An Arrow Based Semantics for Interactive Applications

Peter Achten[1], Marko van Eekelen[2], Maarten de Mol[1], 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,marko,M.deMol,rinus}@cs.ru.nl

Abstract

Interactive applications, whether they run on the desktop or as a web application, can be considered to be collections of interconnected editors that allow users to manipulate data. This is the view that is advocated by the *GEC Toolkit* and the *iData Toolkit*, which offer a high level of abstraction to desktop and web GUI applications respectively. Special features of these toolkits are that editors have *shared, persistent* state, and that they *handle events* individually. In this paper we cast these toolkits within the *Arrow* framework and present a single, unified semantic model that handles shared state and the event handling behavior. We study the definedness properties of the semantic model, and of editors in particular. We demonstrate that this is important when using the *Arrow* combinators with primitive combinators that have different definedness properties. We use the proof assistant *Sparkle* to create and check the proofs. In the process, we identify a missing tactic in *Sparkle*.

category: *Research Article*

1 INTRODUCTION

Graphical User Interfaces in current day applications are either based on a desktop widget set, or they use web technology. Although one can still see rather quickly whether an application is constructed with which technology, we signal the trend that these applications are gradually taking over each other's functionality. Examples are the integration of browsing behavior and back/forward buttons within desktop GUI applications, and local state and local rendering within multiple windows within web applications.

The programming paradigms of desktop GUIs and web applications are radically different: the desktop GUI paradigm basically utilizes a state based callback evaluation model and *widgets* to visualize the interface, whereas the web programming paradigm relies on the stateless *http* request-response sequence model and *HTML* for visualization purposes.

In the past we have developed toolkits for programming desktop GUIs (*GEC Toolkit* [1, 2, 3]) and web applications (*iData Toolkit* [16, 17]). Despite their radically different *implementations*, the programs that are written in them are *identical* on the top level. It is our goal to create a single, unified, framework for developing, and reasoning about, interactive applications. Such a single, unified, framework,

must necessarily abstract from details. We have used *generic programming techniques* [9, 4] to obtain a suitable level of abstraction.

In this paper we consider interactive applications to be a *collection* of *interconnected editors* with which the user can alter *data* of arbitrary *type*. Generic programming allows us to abstract from earthly details such as programming widget sets vs *HTML* code, event handling models, rendering issues, and so on. Instead, we can reason about interactive objects as having a well-typed *state*. User actions are then edit operations on that state, i.e. the request to change the current value of an editor into a new value of the same type.

Programs that we have developed in the past with these toolkits have used the full expressive power of the functional host language, in our case *Clean*. In order to facilitate reasoning we restrict ourselves to a combinator approach, based on the *Arrow* framework by Hughes [12], and its extension with *loops*, by Paterson [15].

The operational semantics of our framework is an *event handling* system, where the events model user edit operations on editors. This is different from the standard approach to *Arrow* based systems, where the value of a system is determined by evaluating the *Arrow* system from the start until the end. Event based systems necessarily need to 'break into' the *circuit* that is created by the arrow expression, because an event causes an effect only after the targeted editor.

Another unusual feature of interactive applications is *sharing* editor states. Editors are identified objects. Two (or more) editor objects with the same identifier conceptually refer to the same object, and hence, the same state. In the realization, any two shared editors are mapped to a single appearance in the concrete user interface that is presented to the user. In this way, complex interconnection patterns can be constructed. Despite these differences, we show that our framework satisfies the standard set of laws that are imposed on *Arrow*s (with loops).

Part of these proofs have been conducted with *Sparkle* [7], the interactive proof assistant that comes with *Clean*. We aim to handle all proofs with *Sparkle*. It turns out that this is not yet possible, because currently, *Sparkle* is not able to handle *cyclic let definitions*. These are crucial in the definition of the *loop* combinator, and hence, we are not able to prove the loop laws with *Sparkle*. Despite this drawback, we have noticed that the use of a proof tool helps us to gain insight in the properties of the semantic framework. At several cases, the proof in progress pointed to situations that were clearly undesired, but that had escaped our scrutiny.

The remainder of this paper is structured as follows. We first present the two toolkits in Sect. 2 and give a small, yet intricate, example of an interactive application. In Sect. 3 we give the semantic model of the two toolkits. The *Arrow* laws and definedness properties of the semantic model are discussed in Sect. 4. In Sect. 5, we present a variation of the example application and prove the equivalence between their semantic models. Related work is presented in Sect. 6 and we end with conclusions in Sect. 7.

2 THE GEC AND IDATA TOOLKITS

The basic building block in the *GEC Toolkit* and the *iData Toolkit* is an *editor*. An editor is a typed unit that provides the application user with a GUI (desktop in case of the *GEC Toolkit*, and *HTML* in case of the *iData Toolkit*) that allows her to edit values of that given type only. In the *GEC Toolkit* the editor is known as a *GEC* (Graphical Editor Component), and in the *iData Toolkit* it is known as an *iData* element. In both toolkits we have a single, generic function that creates editors, given an initial value of the desired type, and some kind of identification. If we ignore the details of creating editors in both toolkits, then they both offer the following core function to create an editor:

```
edit :: ID d *Env → (Edit d,*Env) | gFuncs d
```

Here, *ID* is some kind of identification value (typically an augmented `String`), and *Env* is some kind of opaque environment that takes care of the administration (a desktop GUI environment (`PSt ps`) in case of the *GEC Toolkit*, and an *HTML* environment `HSt` in case of the *iData Toolkit*). The resulting editor can be used in the program to display the editor to the user. In both toolkits the function uses a collection of *generic* functions that are collected in the type class `gFuncs`. These generic functions are used to create a GUI element with which the user can edit values of given type `d`.

An interactive application is a collection of *interconnected* editors. The toolkits allow a 'freeform' style and a 'disciplined' style. In the freeform style, any environment function of the correct type is admissible to define the interconnection relation. This provides the application programmer with a great deal of freedom, as she can use all standard functional programming techniques to create an interactive application. The disciplined style is based on the *Arrow* combinator approach by Hughes, and has been extended with loops by Paterson. In this paper we have chosen to use the disciplined style.

The *Arrow* class contains three combinator functions, viz. `arr`, `>>>`, and `first`. Paterson extends this set with the `loop` combinator. In our toolkits we integrate the editor creation function as the final combinator. Expressed as a *Clean* type constructor class we have:

```
class Arrow a where
    arr   :: (b → c) → a b c
    (>>>) :: (a b c) (a c d) → a b d
    first :: (a b c) → a (b,d) (c,d)
    loop  :: (a (b,d) (c,d)) → a b c
    edit  :: ID → a d d
```

It should be observed that `edit` as an *Arrow* combinator function has no need for an initial value parameter as it will get one within the *Arrow* context.

The behavior of desktop GUI applications is typically event driven, whereas the behavior of web applications is triggered by the user entering data in forms and submitting changes to the application at the server side. In the toolkits we

unify these operational behaviors by defining the behavior of our applications as *edit driven*. This means that whenever the user *edits* the current value of any of the editors that is currently available, the application responds with recomputing the states of all subsequently interconnected editors.

To illustrate the use of these *editor Arrow* combinators Fig. 1 displays the key fragment of a money-converting application. The function `converter` introduces two interactive elements, labeled `euroId` and `usdId`, that are mutually interconnected in such a way that if either of them is altered by the user, then the other element responds with the amount of money expressed in the local currency. The program exploits the fact that editors have shared state: all occurrences of (`edit` *id*) within an *Arrow* expression refer *to the same* editor, and hence the same state. Hence, (`feedback f g`) creates a mutual interconnection between the editors within `f` and `g` simply by repeating `f`.

```
:: USDollars = { dollars :: Real }
:: Euros     = { euros   :: Real }

converter
    = feedback (edit euroId >>> arr toUSD)
               (edit usdId >>> arr toEuro)
where
    toUSD  { euros }   = { dollars = euros * exchangerateEuroUSD }
    toEuro { dollars } = { euros = dollars * exchangerateUSDEuro }
    euroId             = ...
    usdId              = ...

feedback f g           = f >>> g >>> f
```

FIGURE 1. The *iData* money converter program.

3 THE SEMANTICS OF EDITOR ARROWS

In this section we present the complete semantic model of editor arrows. It has been *fully defined within the programming language Clean.*

Advantages to this approach are that: the language aids us in detecting shallow mistakes such as typing errors and type errors that lead to undefined entities; the final model is executable, and can be tested in an ad-hoc way or more rigidly with the *G∀ST* test tool [13]; we can ensure a short distance between the semantic model and real programs; and last but not least, we can use the proof tool assistant *Sparkle* to prove the *Arrow* laws.

There are also *disadvantages* to this approach: we are forced to implement everything of the semantic model; there is lack of standard mathematical concepts

such as sets; and the strong type system of the language sometimes hampers intuitive definitions and forces us to define work-around solutions.

Despite the disadvantages, we have chosen to use *Clean* as the modeling language. We want to emphasize that in the remainder of this paper we assume that every instance of the semantic model is always derived from a well-typed source program, using the editor *Arrow* definition that was given in Sect. 2.

Editors are identified objects that contain a state value of some type. In our semantic model the identifier type `ID` is a synonym type of integers. Modeling collections of arbitrarily typed states is a nuisance within a strongly typed language. Solutions can be forged using exotic constructs such as existential types and phantom types, but this will hinder reasoning. For this reason we take a pragmatic approach without loss of generality. A state is either basic or a pair structure. If it is basic it could be any type, but we restrict ourselves to integers. Hence we obtain the following definitions:

```
:: EDITSTATES  :== [(ID,STATE)]
:: ID          :== Int
:: STATE       =   Basic Int | Pair STATE STATE
```

For readability, we use standard tuple notation `(a,b)` instead of `(Pair a b)`. `EDITSTATES` models the set of all editors, along with their current state values. The function `write` updates this set and `read` reads the current value of the indicated editor.

```
write :: !EDITSTATES !(!ID,!STATE) → EDITSTATES
write sts (i,v)       = map (set (i,v)) sts
where set :: !(!ID,!STATE) !(!ID,!STATE) → (!ID,!STATE)
      set (i,v) (j,w)
          | i == j    = (i,v)
          | otherwise = (j,w)

read :: !EDITSTATES !ID → STATE
read [(id,v):idvs] id‘
    | id == id‘       = v
    | otherwise       = read idvs id‘
```

An editor provides the user with a GUI to allow her to alter the current value of its state that is displayed. Such an event is modeled with a pair of the identifier of the editor and the new value of the state of the editor.

```
:: EVENT       =   Event ID STATE
```

The current state of an editor can be altered either via an event, as described above, or because an editor that occurs earlier within the interconnection relation has been edited. The primitive `edit` combinator function defines this behavior. It has type `:: !ID !EVENT !(D a) → (D a)`. Its first argument is the identifier of the editor; the second argument is the event to which the system should respond; the last argument is the domain `D a` on which all combinators operate. It is a triplet of all editor states (of type `EDITSTATES`), the current value (of type `a`) that is

passed along the *Arrow* expression, and a flag (the boolean synonym type `EDITED` indicating that an edit action has taken place).

By identifying the required type for the `edit` combinator, we also identified the type to use as the basic type for the editor *Arrow* combinators which have no `ID`. This type is `EditF a b` which is a synonym for `:: !EVENT !(D a) → D b`. This type `EditF a b` is also called ***the** Arrow* type.

```
:: D a          :== (EDITSTATES, a, EDITED)
:: EDITED       :== Bool
:: EditF a b    :== !EVENT !(D a) → D b

edit :: !ID !EVENT !(D a) → D a
```

We can now define the behavior of an editor that is defined with identifier `id`:

```
edit :: !ID !EVENT !(D a) → D a                                    1.
edit id (Event id' v') (sts, v, edited)                            2.
    | evalSTATE v                                                  3.
        | edited    = (write sts (id,v), v, True)                  4.
        | id == id' = (write sts (id,v'),v',True)                  5.
        | otherwise = (sts,     read sts id,False)                 6.
```

Editors display their current state value. Consequently, the source of these values need to be fully evaluated values. This is expressed by the guard in line 3, which enforces the value `v` to be in normal form. If an editor that occurs earlier within the interconnection relation has been edited, then the alternative at line 4 is satisfied. In that case, the editor destructively updates its current state with the current value that is passed along the interconnection relation. If no editor was edited yet, but the event is targeted for this editor (their identification values are equal), then the alternative at line 5 is satisfied. In that case, the editor destructively updates its current state with the value in the event. This value is also the value that is passed along to the remaining editors that are interconnected with this editor. They need to alter their states because this editor has been edited, and therefore the `EDITED` flag is set to `True`. Finally, if no editor was edited yet, and neither was this editor, then the value that should be passed along the interconnected editors is the current state of this editor (line 6).

The `edit` combinator introduces a number of subtle cases that need careful consideration. It is for instance not allowed to swap editors:

$$\texttt{edit i>>>edit j} \neq \texttt{edit j>>>edit i} \; (\texttt{i} \neq \texttt{j})$$

This is caused by the fact that events 'break into' the *Arrow* structure: an event for editor `j` does not affect the state of editor `i` on the left hand side, but it does on the right hand side. This matter is discussed in Sect. 4.3 where we study a number of editor laws.

In our semantic framework, we can define suitable instances of the *Arrow* member functions as follows[1]:

[1]For presentational reasons we use the familiar type class and instance approach. In the actual

```
instance Arrow EditF where
    arr    :: (a → b) → EditF a b
    arr    f e (sts, v, b) = (sts, f v, b)

    (>>>) :: (EditF b c) (EditF c d) → EditF b d
    (>>>) f g e d          = g e (f e d)

    first  :: (EditF b c) → EditF (b,d) (c,d)
    first  g e (sts, v_d, b)
      = case g e (sts, fst v_d, b) of (sts', v', b') = (sts', (v',snd v_d), b')

    loop   :: (EditF (b,d) (c,d)) → EditF b c
    loop   g e (sts, v, b)
      = let (sts', (v',d), b') = g e (sts, (v,d), b) in (sts', v', b')
```

The definitions above describe precisely how a single edit event is manipulated by the system. What remains to be defined is the initialization of the system and how the system handles arbitrary edit event sequences, which we call *scenarios*. This is given by the following definitions:

```
meaning :: a (EditF a b) EDITSTATES [EVENT] → (EDITSTATES,[b])
meaning a editF sts scenario
    = let (initializedSTATES,b,_) = editF dummyEVENT (sts,a,True)
      in handle_scenario a editF initializedSTATES scenario
where
    handle_scenario :: a (EditF a b) EDITSTATES [EVENT] → (EDITSTATES,[b])
    handle_scenario _ _ sts []
        = (sts,[])
    handle_scenario a editF sts [e:es]
        ♯ (sts,b,_) = editF e (sts,a,False)
        ♯ (sts,bs)  = handle_scenario a editF sts es
        = (sts,[b:bs])
```

We assume the following, reasonable, conditions for each (`meaning init editF sts scenario`):

- `sts` has an entry `(i,v)` for each and every `edit i` within `editF` such that `v` is a value of the correct type.
- `scenario` contains only proper edit events for editors within `editF`.
- `dummyEVENT` is an event which `ID` value does not equal that of any editor within `editF`.

This completes the semantic model of editor *Arrow* combinators. There are a number of final remarks:

semantic model we have defined individual functions with the same, non-synonym types. Also, for readability, we have chosen to present the *intended* type signatures of the member functions, instead of the type correct signatures that have `STATE` for every type parameter of `EditF`.

- Editors with the same identifier may have several occurrences within a definition. Consider for instance:

  ```
  self f i = edit i >>> arr f >>> edit i
  ```

  All occurrences of (`edit` *i*) in the *Arrow* relation relate to a single on-screen occurrence. This particular pattern is rather useful: whenever the user edits its value to `v`, the editor applies the canonization function `f` to that value, and then adapts its state to `f v`.

- Editors in the semantic framework have very limited functionality: they only serve as stores of their states, and they can respond to events. In the *GEC Toolkit* and the *iData Toolkit*, editors usually manipulate incoming values with some function `f`, and outgoing values with some function `g`. This can be expressed directly as `f @> edit i >@ g`, with:

  ```
  (@>) f g = arr f >>> g
  (>@) g f = g >>> arr f
  ```

- We assume that editors with the same identifier have a state of the same type. This is not enforced by our semantic model, but it can be done within the toolkits. We can ignore the issue because we assume that our instances are derived from well-typed programs (see the beginning of this section).

4 SEMANTIC MODEL PROPERTIES

In this section we investigate the properties of the semantic model that has been presented in Sect. 3. We discuss the "classic" *Arrow* laws (Sect. 4.1), the "loop" laws (Sect. 4.2), and the "editor" laws (Sect. 4.3). After that we investigate the definedness properties of the combinators (Sect. 4.4).

4.1 The "classic" arrow laws

Def.1 summarizes the "classic" *Arrow* laws, as introduced by Hughes.

Definition 1 (Arrow Laws)

$$
\begin{array}{rcl}
\mathit{arr}\ \mathit{id} \ggg f & = f = & f \ggg \mathit{arr}\ \mathit{id} \\
f \ggg (g \ggg h) & = & (f \ggg g) \ggg h \\
\mathit{arr}\ (g\ o\ f) & = & \mathit{arr}\ f \ggg \mathit{arr}\ g \\
 & & \\
\mathit{first}\ (f \ggg g) & = & \mathit{first}\ f \ggg \mathit{first}\ g \\
\mathit{first}\ f \ggg \mathit{arr}\ \mathit{fst} & = & \mathit{arr}\ \mathit{fst} \ggg f \\
\mathit{first}\ f \ggg \mathit{arr}\ (\mathit{id} \times g) & = & \mathit{arr}\ (\mathit{id} \times g) \ggg \mathit{first}\ f \\
\mathit{first}\ (\mathit{first}\ f) \ggg \mathit{arr}\ \mathit{assoc} & = & \mathit{arr}\ \mathit{assoc} \ggg \mathit{first}\ f \\
\textbf{where} & & \\
(\times)\ f\ g\ (a,b) & = & (fa, gb) \\
\mathit{assoc}\ ((a,b),c) & = & (a,(b,c))
\end{array}
$$

Note that we omit two laws:

$$
\begin{array}{rcl}
\mathit{arr}\ (f \ggg g) & = & \mathit{arr}\ f \ggg \mathit{arr}\ g \\
\mathit{first}\ (\mathit{arr}\ f) & = & \mathit{arr}\ (\mathit{first}\ f)
\end{array}
$$

In our toolkits `arr` lifts pure functions to the *Arrow* domain. In such a context $\mathit{arr}\ (f \ggg g)$ coincides with $\mathit{arr}\ (g\ o\ f)$. The same reasons apply for the second law.

These laws can be proven completely with *Sparkle* in a few steps each.

4.2 The "loop" arrow laws

Def. 2 summarizes the "loop" laws, as introduced by Paterson.

Definition 2 (Loop Laws)

$$
\begin{array}{rcl}
\mathit{loop}\ (\mathit{first}\ h \ggg f) & = & h \ggg \mathit{loop}\ f \\
\mathit{loop}\ (f \ggg \mathit{first}\ h) & = & \mathit{loop}\ f \ggg h \\
\mathit{loop}\ (f \ggg \mathit{arr}\ (\mathit{id} \times k)) & = & \mathit{loop}\ (\mathit{arr}\ (\mathit{id} \times k) \ggg f) \\
\mathit{loop}\ (\mathit{loop}\ f) & = & \mathit{loop}\ (\mathit{arr}\ \mathit{assoc}^{-1} \ggg f \ggg \mathit{arr}\ \mathit{assoc}) \\
\mathit{second}\ (\mathit{loop}\ f) & = & \mathit{loop}\ (\mathit{arr}\ \mathit{assoc} \ggg \mathit{second}\ f \ggg \mathit{arr}\ \mathit{assoc}^{-1}) \\
\mathit{loop}\ (\mathit{arr}\ f) & = & \mathit{arr}\ (\mathit{simple\_loop}\ f) \\
\textbf{where} & & \\
\mathit{simple\_loop}\ f\ b & = & \textbf{let}\ (c,d) = f\ (b,d)\ \textbf{in}\ c \\
\mathit{second}\ f & = & \mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ f \ggg \mathit{arr}\ \mathit{swap} \\
\mathit{swap}\ (a,b) & = & (b,a)
\end{array}
$$

Currently, the loop laws cannot be proven completely with the aid of *Sparkle*. The `loop` combinator relies essentially on a cyclic let-definition, and it is currently beyond the capacity of *Sparkle* to deal with such definitions. In these cases, we resort to manual proofs. Fig. 2 shows the proof of the *sliding* law. The proofs of the other laws are analogous.

$$
\begin{array}{ll}
 & \underline{loop\ (f \ggg arr\ (id \times k))\ e\ (sts, v, b)} \\
= & \textbf{let}\ (sts', (v', d), b') \quad = \underline{(f \ggg arr\ (id \times k))\ e\ (sts, (v, d), b)} \\
 & \textbf{in}\ (sts', v', b') \\
= & \textbf{let}\ (sts', (v', d), b') \quad = arr\ (id \times k)\ e\ \underline{(f\ e\ (sts, (v, d), b))} \\
 & \textbf{in}\ (sts', v', b') \\
= & \textbf{let}\ (sts'', (v'', d''), b'') = f\ e\ (sts, (v, d), b) \\
 & \quad (sts', (v', d), b') \quad = \underline{arr\ (id \times k)\ e\ (sts'', (v'', d''), b'')} \\
 & \textbf{in}\ (sts', v', b') \\
= & \textbf{let}\ (sts'', (v'', d''), b'') = f\ e\ (sts, (v, d), b) \\
 & \quad \underline{(sts', (v', d), b') \quad = (sts'', (v'', k\ d''), b'')} \\
 & \textbf{in}\ \underline{(sts', v', b')} \\
= & \textbf{let}\ (sts', (v', d''), b') \quad = f\ e\ \underline{(sts, (v, k\ d''), b)} \\
 & \textbf{in}\ (sts', v', b') \\
= & \textbf{let}\ (sts', (v', d''), b') \quad = \underline{f\ e\ (arr\ (id \times k)\ e\ (sts, (v, d''), b)} \\
 & \textbf{in}\ (sts', v', b') \\
= & \underline{\textbf{let}\ (sts', (v', d''), b') \quad = (arr\ (id \times k) \ggg f)\ e\ (sts, (v, d''), b)} \\
 & \underline{\textbf{in}\ (sts', v', b')} \\
= & loop\ (arr\ (id \times k) \ggg f)
\end{array}
$$

FIGURE 2. Manual proof of the *sliding* law

4.3 Editor Laws

The correctness proofs of the *Arrow* laws do not rely on other combinator functions, hence they are also valid for the `edit` combinator. This means that we get a lot of equivalences 'for free' when `edit` is involved. In addition, we impose the following laws that are specific to `edit` in Def. 3.

Definition 3 (Editor Laws)

$$
\begin{array}{rcll}
edit\ i \ggg edit\ i & = & edit\ i & (edit\ elimination) \\
self\ (g\ o\ f)\ i & = & self\ f\ i \ggg self\ g\ i & (self\ distribution) \\
feedback(edit\ i)(edit\ j) & = & feedback(edit\ j)(edit\ i) & (edit\ swap) \\
feedback(editarr\ i\ f) & & & \\
(editarr\ j\ g) >@ b2a & = & a2b@> feedback(editarr\ j\ g) & \\
 & & (editarr\ i\ f) & (editarr\ swap) \\
\textbf{where} \quad editarr\ i\ f & = & edit\ i >@ f &
\end{array}
$$

The *edit elimination* law states that editors behave as pure stores: it is harmless (and pointless) to store the very same data in the same location in sequence. The *self distribution* law states that function composition distributes over the `self` pattern. The `feedback` combinator is essential in controlling when it is allowed to swap

editors, which can not be done in general. The *edit swap* law states that within the context of a feedback, it is allowed to swap editors, because they always update each other in case of occurring edit events. The exception is in case of an edit event that is not targeted at editor i or j: in that case we must assume the precondition that their state values already have the desired interconnection value. It is sufficient to prove that the precondition holds for the initialized state. Due to the presence of sharing, this law can only be applied within an *Arrow* structure when there are no further occurrences of `edit` i and `edit` j. Finally, the *editarr swap* law extends the *edit swap* law when editors modify their output values. In that case, two additional conversion functions `a2b :: a → b` and `b2a :: b → a` are required when `editarr i f :: EditF a b` and `editarr j g :: EditF b a`. For this law, the same conditions apply: the state values need to have the desired interconnection value, and there should be no other occurrences of the involved editors.

The first two laws have been proven manually, as well as a variant of the last two laws. We expect no issues when redoing the proofs in *Sparkle*.

4.4 Definedness properties of the combinators

Hughes and Paterson have taken great care to construct the *Arrow* combinators in such a way that they impose as little as possible restrictions to the behavior of additional primitive combinators that will be glued together with these combinators. In this section, we study what happens with the *definedness properties* when we extend their combinators with our `edit` combinator. This is motivated by the observation that the `edit` combinator imposes strong definedness properties on events and state values.

Sparkle defines the type class **class** `eval a :: !a → Bool`. It specifies, per instance type, the definedness of its value. In order to shorten the definitions below, we first introduce a synonym type for predicates:

```
:: P a :== a → Bool
```

We first connect the definedness of the domain triplet of type `(D a)` and the tuple tree structure of its value type `a`:

```
instance eval EVENT where eval (Event id v) = eval id && eval v

evalD :: (P a) → P !(D a)
evalD evala (sts, v, edited) = eval sts ∧ evala v ∧ eval edited

evalEditF :: (P a) (P b) → P (EditF a b)
evalEditF evala evalb f
    ⇔ ∀e∈EVENT ∀d∈(D a) [(eval e) → (evalD evala d) → (evalD evalb (f e d))]
```

The `EVENT` instance of `eval` yields `True` only if its argument is completely evaluated. The $\texttt{eval}_D$ predicate combines the definedness predicate of a value of type `a` with the definedness of a domain triplet of type `(D a)`. Predicate $\texttt{eval}_{\texttt{EditF}}$ lifts this property to the concrete *Arrow* domain: given the definedness of the event ar-

gument and domain triplet with a defined input value, then the definedness of the arrow combinator is set by the definedness of the output value.

For functions, we provide a different definedness predicate, that is based on the function's argument and result:

$$\begin{aligned}&\texttt{eval}_F\ \texttt{::}\ \texttt{(P a)}\ \texttt{(P b)} \rightarrow \texttt{P}\ \texttt{(a} \rightarrow \texttt{b)}\\ &\texttt{eval}_F\ \texttt{eval}_a\ \texttt{eval}_b\ \texttt{f}\\ &\quad \Leftrightarrow \forall_{\texttt{a}\in\texttt{a}}\ \texttt{[(eval}_a\ \texttt{a)} \rightarrow \texttt{(eval}_b\ \texttt{(f a))]}\end{aligned}$$

We can now specify the definedness properties of the *Arrow* combinators. These are given in Def. 4.

Definition 4 (Definedness of the *Arrow* combinators)

$$\begin{array}{rcll}
\texttt{eval}_F\ \texttt{eval}_a\ \texttt{eval}_b\ f & \Rightarrow & \texttt{eval}_{\texttt{EditF}}\ \texttt{eval}_a\ \texttt{eval}_b\ (\texttt{arr}\ f) & (\textit{arr})\\
\texttt{eval}_{\texttt{EditF}}\ \texttt{eval}_a\ \texttt{eval}_b\ f & & & \\
\quad \wedge & \Rightarrow & \texttt{eval}_{\texttt{EditF}}\ \texttt{eval}_a\ \texttt{eval}_c\ (f \ggg g) & (\ggg)\\
\texttt{eval}_{\texttt{EditF}}\ \texttt{eval}_b\ \texttt{eval}_c\ g & & & \\
 & & \texttt{eval}_{\texttt{EditF}}\ (\texttt{eval}_a \circ \texttt{fst}) & \\
\texttt{eval}_{\texttt{EditF}}\ \texttt{eval}_a\ \texttt{eval}_b\ f & \Rightarrow & \quad (\texttt{eval}_b \circ \texttt{fst}) & (\textit{first})\\
 & & \quad (\texttt{first}\ f) & \\
\texttt{eval}_{\texttt{EditF}}\ (\texttt{eval}_a \circ \texttt{fst}) & & & \\
\quad (\texttt{eval}_b \circ \texttt{fst}) & \Rightarrow & \texttt{eval}_{\texttt{EditF}}\ \texttt{eval}_a\ \texttt{eval}_b\ (\texttt{loop}\ f) & (\textit{loop})\\
\quad f & & &
\end{array}$$

The above properties have been proven correct with *Sparkle*.

Finally, we can declare the definedness property of `edit`. Editors expect fully evaluated input values and produce fully evaluated output values, within domain triplets that are in *rnf*. The `STATE` instance of `eval` enforces *nf* values:

```
instance eval STATE where eval (Basic v)    = eval v
                          eval (Pair s1 s2) = eval s1 && eval s2
```

Def. 5 gives the definedness axiom for editors.

Definition 5 (Definedness axiom of the *edit* combinator)

$$\texttt{eval}_{\texttt{EditF}}\ \texttt{eval}\ \texttt{eval}\ (\texttt{edit}\ i) \qquad (\textit{edit})$$

Also this property has been proven correct with *Sparkle*.

5 CASE STUDY

In Sect. 2 we have given the toolkit code for a money converter application. The key part of the code was:

```
converter = feedback (edit euroId>>>arr toUSD)
                     (edit usdId >>>arr toEuro)
```

In this example, the *euro* editor happens to be the first editor in the sequence, and the *dollar* editor is second. Hence, the initial value of this application must be of type `Euros`, and the final value is of type `USDollars`. Intuitively, the meaning of the program should be the same if we had chosen to start with the *dollar* editor instead. Put in other words, the following program should have the same meaning:

```
converter‘
    = toUSD >@ feedback (edit usdId >>>arr toEuro)
                        (edit euroId>>>arr toUSD) >@ toUSD
```

The conversion functions `toUSD` need to be added in order to obtain an application of the same type.

Before we start to prove the equivalence between `converter` and `converter‘`, we need to derive their semantic models. We start with the semantic model of `converter`:

```
convertermodel = feedback (edit euroId >>> arr toUSD‘)
                          (edit usdId  >>> arr toEuro‘)
where euroId   = 1
      usdId    = 2
```

The main difference is that we can not use the `USDollars` and `Euros` state types. Instead, we model these values as `Basic Int` values, and introduce integer manipulation functions `toUSD‘` and `toEuro‘`. The only property they should obey is that they are each other's inverse (so `toUSD‘ o toEuro‘ = id` and `toEuro‘ o toUSD‘ = id`). The semantic model of `converter‘` is derived similarly:

```
convertermodel‘
    = toUSD‘ >@ feedback (edit usdId  >>> arr toEuro‘)
                        (edit euroId >>> arr toUSD‘) >@ toUSD‘
```

This demonstrates that the distance between the source code and the semantic model is short.

The proof is rather straightforward:

$$
\begin{aligned}
& \underline{convertermodel} \\
= \;& feedback\ (\underline{edit\ euroId \ggg arr\ toUSD'})\ (\underline{edit\ usdId \ggg arr\ toEuro'}) \\
= \;& \overline{feedback\ (editarr\ euroId\ toUSD')\ (editarr\ usdId\ toEuro')} \\
= \;& \overline{feedback\ (editarr\ euroId\ toUSD')\ (editarr\ usdId\ toEuro')} \ggg arr\ \underline{id} \\
= \;& feedback\ (editarr\ euroId\ toUSD')\ (editarr\ usdId\ toEuro') \ggg \overline{arr\ (toUSD'\ o\ toEuro')} \\
= \;& \overline{feedback\ (editarr\ euroId\ toUSD')\ (editarr\ usdId\ toEuro') \ggg arr\ toEuro'} \ggg arr\ toUSD' \\
= \;& \overline{feedback\ (editarr\ euroId\ toUSD')\ (editarr\ usdId\ toEuro') >@ toEuro'} \ggg arr\ toUSD' \\
= \;& \overline{toUSD' @> feedback\ (editarr\ usdId\ toEuro')\ (editarr\ euroId\ toUSD')} \ggg arr\ toUSD' \\
= \;& \overline{toUSD' @> feedback\ (\underline{editarr\ usdId\ toEuro'})\ (\underline{editarr\ euroId\ toUSD'}) >@ toUSD'} \\
= \;& \overline{toUSD' @> feedback\ (edit\ usdId \ggg arr\ toEuro')\ (edit\ euroId \ggg arr\ toUSD') >@ toUSD'} \\
= \;& convertermodel'
\end{aligned}
$$

Because we use the *edit swapping* law, we need to show that the precondition holds for the initial value of the system. This is done with a symbolic calculation

of the domain after initialization:

$$
\begin{aligned}
& converter\ dummyEVENT\ ([(euroId, v), (usdId, w)], init, True) \\
= \ & ([(euroId, \underline{toEuro(toUSD\ init)}), (usdId, toUSD\ init)], toUSD\ \underline{(toEuro(toUSD\ init))}, True) \\
= \ & ([(euroId, init), (usdId, toUSD\ init)], toUSD\ init, True)
\end{aligned}
$$

It is clear that the desired relation between the state of the *euroId* and *usdId* editors holds.

6 RELATED WORK

We have presented a semantic model for interactive applications. The model is inspired on our work on high level toolkits for desktop GUI applications and web applications, viz. the *GEC Toolkit* and the *iData Toolkit*. It uses the same level of abstraction as the toolkits by considering the elementary interactive components as being editors of arbitrary values that are rendered, and can be edited by the user. The elementary elements are glued together by means of the *Arrow* combinator functions. 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. These systems also had to resort to *Arrow*s in order to eliminate subtle performance problems. In our case, we use them chiefly to structure our programs in order to facilitate reasoning.

Another way of modeling 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) [14], CSP (Communicating Sequential Processes) [10], ACP (Algebra of Communicating Processes) [5], and *μCRL* (micro Common Representation Language) [8]. 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. We hope to have demonstrated that the use of a disciplined, functional style is well suited to create intricate interactive applications that can still be reasoned about with traditional equational reasoning techniques.

7 CONCLUSIONS

In this paper we have introduced the formal semantic model of the *GEC Toolkit* and the *iData Toolkit*. This model is based on the *Arrow* framework by Hughes, and includes the *loop* combinator by Paterson. The semantic model extends the framework with a single primitive combinator function, *edit*, for creating editors with shared state.

We have proven that the associated *Arrow* laws hold within our model, and extended the set with a number of laws for editors. Editors have subtle behavior, and

this is expressed clearly with the conditions that are imposed by the swapping laws. We have further investigated the definedness properties of the semantic model. This is relevant because the *edit* combinator imposes very strict requirements on its input values, output values and events that are passed through the system, which is in contrast with the requirements of the *Arrow* combinators. We have shown that the *Arrow* combinators are quite liberal with respect to these values. When instantiating the *Arrow* laws with the *edit* combinator, these definedness properties must be taken into account.

Most of the proofs have been conducted with the aid of *Sparkle*. We were not able to prove the loop laws due to a missing tactic to handle cyclic let definitions. The use of *Sparkle* in case of the proven laws increases confidence in their correctness. In addition, working on the proofs with a tool assistant has helped us identify issues that escaped our attention when constructing theorems.

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] A. Alimarine and R. Plasmeijer. A Generic Programming Extension for Clean. In T. Arts and M. Mohnen, editors, *The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers*, volume 2312 of *LNCS*, pages 168–186. Älvsjö, Sweden, Springer, Sept. 2002.

[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] J. Groote and M. Reniers. Algebraic Process Verification. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier Science B.V., 2001.

[9] R. Hinze. A new approach to generic functional programming. In *The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. Boston, Massachusetts, January 2000.

[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] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Springer, 2003.

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

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

[16] R. Plasmeijer and P. Achten. Generic Editors for the World Wide Web. In *Central-European Functional Programming School - Revised Selected Lectures*, volume 4164 of *LNCS*, pages 1–34, Eötvös Loránd University, Budapest, Hungary, Jul 4-16 2005. Springer.

[17] 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 - Revised Selected Papers, 17th International Workshop, IFL05*, LNCS 4015, Department of Computer Science, Trinity College, University of Dublin, September 19-21 2006.