A Common Arrow Based Semantics for GEC and iData Applications

PETER ACHTEN, MARKO VAN EEKELEN, MAARTEN DE MOL,

RINUS PLASMEIJER

Institute for Computing and Information Sciences, Radboud University Nijmegen
(*e-mail:* {P.Achten,M.vanEekelen,M.deMol,R.Plasmeijer}@cs.ru.nl)

Abstract

State-based interactive applications, whether they run on the desktop or as a web application, can be considered as collections of interconnected editors of structured values that allow users to manipulate data. This is the view that is advocated by the *GEC* and *iData* toolkits, which offer a high level of abstraction to programming 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 defines shared state and event handling. We study the properties of this *EditorArrow* model, and of editors in particular. Furthermore, we present the definedness properties of the combinators. A reference implementation of the *EditorArrow* model is given with some small program examples. We discuss formal reasoning about the model using the proof assistant *Sparkle*.

1 Introduction

Building Graphical User Interfaces (GUI) is a labor intensive endeavor, whether they are being programmed based on a desktop widget set, or based on the web. Consider the effort of creating a frequently occurring application-user dialog, in which the user is required to enter a number of data items in order to advance the program. When programming for the desktop, the programmer needs to declare, create, manage, and eventually destroy widgets (at least one for each input element, and typically several to contain them and provide proper layout); for each widget several callback routines need to be programmed that implement both the behavior of the widget, and its effect to other widgets. Callback functions must terminate timely (the $\frac{1}{2}s$ rule) to provide the application user the impression that the application is sufficiently responsive to her actions. When programming for the web, the programmer needs to create the proper *HTML* pages containing the forms that hold the input elements; the state of these elements needs to managed by the programmer because of the stateless nature of the web; the communication, which is typically string based, between client browser and server application has to be programmed, and is untyped, which is a known source of errors. The code that computes the page needs to terminate sufficiently fast, otherwise the browser will

give up. In both situations, the resulting code can easily result in hundreds of lines of code that is intricately interdependent. How can you convince yourself, or other stakeholders, that the program is correct with respect to its requirements? Ideally, one would like to prove (once and for all) that the program satisfies well stated properties in a formal, and computer supported, way. Unfortunately, even if the host programming language supports formal reasoning, neither the desktop nor the web has a formally specified reasoning model. Without some kind of underlying model one will have to resort to informal reasoning or to (model-based) testing. Model-based testing does not require a formal model of the implementation, but only a formal specification of the required properties. Due to automation, model based test systems can rapidly and repeatedly explore vast numbers of test scenarios, and generate reports when issues are being found. Model based testing can be an extremely valuable tool to increase the confidence in the correctness of an implementation, but it still does not provide a proof.

In this paper we create a common underlying formal model for a certain class of desktop and web programs making formal reasoning applicable for such programs. Reconsider the task of creating an application user dialog in which the user needs to provide several data items to the application. Another way of looking at this task is to consider *the data items to be an instance of a structured data type*, and to *derive the corresponding GUI automatically from this data type*. The derived GUI acts effectively *as an editor of values of the given structured type*. This reduces the programming effort to specifying a suitable structured data type, and invoking the derivation mechanism to create its GUI. What remains to be done is to *interconnect* the elements of the data type in a suitable way. This avenue has been explored in our previous work on generating GUIs for the desktop resulting in the *GEC* toolkit (Achten *et al.*, 2003; Achten *et al.*, 2004b; Achten *et al.*, 2004a) as well as for the web, resulting in the *iData* toolkit (Plasmeijer & Achten, 2005; Plasmeijer & Achten, 2006). The host language is the pure and lazy functional programming language *Clean* (for readers who are more familiar with the functional language *Haskell*, we refer to (Achten, 2007) for a concise overview of the differences between *Clean* and *Haskell*). We use a functional language because they are known to support formal reasoning well; we use *Clean* because it comes with the interactive proof assistant *Sparkle* (de Mol *et al.*, 2002; de Mol *et al.*, 2008), which allows us to reason about *Clean* programs. Furthermore, because the above mentioned toolkits have been implemented in *Clean*, we wish to reason about the programs, and not a derived model of a program. Finally, the built in *generic programming* support in *Clean* is used for the automatic derivation of GUIs.

With the *GEC* and *iData* toolkit, the programmer creates dialogs, or forms, by means of designing a structured data type that identifies the values that can be edited by the user. Whenever such a value has been edited, it may invoke an effect on other dialogs, or forms. Put in other words, these dialogs are *interconnected*. In the toolkits, we have explored two paths to define this interconnection relation.

- In the first approach, the *'freestyle'* approach, the editors are interconnected

by means of a function that invokes editors when needed. This provides the programmer with the full expressive power of the host language.

- In the second approach, we capture the interconnection relation by means of the *Arrow* framework (Hughes, 2000). In this way, the programmer exchanges freedom of expressiveness with the rigor of a small set of combinator functions.

It is our goal to reason formally about interactive GUI programs written in either the *GEC* or the *iData* toolkit. Eventually, we want to be able to do this for programs written in either of the above styles, but for now we restrict ourselves to the combinator based approach. The point-free style of *Arrow* combinators makes them particularly amenable to formal reasoning. We will use the proof assistant *Sparkle*, not only because it will aid us in managing with the proofs, but also because every *complete Sparkle* proof takes *definedness* properties into account, i.e. reasoning about how a program deals with undefined values ($\perp$) and under which conditions $\perp$ values are yielded (van Eekelen & de Mol, 2005; van Eekelen & de Mol, 2007). With the aid of *Sparkle*, we have been able to formalize definedness relations of the *Arrow* combinators of our framework.

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 *EditorArrow* model satisfies the standard set of laws that are imposed on *Arrow* models. In addition, we identify a number of specific laws for editors and we identify definedness properties for our editor arrows.

In summary, we propose a common formal semantic model for interactive *GEC* and *iData* programs written in an *Arrow* combinator style. We define both denotational and operational semantics for the *EditorArrow* model. Programming applications of *EditorArrow* combinators are expressed in *Clean*, which allowed us to use the interactive proof assistant *Sparkle*. In some cases, this pointed to situations that were clearly undesired, but that had escaped our scrutiny. This has led to changes both on the semantical level and in the specification of the properties. We handle some programming examples and reasoning case studies, some of which concern reasoning about definedness.

The remainder of this paper is structured as follows. We first present the two toolkits in Sect. 2 discussing the differences and correspondences. A common *EditorArrow* model abstracting from these toolkits is defined denotationally and operationally in Sect. 3. Sect. 4 gives the standard *Arrow* laws, and identifies iteration and editor laws, as well as definedness laws for the basic combinators of this semantic

model. In Sect. 5 we present a reference implementation of the *EditorArrow* model in *Clean*. We give some small example *EditorArrow* programs and we also discuss formal reasoning about these programs with *Sparkle*. Related work is presented in Sect. 6 and we end with conclusions in Sect. 7.

2 The *GEC* and *iData* Toolkits

In this section we briefly introduce the two toolkits, *GEC* and *iData*, discuss their similarities and differences, and identify a common *api*, for which a semantic model will be defined and used in the remainder of this paper.

The *GEC* and *iData* toolkits have been designed for different contexts (widget based versus web-based GUIs), but with the same goals and design principles: to automatically generate GUIs from structured types, and to consider such a GUI as an *editor* for values of that type. Hence, an editor is a typed unit that provides the application user with a GUI to edit values of that given type only. The concept of type parameterized editors provides a strong abstraction mechanism to eliminate the differences of the two back-ends of the toolkits. In this way, they become closely related. There are however also many differences between these toolkits with respect to behavior, implementation, and use. Before we distill a common *api*, we first discuss the differences.

The *GEC* Toolkit has been designed and implemented to create desktop GUI applications. It has been implemented in the GUI toolkit of *Clean*, *Object I/O* (Achten & Plasmeijer, 1998). An editor is an interactive element that resides in a window. In addition, the state of the editors is resident. Just like any other interactive element of *Object I/O*, editors are managed by the program: they can be created, altered, and closed. The internal implementation of an editor basically copies the generic decomposition of the editor's value to a (large set of) GUI-fragment/receiver pairs. This allows to refresh only the significant parts of the GUI when values are modified by the user. The editor responds to such a user action by means of a callback function, as is usually done in desktop GUI applications, and *Object I/O* as well. In this callback function, the programmer has access to the full *Object I/O* library and all other editors.

The *iData* Toolkit has been designed and implemented to be a web application. An *iData* application can be opened within any browser, and navigated with the usual back and forth buttons. Editors are interactive elements that reside within a browser window as form elements. With each user action, a new web page needs to be rebuilt. This is an essential difference with the *GEC* toolkit that also has its impact at the programmer's level: in the *GEC* toolkit editors need to be closed explicitly, whereas this is not required of the programmer in the *iData* toolkit. For this reason, *iData* programming is much easier than *GEC* programming. In contrast with the *GEC* toolkit, the value of an *iData* element is not decomposed generically, but rather kept intact, and is 'patched' by a generic function whenever the user alters part of the state of the corresponding *iData* element. Editors in the *iData* toolkit have no callback functions to alter each

other. Instead, it is the program that manipulates the editors and makes their values depend on each other. When computing web pages, the toolkit encodes the editor states, which can reside within the *HTML* page, or stay on the server side. An *iData* application computes *HTML* forms that are generated from the type of the corresponding editor. Within the program, the programmer has access to the generated *HTML*, and she can define additional content in terms of *HTML*, and control the layout of editors.

In order to illustrate the rather large differences between *GEC* programs and *iData* programs, we first give an off-the-shelf, 'freestyle', implementation in the two toolkits of a case study.

2.1 Example: Variable Sum List in GEC and iData

The case study is an interactive program that allows the user to enter a positive number in one input field. This number determines the total number of other integer input elements. These input elements can also be edited by the user. After each such edit action, their sum should be displayed in a final integer editor. This can be repeated as many times as the user likes. She can increase and decrease the number of integer input elements, alter their values, and is informed of the sum of their values.

For both toolkits we present the 'freestyle' versions of this case study. We start with the *GEC* toolkit.

The *GEC* program (Fig. 1) needs to import the *GEC* library `StdGEC`, as well as the *Clean* prelude `StdEnv` (line 2). The main wrapper function of the *GEC* toolkit is `startGEC`, which expects a *GEC* function (`varsumlist`) that creates the GUI that belongs to this program. Being based on *Object I/O*, *GEC* editors are interactive elements that are parameterized with callback functions that define the response of each editor to a change of value. However, the *GEC* toolkit deviates from the *Object I/O api* convention that their constructor functions (`gecEdit` and `gecHide`) yield a *GEC* handle to the created *GEC* editor rather than being provided with one. The first editor that is created is the display of the sum of all values (line 5). Because it does not have to direct its output to another editor, its callback function is simply `noUpdate`, which does not change the environment. The second editor that is created stores the current list of argument editors, which is initially empty (line 6). (A store, created with `hideNGEC`, is just an invisible editor.) These editors are needed to close them afterwards. In line 7, the integer editor is created in which the user can enter the desired number of input fields. Its callback function is `createNrFields` which is parameterized with the *GEC* references `sumGEC` (the sum display) and `argGEC` (the stored list of current editors). When its value is altered, it checks whether the new number is less than the current number of editors. In that case, it closes the appropriate editors (lines 14-19). In the other case, new editors should be created (lines 21-25). Each sum argument editor has the same callback function, (`sumField sumGEC`) (lines 26-28) which first determines the current value of the sum display, and updates it with the new value.

```
module varsumlist_GEC_FreeStyle                                                    1.

import StdEnv, StdGEC                                                              2.

Start world       = startGEC Void varsumlist world                                 3.

varsumlist pSt                                                                     4.
# (sumGEC,pSt)    = gecEdit 0 OutputOnly noUpdate pSt                              5.
# (argGEC,pSt)    = gecHide [] pSt                                                 6.
# (nrGEC, pSt)    = gecEdit 0 Interactive (createNrFields sumGEC argGEC) pSt       7.
= pSt                                                                              8.
where                                                                              9.
  createNrFields sumGEC argGEC _ n pSt                                            10.
  # (argGECs,pSt) = argGEC.gecGetValue pSt                                         11.
  # curNrArgs     = length argGECs                                                 12.
  | n < curNrArgs                                                                  13.
    # (keep,away) = splitAt n argGECs                                            14.
    # (vs,pSt)    = seqList (map get away) pSt                                   15.
    # pSt         = foldr (closeGEC o closeGUI) pSt away                         16.
    # pSt         = set gec NoUpdate keep pSt                                    17.
    # pSt         = sumField sumGEC Enquire (~(sum vs)) pSt                      18.
    = pSt                                                                        19.
  | otherwise                                                                      20.
    # (new,pSt)   = seqList [  gecEdit 0 Interactive (sumField sumGEC)           21.
                            \\ i <- [1..n-curNrArgs]                             22.
                            ]  pSt                                               23.
    # pSt         = set gec NoUpdate (argGECs ++ new) pSt                        24.
    = pSt                                                                        25.

  sumField sumGEC _ v pSt                                                        26.
  # (curSum,pSt)  = get sumGEC pSt                                               27.
  = set sumGEC YesUpdate (curSum + v) pSt                                        28.

// Auxiliary functions for this example
gecEdit v d f     = createNGEC "Example" d True v f                              29.
gecHide v         = hideNGEC   title OutputOnly True v noUpdate                  30.
noUpdate _ _ env  = env                                                          31.
closeGEC gec      = gec.gecClose                                                 32.
closeGUI gec      = gec.gecCloseGUI SkipCONS                                     33.
get      gec      = gec.gecGetValue                                              34.
set      gec      = gec.gecSetValue                                              35.
```

Fig. 1. The *GEC* `varsumlist` program in 'free-style'.

The *iData* program (Fig. 2) needs to import the *iData* toolkit, besides the *Clean* prelude (line 2). Its main wrapper function is `doHtmlWrapper`, which expects a function (`varsumlist`) that computes a *HTML* page, that may contain forms, created as editors for *iData*. In *iData*, an editor is created with the function `mkEditForm`. *iData* follows the *Object I/O* convention to parameterize constructor functions with their handles, rather than yielding such a value. When an editor's value depends on the value of other editors, then its value must be `Set`. An example is in line 9, where

```
module varsumlist_iData_FreeStyle                                              1.

import StdEnv, StdiData                                                        2.

Start world   = doHtmlWrapper varsumlist world                                 3.

varsumlist hSt                                                                 4.
# (nrF,  hSt) = mkEditForm (Init,nFormId "nr" 0) hSt                           5.
# (argFs,hSt) = seqList [  mkEditForm (Init,nFormId ("arg " <+++ i) 0)        6.
                        \\ i <- [0..value nrF-1]                               7.
                        ]  hSt                                                 8.
# (sumF, hSt) = mkEditForm (Set,ndFormId "sum" (sum (map value argFs))) hSt    9.
= mkHtml "Example"                                                             10.
    [mkColForm (map (BodyTag o form) ([nrF] ++ argFs ++ [sumF]))] hSt          11.

// Auxiliary functions for this example
value form    = form.value                                                     12.
form  form    = form.form                                                      13.
```

Fig. 2. The *iData* `varsumlist` program in 'free style'.

the sum display is defined: its value must be the sum of the values of the argument editors. The other editors have just `Init`ial values (line 5 and 6). The *HTML* page (lines 10-11) that is computed is a single column of all form renderings of all editors, starting with the number editor, followed by the list of value editors, and closed with the sum display.

The two programs behave similarly, yet their specifications are very different and the implementation of the underlying toolkits are even more different. The *iData* version is much shorter and more declarative than the *GEC* version, because it only specifies which editors depend on which other editors: if the user enters a lower number of editors, then the *iData* toolkit only includes the remaining editors in the *HTML* page. This is very different from the *GEC* version, in which the program must close the editors itself.

Having discussed the main differences, we can now turn our attention to the similarities and then extract a common core.

2.2 Abstractions towards Editor Arrows

Despite the above mentioned differences, these programs have a lot in common: both use editors to interact with the user, and both programs specify the same interconnection relation: an integer value is displayed that is the sum of the values of n integer editors, where n is the value edited by the user in some first editor. Clearly, the *iData* example in Fig. 2 has the closest match with this interconnection relation. We can rearrange the *GEC* toolkit in such a way that its behavior is similar to that of an *iData* program: immediately after each user event, all editor GUIs are closed, and are reopened only if they are created during execution. In fact, this behavior is already implemented at the level of each individual *GEC*: the user can

switch between the constructors of a value of an algebraic data type without having to reconstruct the intermediate values. The other change that needs to be made is that editors are identified by means of a label instead of a *GEC* value. Again, the adapted toolkit can maintain an administration in which labels are associated with *GEC* values. With these arrangements the freestyle *GEC* version can be expressed much shorter and results in the code displayed in Fig. 3.

```
module varsumlist_GEC_FreeStyle2                                              1.

import StdEnv, StdGEC                                                         2.

Start world       = startGEC Void varsumlist world                            3.

varsumlist pSt                                                                4.
# (sumGEC,pSt)    = gecEdit 0 OutputOnly noUpdate pSt                         5.
# (nrGEC, pSt)    = gecEdit 0 Interactive (createNrFields sumGEC) pSt         6.
= pSt                                                                         7.
where                                                                         8.
  createNrFields sumGEC _ n                                                   9.
  = snd o seqList [  gecEdit ("arg "<+++i,0) Interactive (sumField sumGEC)    10.
                  \\ i <- [0..n-1]                                            11.
                  ]                                                           12.

  sumField sumGEC _ v pSt                                                     13.
  # (curSum,pSt)  = get sumGEC pSt                                            14.
  = set sumGEC YesUpdate (curSum + v) pSt                                     15.
```

Fig. 3. The *GEC* `varsumlist` program in 'free-style', with automatic closing and reopening of editor GUIs.

Even though the adapted *GEC* version still uses callback functions to specify the interconnection of editors, its resemblance with the *iData* version has increased significantly. We continue to eliminate the differences between the two toolkits by means of abstraction. These abstractions are:

- *We ignore all layout issues.* In the *GEC* toolkit, editors can reside in different windows. In the *iData* toolkit, all editors reside in the same browser window. We also ignore where the editors within a window appear, what they look like, and what their dimensions are.
- *We abstract from residence of state.* We simply assume that every editor has access to its state value.
- *We abstract from representation (widgets versus forms).* We are only concerned with editors that respond to value changes. We know that we can derive a rendering for each and every type and do not wish to reason about these renderings.
- *We abstract from the communication method (events versus post/get).* Instead, we consider user actions to be just editing actions which can be modeled conveniently as a whole new value of the same type of value that is maintained by the editor.

- *We abstract from specific typing and type classes issues.* There are many different (generic) classes in the two toolkits, but essentially they all make sure that an editor can be rendered, its value (de)serialized and changed.

As a result of these abstractions we can consider editors of values of any type as basic building blocks.

The next step to undertake is to unify editor declarations and the means to interconnect them. The examples in Fig. 1 and Fig. 2 illustrate that it is very unlikely that we will succeed in doing this for 'freestyle' programs (even for the modified *GEC* toolkit example in Fig. 3). As stated in Sect. 1, we use a combinator approach based on the *Arrow* framework for this. Hence, the editor declaration will become a basic *Arrow* combinator. We adopt the conventions of the *iData* toolkit:

- *Editors are identified by means of a unique label and initial value.*
  In the *GEC* toolkit, the programmer needs to use the handle to a *GEC* editor for this purpose, which is only available *after* creating the editor. This leads in many cases to reversed editor creation, as is also illustrated in Fig. 1 in which the sum display editor needs to be created first because it is manipulated by the other editors. The *iData* approach is actually similar to the one taken in *Object I/O*, in which identifiers are created independently of the elements that they identify.
- *Editors are shared by means of declaring an editor with the same identifier.*
  In the *GEC* toolkit, sharing is realized by manipulating the handle to the *GEC* editor. Again, the use of two different means to identify the same editor is uncomfortable, and we prefer the uniform approach of the *iData* toolkit.
- *Editor values are read and set by subsequent declarations of editors with the same identifier.*
  In the *GEC* toolkit, the value of an editor can be read and set via its handle, and when the editor is created. Because we do not want to have two different forms of access, we combine reading and setting the value of an editor with its declaration.

Both the *GEC* and *iData* toolkit have *one* primitive generic editor creation function (`gGEC` used by `createNGEC` and `hideNGEC` in the *GEC* toolkit, and `mkViewForm` used by `mkEditForm` in the *iData* toolkit). We can chose to use a single editor creation combinator function as well, but instead we prefer to emphasize the *two* different ways of using an editor, each expressed with a separate combinator function, viz. `editread` and `editset`. They have slightly different signatures: both receive an identifier value (unique label and initial value) via the arrow state, but `editset` is also provided with the new value of the editor. Both editors behave the same when manipulated by the user: they receive a new value and emit that value via the arrow state. The difference shows up when an editor that appears earlier within the arrow relation has been manipulated by the user: the `editread` editor simply *echoes* its current value via the arrow state, whereas the `editset` editor *copies* the value that is received via the arrow state as its new value, and emits that new value via the arrow state. Note that the editor that appears earlier within the arrow relation can be *the*

same editor, by using the same identifier. In this way, intricate relationships can be defined via sequential composition rather than a cyclic combination of editors.

2.3 Example: Variable Sum List in Arrow style

In Sect. 2.1 we have presented the 'freestyle' versions of the variable sum list case study in both the *GEC* and *iData* toolkit. We now show what the respective solutions look like in the two toolkits with the *Arrow api*. The solutions are shown in Fig. 4 and Fig. 5. The most important aspect is that the `varsumlist` *Arrow* expression is *identical* in both programs. They still need different wrapper functions (`startGEC` and `doHtmlWrapper` respectively) and need to import different toolkits.

```
module varsumlist_GEC_ArrowStyle                                              1.

import StdEnv, StdGEC                                                         2.

Start world = startGEC Void (startCircuit varsumlist 0) world                 3.

varsumlist  =   arr          (λx → nrId)                                      4.
            >>> editread                                                      5.
            >>> arr          (λn → (n,0))                                     6.
            >>> iterateN     (   first    (arr argId >>> editread)            7.
                             >>> arr      (uncurry (+))                       8.
                             )                                                9.
            >>> arr          (λt → (sumId,t))                                 10.
            >>> editset                                                       11.

    nrId    = ("nr", 0)                                                       12.
    sumId   = ("sum",0)                                                       13.
    argId n = ("arg "<+++n,0)                                                 14.
```

Fig. 4. The *GEC* `varsumlist` program in *Arrow* style.

The `varsumlist` expression uses all standard *Arrow* combinators: `arr` f which lifts a pure function f to the *Arrow* domain (`arr :: (a → b) → Arrow a b`), $f >>> g$ which performs g after f (`>>> :: (Arrow a b) (Arrow b c) → Arrow a c`), and `first` f which bypasses information that is not needed by f (`first :: (Arrow a b) → Arrow (a,c) (b,c)`). Of course, it also uses the two editor combinators: `editread :: Arrow (EditId a) a` and `editset :: Arrow (EditId a,a) a`. A primitive recursion combinator is available as `iterateN :: (Arrow (Int,a) a) → Arrow (Int,a) a`, which repeats its argument arrow as many times as is indicated by the `Int` input. The argument arrow operates on the current index and a local state; in the case study, the current index is used to generate a unique identifier for the editors.

Finally, for each program we show a small scenario of using both applications. The *GEC* sequence is given in Fig. 6, and the *iData* sequence is shown in Fig. 7. In both cases, the following scenario has been performed: we start with the initial GUI (screenshot 1). Next the user has entered the number 2 in the number editor,

```
module varsumlist_iData_ArrowStyle                                              1.

import StdEnv, StdiData                                                          2.

Start world = doHtmlWrapper (startCircuit varsumlist 0) world                    3.

varsumlist  =   arr          (λx → nrId)                                         4.
            >>> editread                                                         5.
            >>> arr          (λn → (n,0))                                        6.
            >>> iterateN     (   first    (arr argId >>> editread)               7.
                             >>> arr      (uncurry (+))                          8.
                             )                                                   9.
            >>> arr          (λt → (sumId,t))                                    10.
            >>> editset                                                         11.

    nrId    = ("nr", 0)                                                          12.
    sumId   = ("sum",0)                                                          13.
    argId n = ("arg "<+++n,0)                                                    14.
```

Fig. 5. The *iData* `varsumlist` program in *Arrow* style.

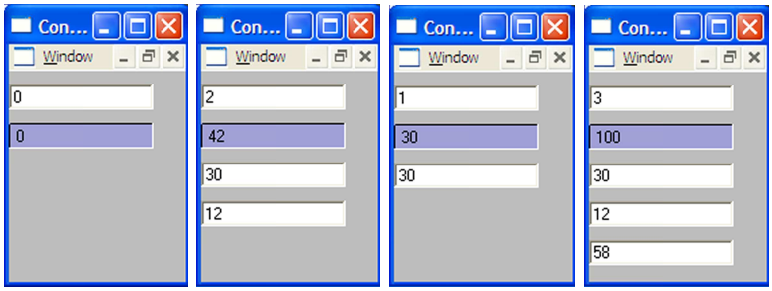

Fig. 6. Editing the *GEC* `varsumlist` program

and the values 30 and 12 in the editors that have appeared due to the previous action (screenshot 2). The next action of the user is to decrease the value 2 to 1 in the number editor, which makes the GUI of the second editor disappear, but its value does not disappear, and neither is the value of the first editor (screenshot 3). If the user now increases the value to 3, then all previous editors reappear, and a new one is added. Entering 58 in the new editor creates a sum of 100 (screenshot 4).

3 Editor Arrows

Both in the *GEC* and in the *iData* toolkit an editor can be regarded as a uniquely named, typed storage for a single value. It presents a GUI to the application user to alter this value. When *connected* to another editor, the editor communicates its stored value both when its value is changed by the user and when a change of another editor is received.

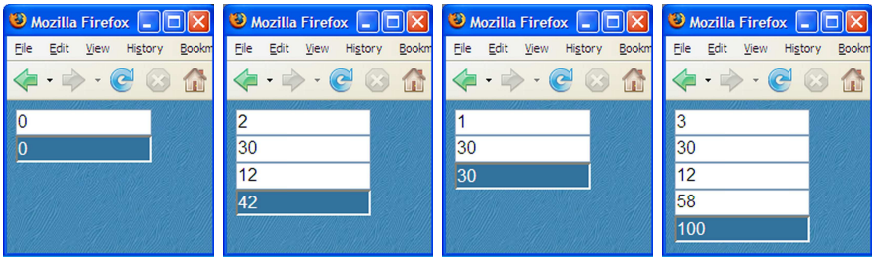

Fig. 7. Editing the *iData* `varsumlist` program

An interactive application is a collection of such connected editors. We will define and use *EditorArrow* combinators to define the connections between editors in a point-free style.

3.1 Denotational Semantics for Editor Arrows

In the classic approaches to functional reactive programming (Elliott & Hudak, 1997; Courtney *et al.*, 2003; Hudak *et al.*, 2003a) the basic building block is formed by signals, defined as time-varying values:

$$Signal\ a = Time \rightarrow a$$

Signals are therefore well suited to define values that vary smoothly over time. They can also be used to accommodate the discrete nature of *events* as they occur in GUIs (Courtney & Elliott, 2001): at time t either an event e is available ($Just\ e$) or it is not ($Nothing$). Hence, by defining

$$Event\ a = Maybe\ a$$

event streams can be included as $Signal\ (Event\ a)$ functions.

From the account in Sect. 2, it follows that in the case of editors we are only concerned with events and event streams. In our framework a $Signal\ (Event\ a)$ simplifies to a list based event stream.

So, in the *EditorArrow* framework an interactive program processes a stream of events, *EditEvents*, which is modelled conveniently as a list of events.

$$EditEvents = [EditEvent]$$

Interactive programs consist of arbitrarily many editors, each having a value of possibly different type. If we would model this with a strongly typed programming language (as we will in Sect. 5) this would lead to the use of existential or dependent types or some other mechanism. Here, we just assume a *Value* domain, and use lists of values abstracting from the way this is specified in a programming language.

When the user manipulates an editor that is identified via $eid : ID$, (s)he eventually generates a new value $v : Value$. This event is modeled as a pair of the $eid : ID$ value of the editor, and the new value $v : Value$ that the user has generated. The ID consists of the name of the editor and its initial value which it will have as long

as no event for it has occurred.

$$\begin{array}{lcl} \mathit{EditEvent} & = & \mathit{ID} \times \mathit{Value} \\ \mathit{ID} & = & \mathit{Name} \times \mathit{Value} \end{array}$$

As stated above, an interactive program consists of arbitrarily many editors that have a data value that can be manipulated by the user. We collect these *editable data* in a set of pairs:

$$\mathit{EditableData} = \wp(\mathit{ID} \times \mathit{Value})$$

We want all values in the *EditableData* domain to be fully defined since these are the values that are to be displayed. We can "read" and "write" pair values from this set using two primitives, *read* and *write*. We assume an access function *initvalue* to take from an *event identifier* of type *ID* the value part which holds its initial value. Note that these primitives require their arguments to be fully defined since the resulting *EditableData* domain is fully defined.

$$\mathit{read}\ \mathit{eid}\ s \quad = \quad \begin{cases} d & \textbf{if}\ (\mathit{eid}, d) \in s \\ \mathit{initvalue}\ \mathit{eid} & \textbf{if}\ (\mathit{eid}, d) \notin s \end{cases}$$

$$\mathit{write}\ \mathit{eid}\ v\ s \quad = \quad \begin{cases} (\mathit{eid}, v) \cup s \backslash (\mathit{id}, d) & \textbf{if}\ (\mathit{eid}, d) \in s \\ (\mathit{eid}, v) \cup s & \textbf{if}\ (\mathit{eid}, d) \notin s \end{cases}$$

The *ID* values serve as unique keys in $s : \mathit{EditableData}$:

$$\forall \mathit{eid} : \mathit{ID}, s : \mathit{EditableData}.(\mathit{eid}, d) \in s \wedge (\mathit{eid}, d') \in s \Rightarrow d = d'.$$

In Sect. 2 we stated that we want to construct programs by means of the *Arrow* combinators. An *Arrow* program fragment processes an event. This is modeled by $\mathit{EventStatus} = \{\mathit{Pending}, \mathit{Processed}\}$. We define two predicates *pending* and *processed* that hold only if their *EventStatus* argument has the corresponding value. Processing an event possibly updates the existing editable data. In addition, it expects an incoming value of type a, and emits an outgoing value of type b. The editable data together with an incoming or outgoing value and the status of event processing are put in one triplet: the *EState*. A program fragment is an *E*ditable *D*ata and *E*vent *T*ransformer function, abbreviated as *EDET*:

$$\begin{array}{lcl} \mathit{EState}\ a & = & \mathit{EditableData} \times a \times \mathit{EventStatus} \\ \mathit{EDET}\ a\ b & = & \mathit{Event} \rightarrow \mathit{EState}\ a \rightarrow \mathit{EState}\ b \end{array}$$

In contrast to classic reactive programming with *Signal*s, where state is always local (introduced by the use of *loop*), we are modelling a situation where essentially global data are edited. Hence, we take as the basis of our *Arrow* modelling the type $\mathit{EDET}\ a\ b$.

The arrow expressions that we allow are built in the following way:

$$\begin{array}{lcl} \mathit{EdArrow} & ::= & \mathit{arr}\ \mathit{Fun} \mid \mathit{EdArrow} \ggg \mathit{EdArrow} \mid \mathit{first}\ \mathit{EdArrow} \\ & & \mid \mathit{left}\ \mathit{EdArrow} \mid \mathit{iterate}\ \mathit{EdArrow} \\ & & \mid \mathit{editread} \mid \mathit{editset} \end{array}$$

where *Fun* represents functions as expressed in a functional language.

Denotationally, we define a partial function $[\![-]\!]$ from these arrow expressions to the functions on the *EDET* domain. Why this is a *partial* function will be explained later in section 4.4.

$$[\![-]\!] : \mathit{EdArrow} \hookrightarrow \mathit{EDET}\ a\ b$$

Below we specify the meaning for each of the combinators denotationally. We use tuples and lists for lambda arguments and standard **case**, **if** and non-recursive **let** constructs to keep the definitions concise and readable.

The basic classic combinators (`arr`, $\ggg$, and `first`) are easily defined. For the meaning of f in the *arr* rule we rely on standard lazy functional language semantics $[\![-]\!]_{\lambda\perp}$ (Cartwright & Donahue, 1982), using domains that are *lifted* by adding $\perp$ to them as domain value. It is important to note that the specific domains for this model (*EditEvent*, *EditableData* and their components) are *not* lifted.

$$\begin{array}{lcl}
[\![\mathit{arr}\ f]\!] & = & \lambda e.\lambda(s,a,p).(s,[\![f]\!]_{\lambda\perp}\ a,p) \\
[\![f \ggg g\ ]\!] & = & \lambda e.([\![g]\!]\ e) \circ ([\![f]\!]\ e) \\
[\![\mathit{first}\ f]\!] & = & \lambda e.\lambda(s,bd,p). \\
 & & \quad \textbf{let}\ (b,d) \quad = bd \\
 & & \quad \textbf{let}\ (s',c,p') = [\![f]\!]\ e\ (s,b,p) \\
 & & \quad \textbf{in}\ (s',(c,d),p')
\end{array}$$

The definition of *first* has an interesting aspect. If the pattern (b,d) is undefined then the result of the meaning function may still be a triplet with a defined or undefined second triplet element, all depending on the meaning of f.

For our purposes, we also need some choice combinator. The standard way to do this is to use a *left* combinator. Based on *left*, different kinds of choice combinators can be created using the lifted standard *Either* type. Since this domain is lifted, the result of the case definition can be a partially defined function.

$$\begin{array}{lcl}
[\![\mathit{left}\ f]\!] & = & \lambda e.\lambda(s,\mathit{eitherlr},p). \\
 & & \quad \textbf{case}\ \mathit{eitherlr}\ \textbf{of} \\
 & & \quad\ \mathit{Left}\ a \quad = \quad \textbf{let}\ (s',b,p') = f\ e\ (s,a,p)\ \textbf{in}\ (s',\mathit{Left}\ b,p') \\
 & & \quad\ \mathit{Right}\ c \quad = \quad (s,\mathit{Right}\ c,p)
\end{array}$$

The meaning of the two combinators for the basic editor variants, *editread* and *editset*, are defined straightforwardly using the *read*, *write* and *pending* functions. We follow the intuitive meaning described in the previous section on page 9 quite closely. Since the *eid* and *eida* event identifiers are lifted and they are passed to the *read* and *write* primitives which require non lifted values, this is a partial definition.

$$
\begin{array}{lcl}
[\![\mathit{editread}]\!] & = & \lambda(\mathit{eid}', v).\lambda(s, \mathit{eid}, p). \\
& & \begin{cases} (\mathit{write}\ \mathit{eid}\ v\ s, v, \mathit{Processed}) & \textbf{if}\ \mathit{pending}(p) \wedge \mathit{eid} = \mathit{eid}' \\ (s, \mathit{read}\ \mathit{eid}\ s, p) & \textbf{if}\ \neg\mathit{pending}(p) \vee \mathit{eid} \neq \mathit{eid}' \end{cases} \\
[\![\mathit{editset}]\!] & = & \lambda(\mathit{eid}', v).\lambda(s, \mathit{eida}, p). \\
& & \textbf{let}\ (\mathit{eid}, a) = \mathit{eida} \\
& & \textbf{in} \\
& & \begin{cases} (\mathit{write}\ \mathit{eid}\ v\ s, v, \mathit{Processed}) & \textbf{if}\ \mathit{pending}(p) \wedge \mathit{eid} = \mathit{eid}' \\ (\mathit{write}\ \mathit{eid}\ a\ s, a, p) & \textbf{if}\ \neg\mathit{pending}(p) \vee \mathit{eid} \neq \mathit{eid}' \end{cases}
\end{array}
$$

We will also need some kind of recursion. Both the *GEC* and the *iData* toolkit have the property that they have a single arrow expression called by a wrapper (which is essentially an event loop that deals with consecutive events recursively). These editor arrow expressions build a finite, fully evaluated interface for the user. This interface may be dynamic in the sense that the user can influence its values and its size but it will always be finite and fully evaluated. For modeling recursion on the level of such editor arrow expressions we need nothing more than primitive recursion. As the basic building block for primitive recursion we use the *iterate* combinator that iterates its argument arrow a finite number of times using a lifted natural number n. Analogous to the choice combinator *left* the result may be partially defined since the (n, a) value is in a lifted domain.

$$
\begin{array}{lcl}
[\![\mathit{iterate}\ f]\!] & = & \lambda e.\lambda(s, (n, a), p). \\
& & \begin{cases} (s, a, p) & \textbf{if}\ n\ =\ 0 \\ \textbf{let}\ (s', a', p') = [\![f]\!]\ e\ (s, (n, a), p) & \textbf{if}\ n\ >\ 0 \\ \ \textbf{in}\ [\![\mathit{iterate}\ f]\!]\ e\ (s', (n-1, a'), p') & \end{cases}
\end{array}
$$

The above denotational semantics states what the meaning is of an arrow expression on a single event. To define what happens with an event stream, consisting of a list of *EditEvents* we need to model the toolkit wrappers' event loops.

$$[\![f]\!]_{\mathit{eventstream}} \quad = \quad [\![\mathit{eventloop}\ f]\!]$$

The toolkit wrappers are modeled by a loop combinator as is introduced by Paterson. The loop combinator is defined using the standard least fixed point combinator $\mathbf{Y}$. In our case however, this loop combinator will occur exactly once (note that it is not part of the definition of *EdArrow*), on the outside of an editor arrow expression. To avoid confusion we have not called this a loop combinator but an *eventloop* combinator.

$$
[\![\mathit{eventloop}\ f]\!] = \mathbf{Y}\left(\begin{array}{l}
\lambda \mathit{evloopf}.\lambda(s, a).\lambda \mathit{es}. \\
\begin{cases} s & \textbf{if}\ \mathit{es} = [] \\ \textbf{let}\ (s', b, p) = [\![f]\!]\ (\mathit{hd}\ \mathit{es})\ (s, a, \mathit{Pending}) & \textbf{if}\ \mathit{es} \neq [] \\ \ \textbf{in}\ \mathit{evloopf}\ (s', a)\ (\mathit{tl}\ \mathit{es}) & \end{cases}
\end{array}\right)
$$

Using *iterate* within arrow expressions and one single *eventloop* on the outside we have exactly the right expressive power for the *EditorArrow* model.

3.2 Operational Semantics for Editor Arrows

For implementing the *EditorArrow* model we also need operational semantics. They are derived straightforwardly from the denotational semantics. We take again the same domains. The operational semantics are defined in the standard way using 'big-step' semantics. The relation $\longrightarrow$ is suffixed with the handled event $e : (id, v)$ which is assumed to be always defined. It relates the argument triplet (s, a, p) of store, value and boolean to a result triplet. The rules define what the semantics is for defined triplets. For other cases the semantics is undefined.

The rules for the basic combinators are given below. With $\rightarrow_{\lambda\perp}$ we denote the standard reduction from functional languages.

$$\frac{f\ a \rightarrow_{\lambda\perp} a'}{arr\ f\ (s,a,p) \rightarrow_{e:(id,v)} (s,a',p)}\ (arr)$$

$$\frac{f\ (s,a,p) \rightarrow_{e:(id,v)} (s',a',p') \qquad g\ (s',a',p') \rightarrow_{e:(id,v)} (s'',a'',p'')}{f \ggg g\ (s,a,p) \rightarrow_{e:(id,v)} (s'',a'',p'')}\ (seq)$$

The *first* rule requires two alternatives since the value domain is lifted and we want a lazy variant of *first* consistent with the denotational definition.

$$\frac{f\ (s,a,p) \rightarrow_{e:(id,v)} (s',a',p')}{first\ f\ (s,(a,c),p) \rightarrow_{e:(id,v)} (s',(a',c),p')}\ (first)$$

$$\frac{f\ (s,\perp,p) \rightarrow_{e:(id,v)} (s',a',p')}{first\ f\ (s,\perp,p) \rightarrow_{e:(id,v)} (s',(a',\perp),p')}\ (first_\perp)$$

Operationally, we need for the *left* combinator the following choice rules (we do not have an extra undefined rule here, we use a partial definition instead):

$$\frac{f\ (s,a,p) \rightarrow_{e:(id,v)} (s',a',p')}{left\ f\ (s,Left\ a,p) \rightarrow_{e:(id,v)} (s',Left\ a',p')}\ (choice\_left)$$

$$\frac{}{left\ f\ (s,Right\ a,p) \rightarrow_{e:(id,v)} (s,Right\ a,p)}\ (choice\_right)$$

Both the editor combinators distinguish between the case where the event is pending (in which case it has to be processed when it matches the id of the editor) or not. The operational semantics employs the same primitives (*pending*, *read* and

write) as the denotational semantics.

$$\frac{s' = write\ id\ v\ s \qquad pending(p)}{editread\ (s, id, p) \rightarrow_{e:(id,v)} (s', v, Processed)}\ (editread\_pending)$$

$$\frac{a = read\ id\ s \qquad id \neq id' \vee \neg pending(p)}{editread\ (s, id, p) \rightarrow_{e:(id',v)} (s, a, p)}\ (editread\_other)$$

$$\frac{s' = write\ id\ v\ s \qquad pending(p)}{editset\ (s, (id, a), p) \rightarrow_{e:(id,v)} (s', v, Processed)}\ (editset\_pending)$$

$$\frac{s' = write\ id\ a\ s \qquad id \neq id' \vee \neg pending(p)}{editset\ (s, (id, a), p) \rightarrow_{e:(id',v)} (s', a, p)}\ (editset\_other)$$

Iteration is defined through two rules (using a natural number). We have one rule for the base case and another for the iterating case using the natural number to count the number of iterations.

$$\frac{}{iterate\ f\ (s, (0, a), p) \rightarrow_e (s, a, p)}\ (iter\_base)$$

$$\frac{f\ (s, (n{+}1, a), p) \rightarrow_e (s', a', p') \quad iterate\ f\ (s', (n, a'), p') \rightarrow_e (s'', a'', p'')}{iterate\ f\ (s, (n+1, a), p) \rightarrow_e (s'', a'', p'')}\ (iter\_next)$$

Finally, the event loop is defined straightforwardly dealing with events one by one and passing the resulting store to the next event. We only yield the store as result since, at each new event the store is augmented to a triplet with the same initial value and the same boolean indicating that the event has not been processed yet.

$$\frac{}{f\ (s, a, Pending) \rightarrow_{[]} s}\ (events\_end)$$

$$\frac{f\ (s, a, Pending) \rightarrow_{e:(id,v)} (s', a', p') \qquad f\ (s', a, Pending) \rightarrow_{es} s''}{f\ (s, a, Pending) \rightarrow_{[e:es]} s''}\ (events\_next)$$

It is easy to prove that the operational semantics is sound with respect to the denotational semantics. The operational semantics will be used as the basis for a reference implementation of the framework in the programming language *Clean* in Sect. 5.

4 Properties of Editor Arrows

In this section we state the basic properties of the semantic model that has been presented in the previous section. The "classic" *Arrow* laws, as described by Hughes and Paterson, are valid for this model. These laws are given as Def. 1.

In Sect. 4.1 we introduce "iterate" laws and in Sect. 4.2 we give properties of the "eventloop". We introduce basic "editor" laws in Sect. 4.3. Finally, we provide "definedness" laws in Sect. 4.4.

Definition 1 (Classic Arrow Laws)

$$
\begin{array}{rcl}
arr\ id \gg f & \overset{(Left\ unit)}{=} f \overset{(Right\ unit)}{=} & f \gg arr\ id \\
f \gg (g \gg h) & \overset{(associativity\ of \gg)}{=} & (f \gg g) \gg h \\
arr\ (g \circ f) & \overset{(\circ\ preserves \gg)}{=} & arr\ f \gg arr\ g \\
 & & \\
first\ (arr\ f) & \overset{(first\ extension)}{=} & arr\ (f \times id) \\
first\ (f \gg g) & \overset{(first\ preserves \gg)}{=} & first\ f \gg first\ g \\
first\ f \gg arr\ (id \times g) & \overset{(first\ swap)}{=} & arr\ (id \times g) \gg first\ f \\
first\ f \gg arr\ fst & \overset{(fst\ eliminates\ first)}{=} & arr\ fst \gg f \\
first\ (first\ f) \gg arr\ assoc & \overset{(assoc\ eliminates\ first)}{=} & arr\ assoc \gg first\ f \\
 & & \\
left\ (arr\ f) & \overset{(left\ extension)}{=} & arr\ (f \oplus id) \\
left\ (f \gg g) & \overset{(left\ functor)}{=} & left\ f \gg left\ g \\
left\ f \gg arr\ (id \oplus g) & \overset{(left\ exchange)}{=} & arr\ (id \oplus g) \gg left\ f \\
arr\ Left \gg left\ f & \overset{(left\ unit)}{=} & f \gg arr\ Left \\
left\ (left\ f) \gg arr\ assocsum & \overset{(left\ association)}{=} & arr\ assocsum \gg left\ f
\end{array}
$$

where

$$
\begin{array}{lcl}
fst\ (a, b) & = & a \\
 & & \\
f \times g\ (a, b) & = & (f\ a, g\ b) \\
 & & \\
f \oplus g\ (Left\ a) & = & Left\ (f\ a) \\
f \oplus g\ (Right\ b) & = & Right\ (g\ b) \\
 & & \\
assoc\ ((a, b), c) & = & (a, (b, c)) \\
 & & \\
assocsum\ (Left\ (Left\ a)) & = & Left\ a \\
assocsum\ (Left\ (Right\ b)) & = & Right\ (Left\ b) \\
assocsum\ (Right\ c) & = & Right\ (Right\ c)
\end{array}
$$

4.1 Iterate Laws

Def. 2 states the two iterate laws. There is a rule for the base case and a rule for the iteration. They are described nicely using an auxiliary function $\odot$. This auxiliary function puts an argument number a in a pair with the arrow result value, that is being passed, such that *iterate* can use this number to count the iterations.

The $iterate - base$ law expresses the fact the argument is applied zero times.

Definition 2 (Iterate Laws)

$$
\begin{array}{rcl}
\mathit{iterate}\ f \odot 0 & \overset{(\mathit{iterate}-\mathit{base})}{=} & \mathit{arr}\ \ \mathit{id} \\
\mathit{iterate}\ f \odot (m+1) & \overset{(\mathit{iterate}-\mathit{next})}{=} & f \odot (m+1) \gg \mathit{iterate}\ f \odot m
\end{array}
$$

where

$$f \odot a \quad = \quad \mathit{arr}\ (\lambda x \rightarrow (a, x)) \gg f$$

Definition 3 (Eventloop Properties)

$$
\begin{array}{rcl}
\mathit{eventloop}\ f\ (s,a)\ [] & \overset{(\mathit{eventloop}-\mathit{end})}{=} & s \\
\mathit{eventloop}\ f\ (s,a)\ [e:es] & \overset{(\mathit{eventloopnext})}{=} & \mathit{eventloop}\ f \\
 & & (\mathit{drop}\ ((\mathit{dupl} \gg \mathit{first}\ f \gg \mathit{pop})\ e\ (s, a, \mathit{Pending}))) \\
 & & es
\end{array}
$$

where

$$
\begin{array}{rcl}
\mathit{dupl} & = & \mathit{arr}\ (\lambda a \rightarrow (a, a)) \\
\mathit{pop} & = & \mathit{arr}\ \mathit{snd} \\
\mathit{drop} & = & \lambda(s, a, p) \rightarrow (s, a)
\end{array}
$$

The $\mathit{iterate} - \mathit{next}$ law expresses the fact that the argument is applied $m+1$ times consecutively with decreasing values starting with $m+1$.

4.2 Eventloop Properties

The properties of the eventloop are given in Def. 3. There are two properties. The property $\mathit{eventloop} - \mathit{end}$ expresses that, when there is no event anymore, the result is the store. The $\mathit{eventloop} - \mathit{next}$ property expresses that the events are dealt with one after the other passing the state and using the same initial value and event status over and over again. This last property requires some auxiliary "plumbing" functions.

4.3 Editor Laws

The proofs of the classic arrow laws, the iterate laws and the eventloop properties do not rely essentially on the definitions of edit combinators, hence they are also valid for the *editread* and *editset* combinators. This means that we get already a lot of equivalences 'for free' when the edit combinators are involved.

In addition, we introduce ten laws that are specific to uses of *editread* and *editset*. They are given as Def. 4.

- We distinguish four *edit elimination* laws (one for each combination of the two different *edit* arrow combinators) expressing that editors behave as pure stores: it is harmless (and pointless) to store the very same data in the same location in sequence in two occurrences of the same editor (i.e. with the same *id*). These laws are expressed again with the auxiliary function $\odot$ to put the

Definition 4 (Editor Laws)

$$
\begin{array}{rcl}
\mathit{editread} \boxdot i \gg \mathit{editread} \boxdot i & \overset{(\mathit{read-read\ elimination})}{=} & \mathit{editread} \boxdot i \\
\mathit{editread} \boxdot i \gg \mathit{editset} \odot i & \overset{(\mathit{read-set\ elimination})}{=} & \mathit{editread} \boxdot i \\
\mathit{editset} \odot i \gg \mathit{editread} \boxdot i & \overset{(\mathit{set-read\ elimination})}{=} & \mathit{editset} \odot i \\
\mathit{editset} \odot i \gg \mathit{editset} \odot i & \overset{(\mathit{set-set\ elimination})}{=} & \mathit{editset} \odot i
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{editread} \boxdot i \ast\!\ast\!\ast \mathit{editread} \boxdot j & \overset{(\mathit{read-read\ swap})}{=} & \mathit{editread} \boxdot j \star\!\star\!\star \mathit{editread} \boxdot i \\
\mathit{editread} \boxdot i \ast\!\ast\!\ast \mathit{editset} \odot j & \overset{(\mathit{read-set\ swap})}{=} & \mathit{editset} \odot j \star\!\star\!\star \mathit{editread} \boxdot i \\
\mathit{editset} \odot i \ast\!\ast\!\ast \mathit{editread} \boxdot j & \overset{(\mathit{set-read\ swap})}{=} & \mathit{editread} \boxdot j \star\!\star\!\star \mathit{editset} \odot i \\
\mathit{editset} \odot i \ast\!\ast\!\ast \mathit{editset} \odot j & \overset{(\mathit{set-set\ swap})}{=} & \mathit{editset} \odot j \star\!\star\!\star \mathit{editset} \odot i
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{self}\ f\ i \gg \mathit{self}\ g\ i & \overset{(\mathit{self\ composition})}{=} & \mathit{self}\ (g \circ f)\ i \\
\mathit{feedback}\ i\ j & \overset{(\mathit{feedback\ swap})}{=} & \mathit{feedback}\ j\ i
\end{array}
$$

where

$$
\begin{array}{rcl}
f \boxdot a & = & \mathit{arr}\ (\lambda x \rightarrow a) \gg f \\
f \odot a & = & \mathit{arr}\ (\lambda x \rightarrow (a, x)) \gg f \\
f \ast\!\ast\!\ast g & = & \mathit{first}\ f \gg \mathit{second}\ g \\
f \star\!\star\!\star g & = & \mathit{second}\ f \gg \mathit{first}\ g \\
\mathit{self}\ f\ i & = & \mathit{editread} \boxdot i \gg \mathit{arr}\ f \gg \mathit{editset} \odot i \\
\mathit{feedback}\ i\ j & = & \mathit{editread} \boxdot i \gg \mathit{editset} \odot j \gg \mathit{editset} \odot i
\end{array}
$$

id at the right place for *editset* and another auxiliary function $\boxdot$ to put the *id* at the right place for *editread*.

- The four *edit swap* laws express the property of independence of the order of two editors of values in the first and the second part of a tuple. In each of these laws it is assumed that i and j are different. The *edit swap* laws are expressed nicely in a symmetric way using the standard combinator $\ast\!\ast\!\ast$ and its "mirrored" variant $\star\!\star\!\star$.

Finally, we have two laws for often used standard application patterns of the edit arrow combinators: *self* and *feedback*.

- The *self* pattern is used to apply a function on the value that is edited by a user and store its result for this editor. In this way, editors can control the values that they contain. The *self composition* law states that function composition distributes over this *self* pattern.
- The *feedback* pattern is used for two editors to feed their results directly back to each other. In general, you cannot swap the order of different subsequent editors because they will respond differently to the same event sequence. The *feedback swap* law states that in the case of mutual feedback the order of the editors is irrelevant. In the case that i equals j this is of course a trivial consequence of applying the *edit elimination* laws.

4.4 Definedness Laws

In the *EditorArrow* model we have assumed that editors are only able to operate on values that are fully defined, which was modeled by restricting the access functions *read* and *write* to values from the *Value* domain. This has subsequent consequences for the entire model, which were left implicit in Sect. 3. In this section, these consequences will be made explicit by means of formulating definedness laws.

Modeling the definedness behavior of editors has consequences for both the used domains and the meaning function. On the domain level, the value part of an *EState* must be lifted by explicitly incorporating $\bot$ in it. When values are constructed with tuples or eithers, multiple lifts may even be necessary. This affects the allowed input (and the produced output) of each editor arrow as follows:

Definition 5 (value transformation of editor arrows)

editor arrow	**allows input**	**and produces**	**assuming**
arr f	A	B	$f \in A \to B$
$f \ggg g$	A	C	$f :: A \to B, g :: B \to C$
first f	$(A \times C)_\bot$	$(B \times C)_\bot$	$f :: A \to B$
left f	$(Either\ A\ C)_\bot$	$(Either\ B\ C)_\bot$	$f :: A \to B$
iterate f	$(\mathbb{N}_\bot \times A)_\bot$	A	$f :: A \to A$
editread	$ID_\bot$	*Value*	–
editset	$(ID_\bot \times A)_\bot$	*Value*	–

Here, $A_\bot$ denotes $A \cup \{\bot\}$, and $f :: A \to B$ denotes that the arrow f transforms values of type A to values of type B (ignoring the other elements of the *EState*, which are of the same type for all editor arrows). For instance, if $f :: A \to A$ and $a \in A$, then $(0, a)$, $\bot$ and $(\bot, a)$ are all valid input for *iterate* f. Note that *editread* and *editset* both produce an element of *Value*, which is assumed to be the unification set of the defined values of all allowed types. The 'A' input of *editset*, on the other hand, does not necessarily have to be defined.

The behavior of the editor arrows on all their allowed inputs was described in Sect. 3.1 and Sect. 3.2, and is the same for the denotational and operational semantics. In the case of $\bot$ values, this behavior can be summarized as follows:

Case 1: It does not matter that (part of) the input is $\bot$, because no structural information is required at that point. Now, computation can continue normally. This case covers the following situations:
arr f on $\bot$; $f \ggg g$ on $\bot$; *first f* on $(\bot, x)$ and $(x, \bot)$;
left f on *Left* $\bot$ and *Right* $\bot$; and *iterate f* on $(n, \bot)$.

Case 2: A $\bot$ occurs where structural information is required, but it is possible to continue computation normally anyway. This case occurs only when *first f* is applied on $\bot$, which is considered to be equal to applying *first f* on $(\bot, \bot)$.

Case 3: A $\bot$ occurs where structural information is required, and it is not possible to continue computation normally. This case covers the following situations:

left f on $\bot$	(cannot decide whether to apply f or not);
iterate f on $\bot$ and $(\bot, x)$	(cannot decide how many times to apply f);
editset on $\bot$	(cannot obtain id and value).

In these situations, we have chosen not to produce any result at all.

Case 4: A $\perp$ occurs when either a defined *ID* or a defined *Value* is required to access the editable data. This case covers the following situations:

editread on $\perp$; *editset* on $(\perp, a)$; and

editset on $(id, \perp)$ (when no event is pending for id).

Again, in these situations we have chosen not to produce any result at all.

Due to cases 3 and 4, the semantics of editor arrows becomes a partial function that does not always produce an *EState* triplet. In order to determine in which situations a result is produced, the following definedness laws can be used:

Definition 6 (definedness relation for editor arrows)

$$Def(f, A, B) \Leftrightarrow \forall_{a \in A} \forall_{ev,s,p} \exists_{b \in B} \exists_{s',p'} .[f\ ev\ (s, a, p) = (s', a', p')]$$

Definition 7 (definedness laws for editor arrows)

$f \in A \to B$	$\overset{(arr\ def)}{\Rightarrow}$	$Def(arr\ f,$	$A,$	$B)$
$Def(f, A, B), Def(g, B, C)$	$\overset{(\ggg\ def)}{\Rightarrow}$	$Def(f \ggg g,$	$A,$	$C)$
$Def(f, A, B)$	$\overset{(first\ def\ 1)}{\Rightarrow}$	$Def(first\ f,$	$A \times C,$	$B \times C)$
$Def(f, \{\perp\}, B)$	$\overset{(first\ def\ 2)}{\Rightarrow}$	$Def(first\ f,$	$\{\perp\},$	$B \times \{\perp\})$
$Def(f, A, B)$	$\overset{(left\ def)}{\Rightarrow}$	$Def(left\ f,$	$Either\ A\ C,$	$Either\ B\ C)$
$Def(f, A, A)$	$\overset{(iterate\ def)}{\Rightarrow}$	$Def(iterate\ f,$	$\mathbb{N} \times A,$	$A)$
	$\overset{(editread\ def)}{}$	$Def(editread,$	$ID,$	$Value)$
	$\overset{(editset\ def)}{}$	$Def(editset,$	$ID \times Value,$	$Value)$
$Def(f, A, B), Def(f, C, D)$	$\overset{(combine\ def)}{\Rightarrow}$	$Def(f,$	$A \cup C,$	$B \cup D)$

(editset-def has been simplified and does not check whether an event is pending or not)

For any given editor arrow f, these laws can be used to come up with sets A and B such that $Def(f, A, B)$ can be inferred. This then shows that f produces a result as long as its input value is an element of A.

5 Programming with Editor Arrows

In this section, we build a direct implementation of the semantic *EditorArrow* model that was described in Sect. 3. The implementation is realized by means of a library in *Clean* and is named '*EditorArrowCore*'. The library serves two purposes. Firstly, it is a reference implementation: execution in *EditorArrowCore* results in the abstract desired behavior of an editor arrow, and execution in *GEC* and *iData* must result in graphical representations of this same abstract behavior. Secondly, it is a basis for formal reasoning, because it allows the laws of Sect. 4 to be verified with *Clean*'s proof assistant *Sparkle*.

This section is structured as follows. First, we describe the realization of the base editor arrows in Sect. 5.1. Then, we define composed arrow operations in Sect. 5.2, which are used to make programming with arrows easier. In Sect. 5.3, we then express two example programs as editor arrows, and compare their execution behaviors in *EditorArrowCore*, *GEC* and *iData*. Finally, in Sect. 5.4 we discuss the

formalization in *Sparkle* of the earlier provided arrow laws, and we compare the definedness of *EditorArrowCore* with respect to the *EditorArrow* model.

5.1 Base editor arrows in the EditorArrowCore library

The *EditorArrowCore* library is a direct implementation of the *EditorArrow* model that was already described concisely in Sect. 3. On the top level, it defines the concept of Editable Data and Event Transformers, by means of the following types:

```
:: EDET a b          :== Event → (EState a) → (EState b)          1.
:: EState a          :== (EditableData, a, EventStatus)          2.
:: EditableData      :== [(EditorId, SerializedValue)]           3.
:: EventStatus       =   Processed | Pending                     4.

:: EditorId          :== (EditorName, InitialValue)              5.
:: EditorName        :== String                                  6.
:: Event             :== (EditorId, SerializedValue)             7.
:: InitialValue      :== SerializedValue                         8.
:: SerializedValue   :== String                                  9.
```

With respect to the *EditorArrow* model, there are only two differences. Firstly, an association list is used to represent `EditableData` (line 3), instead of an association set. This is of no consequence, because `EditableData` will only be operated on by functions that are guaranteed never to create duplicates.

Secondly, values are serialized to `Strings` (line 9) before they are stored in the `EditableData` (line 3). Basically, this is a poor man's solution to implementing stores in which the values can be of arbitrary different types. The serialize and deserialize functions must be provided by the user explicitly, by means of the following class:

```
class editable a                                                 1.
where                                                            2.
    serialize       :: a → String                                3.
    deserialize     :: String → a                                4.
```

In *EditorArrowCore*, each editor must be overloaded with an instance of the `editable` class. Furthermore, in order for serialized values to work correctly, the instance must also satisfy the following properties:

- $\forall_a.[a = \bot \Leftrightarrow$ `serialize` $a = \bot]$; and
- $\forall_s.[s = \bot \Leftrightarrow$ `deserialize` $s = \bot]$; and
- $\forall_a.[$`deserialize` (`serialize` $a) = a]$

The first two properties state that the definedness of serialized values is identical to the definedness of deserialized values, which is necessary to ensure that the definedness properties of the *EditorArrow* model carry over to *EditorArrowCore*. The third property is necessary to make sure that editors do not change values on their own. Unfortunately, it is not possible in *Clean* to enforce properties explicitly for all instances of a class. It is therefore the responsibility of the user to provide instances of the `editable` class that satisfy the required conditions.

In Sect. 3.1, a grammar was introduced for editor arrows (*EdArrow* ::= *arr Fun* | *EdArrow* ⋙ *EdArrow* | ...), and a meaning function was defined on top of it. For

type technical reasons, this approach cannot be translated to *Clean* directly. The problem is that explicit instantiation of *EdArrow* is necessary for the meaning function (i.e. $[\![]\!] :: (EdArrow\ a\ b) \rightarrow EDET\ a\ b$), but can never be realized because the types of the arrow operations are not unifiable[1].

In *EditorArrowCore*, each arrow operation is therefore defined directly by means of a function of the appropriate `EDET` type. This approach is typeable, but has the disadvantage that argument editor arrows can only be typed by means of `EDET` as well, and are therefore no longer restricted to wellformed arrows ($\in EdArrow$). This is corrected by making the `EDET` type abstract. Finally, note that in *EditorArrowCore* arrows are not defined by means of classes, because in the context of editors we are only interested in the `EState` instance.

The effect of the arrow operations is simply a transformation of the `EState` based on an incoming `Event`. First, the standard operations $\ggg$, `arr` and `first` are defined:[2]

```
(>>>) :: (EDET a b) (EDET b c) → EDET a c                                  1.
(>>>) f g event state=:(_,_,_)                                             2.
    = g event (f event state)                                              3.

arr :: (a → b) → EDET a b                                                  4.
arr f event (data, a, status)                                              5.
    = (data, f a, status)                                                  6.

first :: (EDET a b) → EDET (a,c) (b,c)                                     7.
first f event (data, ac, status)                                           8.
    # (data, b, status)      = f event (data, fst ac, status)              9.
    = (data, (b, snd ac), status)                                          10.
```

These functions behave identically to their counterparts in Sect. 3. Note that analogously to the operational semantics, $\ggg$ performs a pattern match on the `EState` triple (line 2), and `first` does not perform a pattern match on the input value `ac` (line 8). This has to do with desired definedness properties, and will be explained further in Sect. 5.4.

Next, the operations `left` (for the realization of choice) and `iterate` (for the realization of the most basic form of recursion) are defined. Note that *Clean* defines a function `iterate` in its standard environment already; the arrow operation is therefore renamed to `iterateN`.

```
:: Either a b       =   Left a | Right b                                   1.

left :: (EDET a b) → EDET (Either a c) (Either b c)                        2.
left f event (data, Left a, status)                                        3.
    # (data, b, status)      = f event (data, a, status)                  4.
    = (data, Left b, status)                                               5.
left f event (data, Right c, status)                                       6.
```

[1] For instance, *first f* can only be a member of *EdArrow* if tuples are **always** produced, which is undesirable for the other arrow operations

[2] For reasons of clarity, we have simplified the types of the arrow operations; the actual types are more complex, because in *Clean* the number of type arguments must be equal to the number of function arguments, resulting in for instance: `>>> :: (EDET a b) (EDET b c) Event (EState a) -> EState c`

```
    = (data, Right c, status)                                                    7.

iterateN :: (EDET (Int,a) a) → EDET (Int,a) a                                    8.
iterateN f event (data, (n, a), status)                                          9.
    | n ≤ 0                          = (data, a, status)                         10.
    ♯ (data, a, status)          = f event (data, (n, a), status)                11.
    = iterateN f event (data, (n-1, a), status)                                  12.
```

The definition of `left` is identical to the operational semantics. The definition of `iterateN` is slightly different, because *Clean* does not provide a type for natural numbers, but only one for whole numbers (`Int`). The base case therefore has to check for $n \leq 0$ (line 10) instead of $n = 0$, and the recursive case goes from n to $n - 1$ (line 12) instead of from $n + 1$ to n. Note that the recursion in `iterateN` always terminates, because the loop variable cannot be changed by the recursive arrow (see line 11: `n` is input of `f`, but not output).

Next, the accessor functions *read* and *write* will be defined, which will be used later to describe the operations *editread* and *editset*. In the *EditorArrow* model, the purpose of *read* and *write* is twofold: they are not only used to update the editable data, but they are also used to implicitly enforce definedness properties. The required definedness properties of *read* and *write* are as follows:

- In the *EditorArrow* model, *read* can be regarded as a partial function in the lifted domain that only produces a result for identifiers that are defined. This is then subsequently used to restrict the behavior of *editread*.
  In *Clean*, partial functions can be modeled by producing $\perp$ for the input values that are not in its domain. In *EditorArrowCore*, *read* will therefore be defined in such a way that it produces $\perp$ if $id = \perp$, and performs the required read operation on the editable data otherwise.
- In the *EditorArrow* model, *write* can be regarded as a partial function in the lifted domain that only produces a result for identifiers and values that are defined. This is then subsequently used to restrict the behavior of *editset*.
  In *EditorArrowCore*, *write* will be defined in such a way that it produces $\perp$ if either $id = \perp$ or $v = \perp$, and performs the required write operation on the editable data otherwise.

This leads to the following definitions of *read* and *write*:

```
evalString :: !String → Bool                                                     1.
evalString s                                                                     2.
    = True                                                                       3.

evalEditorId :: EditorId → Bool                                                  4.
evalEditorId (name, value)                                                       5.
    = evalString name && evalString value                                        6.

read :: EditorId EditableData → SerializedValue                                  7.
read id data                                                                     8.
    | not (evalEditorId id)          = ⊥                                         9.
    = read‘ id data                                                              10.
    where                                                                        11.
```

```
        read` id [record:data]                                          12.
            | fst record == id      = snd record                        13.
            | otherwise             = read` id data                     14.
        read` id []                                                     15.
            = snd id                                                    16.

write :: EditorId SerializedValue EditableData → EditableData           17.
write id value data                                                     18.
    | not (evalEditorId id)         = ⊥                                 19.
    | not (evalString value)        = ⊥                                 20.
    = write` id value data                                              21.
    where                                                               22.
        write` id value [record: data]                                  23.
            | fst record == id      = [(id,value):data]                 24.
            | otherwise             = [record: write` id value data]    25.
        write` id value []                                              26.
            = [(id,value)]                                              27.
```

The definedness conditions are checked by `read` and `write` on lines 9, 19 and 20. For checking the definedness of a `SerializedValue` (which is actually a `String`), the function `evalString` (lines 1-3) is used. By means of its strictness annotation, it produces `True` for defined values and ⊥ for undefined ones. The definedness of a `EditorId` is checked with `evalEditorId` (lines 4-6), which makes use of pattern matching and translates to two calls of `evalString`. Because of the explicit pattern match, it does not need a strictness annotation in front of its `EditorId` argument.

Using `read` and `write`, the operations *editread* and *editset* can now be defined in *EditorArrowCore* as follows:

```
editread :: EDET EditorId a | editable a                                1.
editread (ev_id, v) (data, id, status)                                  2.
    | status == Pending && ev_id == id                                  3.
        #! data                     = write id v data                   4.
        = (data, deserialize v, Processed)                              5.
    | otherwise                                                         6.
        #! read_v                   = read id data                      7.
        = (data, deserialize read_v, status)                            8.

editset :: EDET (EditorId, a) a | editable a                            9.
editset (ev_id, v) (data, (id, a), status)                              10.
    | status == Pending && ev_id == id                                  11.
        #! data                     = write id v data                   12.
        = (data, deserialize v, Processed)                              13.
    | otherwise                                                         14.
        #! data                     = write id (serialize a) data       15.
        = (data, a, status)                                             16.
```

These functions model the operational semantics directly. The strict lets (denoted by `#!`) on lines 4, 7, 12 and 15 model the definedness conditions imposed by *read* and *write*. These strict lets compute a value, and if this value is ⊥ cause `editread` and `editset` to produce ⊥ as a whole. Note that as was discussed earlier, explicit conversion to and from `SerializedValue` is necessary in *EditorArrowCore* for storing values of different types in a single editable data.

Finally, the execution of an arrow on a scenario is realized by applying events one by one on the arrow. This *eventloop* is defined in a general way for all editor arrows of type `EDET a b`. It requires an initial value of type `a`, which is needed at every event to get started, and it throws away the result value of type `b`, assuming instead that the editable data is used for transferring information from one event to the next. It also requires an initial editable data.

```
:: Scenario          :== [Event]                                                    1.

eventloop :: (EDET a b) (EditableData, a) Scenario → EditableData                   2.
eventloop f (data, a) [event:events]                                                3.
    # (data, _, _)       = f event (data, a, Pending)                              4.
    = eventloop f (data, a) events                                                  5.
eventloop f (data, a) []                                                            6.
    = data                                                                          7.
```

To execute an arrow in *EditorArrowCore*, it must be wrapped in an application of `eventloop`. For the initial editable data, `[]` can be filled in to indicate that all editors should start at their specified initial values. The scenario input corresponds to user actions which must be processed by the arrow and can be chosen freely. The `varsumlist` arrow of Sect. 2.3 can be wrapped in *EditorArrowCore* as follows:

```
module varsumlist_EAC                                                               1.

import StdEnv, EditorArrowCore                                                      2.

Start = eventloop varsumlist ([], ⊥)                                                3.
            [(nrId, "2"), (argId 1, "30"), (argId 2, "12"),                         4.
             (nrId, "1"), (nrId, "3"), (argId 3, "58")]                             5.
```

Note that `varsumlist` does not use its initial value, therefore ⊥ can be used for it safely (line 3). The user actions of Sect. 2.3 have been modeled by a list of six events (lines 3-5). Note that the value in each event must be provided in serialized format.

5.2 Derived editor arrows in the EditorArrowCore library

The base arrow operations of *EditorArrowCore* are sufficiently powerful to express many example programs, but are still rather unfriendly for programming purposes. In this section, a layer of derived arrow operations will therefore be defined on top of the base layer. The derived operations are applications of existing arrows only, and can be used in *EditorArrowCore*, *GEC* and *iData*. In Sect. 5.3, the derived operations will be used to construct example programs with ease.

The derived arrow operations consist of useful abbreviations for commonly used functionality, an operation for branching into separate computations, operations for performing choice based on the arrow state, and an arrow version of *map*. First, abbreviations are introduced for functions that are often lifted to the arrow level:

```
dupl        :== λx → (x,x)                                                          1.
set a       :== λx → a                                                              2.
add1 a      :== λb → (a,b)                                                          3.
add2 b      :== λa → (a,b)                                                          4.
```

The function `dupl` (line 1) duplicates an arrow state, which is useful if an operation is applied that unwantedly consumes its input. The functions `set`, `add1` and `add2` (line 2-4) introduce a constant into in the arrow state, which is useful for operations that need constant input only (apply `set` beforehand) and operations that need a combination of state and constant input (apply `add1` or `add2` beforehand).

The following abbreviations introduce special notations for specific applications of `arr` that are often needed in combined arrow expressions:

```
arr2 f      :== arr (λ(a,b) → f a b)                    1.
(@) f g     :== arr g >>> f                             2.
```

The operation `arr2` (line 1) combines two separately computed values to a single one by means of the application of a function. The infix operation `@` (line 2) inserts an `arr` before an abritrary operation, which is useful if the operation requires a small state transformation to be applicable. In particular, it is handy for providing editor ids to `editread` and `editset` by means of (`editread @ set id`) and (`editset @ add1 id`), which in Sect. 4.3 were even abbreviated further to *editread* ⊡ *id* and *editset* ⊙ *id*.

Arrows often require separate computations to be carried out independently, after which the results are combined again. This behavior can be achieved by means of `first` and its dual `second`, but they both require the arrow state to be a tuple in the first place. In order to conveniently start separate computations from a single value, the operation `branch` is defined:

```
second :: (EDET a b) → EDET (c,a) (c,b)                 1.
second f = arr swap >>> first f >>> arr swap            2.
       where swap (x,y) = (y,x)                         3.

branch :: (EDET a b) (EDET a c) → EDET a (b, c)         4.
branch f g = arr dupl >>> first f >>> second g          5.
```

The well-known operation `second` (lines 1-3) is the dual of `first` and allows an arrow to be executed on the right-hand-side of a tuple only. The operation `branch` (lines 4-5) duplicates its input value, which in fact creates two separate branches, and executes its first argument on the first branch and its second argument on the second branch. Combining the values afterwards must be performed separately.

For programming purposes, it is important that an arrow operation is available that chooses between computations based on the contents of the arrow state. The base layer does not define such an operation, but it can be expressed in terms of `left` as follows:

```
right :: (EDET b c) → EDET (Either a b) (Either a c)    1.
right f     = arr swap >>> left f >>> arr swap           2.
            where   swap (Left a)   = Right a           3.
                    swap (Right b)  = Left b            4.

choice :: (EDET l b) (EDET r b) → EDET (Either l r) b   5.
choice f g  = left f >>> right g >>> arr remove_either  6.
            where   remove_either (Left x)  = x          7.
                    remove_either (Right x) = x          8.

ifthenelse :: (a → Bool) (EDET a b) (EDET a b) → EDET a b   9.
```

```
ifthenelse p f g =  arr (λa → if (p a) (Left a) (Right a))                    10.
                 >>> choice f g                                               11.
```

The operation `right` (lines 1-4) is the dual of `left`. The standard operation `choice` (lines 5-8) chooses between its arguments on the basis of the arrow state: a `Left` triggers execution of the first argument and a `Right` execution of the second. The operation `ifthenelse` (lines 9-11) lifts choice to predicates by internally converting to an `Either` based on the outcome of the predicate.

In a truly functional manner, it is possible to lift basic arrow operations to lists as well. We will demonstrate this by realizing a *map* in terms of `iterateN`. The idea is to repeatedly pop the first element of the list, apply the arrow to it and put the transformed element back at the *end* of the list. This must be iterated exactly as many times as the list is long:

```
mapA :: (EDET a a) → EDET [a] [a]                                             1.
mapA f = arr (λas → (length as,as)) >>> iterateN (inner_app f)                2.
       where inner_app f    =  arr    (λ(_, [a:as]) → (a,as))                3.
                               >>> first f                                    4.
                               >>> arr    (λ(a,as) → as ++ [a])               5.
```

Many other derived applications can of course be defined as well, and the actual *EditorArrowCore* library contains more operations than are defined in this section. It is not the purpose of this paper to list all these operations, however.

5.3 Some Small Editor Arrows Programs

In Sect. 2.3, an example editor arrow was described with which the sum of a variable number of editors was computed. Using the derived operations of *EditorArrowCore*, this editor arrow can now be expressed much more elegantly, as follows:

```
variable_sum_arrow :: EDET Int Int                                            1.
variable_sum_arrow                                                            2.
    =   editread    @ (set nrId)                                              3.
   >>>  iterateN    (first (editread @ argId) >>> arr2 (+)) @ (add2 0)         4.
   >>>  editset     @ (add1 sumId)                                            5.
```

The main difference is that all applications of `arr` which were used to add a constant value to the arrow value have been replaced with applications of `@`. This is not only more compact, but also describes the intention of these constant values (they are used as fixed input for the next arrow) more clearly.

This editor arrow can be executed in *EditorArrowCore*. We will use the scenario of Sect. 2.3, modeling the user actions with a list of `Events`. By printing the events and the intermediate states, this results in the following output in *EditorArrowCore*:

```
[]                                                                            1.
→ Event(nr, 2)                                                                2.
[nr=2; sum=0]                                                                 3.
→ Event(arg 1, 30)                                                            4.
[nr=2; arg 1=30; sum=30]                                                      5.
→ Event(arg 2, 12)                                                            6.
[nr=2; arg 1=30; arg 2=12; sum=42]                                            7.
```

```
→ Event(nr, 1)                                                    8.
 [nr=1; arg 1=30; arg 2=12; sum=30]                               9.
→ Event(nr, 3)                                                    10.
 [nr=3; arg 1=30; arg 2=12; sum=42]                               11.
→ Event(arg 3, 58)                                                12.
 [nr=3; arg 1=30; arg 2=12; arg 3=58; sum=100]                    13.
```

The incoming events are shown on lines 2,4,6,8,10 and 12. The editable data, which contain the current values of the editors, are shown on lines 1,3,5,7,9,11 and 13. Note that editors that do not have an entry in the editable data are still at their initial value (which is 0 for all editors in this example). The states at lines 1, 7, 9 and 13 correspond with the screenshots in Sect. 2.3.

Another interesting example is a convertor between euro's and dollars. It consists of a euro editor and a dollar editor which are connected in such a way that a change in one editor causes the other editor to be updated. In arrow style, this can be realized by a shared feedback of the form *euro* ⋙ *dollar* ⋙ *euro*, as follows:

```
convert_arrow :: EDET a Real                                      1.
convert_arrow                                                     2.
    =   editread    @ (set euroId)                                3.
   >>>  arr         toDollar                                      4.
   >>>  editset     @ (add1 dollarId)                             5.
   >>>  arr         toEuro                                        6.
   >>>  editset     @ (add1 euroId)                               7.
    where                                                         8.
        toDollar euro   = euro * 1.592                            9.
        toEuro dollar   = dollar / 1.592                          10.
```

Finally, the following editor arrow changes indicated values in a list. It consists of two editors, one to input the index of the element, and one to change its value. The list itself is stored in the arrow state, and is never sent to an editor. Therefore, this example works both for finite and for infinite lists.

```
list_editor :: EDET [a] [a] | editable a                          1.
list_editor                                                       2.
    =   branch      (editread @ set indexId) skip                 3.
   >>>  arr         (λ(i,list) → (list!!i, (i,list)))             4.
   >>>  first       (editset @ (add1 fieldId))                    5.
   >>>  arr         (λ(n,(i,list)) → updateAt i n list)           6.
```

5.4 Arrow laws for Sparkle

By implementing the *EditorArrow* model in *Clean*, it also becomes possible to make use of its integrated proof assistant *Sparkle* (de Mol *et al.*, 2008). In this section, we will translate the laws of Sect. 4 to *EditorArrowCore*, which allows their correctness to be verified by proving them with *Sparkle*.

The realization of editor arrows in *Clean* follows the operational semantics as closely as possible. As a result, there is only one difference between the behavior of *EditorArrowCore* and *EditorArrow*. This difference is due to the lazy semantics of *Clean*, which makes it possible for an editor arrow to get an undefined event,

editable data or event status as input. The behavior in these cases has not been defined by the semantics, and may falsify the laws of Sect. 4.

If the incoming event, editable data and event status are all defined, then editor arrows in *EditorArrowCore* behave exactly the same as in the *EditorArrow* model. By explicitly enforcing these definedness conditions, the laws can be transferred to *Sparkle* directly. For this purpose, we implement the following eval functions:

```
evalEvent :: Event → Bool                                                    1.
evalEvent (id, v)                                                            2.
    = evalEditorId && evalValue v                                            3.

evalEState :: (a → Bool) (EState a) → Bool                                   4.
evalEState eval_a (data, a, status)                                          5.
    = evalEditableData data && eval_a a && evalEventStatus status            6.

evalEditableData :: EditableData → Bool                                      7.
evalEditableData [(id, v): data]                                             8.
    = evalEditorId id && evalValue v && evalEditableData data                9.
evalEditableData []                                                          10.
    = True                                                                   11.

evalEventStatus :: EventStatus → Bool                                        12.
evalEventStatus Pending     = True                                           13.
evalEventStatus Processed   = True                                           14.
```

Note that `evalEditorId` and `evalValue` were already defined in Sect. 5.1. The other eval functions are defined here in the same manner. The function `evalEState` (lines 4-6) has been augmented with a custom eval predicate for values because this additional predicate is needed for translating the definedness laws of Sect. 4.4.

The laws of Sect. 4 can now be transferred to *Sparkle* directly. We demonstrate this for the following three laws:

Law '$\ggg$ def': $Def(f, A, B) \Rightarrow Def(g, B, C) \Rightarrow Def(f \ggg g, A, C)$
Sparkle:
```
evalEvent ev
-> evalEState A state
-> ([e][s] evalEvent e -> evalEState A s -> evalEState B (f e s))
-> ([e][s] evalEvent e -> evalEState B s -> evalEState C (g e s))
-> evalEState C ((f >>> g) ev state)
```
Notes: With additional definedness conditions, the translation of $Def(f, A, B)$ is `[e][s] evalEvent e -> evalEState A s -> evalEState B (f e s)`. The *Sparkle* law can be obtained by applying this translation three times, and eliminating the outer universal quantors (which are optional in *Sparkle*).

Law 'assoc eliminates first': $first\ (first\ f) \ggg assoc = arr\ assoc \ggg first\ f$
Sparkle:
```
evalEvent ev
-> evalEState (A o fst o fst) state
-> ([e][s] evalEvent e -> evalEState A s -> evalEState B (f e s))
-> (first (first f) >>> arr assoc) ev state
   = (arr assoc >>> first f) ev state
```
Notes: The original law can be found in the last line of the translation. The first two lines ensure that the incoming event and state are defined, and that A holds

for the *fst* of the *fst* of the state. The third line corresponds to $Def(f, A, B)$, and ensures that applying f on the *fst* of the *fst* of the state yields a defined result.

Law 'read-read elimination': $editread \boxdot i \ggg editread \boxdot i = editread \boxdot i$

Sparkle:
```
evalEditorId id
-> evalEState A state
-> (editread @ (set id) >>> editread @ (set id)) ev state
   = (editread @ (set id)) ev state
```

Notes: The original law can be found in the last line of the translation, realizing $\boxdot i$ with `@ (set id)`. The additional definedness conditions ensure that the editor id and the incoming state are both defined.

Proofs of all the transferred laws can be constructed easily in *Sparkle*. This ensures that the laws in Sect. 4 are indeed correct.

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. The model uses the same level of abstraction as the toolkits by considering the elementary interactive components as being editors of arbitrary values that can be edited by the user. The elementary elements are glued together by means of the *EditorArrow* 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* (Courtney & Elliott, 2001) and *Fran* (Hudak *et al.*, 2003b) have taken this route as well. These systems 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 (Milner, 1980), CSP (Hoare, 1985), ACP (Baeten & Weijland, 1990), and μCRL (Groote & Reniers, 2001). 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

We have introduced the formal *EditorArrow* semantic model of the *GEC* and the *iData* toolkit. This model is based on the *Arrow* framework. It essentially extends

the basic framework with *iteration* instead of loops and with primitive combinator functions, *editread* and *editset*, for creating editors with shared state.

Apart from the classic associated *Arrow* laws we have formulated a number of additional laws for iteration and for editors. Furthermore, we have introduced definedness laws for the semantic model. This is relevant because the *edit* combinators impose very strict requirements on their input values, output values and events that are passed through the system, which is in contrast with the requirements of the standard *Arrow* combinators.

The use of *Sparkle* greatly increased confidence in the correctness of the proven laws. In addition, working with this proof assistant helped us to identify issues that escaped our attention in the process of specifying the model and its theorems.

References

Achten, Peter. 2007 (July 13). *Clean for Haskell98 Programmers – A Quick Reference Guide –*. Available at: `http://www.st.cs.ru.nl/papers/2007/CleanHaskellQuickGuide.pdf`.

Achten, Peter, & Plasmeijer, Rinus. (1998). Interactive Functional Objects in Clean. *Pages 304–321 of:* Clack, C., Hammond, K., & Davie, T. (eds), *Proc. of the 9th International Workshop on the Implementation of Functional Languages, IFL 1997, St.Andrews, UK, Selected Papers.* LNCS, vol. 1467. Springer Verlag.

Achten, Peter, van Eekelen, Marko, & Plasmeijer, Rinus. (2003). Generic Graphical User Interfaces. Michaelson, Greg, & Trinder, Phil (eds), *Selected Papers of the 15th Int. Workshop on the Implementation of Functional Languages, IFL03, Edinburgh, UK*. LNCS, vol. 3145. Springer Verlag.

Achten, Peter, van Eekelen, Marko, Plasmeijer, Rinus, & van Weelden, Arjen. (2004a). Automatic Generation of Editors for Higher-Order Data Structures. *Pages 262–279 of:* Wei-Ngan Chin (ed), *Second Asian Symposium on Programming Languages and Systems (APLAS 2004).* LNCS, vol. 3302. Springer Verlag.

Achten, Peter, van Eekelen, Marko, & Plasmeijer, Rinus. (2004b). Compositional Model-Views with Generic Graphical User Interfaces. *Pages 39–55 of: Practical Aspects of Declarative Programming, PADL04.* LNCS, vol. 3057. Springer Verlag.

Baeten, J.C.M., & Weijland, W.P. (1990). *Process Algebra.* Cambridge Tracts. in Theoretical Computer Science, vol. 18. Cambridge University Press.

Cartwright, Robert, & Donahue, James. (1982). The semantics of lazy (and industrious) evaluation. *Pages 253–264 of: Lfp '82: Proceedings of the 1982 acm symposium on lisp and functional programming.* New York, NY, USA: ACM.

Courtney, A., & Elliott, C. 2001 (September). Genuinely Functional User Interfaces. *Proceedings of the 2001 Haskell Workshop.*

Courtney, Antony, Nilsson, Henrik, & Peterson, John. (2003). The Yampa arcade. *Pages 7–18 of: Proceedings of the 2003 ACM SIGPLAN Haskell workshop (Haskell'03).* Uppsala, Sweden: ACM Press.

de Mol, Maarten, van Eekelen, Marko, & Plasmeijer, Rinus. (2002). Theorem Proving for functional Programmers - Sparkle: A Functional Theorem Prover. *Pages 55–72 of:* Arts, Thomas, & Mohnen, Markus (eds), *The 13th International Workshop on Implementation of Functional Languages, IFL 2001, Stockholm, Sweden, Selected Papers.* LNCS, vol. 2312. Springer Verlag.

de Mol, Maarten, van Eekelen, Marko, & Plasmeijer, Rinus. (2008). Proving Properties of Lazy Functional Programs with SPARKLE. Horváth, Zoltán (ed), *2nd Central-European*

Functional Programming School, CEFP 2007, Cluj-Napoca, Romania. LNCS Tutorial Series. Springer Verlag. To appear.

Elliott, Conal, & Hudak, Paul. 1997 (June). Functional reactive animation. *Pages 163–173 of: International conference on functional programming.*

Groote, J.F., & Reniers, M.A. (2001). Algebraic Process Verification. *Chap. 17, pages 1151–1208 of:* Bergstra, J.A., Ponse, A., & Smolka, S.A. (eds), *Handbook of Process Algebra.* Elsevier Science B.V.

Hoare, C.A.R. (1985). *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall International.

Hudak, Paul, Courtney, Antony, Nilsson, Henrik, & Peterson, John. (2003a). Arrows, robots, and functional reactive programming. *Pages 159–187 of: Summer school on advanced functional programming 2002, oxford university.* Lecture Notes in Computer Science, vol. 2638. Springer-Verlag.

Hudak, Paul, Courtney, Antony, Nilsson, Henrik, & Peterson, John. (2003b). Arrows, Robots, and Functional Reactive Programming. *Pages 159–187 of:* Jeuring, Johan, & Peyton Jones, Simon (eds), *Advanced Functional Programming, 4th International School, Oxford.* LNCS, vol. 2638. Springer Verlag.

Hughes, John. (2000). Generalising Monads to Arrows. *Science of Computer Programming*, **37**(May), 67–111.

Milner, Robert. (1980). *A Calculus of Communicating Systems.* LNCS, vol. 92. Springer Verlag.

Plasmeijer, Rinus, & Achten, Peter. (2005). Generic Editors for the World Wide Web. *Pages 1–34 of: Central-European Functional Programming School, Eötvös Loránd University, Budapest, Hungary - Revised Selected Lectures.* LNCS, vol. 4164. Springer Verlag.

Plasmeijer, Rinus, & Achten, Peter. 2006 (September 19-21). The Implementation of iData - A Case Study in Generic Programming. *Pages 106–123 of:* Andrew Butterfield and Clemens Grelck and Frank Huch (ed), *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Dublin, Ireland, September 19-21, 2005, Revised Selected Papers.* LNCS 4015.

van Eekelen, Marko, & de Mol, Maarten. 2007 (December 17). Proving Lazy Folklore with Mixed Lazy/Strict Semantics. *Pages 87–100 of:* Barendsen, Erik, Capretta, Venanzio, Geuvers, Herman, & Milad, Niqui (eds), *Reflections on Type Theory, λ-Calculus, and the Mind - Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday.* Radboud University Nijmegen, The Netherlands.

van Eekelen, Marko C. J. D., & de Mol, Maarten. (2005). Proof Tool Support for Explicit Strictness. *Pages 37–54 of:* Butterfield, Andrew, Grelck, Clemens, & Huch, Frank (eds), *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Dublin, Ireland, September 19-21, 2005, Revised Selected Papers.* LNCS, vol. 4015. Springer Verlag.