Towards A Unified Semantic Model For Interactive Applications Using Arrows And Generic Editors

Peter Achten[1], Marko van Eekelen[2], and Rinus Plasmeijer[1]

{[1]Software Technology, [2]Security of Systems} Nijmegen Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 1, 6525ED Nijmegen, Netherlands

P.Achten@cs.ru.nl marko@cs.ru.nl rinus@cs.ru.nl

Abstract

In slightly less than two decades Graphical User Interfaces (GUI) have become standard in presenting the user a pleasant and intuitive interface to software applications. Two major paradigms have contributed to this success: the desktop GUI for widget based programming on a single computer, and the World Wide Web for web applications with a multitude of client computers. Applications that are created within these paradigms are constructed in entirely different ways. Desktop applications can use the vast platform dependent API, or rely on platform independent toolkits. Web applications are confined within web browsers, but they can stretch these limits by relying on a wide variety of scripting languages. This has resulted in 'typical' desktop and web applications. However, one can observe the *trend* that desktop and web applications are taking over each others functionality: former desktop GUI applications now also offer *back*/*forward* buttons, *hyperlinks* from web applications, and web applications use multiple windows/dialogues, disallow *browser window cloning*, and so on. Hence, the clear separation between desktop and web applications is blurring. This implies that in order to reason rigidly about interactive applications in general, we need a semantic model that is capable of handling these two, formerly separated, paradigms. In this paper we present a first step towards such a framework. This work is based on our earlier work on the iData and GEC toolkits.

1 INTRODUCTION

In slightly less than two decades Graphical User Interfaces (GUI) have become standard in presenting the user a pleasant and intuitive interface to software applications. Two major paradigms have contributed to this success: the desktop GUI for widget based programming on a single computer, and the World Wide Web for web applications with a multitude of client computers. At a cursory glance, these two paradigms are closely related as they use the same Windows, Icons, Menus and Pointing device (WIMP) interface style. However, they are completely different. Semantically speaking, desktop GUI applications are essentially a state-based iterative event-handling process (callback semantics), whereas web applications are essentially a stateless, single-step procedure. Desktop applications can use the vast platform dependent APIs that are available, or rely on platform independent libraries, such as wxWidgets. Web applications are confined within web browsers, but they can stretch these limits by relying on a wide variety of scripting languages.

Until recently, one could immediately tell whether an application was a desktop or a web application. However, we can observe the *trend* that desktop and web applications are taking over each others functionality: formerly 'pure' desktop GUI applications now also offer *back*/*forward* buttons, and *hyperlinks* from the web interface, whereas formerly 'pure' web applications are using multiple windows and dialogues, and disallow *browser window cloning*. Hence, the clear-cut separation between desktop and web applications is blurring. This implies that in order to reason rigidly about interactive applications in general, we need a semantic model that is capable of handling these two, formerly separated, paradigms. In this paper we present a first step to a *unified semantic model* for interactive applications.

The unified semantic model that we propose needs to strike a balance between abstraction and practicality. We obtain an abstract semantic model by postponing reasoning about GUI aspects such as the exact set of interactive elements and their layout and look and feel. This can be done by adopting the concept of *generic editor* that has been used in our earlier work on desktop GUI applications on the one hand, the GEC Toolkit [1, 2, 3, 4], and web applications on the other hand, the iData Toolkit [15, 16, 17]. In these toolkits interactive elements are modelled by means of data types and functions. This allows us to concentrate on the data that is maintained by the interactive elements of an application. We obtain a practical semantic model by building it on Arrows [12, 14]. With Arrows we get a disciplined way of modelling the information flow between interactive elements, and still handle arbitrarily complex computations.

As an example of an application within this semantic model, Fig. 1 displays the key fragment of a money-converting application. Function c_0 introduces two interactive elements, labelled with `euroId` and `poundId`, that are mutually interconnected in such a way that if either of them is altered by the user, that the other element responds with the amount of money expressed in the local currency. This is not altogether obvious when considering the mutual dependency of this simple program. We prove this property using the unified semantic model (Sect. 5).

Contributions presented in this paper are:

- We present a uniform semantic framework that captures the essence of desktop and web GUI applications.
- We demonstrate that the framework is practical: real desktop and web applications can be created with it.
- We demonstrate that the framework is accurate: we can reason thoroughly about the framework and applications that are created with it.

The layout of this paper is as follows. We first present the major design decisions that our unified framework needs to satisfy in Sect. 2. We define the unified semantic model in Sect. 3 and present a number of its properties in Sect. 4. We illustrate the use of the model by means of a case study in Sect. 5. We present related work in Sect. 6, and conclusions in Sect. 7.

```
:: Pounds = { pounds :: Real }
:: Euros  = { euros  :: Real }

c0 :: A Euros Pounds
c0 = feedback (edit euroId  >>> arr toPound)
              (edit poundId >>> arr toEuro)
where
    toPound { euros } = { pounds = euros * exchangerateEuroPound }
    toEuro { pounds } = { euros = pounds * exchangeratePoundEuro }
    euroId            = ...
    poundId           = ...
```

FIGURE 1. The iData **money converter program.**

2 DESIGN RATIONALE

The unified semantic model has been inspired mainly by our earlier work on the GEC Toolkit and iData Toolkit. The model consists of two levels: at the bottom level we have the main building blocks, the interactive elements, also known as *objects*. These objects represent the interactive elements of an interactive application. At the second level, we have *programs* that create collections of objects and interconnect them: the state of objects can depend on the state of other objects. Briefly, we have made the following design decisions for the unified semantic model:

- Objects correspond with primitive interactive elements such as text input boxes, buttons, scroll bars, and so on but also with arbitrary compositions of interactive elements. In this paper we abstract from all issues related with interactive elements, such as their number, layout, and so on. Instead, we will *model* them by a (composite) data type, that represents the state of the object. From our earlier work on the GEC Toolkit and iData Toolkit, we know that we can generate arbitrarily complex user interfaces from values with generic programming techniques.

- User manipulations of objects are modelled as *editing operations* on the state of these objects. This provides us with an object style semantics, in which user actions update the state of an existing object rather than creating a new object (a functional style semantics).

- Objects can be *shared* within our model. Shared objects have multiple occurrences within an interactive application, but they share the same state. Sharing of objects turns out to provide us with concise and intuitive ways of handling feedback (Sect. 3).

- Programs create and interconnect objects. The expressive power of functions provides programmers with a great deal of freedom to manipulate and create intricate interfaces. For the unified semantic model, we will not yet tackle functional expressions in general. Currently, we restrict ourselves to the Arrow framework. The Arrow framework imposes a typed discipline on top of the untyped unified semantic model.

3 THE UNIFIED SEMANTIC MODEL

In this section we present the unified semantic model. We express the semantic model in Clean. This has the following advantages:

- It helps us to detect and remove shallow errors in the semantic definition as we can rely on the language compiler to detect these mistakes at compile time.

- We can *test* the semantic model with concrete examples by defining a meaning function that can actually compute the output of an interactive program, given a scenario.

- We can use the *proof tool* Sparkle [7] to guide us in constructing a complete proof of the desired properties. We intend to do this in a similar style as done in [8]. At this stage we have not yet completed the proofs, but already the use of Sparkle has proven productive in structuring the semantic definitions.

The main disadvantage of this approach is that the semantics needs to deal with collections of elements of arbitrary types. In order to escape typing problems, we have chosen to use a flat type to represent types with:

```
:: STATE = INT !Int | REAL !Real | TUPLE !STATE !STATE
```

This does not cause loss of generality of the results, but it does require extra attention by ourselves to make sure that no intentional type errors are introduced that would otherwise be intercepted by the type checker.

The structure of the semantic model is as follows. We first present the arrows in which programs are to be represented (Sect. 3.1). Based on these arrows, we first construct a rather straightforward functional interpretation semantics of these arrows (Sect. 3.2). This interpretation is a natural semantics of pure server side web applications. Next, we construct a more complex, object oriented, semantics of these arrows (Sect. 3.3). This semantics fits closely to desktop applications. Finally, we demonstrate that these two semantics are equivalent in the traditional functional sense (Sect. 3.4).

3.1 Programs as Arrows

We start with a representation of arrows in which programs will be constructed.

```
::  A   = Arr     !(STATE → STATE)
        | (>>>>)  infixr 1 !A !A
        | First   !A
        | Loop    !A
        | Edit    !UID
```

The first four data constructors correspond with the standard arrow combinator functions `arr`, $>>>$, `first`, and `loop`. The `Edit` data constructor is specific for interactive applications: it should create an interactive element that presents to the user a state and that can be edited by the user. We arrange *sharing* of these editors by tagging them with values of type `UID`:

```
::  UID :== Int.
```

Whenever the same integer is used, the corresponding editors are shared. We assume that the `UID` parameter of an `Edit` data constructor is a positive integer, and that the intended types of editors with the same `UID` value are equal.

As an example, the program in Fig.1 is represented as:

```
c0 = feedback (Edit euroId  >>>> Arr toPound)
              (Edit poundId >>>> Arr toEuro)
where euroId               = 1
      poundId              = 2
      toPound (REAL euros)  = REAL (euros  * exchangerateEuroPound)
      toEuro  (REAL pounds) = REAL (pounds * exchangeratePoundEuro)
```

Within the semantic model, the need arises to generate `UID` values itself. In order to prevent clashes with the values from the programs, these will always be negative. These `UID` values are generated with the trivial functions:

```
initFreshUID :: UID
initFreshUID = -1

nextFreshUID :: !UID → UID
nextFreshUID uid = uid-1
```

Both web and desktop applications need to maintain state that persists between distinct web page creations and handling of events. For web applications this can be done by serializing the required states in the web page, and for desktop applications this is usually done by storing the states within the interactive elements themselves. In any case, the semantic model needs to have these states. In the model, the collection of states is modelled by means of a straightforward association set:

```
::  ASet k v    = ASet ![AElt k v]
::  AElt k v    = AElt !k !v

key             :: !(AElt k v) → k
val             :: !(AElt k v) → v
```

```
instance zero (ASet k v)
instance == (ASet k v) | == k

isEmptyASet     ::             !(ASet k v) → Bool
isMemberAElt    :: !k          !(ASet k v) → Bool     | == k
findAElt        :: !k          !(ASet k v) → AElt k v | == k
replaceAElt     :: !(AElt k v) !(ASet k v) → ASet k v | == k
replaceOrAddAElt:: !(AElt k v) !(ASet k v) → ASet k v | == k
```

The definitions of these access functions are such that a set never contains two elements with the same key value.

Within an arrow program, events can be created for only the editors that are defined by the `Edit` data constructor. Because we consider interactive elements to be editors of values of their state, an event can be modelled by a pair of the `UID` value of the editor, and a new value for its `STATE`:

```
::  Event :== ( !UID, !STATE )
```

Obviously, we assume that the intended type of the state is equal to the actual type of the state of the indicated editor.

Finally, given a program that is represented by an arrow expression of type `A`, we can extract the initial states of this program:

```
arrowSTATES :: !A !STATE → (!STATE,!ASet UID STATE)
arrowSTATES f a   = (b,inits)
where (b,inits,_) = arrowSTATES' f (a,zero,initFreshUID)

      arrowSTATES' :: !A !(!STATE,!ASet UID STATE,!UID)
                      → (!STATE,!ASet UID STATE,!UID)
      arrowSTATES' (Arr f) (a,states,uid)
          # states           = replaceOrAddAElt (AElt uid (f a)) states
          = (f a,states,nextFreshUID uid)
      arrowSTATES' (f>>>>g) (a,states,uid)
          = ((arrowSTATES' g) o (arrowSTATES' f)) (a,states,uid)
      arrowSTATES' (First f) (v=:(TUPLE a c),states,uid)
          # states           = replaceOrAddAElt (AElt uid v) states
          # (b,states,uid_1)  = arrowSTATES' f (a,states,nextFreshUID uid)
          = (TUPLE b c,states,uid_1)
      arrowSTATES' (Loop f) (b,states,uid)
          # states           = replaceOrAddAElt (AElt uid b) states
          # (TUPLE c d,states,uid_1)
                             = arrowSTATES' f
                                 (TUPLE b undef,states,nextFreshUID uid)
          = (c,states,uid_1)
      arrowSTATES' (Edit id)  (a,states,uid)
          = (a,replaceOrAddAElt (AElt id a) states,uid)
```

3.2 Interpretation Semantics

In this section we present a more or less straightforward interpretation semantics for arrow expressions of type A that have been introduced in Sect. 3.1. This semantics is well suited to capture the nature of pure web server based applications. The interpretation is a state based functional version:

```
::  CircF  e a b :== (CircSt e a) → (CircSt e b)
::  CircSt e a   :== (a,e)

(>>>) infixr 1 :: (CircF e a b) (CircF e b c) → (CircF e a c)
(>>>) f g = g o f
```

The environment that these functions will manipulate consists of the states of the objects, as discussed above, as well as the current event that is inspected:

```
::  Env :== ( !ASet UID STATE, !Maybe Event )
```

Initially, the (Maybe Event) component is set to (Just e) for an event e. It is set to Nothing by the concrete object for which the event was intended (see interpretation of object below). For each of the components of the arrow program, we define a functional interpretation:

```
arrowFunctional :: !A → CircF Env (Maybe STATE) (Maybe STATE)
```

Please note that one might expect (CircF Env STATE STATE) as the type of functionals. However, this presumes that values are always sent through the complete program. This is in general not the case for interactive programs. Any of the interactive elements that are 'in front' of the interactive element that has been manipulated should not be changed. For this reason, when handling an event, the system starts with a Nothing value.

Here is the functional interpretation of an arrow program:

```
arrowFunctional :: !A → CircF Env (Maybe STATE) (Maybe STATE)
arrowFunctional (Arr   f)   = arr_W f
arrowFunctional (f >>>> g)  = arrowFunctional f>>>arrowFunctional g
arrowFunctional (First f)   = first_W (arrowFunctional f)
arrowFunctional (Loop f)    = loop_W (arrowFunctional f)
arrowFunctional (Edit uid)  = object (uid,const id)
```

The functions arr_W, first_W, and loop_W are the standard functional definitions, but need to handle these awkward STATE values instead of straight values and tuples:

```
arr_W :: !(STATE → STATE) → CircF Env (Maybe STATE) (Maybe STATE)
arr_W fun
   = λ(ma,env) → case ma of
                   Just a  → (Just (fun a),env)
                   nothing → (Nothing,env)

first_W :: !(CircF Env (Maybe STATE) (Maybe STATE))
          → CircF Env (Maybe STATE) (Maybe STATE)
first_W f
```

```
   = λ(m_ac,env) → case m_ac of
                     Just (TUPLE a c) → let (mb,env') = f (Just a,env)
                                        in  (Just (TUPLE (fromJust mb) c),env')
                     nothing          → (Nothing,env)

loop_W :: !(CircF Env (Maybe STATE) (Maybe STATE))
          →CircF Env (Maybe STATE) (Maybe STATE)
loop_W f = loop' f
where
    loop' f (mb,env)
        = case mb of
            Just b  → let  (mt,env_1)   = f (Just (TUPLE b d),env)
                           (c,d)        = toTuple (fromJust mt)
                      in   (Just c,env_1)
            nothing → (Nothing,env)
```

Of more interest is the functional interpretation of the `Edit` alternative.

```
::  F :== (ASet UID STATE) STATE → STATE

object :: !(!UID,!F) !(!Maybe STATE,!(!ASet UID STATE,Maybe Event))
                     → (!Maybe STATE,!(!ASet UID STATE,Maybe Event))
object (uid,f) (ma,(states,event))
    = case (ma,event) of
        (_,Just (id,v))
            | uid==id   = let b = f states v
                          in  (Just b, (replaceAElt (AElt uid v) states,Nothing))
        (Just a,Nothing)= let b = f states a
                          in  (Just b, (replaceAElt (AElt uid a) states,Nothing))
        otherwise       = (Nothing,(states,event))
```

This definition shows that the object to which an event is directed (first alternative of the case distinction), takes over the value of the event and updates its state in the environment. In addition, it signals the fact that it has handled the event by setting the event to `Nothing`. The new value that should be propagated through the program depends on this new value. The second case states that whenever such a value has been set, that the interactive elements that depend on this value use this value. Finally, the third case states that elements that occur before the modified element are not affected.

The meaning of a functional style program is to apply the functional interpretation to each and every event that is directed to the program. Of course, first the program needs to be initialized. This is expressed by the following, general, `meaning` function:

```
meaning :: !(A (!Event,!Env) → (!STATE,!Env))
           !A !STATE ![Event] → (![STATE],!Env)
meaning mf f a es   = ([b:bs],env)
where (b,states)    = arrowSTATES f a
      (bs,env)      = mapSt (mf f) (es,(states,Nothing))
```

A functional style interpretation therefore has the following meaning:

```
meaning_W :: (A STATE [Event] → (![STATE],!Env))
meaning_W = meaning handle_event_W

handle_event_W :: !A !(!Event,!Env) → (!STATE,!Env)
handle_event_W f (e,(states,_))
   = let (mb,env_1) = arrowFunctional f (Nothing, (states,Just e))
     in  (fromJust mb,env_1)
```

3.3 Object Semantics

In this section we present an object based semantics of arrow expressions of type A. It turns out that this is a much more complicated semantic model than the functional interpretation semantics as given above in Sect. 3.2. This semantics is well suited for desktop GUI applications that distributes the interactive behavior over a collection of persistent widgets with state and callback functions.

It is useful break up the concept of a widget with state and callback functions into its state and its callback functions. The environment Env already contains all states of all elements, so what remains are the callback functions. The key idea is to present programs of type A in a different way, namely as the sequence of callback functions that they would call in case of occurring events. We call this a *wiring*:

```
:: Wiring  :== [Wire]
:: Wire    :== ( !UID, !F )
```

Every program of type A can be transformed into a linearized form of type Wiring:

```
arrowWiring :: !A → Wiring
arrowWiring f             = fst (arrowWiring' f ( [], initFreshUID ))
where
    arrowWiring' :: !A !(!Wiring,!UID) → (!Wiring,!UID)
    arrowWiring' (Arr fun) (wiring,uid)
        = (appendWire (uid,const fun) wiring,nextFreshUID uid)
    arrowWiring' (f >>>> g) (wiring,uid)
        = ((arrowWiring' g) o (arrowWiring' f)) (wiring,uid)
    arrowWiring' (First f) (wiring,uid)
        # wiring          = appendWire (uid,F_1) wiring
        # (wiring,uid_1)  = arrowWiring' f (wiring,nextFreshUID uid)
        # wiring          = appendWire (uid,F_2) wiring
        = (wiring,uid_1)
    where
        F_1 _ (TUPLE a c) = a
        F_2 states b      = let (TUPLE a c) = val (findAElt uid states)
                            in  TUPLE b c
    arrowWiring' (Loop f) (wiring,uid)
        # wiring          = appendWire (uid,F_1) wiring
        # (wiring,uid_1)  = arrowWiring' f (wiring,nextFreshUID uid)
        # wiring          = appendWire (uid,F_2) wiring
        = (wiring,uid_1)
    where
```

```
        F_1 _ b           = TUPLE b undef
        F_2 b (TUPLE c d) = c
    arrowWiring' (Edit eid) (wiring,uid)
        = (appendWire (eid,const id) wiring,uid)

appendWire :: !Wire !Wiring → Wiring
appendWire w wires = wires ++ [w]
```

This definition states that every element of an arrow is mapped to an object with state and callback function. We only have objects in this semantics. For this reason, the meaning of an object based semantics is to let the objects that can be derived from an arrow based program of type `A` (done by `arrowWiring`) handle each and every event. For this we can reuse the `object` function that we have already defined:

```
meaning_D :: (A STATE [Event] → (![STATE],!Env))
meaning_D       = meaning (handle_event_D o arrowWiring)

handle_event_from_start_D :: !Wiring !(!Event,!Env) → (!STATE,!Env)
handle_event_from_start_D wiring (event,env)
    # (mb,env)  = foldl (flip object) (Nothing,env) wiring
    = (fromJust mb,env)
```

3.4 Equivalence of Interpretation vs Object Semantics

In the above two sections, we have introduced two meanings of arrow programs of type `A`:

```
meaning_W :: (A STATE [Event] → (![STATE],!Env))
meaning_W = meaning handle_event_W

meaning_D :: (A STATE [Event] → (![STATE],!Env))
meaning_D = meaning (handle_event_D o arrowWiring)
```

We need to prove that `meaning_W = meaning_D`. As stated earlier, we want to do the proof in the proof assistant Sparkle. Unfortunately, at this stage we have not yet finished the proof. The proof proceeds by case distinction on the structure of `A`.

4 UNIFIED MODEL PROPERTIES

We define a number of semantic equivalence relations:

Definition 1 (Initial I/O Equivalence)
Two systems are said to be *initially I/O equivalent* if for the initial event the corresponding output is the same.

Definition 2 (Behavioural I/O Equivalence)
Two systems are said to be *behaviorally I/O equivalent* if for each list of input events the list of corresponding outputs is the same.

Definition 3 (Event Identical)
Two systems are said to be *event identical* w.r.t. to a set of *UntypedId* if for each list of input events the collection of the *Obj*ects that correspond with the ids are identical.

Definition 4 (Identical)
Two systems are said to be *identical* if the collection of their *Obj*ects is identical up to a global renaming of their id_{self}'s.

In the unified model it is straightforward to prove that the interpretation semantics as given in Sect. 3.2 satisfies the usual collection of Arrow laws:

Definition 5 (Arrow Laws)

$$
\begin{array}{rcl}
arr\ id \ggg f & = f = & f \ggg arr\ id \\
f \ggg (g \ggg h) & = & (f \ggg g) \ggg h \\
arr\ (f \ggg g) & = & arr\ f \ggg arr\ g \\
arr\ (g\ o\ f) & = & arr\ f \ggg arr\ g \\
 & & \\
first\ (f \ggg g) & = & first\ f \ggg first\ g \\
first\ f \ggg arr\ fst & = & arr\ fst \ggg f \\
first\ (arr\ f) & = & arr\ (first\ f) \\
first\ f \ggg arr\ (id \times g) & = & arr\ (id \times g) \ggg first\ f \\
first\ (first\ f) \ggg arr\ assoc & = & arr\ assoc \ggg first\ f \\
 & & \\
loop\ (first\ h \ggg f) & = & h \ggg loop\ f \\
loop\ (f \ggg first\ h) & = & loop\ f \ggg h \\
loop\ (f \ggg arr\ (id \times k)) & = & loop\ (arr\ (id \times k) \ggg f) \\
loop\ (loop\ f) & = & loop\ (arr\ assoc^{-1} \ggg f \ggg arr\ assoc) \\
second\ (loop\ f) & = & loop\ (arr\ assoc \ggg second\ f \ggg arr\ assoc^{-1}) \\
loop\ (arr\ f) & = & arr\ (simple\_loop\ f)
\end{array}
$$

where *simple_loop* is the 'stateless' variant of our *loop* combinator:

$simple\_loop\ ::\ ((b,d) \to (c,d)) \to b \to c$
$simple\_loop\ f\ b = \mathbf{let}\ (c,d) = f\ (b,d)\ \mathbf{in}\ c$

Proving the Arrow laws for the object semantics directly is very hard. Even if this can be done, then the result is weaker than that for the interpretative semantics, because the laws must necessarily be *behavioural I/O equivalence* instead of equality (because (`Arr id>>>>f`)) will always create one more state in the environment than `f` by itself). However, using the equivalence result of Sect. 3.4 we need not go through the trouble of proving the Arrow laws directly.

Arrow-structured programs have the further property that every user manipulation ends in a stable state of the application; i.e. events always arrive at the sink element. This is an important property for both desktop GUI applications (the

program ‘freezes’ if this property does not hold) as well as web applications (no new page is computed, causing the browser to fail). This depends solely on the functions that are provided by the programmer when using the *arr* combinator.

5 CASE STUDY

In Sect. 1 we have presented a small money exchange program that has two intricately mutually interconnected objects (Fig. 1). Consider the following variations of this little program:

```
c0 = feedback (edit euroId  >>> arr toPound)
              (edit poundId >>> arr toEuro)
c1 = feedback (edit euroId)
              (arr toPound>>>edit poundId>>>arr toEuro)
c2 = c0>>>arr toEuro
c3 = c1>>>arr toPound
```

For this program, it is reasonable to assume $\texttt{toPound} = \texttt{toEuro}^{-1}$. All programs are event identical w.r.t. $\{\textit{euroId}, \textit{poundId}\}$. These are the objects that are visible to the user, so from a user's perception it does not matter which one to use. Programs $\texttt{c}_0$ and $\texttt{c}_3$ are identical, hence they are also initial I/O and behavioral I/O equivalent. Programs $\texttt{c}_1$ and $\texttt{c}_2$ are Arrow equivalent, hence also initial and behavioral I/O equivalent.

We can use the framework to explain what happens if we drop the assumption that the conversion functions are each others inverse, so $\texttt{toPound} \neq \texttt{toEuro}^{-1}$. Editing the pound editor behaves as before, but every event $(\textit{euroId}, x)$ results in displaying the value $(\textit{toPound}\ x)$ in the *poundId* object, but also modifies the entered value in the *euroId* object into $(\textit{toEuro}\ \ (\textit{toPound}\ x))$.

6 RELATED WORK

This work is about providing a semantic framework that can handle both interactive applications for the desktop as well as web applications, or even a mixture of these systems. Due to the radically different nature of these two paradigms, such a semantic framework needs to be sufficiently abstract without loosing the opportunity to reason about the application logic.

In this project we have chosen to build the semantic model on Arrows. The advantage of using a functional style formalism is that integration of computation can be done within the framework, using functions. Other projects, such as *Fruit* [6] and *Fran* [11] have taken this route as well. In these systems Arrows were also necessary to eliminate subtle performance problems. In our case, we use them chiefly to structure our programs in order to facilitate reasoning.

Another way of modelling interactive programs is to regard them as collections of communicating processes. From this point of view, it seems to be natural to provide a model in terms of a *process algebra*. There is a wide variety of process algebras available, such as CCS (Calculus of Communicating Systems) [13],

CSP (Communicating Sequential Processes) [10], ACP (Algebra of Communicating Processes) [5], and μCRL (micro Common Representation Language) [9]. Especially the latter might be interesting in this context because it augments ACP with algebraic data types in a spirit that is very similar to functional programming. In general, the fine grained control over concurrency that is usually provided by process algebraic models is not necessary when dealing with interactive applications. It is our opinion that the dependency between interactive elements can be dealt with more suitably in a functional style than an inherent concurrent style.

7 CONCLUSIONS AND FUTURE WORK

In this paper we have identified the trend that interactive applications that are developed for either the desktop or the web tend to take over functionality that used to be the exclusive domain of either paradigm. This implies that in order to reason about interactive programs, we need a suitable model that can capture aspects of both paradigms. In this paper we have presented a first step to such a unified semantic model.

We have restricted ourselves deliberately to programs that are expressed in the Arrow framework, because we expected that this would facilitate reasoning over programs. In our experience with the two toolkits, it is very natural to write non-Arrow programs that use the full expressiveness of the functional host language. We want to investigate if reasoning in a much less restricted framework still allows feasible reasoning.

REFERENCES

[1] P. Achten, M. van Eekelen, and R. Plasmeijer. Generic Graphical User Interfaces. In G. Michaelson and P. Trinder, editors, *Selected Papers of the 15th Int. Workshop on the Implementation of Functional Languages, IFL03*, volume 3145 of *LNCS*. Edinburgh, UK, Springer, 2003.

[2] P. Achten, M. van Eekelen, and R. Plasmeijer. Compositional Model-Views with Generic Graphical User Interfaces. In *Practical Aspects of Declarative Programming, PADL04*, volume 3057 of *LNCS*, pages 39–55. Springer, 2004.

[3] P. Achten, M. van Eekelen, R. Plasmeijer, and A. van Weelden. Automatic Generation of Editors for Higher-Order Data Structures. In Wei-Ngan Chin, editor, *Second ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *LNCS*, pages 262–279. Springer, 2004.

[4] P. Achten, M. van Eekelen, R. Plasmeijer, and A. van Weelden. GEC: a toolkit for Generic Rapid Prototyping of Type Safe Interactive Applications. In *5th International Summer School on Advanced Functional Programming (AFP 2004)*, volume 3622 of *LNCS*, pages 210–244. Springer, August 14-21 2004.

[5] J. Baeten and W. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts. in Theoretical Computer Science*. Cambridge University Press, 1990.

[6] A. Courtney and C. Elliott. Genuinely Functional User Interfaces. In *Proceedings of the 2001 Haskell Workshop*, September 2001.

[7] M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - Sparkle: A functional theorem prover. In T. Arts and M. Mohnen, editors, *The 13th International Workshop on Implementation of Functional Languages, IFL 2001, Selected Papers*, volume 2312 of *LNCS*, pages 55–72, Stockholm, Sweden, 2002. Springer.

[8] M. Dowse, A. Butterfield, and M. van Eekelen. Reasoning About Deterministic Concurrent Functional I/O. In C. Grelck, F. Huch, G. Michaelson, and P. Trinder, editors, *Proceedings Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, volume 3474 of *LNCS*, pages 469–480. Springer, September 8-10 2004.

[9] J. Grootte and M. Reniers. *Algebraic Process Verification*, chapter 17, pages 1151–1208. Elsevier Science B.V., 2001.

[10] C. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall International, 1985.

[11] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, Robots, and Functional Reactive Programming. In J. Jeuring and S. Peyton Jones, editors, *Advanced Functional Programming, 4th International School*, volume 2638 of *LNCS*, Oxford, 2003. Springer.

[12] J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[13] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer Verlag, 1980.

[14] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, Sept. 2001.

[15] R. Plasmeijer and P. Achten. Generic Editors for the World Wide Web. In *Central-European Functional Programming School*, Eötvös Loránd University, Budapest, Hungary, Jul 4-16 2005.

[16] R. Plasmeijer and P. Achten. The Implementation of iData - A Case Study in Generic Programming. In A. Butterfield, editor, *Proceedings Implementation and Application of Functional Languages, 17th International Workshop, IFL05*, Dublin, Ireland, September 19-21 2005. Technical Report No: TCD-CS-2005-60.

[17] R. Plasmeijer and P. Achten. iData For The World Wide Web - Programming Interconnected Web Forms. In *Proceedings Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, volume 3945 of *LNCS*, Fuji Susono, Japan, Apr 24-26 2006. Springer Verlag.