= λ{ (s′ , c₃) →
  do (putStrLn s′)            λ _ →
  method c₃ (putᶜ "Over!")    ≫= λ{ (_ , c₄) →
  method c₄ stats             ≫= λ{ (_ , c₅) →
  return unit                 }}}}}

main : NativeIO Unit
main = translateIO translateIOConsoleLocal (program "Hello")
```

8 State-Dependent Objects and IO

8.1 State-Dependent Interfaces

We motivate stateful object interfaces with the implementation of a stack. A stack has two operations: push places an object on the stack, and pop removes the top object and returns it. Consider a stack interface in Java:

```
public interface  Stack<E> {
  void push(E e);

  /** @throws EmptyStackException if the stack is empty **/
  E pop() throws java.util.EmptyStackException;
}
```

A stack underflow happens when the pop method is called on an empty stack. In the Java Development Kit (JDK), the pop method throws a runtime exception, which the programmer is advised but not forced to catch.

In Agda, a safer version of a stack class can be defined, where the type system ensures that a pop operation may only be performed on a non-empty stack. The interface depends on the state of the stack, i.e., the number of elements that are on the stack:

```
StackStateˢ = ℕ
```

A state-dependent object interface in Agda[7] is given by a value of the following record type (superscript $^{\mathsf{s}}$ indicates the state-dependency of the interface).

```
record Interfaceˢ : Set₁ where
    Stateˢ   : Set
    Methodˢ  : (s : Stateˢ) → Set
    Resultˢ  : (s : Stateˢ) → (m : Methodˢ s) → Set
    nextˢ    : (s : Stateˢ) → (m : Methodˢ s) → (r : Resultˢ s m) → Stateˢ
```

The set of methods depends on the state of the object, while the result depends on the state and the invoked method. The nextˢ function determines the successive state after the result of the method invocation has been computed.

To model state-dependent methods, StackMethodˢ needs to be indexed by the size of the stack. The pop method is only available when the size is non-zero, i.e., of the form suc n (*successor* of some natural number n). In Agda, this is realized by an indexed data type, aka *inductive family* (Dybjer, 1994).

```
data StackMethodˢ (A : Set) : (n : StackStateˢ) → Set where
  push  :  ∀ {n}  →  A  →  StackMethodˢ A n
  pop   :  ∀ {n}         →  StackMethodˢ A (suc n)
```

[7] Also known as indexed container (Altenkirch & Morris, 2009).

Pushing to a stack has no return value (Unit), while the result of popping from a stack of As is an element of type A.

```
StackResultˢ  :  (A : Set) → (s : StackStateˢ) → StackMethodˢ A s → Set
StackResultˢ A _  (push _)  =  Unit
StackResultˢ A _  pop       =  A
```

The next state after a push operation is a stack with an increased size (i.e., state (suc n)), while pop leads to a decreased size.

```
stackNextˢ  :  ∀ A n (m : StackMethodˢ A n) (r : StackResultˢ A n m) →  StackStateˢ
stackNextˢ  _  n        (push _)  _  =  suc n
stackNextˢ  _  (suc n)  pop       _  =  n
```

The previous definitions allow us to assemble the state dependent interface for stack objects:

```
StackInterfaceˢ : (A : Set)  →  Interfaceˢ
Stateˢ      (StackInterfaceˢ A)  =  StackStateˢ
Methodˢ     (StackInterfaceˢ A)  =  StackMethodˢ A
Resultˢ     (StackInterfaceˢ A)  =  StackResultˢ A
nextˢ       (StackInterfaceˢ A)  =  stackNextˢ A
```

8.2 State-Dependent Objects

An object for interface I is a coalgebra which has one eliminator objectMethod which for each method of I returns an element of the response type and the adapted object:

```
record Objectˢ (I : Interfaceˢ) (s : Stateˢ I) : Set where
  objectMethod :  (m : Methodˢ I s) →
                  Σ[ r ∈ Resultˢ I s m ] Objectˢ I (nextˢ I s m r)
```

Since the type of the returned object depends, via nextˢ, on the returned result r, we need to type the returned pair via a Σ-type, defined in Agda's standard library. Here Σ[x ∈ A] B denotes Σ A (λx → B), where Σ A C is the dependent product type defined as a record with fields proj_1 : A and proj_2 : C proj_1.

Note Objectˢ is given as a ΠΣ-type rather than ΣΠ. This is a very general form of polynomial functors as known from the theory of Petersson-Synek trees (Petersson & Synek, 1989) and indexed containers (Hancock *et al.*, 2013) (see also Setzer (2016)).

The state-dependent version of an IO object is:

```
record IOObjectˢ (I_io : IOInterface) (I : Interfaceˢ) (s : Stateˢ I) : Set where
  method :  (m : Methodˢ I s) →
                IO I_io ∞ (Σ[ r ∈ Resultˢ I s m ] IOObjectˢ I_io I (nextˢ I s m r))
```

IOObjectˢ is a straightforward adaption to state-dependency of IOObject from Sect. 5. The IO version of a stack object could additionally log its activity on an output channel. However, in the following we restrict it to the non-IO version for clarity of exposition.

The simplest implementation of a stack object is just a wrapper of its data, a stack implemented as a vector (Vec A n) of elements in type A, where n is the current stack size.

```
stack  :  ∀{A}{n : ℕ} (as : Vec A n) →  Objectˢ (StackInterfaceˢ A) n
objectMethod (stack as)          (push a)  =  unit  ,  stack (a :: as)
objectMethod (stack (a :: as))  pop        =  a       ,  stack as
```

In the case of method pop, in principle, we have two cases for the content on the stack: First, the stack is non-empty, i.e., of the form a :: as where a is the top element and as is the rest. This case is handled by the second clause. The other case, the stack being empty, i.e., of the form [], is ruled out by the dependent typing: method pop expects n to be a successor, but [] : Vec A 0 enforces $n = 0$. This allows Agda to conclude that this case is impossible, and no clause has to be written for it. Especially no exception handling is needed.

Objects that may flexibly depend on runtime values may not only be suitable for ensuring runtime invariants, but may also help model extensions of object-oriented programming; for instance, method dispatch may depend on another dimension. Consider *context-oriented programming* (Hirschfeld *et al.*, 2008), where the behavior of an object depends on a given execution context that can be activated dynamically at runtime.

8.2.1 Example of Use of Safe Stack

We consider an example of the use of safe stacks, where type theoretic rules prevent the use of pop when it is empty. A stack machine for evaluating the Fibonacci numbers iteratively using a safe stack serves as an illustration. This is essentially the result of computing the recursive definition of the Fibonacci numbers (which is of course inefficient) using a stack. This definition can easily be generalised to other recursive functions. The presented example is an adaption of Sect. 5.1.4. of Abelson *et al.* (1996).

The stack machine consists of a state, a number $n : \mathbb{N}$, and a stack of size n. The state is either an expression (fib m) to be evaluated or a value (val k) to be returned. The elements of the stack are expressions with a hole •, into which k is to be inserted, once the stack above it has been emptied, and the state has become a value (val k). These elements are of the form (•+fib m), which means that (fib m) has to be added to the result k, or k' +•, which means that the result k has to be added to k'.

```
data FibState : Set where
  fib  : ℕ → FibState
  val  : ℕ → FibState

data FibStackEl : Set where
  _+•     : ℕ → FibStackEl
  •+fib_  : ℕ → FibStackEl

FibStack : ℕ → Set
FibStack = Objectˢ (StackInterfaceˢ FibStackEl)

FibStackmachine : Set
FibStackmachine = Σ[ n ∈ ℕ ] (FibState × FibStack n)
```

The function reduce carries out a one-step reduction, returning either a new stack machine or the value computed, i.e. an element of the disjoint union of the two sets. If the state is (val k), then this expression is used to reduce the top element on the stack. If the state is (fib m), then the machine is supposed to compute (fib m). In case of $m = m' + 2$ we have (fib $(m' + 1)$) as the next state, which when evaluated will be inserted into the whole of the element (•+fib m') pushed onto the stack, computing fib $(m' + 1) +$ fib m'. Note that we never pop from the stack when it is empty.

```
reduce : FibStackmachine →  FibStackmachine ⊎ ℕ
reduce (n      , fib 0 ,                 stack)  =  inj₁ (n  ,  val 1  ,  stack)
reduce (n      , fib 1 ,                 stack)  =  inj₁ (n  ,  val 1  ,  stack)
reduce (n      , fib (suc (suc m)) ,  stack)  =
          objectMethod stack (push (•+fib m))  ▷ λ { (_ , stack₁) →
          inj₁ ( suc n , fib (suc m) , stack₁)           }
reduce (0      , val k ,                 stack)  =  inj₂ k
reduce (suc n  , val k ,                 stack)  =
          objectMethod stack pop                        ▷ λ { (k′ +•      , stack₁) →
          inj₁ (n , val (k′ + k) , stack₁)

                                                              ; (•+fib m  , stack₁) →
          objectMethod stack₁ (push (k +•))        ▷ λ { (_ , stack₂) →
          inj₁ (suc n , fib m , stack₂)                   }}
```

In the above, we used the function _▷_ which can be used to make a method call and—depending on the result—continue with a next operation. We also used anonymous pattern matching in λ-expressions, where the cases are separated by ";".

```
_▷_   :   ∀{A B : Set} → A → (A → B) → B
a ▷ f  =  f a
```

The function computeFibRec applies iteratively reduce until it returns a result. We know it is TERMINATING. But, since we did not calculate how often we should iterate this operation, we have to override the termination checker.

```
{-# TERMINATING #-}
computeFibRec : FibStackmachine → ℕ
computeFibRec s with reduce s
... | inj₁ s′  = computeFibRec s′
... | inj₂ k   = k
```

fibUsingStack computes the Fibonacci function:

```
fibUsingStack : ℕ → ℕ
fibUsingStack m = computeFibRec (0 , fib m , stack [])
```

8.3 Reasoning About Stateful Objects

8.3.1 Bisimilarity

Henceforth, we assume an arbitrary I : Interface^{s} and use O for the type of objects of this interface. Assume

```
I  = record { Stateˢ = S; Methodˢ = M; Resultˢ = R; nextˢ = next }
O = Objectˢ I
```

In Agda the equality used in type checking is definitional equality, which is a decidable equality based on equality of normal forms up to α,η-equality. It is not extensional: for instance, functions are equal if they have the same normal form, not if they return equal values for equal arguments. The standard generic propositional equality in Agda is Martin-Löf's intensional equality type. One can define extensional propositional equality types, but the preservation of such equalities by functions needs to be proved for each instance needed.

The natural extensional equality on coalgebras is bisimilarity, which means that two elements of coalgebras are *bisimilar*, if all eliminators return equal results for equal arguments. Since the result type of an eliminator might refer to the coalgebra, this results in a recursive definition. Because the coalgebra is defined coinductively, it is natural to define the bisimilarity coinductively as well. Essentially, this means that two elements of a coalgebra are bisimilar, if, after repeatedly applying eliminators until one obtains an element of a type which was defined before the coalgebra was introduced, one obtains equal results. Adapted to objects, this means that two objects are *bisimilar* if they yield the same responses if subjected to the same method calls, which is a recursive definition to be understood coinductively.

To express bisimilarity in Agda, let us first define a relation ΣR R on dependent pairs $(a,b),(a',b') \in \Sigma\, A\, B$ that holds iff the first components $a, a' \in A$ are identical and the second components $b, b' \in B\, a$ are related by $R\, a : (b\, b' : B\, a) \to$ Set.

```
data ΣR  {A : Set} {B : A → Set} (R : ∀{a} (b b′ : B a) → Set)
         : (p p′ : Σ[ a ∈ A ] B a) → Set
   where
   eqΣ  :  ∀{a}{b b′ : B a} → R b b′ → ΣR R (a , b) (a , b′)
```

We can establish ΣR $R\ (a,b)\ (a,b')$ using constructor eqΣ, provided we have a proof of $(R\, b\, b')$. This enables us to define the bisimilarity relation coinductively in a very similar way to how we have defined objects.

```
record _≅_ {s : S} (o o′ : O s) : Set where
   bisimMethod  :  (m : M s) →
                      ΣR (_≅_) (objectMethod o m) (objectMethod o′ m)
```

A bisimilarity derivation $o \cong o'$ for two objects $o, o' \in O\, s$ at the same state s is an infinite proof tree which we can, by bisimMethod, query for its node sitting on branch $m : M\ s$ for a valid method call m. This node will consist of an eqΣ constructor certifying the identity of responses and holding a subtree for the equality of the objects after the method invocation.

Reflexivity of bisimilarity is shown corecursively; the proof, as the statement, is rather trivial.

```
refl≅ :  ∀{s} (o : O s) → o ≅ o
bisimMethod (refl≅ o) m  =  let  (r , o′) =  objectMethod o m
                                 in   eqΣ (refl≅ o′)
```

To show that o is bisimilar to itself, we subject it to an arbitrary method call m. Trivially, there is only one result r, which is equal to itself. By the coinduction hypothesis, the new object o' is bisimilar to itself, thus, eqΣ is sufficient to establish bisimilarity.

8.3.2 Verifying stack laws

In this section, we show that two stack laws hold for our implementation of a stack by a vector. Both hold in Agda by computation, so reflexivity of bisimilarity is sufficient to prove them.

The first law states that for an arbitrary stack st constructed from a vector v of elements of type E, if we first push an arbitrary element e and then pop from the stack, we get back e and the original stack.

```
pop-after-push : ∀{n} {v : Vec E n} {e : E} →
  let st          = stack v
      (_ , st₁)   = objectMethod st  (push e)
      (e₂ , st₂)  = objectMethod st₁ pop

  in  (e ≡ e₂) × (st ≅ st₂)

pop-after-push = refl , refl≅ _
```

In Agda, the proof is trivial by expansion of the definition of our stack implementation: First, st_1 computes to stack $(e :: v)$, then the pair $(e_2,\ st_2)$ computes to $(e,$ stack $v)$, and both goals hold by reflexivity.

The second law concerns the opposite order of these operations. If we first pop and element from stack *st* constructed from the non-empty vector $e :: v$, and then push the popped element, we end up with the same stack *st*.

```
push-after-pop : ∀{n} {v : Vec E n} {e : E} →
  let st         = stack (e :: v)
      (e₁ , st₁) = objectMethod st  pop
      (_  , st₂) = objectMethod st₁ (push e₁)

  in  st ≅ st₂

push-after-pop = refl≅ _
```

Again, this lemma is proven by computation.

8.3.3 *Bisimilarity of different stack implementations*

Alternatively to a vector, we can store the stack contents in a finite map implemented naively as a pair of a number $n : \mathbb{N}$ which denotes the stack size, and a function $f : \mathbb{N} \to E$ which gives direct access to the stack elements, with $f\,0$ standing for the top element and $f\,(n-1)$ for the bottom element. The value of $f\,k$ for $k \geq n$ is irrelevant. We can transform such a finite map into a vector through the function (tabulate $n\ f$), which computes the vector $f\,0 :: f\,1 :: \cdots :: f\,(n-1) :: []$ and will be used to relate the finite maps and vectors later.

```
tabulate : ∀ (n : ℕ) (f : ℕ → E) → Vec E n
tabulate 0       f = []
tabulate (suc n) f = f 0 :: tabulate n λ m → f (suc m)
```

The object stackF $n\ f$ implements a stack represented by the finite map (n, f). Pushing a new element e onto the stack will result in increasing the stack size to (suc n) and changing the function f to a new function f' such that top position 0 maps to the new element e and position $m+1$ maps to $(f\ m)$. Basically, we have shifted the old stack content to make space for the new element e in position 0.

```
stackF : ∀ (n : ℕ) (f : ℕ → E) → Objectˢ (StackInterfaceˢ E) n
objectMethod  (stackF  n        f) (push e)  =  _   ,  stackF (suc n) λ
   { 0             → e
   ; (suc m)       → f m }
objectMethod  (stackF  (suc n)  f) pop       =  f 0  ,  stackF n (f  ∘ suc)
```

Popping from the stack returns the top element $f\ 0$ and changes the stack size from (suc n) to n and the representing function from f to $f \circ$ suc.

Given a finite map (n, f) which tabulates to a vector v, we obtain bisimilar stack objects (stackF $n\ f$) and (stack v). After we push a new element e we can invoke the coinduction hypotheses on the new stack objects provided that their data is still in correspondence, (tabulate (suc n) $f' \equiv (e :: v)$). By definition of tabulate the heads of these vectors are both e, and the equality of their tails is the assumption p.

```
impl-bisim : ∀{n f} v (p : tabulate n f ≡ v) → stackF n f ≅ stack v

bisimMethod (impl-bisim v          p) (push e) =
   eqΣ (impl-bisim (e :: v)  (cong (_::_ e)  p))

bisimMethod (impl-bisim (e :: v)  p) pop rewrite cong head p =
   eqΣ (impl-bisim v          (cong tail         p))
```

Here (cong head p) has type $f\ 0 \equiv e$, where f is the hidden argument. The syntax rewrite cong head p makes an implicit case distinction on (cong head p), which in this example equates $f\ 0$ with e.

When popping from a non-empty stack whose vector representation is $e :: v$, we first have to show the equality of the result component, $f\ 0 \equiv e$. This equation is obtained from p : tabulate (suc n) $f \equiv (e :: v)$ by applying head on both sides. After rewriting with this equation, we can proceed with eqΣ and apply the coinduction hypothesis with tabulate $n\ (f \circ$ suc$) \equiv v$, which we get from p by applying tail on both sides.

8.4 State-Dependent IO

State-dependent interactive programs are defined in a similar way as state-dependent objects, except for replacing Methodˢ by Commandˢ and Resultˢ by Responseˢ. Later, we will use state dependent IO in a situation where the components have different type levels (Set vs Set$_1$). We, therefore, define the operations polymorphically in the finite type levels σ, γ, ρ, which from now we consider fixed but arbitrary. Levels have the lowest element lzero, successor operation lsuc and the maximum operation $\sqcup$. [8]

[8] Note that Set α has type Set (lsuc α) and (α : Level) → Set α has type Setω, which is a universe above any Set α. Universe Setω only exists internally in Agda as the type of level-polymorphic

```
record IOInterfaceˢ : Set (lsuc (σ ⊔ γ ⊔ ρ )) where
    Stateˢ      : Set σ
    Commandˢ    : Stateˢ → Set γ
    Responseˢ   : (s : Stateˢ) → Commandˢ s → Set ρ
    nextˢ       : (s : Stateˢ) → (c : Commandˢ s) → Responseˢ s c → Stateˢ
```

State-dependent IO programs are defined in a similar way as state-dependent Objects:

```
record IOˢ (i : Size)  (A : S → Set α) (s : S) : Set (lsuc (α ⊔ σ ⊔ γ ⊔ ρ )) where
  constructor delay
  forceˢ : {j : Size< i} → IOˢ′ j A s

data IOˢ′ (i : Size)  (A : S → Set α) : S → Set (lsuc (α ⊔ σ ⊔ γ ⊔ ρ )) where
  doˢ′      : {s : S} → (c : C s) → (f : (r : R s c) → IOˢ i A (next s c r) )
              → IOˢ′ i A s
  returnˢ′  : {s : S} → (a : A s) → IOˢ′ i A s


returnˢ : ∀{i}{A : S → Set α} {s : S} (a : A s) → IOˢ  I i A s
forceˢ (returnˢ a) = returnˢ′ a

doˢ  :  ∀{i}{A : S → Set α} {s : S}
        (c : C s) (f : (r : R s c) → IOˢ I i A (next s c r)) → IOˢ I i A s
forceˢ (doˢ c f) = doˢ′ c f
```

Translation into NativeIO is as before, however the type level ρ of Responseˢ has to be lzero. Furthermore, the result type needs to be independent of s and in Set:

```
translateIOˢ : ∀{A : Set }{s : S}
  →  (translateLocal : (s : S) → (c : C s) → NativeIO (R s c))
  →  IOˢ I ∞ (λ s → A) s
  →  NativeIO A
```

9 A Drawing Program in Agda

In this section we will introduce a graphics library in Agda and implement a proof-of-concept drawing program with it. The library is using Hudak's SOE Haskell library (Hudak, 2016). The GUI interface in the Agda library has commands for creating, changing, and closing GUI components, and for checking for GUI events such as key pressed (together with the character pressed) or mouse move event (together with the new

universes, it cannot be written by the user. It is needed in type theoretic rules which require to associate a type for any A occurring in a typing judgement $a : A$.

point). The library will refer to data types representing GUI-related data such as a type Window of windows. All these commands and types will be translated in Haskell functions and types using the SOE library.

```
data GraphicsCommands  :  Set where
  getWindowEvent  :  Window  →  GraphicsCommands
  openWindow      :  String  →  Maybe Point  →  ℕ  → ℕ
                             →  RedrawMode   →  Maybe Word32
                             →  GraphicsCommands
  closeWindow     :  Window  →  GraphicsCommands
  drawInWindow    :  Window  →  Graphic → GraphicsCommands


GraphicsResponses  :  GraphicsCommands → Set
GraphicsResponses  (getWindowEvent _)              =  Event
GraphicsResponses  (openWindow _ _ _ _ _ _)  =  Window
GraphicsResponses  (closeWindow _)                 =  Unit
GraphicsResponses  (drawInWindow _ _)             =  Unit

GraphicsInterface                      :   IOInterface
Command   GraphicsInterface   =   GraphicsCommands
Response  GraphicsInterface   =   GraphicsResponses

IOGraphics : Size → Set → Set
IOGraphics i = IO GraphicsInterface i
```

Graphics commands are translated into native IO commands by a function translateNative, which replaces each command by a native function.

```
translateNative : (c : GraphicsCommands) → NativeIO (GraphicsResponses c)
```

We define now a simple drawing program, which opens a window and draws a trace of where the mouse moves. After having started, the state of the program is given by the last point up to which the drawing has already been carried out. After the first mouse movement event at point p, the drawing consists of a single point at position p. Initially there is no such point. So we define the state as

```
State = Maybe Point
```

The loop of the program checks for any window event, which are handled by winEvtHandler. If key `'x'`, representing a request to terminate the program, was pressed, the program closes the window and terminates. If, after we have started (state (just p_1)), a mouse movement event with point p_2 occurs, a line is drawn from p_1 to p_2, and the state is updated to (just p_2). If the same mouse movement event occurs in the initial state nothing, no line is drawn, but as before the state is updated to (just p_2). In all other cases, winEvtHandler calls the loop function without changing the state.

```
mutual

  loop           :   ∀{i} → Window → State → IOGraphics i Unit
  force (loop w s)  =  do'  (getWindowEvent w) λ e →
                        winEvtHandler w s e

  winEvtHandler : ∀{i} → Window → State → Event → IOGraphics i Unit
  winEvtHandler w s (Key c t)              = if charEquality c 'x' then (do (closeWindow w) return)
                                              else loop w s
  winEvtHandler w s (MouseMove p2)  =  case s of
    λ{ nothing     → loop w (just p2)
     ; (just p1)  → do (drawInWindow w (line p1 p2)) λ _ →
                     loop w (just p2)  }
  winEvtHandler w s _                      = loop w s
```

The main program opens a window and then runs the loop. This program is then translated into a native IO program:

```
program : ∀{i} → IOGraphics i Unit
program =
  do (openWindow "Drawing Prog" nothing 1000 1000 nativeDrawGraphic nothing) λ win →
  loop win nothing

translateIOGraphics : IOGraphics ∞ Unit → NativeIO Unit
translateIOGraphics = translateIO translateNative

main : NativeIO Unit
main = nativeRunGraphics (translateIOGraphics program)
```

10 A Graphical User Interface using an Object

10.1 Graphical User Interfaces with Event Handlers

So far our interactive programs were client-side programs: the program issues commands and receives responses. In the drawing program in section 9 we ran a loop which was checking for any events that had occurred and modified the program state accordingly. When dealing with more complex graphical user interfaces this becomes inefficient. A better way is to use event listeners.

In object-oriented programming languages such as Java, when creating a graphical user interface, one uses commands which create GUI elements such as frames, buttons and text fields, and commands for placing them usually within previously created GUI elements. For instance, one can place a button within a frame.

These GUI elements are associated with events, which are usually triggered by user interaction. For example, once we have created a button, a button click event is created, which is activated whenever the button is pressed. Moreover, there are events triggered by the user interface itself, such as the paint event that signals that a window's contents needs to be repainted. Events are handled by event listeners or event handlers. An event handler is an interactive program that is executed whenever the event is triggered and is provided with parameters accordingly. For instance, when a mouse click event is triggered, one obtains the coordinates of the location of the mouse click. An event handler also has other parameters which are implicit, such as the device context. When the event is triggered, the event handler is applied to these arguments.

In object-oriented programming such as Java, the event handlers are usually invoked as methods of several objects. This allows communication between the event handlers. In 10.2, we will introduce an example of a spaceship controlled by a button. The coordinates of the spaceship are changed when a button is pressed. The paint event handler then uses these coordinates to draw the spaceship at a different location.

10.2 wxHaskell

In this section we will translate our IO programs into native IO programs, which then translate into Haskell programs by making use of the Haskell library wxHaskell (Leijen, 2004; wiki.haskell, 2016). This library is suitable for creating GUIs since it has good support for server-side programs based on action handlers. Here "server-side" refers to the notion put forth by Hancock & Setzer (2005); Setzer & Hancock (2004). The wxHaskell library offers bindings for wxWidgets, and an object-oriented (C++) widget toolkit to build GUIs. For each method in C++ there is a wrapper function in C with a pointer to a `struct` representing an object. The Haskell library binds to the C functions. As the C++ methods assume access to a mutable state, wxHaskell makes use of mutable variables. Our examples use mutable variables based on Concurrent Haskell (Jones *et al.*, 1996). In our Agda code, we do not focus on modeling the inheritance relationship between widgets that is present in the C++ library. In wxHaskell, inheritance relationships are modeled as phantom types; however, as it relies on unsafe object casts, it is only an approximation that does not fully represent subtyping of object-oriented programming languages (see the related work of phantom types in section 11).

wxHaskell provides some basic data types such as the device context DC, frame Frame and button Button. Additionally, it provides functions for creating and placing GUI elements. GUI elements have properties, using syntax such as

```
frame [text := "Frame Title"]
```

for creating a frame with the title "Frame Title". Event handlers are associated with GUI elements by using syntax such as

```
set myframe [on paint := prog]
```

which sets the onpaint method (in underlying C++ terms) for frame `myframe` to program `prog`, where `prog` is an element of IO () in Haskell. In order to share information between event handlers, mutable variables are used. As multiple events may occur in parallel, we use variables based on Concurrent Haskell (Jones *et al.*, 1996):

A mutable location `MVar a` is either empty or contains a value. There are commands for creating a mutable location, putting a value into the location, and taking a value out of the location:

```
newMVar  :: a -> IO (MVar a)
putMVar  :: MVar a -> a -> IO ()
takeMVar :: MVar a -> IO a
```

A thread putting a variable blocks until the variable is empty, and then puts a value into that location. If it is taking a variable, it blocks until the variable is non-empty, and then reads the value, leaving the location empty. The dispatch function in the next section utilizes Haskell's MVar semantics to implement thread-safe communication. The Agda Var type corresponds to the Haskell MVar type and, e.g., nativePutVar is a wrapper for `putMVar` in Haskell.

10.3 A Library for Object-Based GUIs in Agda

We will handle variables by forming a list of variables. Since a variable depends on its type—an element of Set—a list of variables is an element of the next type level Set_1 above Set.

```
data VarList : Set₁ where
  []     : VarList
  addVar : (A : Set) → Var A → VarList → VarList
```

We form the product of the set of variables in a VarList. In our example below, we have only one variable of type A. In order to obtain it as product A instead of $A \times \mathsf{Unit}$, we add a special case for the singleton list:

```
prod : VarList → Set
prod []                = Unit
prod (addVar A v [])   = A
prod (addVar A v l)    = A × prod l
```

The function takeVar reads in sequence all the variables, empties them, and returns the product. If a variable is empty, it waits until it is non-empty, before taking it. The function putVar writes all the variables, leaving them non-empty. If a variable is non-empty, it waits until it is empty, before putting the value. At the time of writing, Agda requires to expand the pattern in the third case from (addVar A v l) to (addVar A v (addVar B v’ l)), but we hope this will be fixed in a future implementation of Agda.

```
takeVar : (l : VarList) → NativeIO (prod l)
takeVar []                          = nativeReturn unit
takeVar (addVar A v [])             = nativeTakeVar {A} v
takeVar (addVar A v (addVar B v′ l)) =
  nativeTakeVar {A} v    native>>= λ a →
  takeVar (addVar B v′ l) native>>= λ rest →
  nativeReturn ( a , rest )

putVar : (l : VarList) → prod l → NativeIO Unit
putVar [] _                                  = nativeReturn unit
putVar (addVar A v []) a                     = nativePutVar {A} v a
putVar (addVar A v (addVar B v′ l)) (a , rest) =
  nativePutVar {A} v a native>>= λ _ →
  putVar (addVar B v′ l) rest
```

We have two levels of IO interfaces: The level 1 interface GuiLev1Interface is used for creating and modifying GUI elements without making use of event handlers. It does not, however, allow the use of variables. We omit its definition, which is similar to the one given in Sect. 9.

The main program uses the level 2 interface, which extends the level 1 interface and has commands for adding event handlers that refer to programs written for the level 1 interface. This means that it *negatively* refers to the set of all level 1 IO programs. In order for this to be possible, the set of level 1 programs need to be defined (and therefore the level 1 interface be finished) before we can define the level 2 interface. If we allowed the level 2 interface to refer to itself we would get an *inconsistent type theory* (essentially we need the principle Set : Set). Therefore we could not use Agda for verification.

Because of this, we separate level 1 and level 2 interfaces: The level 1 interface has no event handler. The level 2 interface extends the level 1 interface by the possibility of adding event handlers, which refer to level 1 programs. This is the reason why we call them “level 1” and “level 2”, because level 2 programs can refer to the collection of all level 1 programs.

Event handlers are expected to be executed as independent *threads*, possibly in parallel. In order to communicate between them, we add to the level 2 interface the ability to create and use variables that represent the shared state between the event handlers. Event handlers access the shared variables, modify them and update the shared variables.

A first approximation for the type of an event handler referring to variable list l is

```
eventHandler : prod l → IO GuiLev1Interface (prod l)
```

When translated into a native IO program, the event handler will read the state of all variables, obtaining value a : prod l. Then it will execute program (eventHandler a) which, when terminating, returns an element a’ : prod l which will then be written back to the variables.

Some modifications are needed: One is that eventHandler has two kinds of additional parameters: (1) Some are to be executed when it is first created. Let the types of those parameters be $B_1, \ldots, B_n$. (2) Furthermore, we have some parameters which are used each time eventHandler is activated. Let their types be $C_1, \ldots, C_n$. The reason for the other modification is that one event handler might trigger events which other event handlers then handle. For this to work, one might need to update the variables before this trigger event is activated, so that the other handlers triggered make use of the updated state. The solution is to have a list of event handlers instead of having just one event handler. The event handlers in such a list will be executed in sequence. After each event handler in this list has been executed, the variables are updated. As a consequence, event handlers from other threads from now on will make use of the updated state. Therefore, we can update the state in one event handler of the list, and trigger an event in a later event handler of the list. A handler that refers to the revised state will handle the triggered event. Using these considerations, we obtain that the type of an event handler is as follows:

```
eventHandler : B₁ → ⋯ Bₙ → List ( prod l → C₁ → ⋯ Cₙ
                                   → IO GuiLev1Interface (prod l))
```

The level 2 interface GuiLev2Interface will be a state-dependent IO interface. The state of GuiLev2Interface is the list of variables obtained up to now, i.e. VarList : Set₁:

```
GuiLev2State : Set₁
GuiLev2State = VarList
```

The level 2 interface has as commands level 1 commands, a command for creating a variable, and commands for adding a button handler, and an onPaint handler. The type of event handlers is as discussed before. The type of event handlers will refer to the variables created up to now; therefore, the commands for setting an event handler will depend on this state. Since the type of variables is an element of Set, the type of commands will be an element of Set₁.

```
data GuiLev2Command (s : GuiLev2State) : Set₁ where
  level1C          : GuiLev1Command → GuiLev2Command s
  createVar        : {A : Set} → A → GuiLev2Command s
  setButtonHandler : Button
                     → List (prod s → IO GuiLev1Interface ∞ (prod s))
                     → GuiLev2Command s
  setOnPaint       : Frame
                     → List (prod s → DC → Rect → IO GuiLev1Interface ∞ (prod s))
                     → GuiLev2Command s
```

The responses for level 1 commands are the corresponding level 1 responses. The response for the create variable command is the variable which was created. For all other commands the response is empty (an element of Unit). The type of responses is an element of Set:

```
GuiLev2Response : (s : GuiLev2State) → GuiLev2Command s → Set
GuiLev2Response _ (level1C c)        = GuiLev1Response c
GuiLev2Response _ (createVar {A} a)  = Var A
GuiLev2Response _ _                  = Unit
```

When creating a new variable, the return type will be a new variable; adding the new variable to the list of variables then updates this state. Otherwise the state will remain unchanged. Here we have an example of a state-dependent interface where the next state not only depends on the command executed, but also on the response returned.

```
GuiLev2Next : (s : GuiLev2State) → (c : GuiLev2Command s)
  → GuiLev2Response s c
  → GuiLev2State
GuiLev2Next s (createVar {A} a)  var  =  addVar A var s
GuiLev2Next s _  _                    =  s
```

Combining the above we obtain the resulting interface, which is an element of the second type level Set_2:

```
GuiLev2Interface : IOInterfaceˢ
Stateˢ     GuiLev2Interface  =  GuiLev2State
Commandˢ   GuiLev2Interface  =  GuiLev2Command
Responseˢ  GuiLev2Interface  =  GuiLev2Response
nextˢ      GuiLev2Interface  =  GuiLev2Next
```

When translating an event handler, which refers to variables l, into NativeIO, we obtain a list of functions of type

```
f : prod l → NativeIO (prod l)
```

The function dispatch will translate each of these functions into an element of NativeIO, which takes the variables, obtains a value a, executes f, and then writes back the variable.

```
dispatch : (l : VarList) → (prod l → NativeIO (prod l)) → NativeIO Unit
dispatch l f =  takeVar l  native>>= λ a   →
                f a        native>>= λ a₁  →
                putVar l a₁
```

Dispatch will be used for writing event handlers which are possibly executed in parallel. Any dispatched handler of the form (dispatch $l\,f$) will empty the variables initially. No other dispatched handler then starts, because it waits until the variables are non-empty. When the first dispatched handler has finished, it writes the variables, allowing other dispatched handlers (which run in parallel) to start. Therefore, the execution of dispatched handlers of different threads is mutually exclusive. This is necessary since any intermediate changes of the state are not shared between threads.

The dispatching of a list of such functions is obtained by dispatching each individual function in sequence. Therefore, updates to the variables in one element of the list are shared to all other event handlers before executing the next element of the list. Since the variables are then non-empty, other threads accessing the variables at this point might interrupt execution and change the variables. Therefore, reading the variables again after having written them is necessary, since they may have changed.

```
dispatchList : (l : VarList) → List (prod l → NativeIO (prod l)) → NativeIO Unit
dispatchList l []          = nativeReturn unit
dispatchList l (p :: rest) = dispatch l p  native>>= λ _ →
                             dispatchList l rest
```

We define translateLev1Local : (c : GuiLev1Command) → NativeIO (GuiLev1Response c) similarly to as we did in Sect. 9. The translation of level 2 commands makes use of the dispatch function. Since the event handlers are lists of functions, we need to apply the level 1 translation to each of the elements of this list by using the operation map.

```
translateLev2Local  :  (s : GuiLev2State)
                       → (c : GuiLev2Command s)
                       → NativeIO (GuiLev2Response s c)
translateLev2Local s (level1C c)          =  translateLev1Local c
translateLev2Local s (createVar {A} a)  =  nativeNewVar {A} a
translateLev2Local s (setButtonHandler bt proglist) =
   nativeSetButtonHandler bt
      (dispatchList s (map (λ prog → translateLev1 ∘ prog)  proglist))
translateLev2Local s (setOnPaint fra proglist) =
   nativeSetOnPaint fra (λ dc rect → dispatchList s
      (map (λ prog aa → translateLev1 (prog aa dc rect)) proglist))

translateLev2 : ∀ {A s} → IOˢ GuiLev2Interface ∞ (λ _ → A) s → NativeIO A
translateLev2 = translateIOˢ  translateLev2Local
```

Note that the translation of (setButtonHandler bt proglist) uses nativeSetButtonHandler which creates a new thread running a handler. The handler waits for a button event. If the button event happens the handler code is executed.

10.4 Example: A GUI controlling a Space Ship in Agda

We are going to introduce a program displaying a small spaceship controlled by buttons in Agda. This example is based on the Haskell program of the asteroids game (Leijen, 2004; github, 2015). We will demonstrate only one button in this paper, which moves the spaceship to the right by a fixed amount. We will define three versions, which differ by the type of shared variables. These versions correspond to different methodologies of writing event handlers. The first one uses the datatype of integers $\mathbb{Z}$ as its shared state. It represents the x-coordinate of the spaceship. The second one uses an object for storing the shared state. Here we simply wrap $\mathbb{Z}$ into a cell object. More advanced examples would make use of more complex objects. Finally, the third one follows the common approach in object-oriented programming, namely to define the event handlers as methods of a common object.

All versions will define three event handling functions: onPaint, which handles the onpaint event for drawing the spaceship; moveSpaceShip, which is the first part of the button handler, which updates the state so that next time the spaceship is drawn its coordinates have changed; and callRepaint, which triggers a repaint event. The button will be handled by the two handlers moveSpaceShip and callRepaint in sequence. When the button event is triggered, first moveSpaceShip moves the spaceship by updating and sharing the state of the spaceship with new updated coordinates. Then the callRepaint handler will trigger a repaint event which triggers the paint function to repaint the spaceship with the new coordinates.

It turns out that in all three versions the types of the event handling functions are — except for the type of the shared variable — the same. Furthermore, the definitions of the main program are identical. We will therefore define it at the end.

In the first version, the event handlers will read the x-coordinate, an element of ℤ, and return the updated coordinate. Here +_ is the constructor for ℤ embedding ℕ into ℤ. We define the type of the shared variable and its initial value:

```
VarType = ℤ

varInit : VarType
varInit = (+ 150)
```

The event handling functions are as follows:

```
onPaint : ∀{i} → VarType → DC → Rect → IO GuiLev1Interface i VarType
onPaint z dc rect       = do (drawBitmap dc ship (z , (+ 150)) true) λ _ →
                          return z

moveSpaceShip  : ∀{i} → Frame → VarType → IO GuiLev1Interface i VarType
moveSpaceShip  fra z  =  return (z + (+ 20))

callRepaint        : ∀{i} → Frame → VarType → IO GuiLev1Interface i VarType
callRepaint        fra z  =  do (repaint fra) λ _ → return z
```

In the second version we define the variable via an object. Here, we will take the example of a simple cell, containing an integer with constructor cellℤC.

```
VarType  = Object (cellJ ℤ)

cellℤC : (z : ℤ ) → VarType
objectMethod (cellℤC z) get        = (  z      ,  cellℤC z   )
objectMethod (cellℤC z) (put z′)   = (  unit   ,  cellℤC z′  )

varInit : VarType
varInit = cellℤC (+ 150)
```

The event handlers are defined as before, but now they call methods of the simple cell object for getting and setting the coordinate. We omit their types since these are identical for all 3 versions of this program

```
onPaint c dc rect =
  let (z , c₁) = objectMethod c get  in
  do  (drawBitmap dc ship (z , (+ 150)) true)  λ _ →
  return c₁

moveSpaceShip fra c =
  let (z   ,  c₁)  =  objectMethod c    get
      (_   ,  c₂)  =  objectMethod c₁  (put (z + (+ 20)))
  in   return c₂

callRepaint fra c = do (repaint fra) λ _ →  return c
```

The third version makes use of an object, which has methods onPaintM, moveSpaceShipM, callRepaintM corresponding to the first 3 event handling functions.

```
data GraphicServerMethod : Set where
  onPaintM            :  DC       →  Rect → GraphicServerMethod
  moveSpaceShipM  :  Frame  →  GraphicServerMethod
  callRepaintM       :  Frame  →  GraphicServerMethod

GraphicServerResult : GraphicServerMethod  →  Set
GraphicServerResult  _  = Unit

GraphicServerInterface : Interface
Method  GraphicServerInterface  =  GraphicServerMethod
Result    GraphicServerInterface  =  GraphicServerResult

GraphicServerObject : ∀{i} → Set
GraphicServerObject {i} = IOObject GuiLev1Interface GraphicServerInterface i

graphicServerObject : ∀{i} → ℤ → GraphicServerObject {i}
method (graphicServerObject z) (onPaintM dc rect) =
      do (drawBitmap dc ship (z , (+ 150)) true) λ _ →
      return (unit , graphicServerObject z)
method (graphicServerObject z) (moveSpaceShipM fra) =
      return (unit , graphicServerObject (z + (+ 20)))
method (graphicServerObject z) (callRepaintM fra) =
      do (repaint fra)  λ _ →
      return (unit , graphicServerObject z)

VarType = GraphicServerObject {∞}
```

```
varInit : VarType
varInit = graphicServerObject (+ 150)
```

The event handlers will now simply call the methods of the shared object. The methods' result type is IO (Unit × GraphicServerObject), namely the product of the response type Unit of the methods and of the object itself. We map it using mapIO and the function proj_2 to GraphicServerObject i.e. to VarType.

```
onPaint obj dc rect = mapIO proj₂  (method obj (onPaintM dc rect))

moveSpaceShip fra obj =  mapIO proj₂ (method obj (moveSpaceShipM fra))

callRepaint fra obj = mapIO proj₂ (method obj (callRepaintM fra))
```

The main program, which is identical for all three versions, does the following: It creates a frame and a button, adds the button to the frame, creates a variable of type VarType initialized by varInit and sets the button handler and the onPaint handler to the event handlers defined. The program is then translated into a NativeIO program:

```
program : ∀{i} → IOˢ GuiLev2Interface i (λ _ → Unit) []
program =  doˢ (level1C makeFrame)           λ fra →
           doˢ (level1C (makeButton fra))  λ bt →
           doˢ (level1C (addButton fra bt)) λ _ →
           doˢ (createVar varInit)  λ _ →
           doˢ (setButtonHandler bt (moveSpaceShip fra :: [ callRepaint fra ])) λ _ →
           doˢ (setOnPaint fra [ onPaint  ])
           returnˢ

main : NativeIO Unit
main = start (translateLev2 program)
```

11 Related Work

Typestate-oriented programming (Garcia *et al.*, 2014) is an extension of object-oriented programming (Strom & Yemini, 1986). It models state-dependent interfaces and object behaviour in imperative object-oriented programming. The states are given by a finite number of type states. Executing a method may change the state of an object. Although typestate-oriented programming can express the full range of object-oriented programming including aliases, it lacks a notion of dependent states that can be statically verified. Thus, the approach may catch only some errors statically, while still resorting to runtime checks or assertions (Garcia *et al.*, 2014) to cover all errors. Furthermore, objects with an infinite number of states (such as our stack example with the state being the number of elements on the stack) are out of the scope of typestate-oriented programming and other approaches to typestate.

Abadi and Cardelli (1996) introduce the ζ-calculus, which is similar to the λ-calculus, but for objects. There is a special second order quantifier called Self quantifier $\zeta(X)\varphi(X) := \mu Y.\exists X <: Y.\varphi(X)$. This allows to type objects and classes which make self-referential calls. Dependent types are not studied in their approach.

Coinduction in type theory. In the context of Nuprl's extensional type theory, simple coinductive types (Mendler *et al.*, 1986) such as streams have been considered as greatest fixed-points of functors, using, in modern terminology, the following introduction rule:

$$\frac{\Gamma, n : \mathbb{N} \vdash e : F^n(\top)}{\Gamma \vdash e : \nu F}$$

In talking about the finite approximations $F^n(\top)$ of the coinductive type νF, it resembles sized types. However, in extensional type theory type checking is undecidable. Corecursive definitions have to be justified by proof, here by induction on the natural number n, whereas in Agda's sized types are built into the core language. One could, of course, say that Agda's sized types give the information needed to create such proofs.

Coquand (1994) introduces coinductive types via constructors as non-wellfounded trees—in contrast to the coalgebraic approach to define them via their destructors (Hagino, 1989; Setzer, 2012; Abel *et al.*, 2013). Coquand's work contains the definition of productivity of corecursive definitions and the guarded-by-constructors criterion to ensure productivity. This also extends to proofs as "guarded induction principle", but has limited expressivity, which is overcome by sized types as described in this article.

Coinductive types have been added to Coq's *Calculus of Inductive Constructions* following Coquand's proposal (Giménez, 1996). In Giménez' thesis, it was already noted that dependent pattern matching on coinductive data breaks subject reduction.

Gimenez also suggested a type-based productivity check (Giménez, 1998) with similar proposals occurring at around the same time (Hughes *et al.*, 1996; Amadio & Coupet-Grimal, 1998). Since then, sized types have seen thorough theoretical exploration (Barthe *et al.*, 2004; Blanqui, 2004; Barthe *et al.*, 2008; Abel, 2008, 2007; Sacchini, 2013) and several prototypical implementations (Barthe *et al.*, 2005; Abel, 2010; Sacchini, 2015).

Component-based programming Hancock and Hyvernat (2006) and Granström (2012) (see also the component-based programming language IPL (Granström, 2016)) have suggested the use of interactive programs in component-based programming. A component is a combination of a server-side and a client-side program: It receives a request from the server and then interacts with the client-side until it has computed a response, which is then returned to the server-side. After the request ends, the component waits for the next server request. In this sense, an IOObject is a component having the object interface as a server-side interface and the IO-interface as a client-side interface. CounterCell (Sect. 7) can be considered as a component which communicates with a CellC object on its client-side.

Isabelle has many advantages since it integrates powerful automated theorem provers, especially Sledgehammer. Its built-in equality for coalgebras is already bisimilarity, making proofs much easier. However, it lacks dependent types. Strict positivity is more restrictive than in Agda which allows inductive-recursive (Dybjer & Setzer, 2003) and inductive-inductive definitions (Nordvall Forsberg & Setzer, 2010), which only make sense using dependent types. Lochbihler and Züst (2014) demonstrated how to use Isabelle as a functional programming language. Their type of interactive programs makes use of a type similar to our IO monad. Because of the lack of dependent types, this type has only one command with one type of arguments, and one result type. That could easily be generalised to finitely many methods, but not to the full generality in this paper. Blanchette *et al.* (2015) introduce friendly functions which are allowed on the right-hand side of corecursive definitions. They play a similar rôle to that of size preserving functions in our settings. However, size preserving functions seem to be more general.

Software Transactional Memory. The STM monad (Harris *et al.*, 2008; hackage.haskell.org, 2016) allows to combine a series of actions such as writing and reading variables into one transaction. If such a transaction is interrupted, the transaction is rolled back to the state it was before it was executed. IO actions are not allowed inside such transactions. For this reason, the STM monad is not suitable for our approach, and we use manual locking via `MVars` instead.

Functional Reactive Programming (FRP) is another approach for writing interactive programs in functional programming languages. The idea is that input and output are given by input and output streams, and one has operations for creating new streams from existing ones. The elements of the input streams change as the input changes, which is then reflected in the elements of the streams defined from it, including the output streams. Therefore the output reacts in response to the input. In connection with dependent types, FRP has been studied from the foundational perspective (Sculthorpe & Nilsson, 2009) and for verified programming (Jeffrey, 2013).

Phantom Types for Modeling Inheritance Relationships wxHaskell offers bindings to the C++ GUI-library wxWidgets. The Haskell bindings model inheritance relationships (e.g., between widget classes in C++) as phantom types. However, wxHaskell cannot fully represent subtyping of object-oriented programming languages, as it relies on unsafe object casts. Phantom types are types with an additional type parameter which is not used by its constructors. For instance, we can state

$$\begin{array}{l}\text{data isPerson}\ x\\ \text{data isStudent}\ x\\ \text{data isPhDStudent}\ x\end{array}$$

If we had existential quantifiers over types, one could define

$$\begin{array}{lcl}\text{Person} & = & \exists x.\text{isPerson}\ x\\ \text{Student} & = & \exists x.\text{isPerson}\ (\text{isStudent}\ x)\\ \text{PhDStudent} & = & \exists x.\text{isPerson}\ (\text{isStudent}\ (\text{isPhDStudent}\ x))\end{array}$$

and then obtain: if a : Student then a : Person and if a : PhDStudent then a : Student and a : Person.

However, Haskell does not have existential quantifiers, so instead one defines

$$\begin{aligned}
\text{Person} &= \text{isPerson}\ ()\\
\text{Student} &= \text{isPerson}\ (\text{isStudent}\ ())\\
\text{PhDStudent} &= \text{isPerson}\ (\text{isStudent}\ (\text{isPhDStudent}\ ()))
\end{aligned}$$

Now, an element of Student is no longer an element of Person but we can define an upcasting function

$$\text{upcast} : \text{Student} \to \text{Person}$$

As we can equally define downcast : Person → Student, the definition is unsafe. Furthermore, Student and Person are type synonymes, so they do not really have different constructors or (as objects) methods. One can distinguish them by having operations such as

$$\text{studentNumber} : \text{Student} \to \mathbb{N}$$

which is a postulated function and gets its implementation only from the corresponding C code. In this sense, this use of phantom types is unsafe, meaning we do not really have a type hierarchy but are using potentially unsafe casts which are not type-checked.

Algebraic effects and Idris. In the dependently typed programming language Idris, Brady has created an effects library based on algebraic effects (Brady, 2014). Algebraic effects were introduced in Bauer & Pretnar (2015). Effects can be considered as a version of state dependent IO.

The type Effect of effects is a predicate on the sets *Result*, incoming *InResource*, and outcoming resources *OutResource* : *Result* → Set. Written in Agda syntax, this reads as follows:

```
Effect : Set₁
Effect = (Result : Set) → (InResource : Set) →  (OutResource : Result → Set) → Set
```

This can be considered as a state-dependent interface: The states are Set, the commands for a state s are the effects for which the result component is s, the responses are the *Result* component of the effect, and the next state is determined by the *OutResource* component:

```
effectToIOInterfaceˢ : Effect → IOInterfaceˢ
Stateˢ     (effectToIOInterfaceˢ eff)     = Set
Commandˢ   (effectToIOInterfaceˢ eff)  s  =
   Σ[ Result ∈ Set ] (Σ[ outR ∈ (Result → Set) ] (eff Result s outR))
Responseˢ  (effectToIOInterfaceˢ eff)  s (result , outR , op)  =  result
nextˢ      (effectToIOInterfaceˢ eff)  s (result , outR , op)  =  outR
```

In order to handle several effects in parallel, Brady introduces the type EFFECT, which consists of a state and a state-dependent interface, which can be simplified to

$\mathsf{EFFECT} = \mathsf{Set} \times \mathsf{Effect}$

Then, he defines a second level of state-dependent interfaces Eff, which, as before, are state-dependent interfaces, but with (List EFFECT) as state. This type is not a closed data type but open for new commands to be registered, similar to IO in Haskell.

Furthermore, Brady introduces handlers, which are defined by referring to the predicate based data type Effect. If we replace this type by a set interface with the components $State^{s}$, C, R, and *next*, we obtain the following type of a handler, which depends on an operation $M : \mathsf{Set} \to \mathsf{Set}$:

$$(A : \mathsf{Set}) \to (s : State^{s}) \to (c : C\ s) \to (f : (r : R\ s\ c) \to next\ s\ c\ r \to M\ A) \to M\ A$$

Using a handler, an IO program for the corresponding interface with return type A can be evaluated to an element of $M A$ essentially by evaluating the handler for each effect. This allows, for instance, to write effectful programs having as effects an exception with return type A, and evaluate them to an element of Maybe A.

Brady introduces some very elegant syntax for defining and programming with Effects. As in Bauer & Pretnar (2015), he allows expressions of normal data types to be formed from effectful programs (Brady uses a ! notation). This means that Idris code looks very similar to ML code where we have terms with side effects. However, this requires strict evaluation and that the order of evaluation is fixed.

12 Conclusion

We have seen how to introduce interactive programs and objects in Agda. We demonstrated how to program with them, including introducing graphical user interfaces with action listeners. We have seen the importance of state-dependent interactive programs and objects. One true example of a state-dependent interactive program was the creation of variables, where the new state depends not only on the issued command but also on the response given by the real world, namely the variable which was created. The example in Sect. 10 solved a problem in Agda: The original implementation in Haskell is rather low-level and requires the direct modification of variables. Our program solves this issue by using a shared object which can be accessed by the action listeners while they are executed. This is very close to the way this is actually implemented in standard object-oriented languages.

Our approach is a first step towards introducing object-orientation into dependent type theory. However, object-orientation consists of much more than simple objects. We have not shown how to define objects calling each other recursively; some work is already available in the third author's work (Setzer, 2006). The problem is to find a definition in such a way that it passes the termination checker – a method calling itself immediately would result in black hole recursion. We have introduced a first step towards inheritance, namely to extend an object by additional methods reusing the original method implementations. However, future work is required to develop a methodology for overwriting existing methods. Proper inheritance would require a more expressive form of subtyping as it is currently implemented in Agda. The most challenging problem at this moment seems to be how to define objects dynamically on the heap. This would need some notion of pointers.

We hope that this article is a step towards having a programming language which has both dependent types and object-orientation. This would allow to combine both of these advanced programming paradigms, and to create a language in which programming is considerably easier and safer.

Acknowledgments. The first author acknowledges support from Vetenskapsrådet (Swedish Research Council) through project 621-2014-4864/E0486401 *Termination Certification for Dependently-Typed Programs and Proofs via Refinement Types*. The third author acknowledges support by EPSRC grant EP/G033374/1 *Theory and Applications of Induction Recursion*; CORCON FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638; and COMPUTAL FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962. First and third author were further supported by CA COST Action CA15123 European research network on types for programming and verification (EUTYPES).

Bibliography

Abadi, Martín, & Cardelli, Luca (eds). (1996). *A theory of objects*. Monographs in Computer Science. Springer.

Abbott, Michael, Altenkirch, Thorsten, & Ghani, Neil. (2003). Categories of containers. *In:* Gordon (2003).

Abel, Andreas. (2007). Mixed inductive/coinductive types and strong normalization. *Pages 286–301 of:* Shao, Zhong (ed), *Proc. of the 5th Asian Symp. on Programming Languages and Systems, APLAS 2007*. Lect. Notes in Comput. Sci., vol. 4807. Springer.

Abel, Andreas. (2008). Semi-continuous sized types and termination. *Logical Meth. in Comput. Sci.*, **4**(2:3), 1–33. CSL'06 special issue.

Abel, Andreas. (2010). MiniAgda: Integrating sized and dependent types. *Pages 14–28 of:* Bove, Ana, Komendantskaya, Ekaterina, & Niqui, Milad (eds), *Wksh. on Partiality And Recursion in Interactive Theorem Provers, PAR 2010*. Electr. Proc. in Theor. Comp. Sci., vol. 43.

Abel, Andreas. (2012). Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. *Pages 1–11 of:* Miller, Dale, & Ésik, Zoltán (eds), *Proc. of the 8th Wksh. on Fixed Points in Comp. Sci. (FICS 2012)*. Electr. Proc. in Theor. Comp. Sci., vol. 77. Invited talk.

Abel, Andreas, & Altenkirch, Thorsten. (2002). A predicative analysis of structural recursion. *J. Func. Program.*, **12**(1), 1–41.

Abel, Andreas, & Pientka, Brigitte. (2013). Wellfounded recursion with copatterns: A unified approach to termination and productivity. *Pages 185–196 of:* Morrisett, Greg, & Uustalu, Tarmo (eds), *Proc. of the 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2013*. ACM Press.

Abel, Andreas, Pientka, Brigitte, Thibodeau, David, & Setzer, Anton. (2013). Copatterns: Programming infinite structures by observations. *Pages 27–38 of:* Giacobazzi, Roberto, & Cousot, Radhia (eds), *Proc. of the 40th ACM Symp. on Principles of Programming Languages, POPL 2013*. ACM Press.

Abel, Andreas, Adelsberger, Stephan, & Setzer, Anton. (2016). *ooAgda*. Available from https://github.com/agda/ooAgda.

Abelson, Harold, Sussman, Gerald Jay, & Sussman, Julie. (1996). *Structure and interpretation of computer programs*. 2nd edn. MIT Press.

Agda Wiki. 2011 (11 October). *Malonzo*. http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Docs.MAlonzo.

AgdaTeam. (2016). *The Agda Wiki*.

Altenkirch, Thorsten, & Danielsson, Nils Anders. (2012). Termination checking in the presence of nested inductive and coinductive types. *Pages 101–106 of:* Bove, Ana, Komendantskaya, Ekaterina, & Niqui, Milad (eds), *Wksh. on Partiality And Recursion in Interactive Theorem Provers, PAR 2010*. EPiC Series in Comput. Sci., vol. 5. EasyChair.

Altenkirch, Thorsten, & Morris, Peter. (2009). Indexed containers. *Pages 277–285 of: Proc. of the 24nd IEEE Symp. on Logic in Computer Science (LICS 2009)*. IEEE Computer Soc. Press.

Amadio, Roberto M., & Coupet-Grimal, Solange. (1998). Analysis of a guard condition in type theory (extended abstract). *Pages 48–62 of:* Nivat, Maurice (ed), *Proc. of the 1st Int. Conf. on Foundations of Software Science and Computation Structure, FoSSaCS'98*. Lect. Notes in Comput. Sci., vol. 1378. Springer.

Barthe, Gilles, Frade, Maria João, Giménez, Eduardo, Pinto, Luís, & Uustalu, Tarmo. (2004). Type-based termination of recursive definitions. *Math. Struct. in Comput. Sci.*, **14**(1), 97–141.

Barthe, Gilles, Grégoire, Benjamin, & Pastawski, Fernando. (2005). Practical inference for type-based termination in a polymorphic setting. *Pages 71–85 of:* Urzyczyn, Pawel (ed), *Proc. of the 7th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2005*. Lect. Notes in Comput. Sci., vol. 3461. Springer.

Barthe, Gilles, Grégoire, Benjamin, & Riba, Colin. (2008). Type-based termination with sized products. *Pages 493–507 of:* Kaminski, Michael, & Martini, Simone (eds), *Computer Science Logic, 22nd Int. Wksh., CSL 2008, 17th Annual Conf. of the EACSL*. Lect. Notes in Comput. Sci., vol. 5213. Springer.

Bauer, Andrej, & Pretnar, Matija. (2015). Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, **84**(1), 108 – 123.

Benke, Marcin. (2007). *Alonzo — a compiler for Agda*. Talk given at TYPES 2007. Available from http://www.mimuw.edu.pl/~ben/Papers/TYPES07-alonzo.pdf.

Blanchette, Jasmin Christian, Popescu, Andrei, & Traytel, Dmitriy. (2015). Foundational extensible corecursion: a proof assistant perspective. *Pages 192–204 of:* Fisher, Kathleen, & Reppy, John H. (eds), *Proc. of the 20th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2015*. ACM Press.

Blanqui, Frédéric. (2004). A type-based termination criterion for dependently-typed higher-order rewrite systems. *Pages 24–39 of:* van Oostrom, Vincent (ed), *Rewriting Techniques and Applications, RTA 2004, Aachen, Germany*. Lect. Notes in Comput. Sci., vol. 3091. Springer.

Boehm, Hans-Juergen, & Jr., Guy L. Steele (eds). (1996). *Proc. of the 23rd ACM Symp. on Principles of Programming Languages, POPL'96*. ACM Press.

Brady, Edwin. (2014). Resource-dependent algebraic effects. *Pages 18–33 of:* Hage, Jurriaan, & McCarthy, Jay (eds), *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*. Lect. Notes in Comput. Sci., vol. 8843. Springer.

Coq Community. (2015). *The Coq Proof Assistant* . https://coq.inria.fr/.

Coquand, Thierry. (1994). Infinite objects in type theory. *Pages 62–78 of:* Barendregt, Henk, & Nipkow, Tobias (eds), *Types for Proofs and Programs, Int. Wksh., TYPES'93*. Lect. Notes in Comput. Sci., vol. 806. Springer.

Dagand, Pierre-Évariste, & McBride, Conor. (2014). Transporting functions across ornaments. *J. Func. Program.*, **24**(2-3), 316–383.

Dybjer, Peter. (1994). Inductive families. *Formal Asp. Comput.*, **6**(4), 440–465.

Dybjer, Peter, & Setzer, Anton. (2003). Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, **124**, 1 – 47.

Garcia, Ronald, Tanter, Éric, Wolff, Roger, & Aldrich, Jonathan. (2014). Foundations of typestate-oriented programming. *ACM Trans. on Program. Lang. and Syst.*, **36**(4), 12:1–12:44.

Giménez, Eduardo. 1996 (Dec.). *Un calcul de constructions infinies et son application a la vérification de systèmes communicants*. Ph.D. thesis, Ecole Normale Supérieure de Lyon. Thèse d'université.

Giménez, Eduardo. (1998). Structural recursive definitions in type theory. *Pages 397–408 of:* Larsen, K. G., Skyum, S., & Winskel, G. (eds), *Int. Colloquium on Automata, Languages and Programming (ICALP'98), Aalborg, Denmark*. Lect. Notes in Comput. Sci., vol. 1443. Springer.

Girard, Jean-Yves. (1972). *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de Doctorat d'État, Université de Paris VII.

github. 2015 (Commit f2f5b5a 23 Sept). *wxAsteroids*. `https://github.com/wxHaskell/wxAsteroids`.

Gordon, Andrew D. (ed). (2003). *Proc. of the 6th int. conf. on foundations of software science and computational structures, fossacs 2003*. Lect. Notes in Comput. Sci., vol. 2620. Springer.

Granström, Johan Georg. (2012). A new paradigm for component-based development. *J. Software*, **7**(5), 1136–1148.

Granström, Johan Georg. 2016 (Retrieved 8 July). *Intuitionistic Programming Language*. `http://intuitionistic.org/`.

hackage.haskell.org. 2016 (Retrieved 17 Feb). *Control.Monad.STM*. https://hackage.haskell.org/package/stm-2.4.4/docs/Control-Monad-STM.html.

Hagino, Tatsuya. (1989). Codatatypes in ML. *J. Symb. Logic*, **8**(6), 629–650.

Hancock, Peter, & Hyvernat, Pierre. (2006). Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, **137**, 189 – 239.

Hancock, Peter, & Setzer, Anton. (2000a). Interactive programs in dependent type theory. *Pages 317–331 of:* Clote, Peter, & Schwichtenberg, Helmut (eds), *Computer Science Logic, 14th Int. Wksh., CSL 2000, 9th Annual Conf. of the EACSL*. Lect. Notes in Comput. Sci., vol. 1862. Springer.

Hancock, Peter, & Setzer, Anton. (2000b). Specifying interactions with dependent types. *Workshop on subtyping and dependent types in programming,* *Portugal, 7 July 2000*. Electronic proceedings, available via http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.

Hancock, Peter, & Setzer, Anton. (2005). Interactive programs and weakly final coalgebras in dependent type theory. *Pages 115 – 136 of:* Crosilla, L., & Schuster, P. (eds), *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*. Oxford Logic Guides. Oxford Univ. Press, Oxford.

Hancock, Peter, McBride, Conor, Ghani, Neil, Malatesta, Lorenzo, & Altenkirch, Thorsten. (2013). Small induction recursion. *Pages 156–172 of:* Hasegawa, Masahito (ed), *Typed Lambda Calculi and Applications*. Lecture Notes in Computer Science, vol. 7941. Springer Berlin Heidelberg.

Harris, Tim, Marlow, Simon, Jones, Simon L. Peyton, & Herlihy, Maurice. (2008). Composable memory transactions. *Commun. ACM*, **51**(8), 91–100.

Hirschfeld, Robert, Costanza, Pascal, & Nierstrasz, Oscar. (2008). Context-oriented programming. *J. Obj. Tech.*, **7**(3), 125–151.

Hudak, Paul. 2016 (Retrieved 12 February). *SOE library*. `http://www.cs.yale.edu/homes/hudak/SOE/software1.htm`.

Hughes, John, Pareto, Lars, & Sabry, Amr. (1996). Proving the correctness of reactive systems using sized types. *In:* Boehm & Jr. (1996).

Hurkens, Antonius J. C. (1995). A simplification of Girard's paradox. *Pages 266–278 of:* Dezani-Ciancaglini, Mariangiola, & Plotkin, Gordon D. (eds), *Proc. of the 2nd Int. Conf. on Typed Lambda Calculi and Applications, TLCA '95*. Lect. Notes in Comput. Sci., vol. 902. Springer.

Igried, Bashar, & Setzer, Anton. (2016). *Programming with monadic CSP-style processes in dependent type theory*. To appear in proceedings of TyDe 2016, Type-driven Development, preprint from http://www.cs.swan.ac.uk/∼csetzer/articles/TyDe2016.pdf.

Jeffrey, Alan. (2013). Dependently typed web client applications - FRP in agda in HTML5. *Pages 228–243 of:* Sagonas, Konstantinos F. (ed), *Proc. of the 15th Int. Symp. on Practical Aspects of Declarative Languages, PADL 2013*. Lect. Notes in Comput. Sci., vol. 7752. Springer.

Jones, Simon L. Peyton, Gordon, Andrew D., & Finne, Sigbjorn. (1996). Concurrent haskell. *In:* Boehm & Jr. (1996).

Leijen, Daan. (2004). wxHaskell: A portable and concise GUI library for Haskell. *Pages 57–68 of:* Nilsson, Henrik (ed), *Proc. of the ACM SIGPLAN Workshop on Haskell, Haskell 2004*. ACM Press.

Lochbihler, Andreas, & Züst, Marc. (2014). Programming TLS in Isabelle/HOL. *Isabelle Wksh., Associated with ITP 2014*.

Martin-Löf, Per. (1984). *Intuitionistic type theory*. Naples: Bibliopolis.

Matthes, Ralph. (2002). Tarski's fixed-point theorem and lambda calculi with monotone inductive types. *Synthese*, **133**(1-2), 107–129.

Mendler, Nax P., Panangaden, Prakash, & Constable, Robert L. (1986). Infinite objects in type theory. *Pages 249–255 of: Proc. of the 1st IEEE Symp. on Logic in Computer Science (LICS'86)*. IEEE Computer Society.

Moggi, Eugenio. (1991). Notions of computation and monads. *Inf. Comput.*, **93**(1), 55–92.

Nordvall Forsberg, Fredrik, & Setzer, Anton. (2010). Inductive-inductive definitions. *Pages 454 – 468 of:* Dawar, Anuj, & Veith, Helmut (eds), *CSL 2010*. Lecture Notes in Computer Science, vol. 6247. Springer, Heidelberg.

Norell, Ulf. 2007 (Sept.). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Dept of Comput. Sci. and Engrg., Chalmers, Göteborg, Sweden.

Petersson, Kent, & Synek, Dan. (1989). A set constructor for inductive sets in Martin-Löf's Type Theory. *Pages 128–140 of:* Pitt, David H., Rydeheard, David E., Dybjer, Peter, Pitts, Andrew M., & Poigné, Axel (eds), *Category Theory and Computer Science*. Lecture Notes in Computer Science, vol. 389. London, UK, UK: Springer-Verlag.

Sacchini, Jorge. (2015). *Coq^: Type-based termination in the Coq proof assistant*. Project description, http://qatar.cmu.edu/~sacchini/coq.html.

Sacchini, Jorge Luis. (2013). Type-based productivity of stream definitions in the calculus of constructions. *Pages 233–242 of: 28th IEEE Symp. on Logic in Computer Science (LICS'13)*. IEEE Computer Soc. Press.

Sculthorpe, Neil, & Nilsson, Henrik. (2009). Safe functional reactive programming through dependent types. *Pages 23–34 of:* Hutton, Graham, & Tolmach, Andrew P. (eds), *Proc. of the 14th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2009*. ACM Press.

Setzer, Anton. (2006). Object-oriented programming in dependent type theory. *Pages 91–108 of:* Nilsson, Henrik (ed), *Revised Selected Papers from the 7th Symp. on Trends in Functional Programming, TFP 2006*. Trends in Func. Program., vol. 7. Intellect.

Setzer, Anton. (2012). Coalgebras as types determined by their elimination rules. *Pages 351–369 of:* Dybjer, P., Lindström, Sten, Palmgren, Erik, & Sundholm, G. (eds), *Epistemology versus Ontology*. Logic, Epistemology, and the Unity of Science, vol. 27. Springer Netherlands. 10.1007/978-94-007-4435-6_16.

Setzer, Anton. (2016). How to reason coinductively informally. *Pages 377–408 of:* Kahle, Reinhard, Strahm, Thomas, & Studer, Thomas (eds), *Advances in Proof Theory*. Springer.

Setzer, Anton, & Hancock, Peter. (2004). Interactive programs and weakly final coalgebras in dependent type theory (extended version). Altenkirch, Thorsten, Hofmann, Martin, & Hughes, John (eds), *Dependently Typed Programming 2004*. Dagstuhl Seminar Proc.s, vol. 04381. IBFI, Schloss Dagstuhl, Germany.

Sprenger, Christoph, & Dam, Mads. (2003). On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus. *In:* Gordon (2003).

Strom, Robert E, & Yemini, Shaula. (1986). Typestate: A programming language concept for enhancing software reliability. *Software Engineering, IEEE Transactions on*, 157–171.

Stump, Aaron. (2016). *Verified functional programming in agda*. Morgan & Claypool.

Wegner, Peter. (1987). Dimensions of object-based language design. *OOPSLA'87 Conference Proceedings*, **22**(12), 168—182.

wiki.haskell. 2016 (Retrieved 9 February). *wxHaskell quick start*. https://wiki.haskell.org/WxHaskell/Quick_start.