JFP **23** (2): 185–224, 2013. © Cambridge University Press 2012
doi:10.1017/S0956796812000421 First published online 22 November 2012

EditorArrow: An arrow-based model for editor-based programming

PETER ACHTEN
Institute for Computing and Information Sciences, Radboud University Nijmegen, Nijmegen, The Netherlands
(*e-mail:* `P.Achten@cs.ru.nl`)

MARKO VAN EEKELEN
Institute for Computing and Information Sciences, Radboud University Nijmegen, Nijmegen, The Netherlands
and
School of Computer Science, Open University of The Netherlands, Heerlen, The Netherlands
(*e-mail:* `M.vanEekelen@cs.ru.nl`)

MAARTEN DE MOL
Formal Methods and Tools, University of Twente, Enschede, The Netherlands
(*e-mail:* `M.J.deMol@utwente.nl`)

RINUS PLASMEIJER
Institute for Computing and Information Sciences, Radboud University Nijmegen, Nijmegen, The Netherlands
(*e-mail:* `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 *EditorArrow*: a single, unified semantic model that defines shared state and event handling. We study the properties of *EditorArrow*, 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*. The availability of this tool has proved to be indispensable in this endeavor.

1 Introduction

Programming 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 promptly so that the application is sufficiently responsive (the $\frac{1}{2}s$ rule, see also Shneiderman, 1992). 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 be 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 (see also Thiemann, 2002; Hanus, 2002). The code that computes the page needs to terminate promptly, otherwise the browser will give up. In both situations, the resulting code can easily be hundreds of lines long, with intricate interdependencies.

How can you convince yourself, or other stakeholders, that the program is correct with respect to its requirements? Ideally, one would like to *formally prove* properties for this purpose (Dowse *et al.*, 2004). Unfortunately, neither the desktop nor the web has a formally specified reasoning model. In this paper, we create a common underlying formal model, called *EditorArrow*, that allows formal reasoning to take place for a class of interactive programs that are *event driven* and in which graphical units behave as *model*-*view* components, i.e. they are *identified* units that have a *state* (model) that is *rendered* (view) to allow users to view and alter the state. Note that both widget-based and web-based applications are examples of such interactive programs.

The motivation for this work is our experience in building this class of interactive applications both for widget-based systems and web-based systems. The first resulted in the *Graphical Editor Components* (*GEC*) toolkit (Achten *et al.*, 2003, 2004a, 2004b, 2004c) and the second in the *iData* toolkit (Plasmeijer & Achten, 2005, 2006c). One distinctive feature of both approaches is that they allow the developer to *abstract* from rendering issues *and* event handling issues by using generic programming techniques to derive a GUI and its event handling from a type. Such units are called *editors*, because at a conceptual level, the developer can think of interactive elements as components that render a state that can be edited by the application user. The model values of these editors operate as *editable data* that are altered in an *event-driven* way. Despite the different back-ends of both toolkits, their *api* can be captured in the *Arrow* (Hughes, 2000) framework. For this reason, we base *EditorArrow* on *Arrow*s. We show that for the class of state-based, event-driven interactive programs the *Arrow* framework satisfies the usual *Arrow* laws and other properties that are required for formal reasoning. We think that the resulting model is sufficiently general to capture an interesting range of systems. In Section 6 we argue informally that the well-known *Fudgets* system can be modeled within *EditorArrow*.

In order to verify the correctness of *EditorArrow*, we create a *reference implementation* of *EditorArrow*, called *EditorArrowCore*, in the pure and lazy, graph rewriting, functional programming language *Clean* (Eekelen *et al.*, 1997; Plasmeijer & Eekelen, 1999, 2002) (for readers who are more familiar with the functional language *Haskell* (Hudak *et al.*, 2007), we refer to Groningen *et al.*, 2010, for a detailed overview of the differences between *Clean* and *Haskell* and how to merge the two languages). We use *Clean* because it comes with the interactive proof assistant *Sparkle* (Mol *et al.*, 2002, 2008; Kesteren *et al.*, 2004; Mol, 2009). We use *Sparkle* and its special support for reasoning about strictness (Eekelen & Mol, 2005) to determine and analyze the definedness properties of the *EditorArrow* framework.

The layout and contributions of this paper are as follows:

- We develop *EditorArrow*: a formal *Arrow* model for event-driven state-based interactive applications and give a denotational and operational semantics (Section 2).
- We show that the *Arrow* laws hold within *EditorArrow*, and identify a useful set of additional laws that are concerned with interconnecting editors, the *editor* laws. The *Arrow* model is constructed to support partially defined values in a maximal way, resulting in *definedness* laws (Section 3).
- We develop a reference implementation of the semantic model in *Clean* and use it to formally prove the *Arrow*, *editor*, and *definedness* laws with the *Sparkle* tool (Section 4).
- We demonstrate how to use the results of the framework to formally prove correctness properties of concrete examples (Section 5).

Related work is presented in Section 6 and we end with discussion and conclusions in Section 7.

2 *EditorArrow*

In *EditorArrow*, an editor is regarded as a uniquely named, typed storage for a single value that presents a GUI to allow the application user to alter (*edit*) this *data* value. This edit action is an *event*. 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 the value held by another editor is received. An interactive application is a collection of such connected editors. The *EditorArrow* model uses the *Arrow* combinators to connect editors. The *Arrow* instance is defined in the domain of editable data and event transformer functions.

In this section, we introduce the generalized framework and motivate its components in Section 2.1, together with a small example. In Section 2.2, we give the formal denotational semantics of *EditorArrow*, and in Section 2.3 the operational semantics.

2.1 Design considerations

With *EditorArrow*, we wish to reason about the behavior of interactive applications, regardless of whether these are constructed for the desktop or the web. In this section, we motivate the design considerations for the *EditorArrow* model.

To start with, we introduce a number of abstractions. First of all, we abstract from representation (widget or form) and layout. In *EditorArrow* 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 them. Second, we abstract from residence of state and assume that every editor has access to its state value and that this state value is persistent. Hence, in *EditorArrow* every modeled program leaves its trace during and after execution in terms of the persistent editor states. Third, 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 new value of the same type of value that is maintained by the editor.

As a result of these abstractions, the basic building blocks in *EditorArrow* are editors of values of any type. Editors are identified by means of a unique label and an initial value. Within the arrow relation, *the same* editor can appear multiple times, by using the same identifier. In this way, intricate relationships can be defined via sequential composition rather than a cyclic combination of editors.

The current value of an editor can be *read* and *set*, for which purpose we introduce two combinator functions, viz. `editread` and `editset`. Both functions receive an editor identifier value via the arrow state, and `editset` is also provided with the new value of the editor. When manipulated by the user, an editor receives a new value and emits that value via the arrow state. The difference between the two operations shows 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.

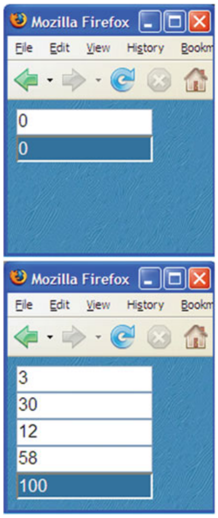
In order to capture repetition, we need some kind of recursion. All interactive applications have some kind of event loop that deals with consecutive events. An editor arrow expression must build a finite, fully evaluated interface for the user. This interface may be dynamic in the sense that the user can influence its values and its size but it must always be finite and fully evaluated. For modeling recursion on the level of such editor arrow expressions, we need nothing more than *primitive* recursion. The corresponding combinator, `iterateN`, receives a number argument that tells how many times a certain *EditorArrow* needs to be repeated.

Finally, as stated in Section 1, we use the standard *Arrow* combinators: `arr`, $\ggg$, and `first` to interconnect editors.

As an example of *EditorArrow*, Figure 1 displays the definition of an interactive program that dynamically creates and removes number input elements depending on the current user input. The example is called `varsumlist`. In Section 5 we formally prove in *EditorArrow* that this program is equivalent to another version. The initial situation of the program is depicted in the upper screenshot: the program only consists of the number input field and a display that shows the sum of the values of the dynamically created number input elements. The number input field, with initial value zero, is identified by `nrId` (line 11) and the sum display, also with initial value zero, is identified by `sumId` (line 12). The number input field is created at line 3 and the sum display at line 9. The read value of the number input field determines the total number of other integer input elements. The *n*th editor is identified by `argId` *n*, and also has initial value zero (lines 5 and 13). These input elements can be edited by the user. After each such edit action, their sum is computed and displayed in the sum display. This can be repeated as many times as the user likes. In the lower screenshot the user has entered 3 in the first editor, resulting in three more editors that have been altered by the user.

```
varsumlist
  =   arr       (λx → nrId)
  >>> editread
  >>> arr       (λn → (n,0))
  >>> iterateN  (   first  (arr argId >>> editread)
                >>> arr    (uncurry (+))
                )
  >>> arr       (λt → (sumId,t))
  >>> editset

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



```
0
0
```

```
3
30
12
58
100
```

Fig. 1. (Colour online) The `varsumlist` program in *Arrow* style and its *iData* rendering.

2.2 Denotational semantics for *EditorArrow*

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

$$Signal\ \ a = Time \to a$$

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 Section 2.1, 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 modeled 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 model this in a strongly-typed programming language (as we will in Section 4), we need existential or dependent types. 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 $eid : ID$, she 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.

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

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.

$$\begin{aligned} &\mathit{read} &&: \mathit{ID} \to \mathit{EditableData} \to \mathit{Value} \\ &\mathit{read}\ \mathit{eid}\ s &&= \begin{cases} d & \textbf{if } (\mathit{eid}, d) \in s \\ \mathit{initvalue}\ \mathit{eid} & \textbf{if } (\mathit{eid}, d) \notin s \end{cases} \end{aligned}$$

$$\begin{aligned} &\mathit{write} &&: \mathit{ID} \to \mathit{Value} \to \mathit{EditableData} \to \mathit{EditableData} \\ &\mathit{write}\ \mathit{eid}\ v\ s &&= \begin{cases} (\mathit{eid}, v) \cup s \backslash (\mathit{eid}, d) & \textbf{if } (\mathit{eid}, d) \in s \\ (\mathit{eid}, v) \cup s & \textbf{if } (\mathit{eid}, d) \notin s \end{cases} \end{aligned}$$

The *ID* values serve as unique keys in s : *EditableData*:

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

Programs are constructed by means of the *Arrow* combinators. An *Arrow* program fragment processes an event. This is modeled by:

$$\begin{aligned} &\mathit{EventStatus} = \{\mathit{Pending}, \mathit{Processed}\} \\ &\mathit{pending} \quad : \ \mathit{EventStatus} \to \mathit{Bool} \\ &\mathit{processed} \ : \ \mathit{EventStatus} \to \mathit{Bool} \end{aligned}$$

The predicates *pending* and *processed* 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*. Why this is a *partial* function will be explained later in Section 3.4.

$$\begin{aligned} \mathit{EState}\ a &= \mathit{EditableData} \times a \times \mathit{EventStatus} \\ \mathit{EDET}\ a\ b &= \mathit{EditEvent} \to \mathit{EState}\ a \hookrightarrow \mathit{EState}\ b \end{aligned}$$

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

The grammar of arrow expressions is:

$$\begin{array}{lll} \mathit{EdArrow} ::= & \mathit{arr}\ \ \mathit{Fun} & \text{(lift pure function to } \mathit{Arrow} \text{ domain)} \\ \mid & \mathit{EdArrow} \ggg \mathit{EdArrow} & \text{(sequential composition)} \\ \mid & \mathit{first}\ \ \mathit{EdArrow} & \text{(bypass information)} \\ \mid & \mathit{left}\ \ \mathit{EdArrow} & \text{(handle left alternatives only)} \\ \mid & \mathit{iterateN}\ \ \mathit{EdArrow} & \text{(primitive recursion)} \\ \mid & \mathit{editread} & \text{(read editor state)} \\ \mid & \mathit{editset} & \text{(write editor state)} \end{array}$$

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

Denotationally, we define a function $[\![-]\!]$ from these arrow expressions to the functions on the *EDET* domain.

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

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.

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

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.

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

The meaning of the two combinators for the basic editor variants, *editread* and *editset*, is defined straightforwardly using the functions *read*, *write* and *pending*. We follow the intuitive meaning described in Section 2.1 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.

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

The primitive recursion *iterateN* combinator 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.

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

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 loop.

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

The toolkit wrappers are modeled by a loop combinator as is introduced by Paterson (2001). The loop combinator is defined using the standard least fixed point combinator **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 *iterateN* within arrow expressions allows us to use primitive recursion over the *Arrow* structure, whereas *eventloop* takes care that we can have an interactive system with persistent state.

2.3 Operational semantics for *EditorArrow*

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 treat specially the case of a "pending" event that must be processed by an editor with matching id. The operational semantics employs the same primitives (*pending*, *read* and *write*) as the denotational semantics.

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

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

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

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

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

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

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

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.

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

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

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 Section 4.

3 Properties of *EditorArrow*

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 (2000) and Paterson (2001), are valid for this model. These laws are given as Definition 1.

In Section 3.1 we introduce "iterate" laws and in Section 3.2 we give properties of the "eventloop." We introduce basic "editor" laws in Section 3.3. Finally, we provide "definedness" laws in Section 3.4.

Definition 1 (Classic Arrow Laws)

$$
\boxed{
\begin{array}{rcl}
arr\ id\ \ggg f & \overset{(Left\_unit)}{=}\ f\ \overset{(Right\_unit)}{=} & f \ggg arr\ id \\
f \ggg (g \ggg h) & \overset{(associativity\ \ of \ggg)}{=} & (f \ggg g) \ggg h \\
arr\ (g\ \circ\ f) & \overset{(o\ \ preserves\ \ \ggg)}{=} & arr\ f \ggg arr\ g \\
 & & \\
first\ (arr\ f) & \overset{(first\ \ extension)}{=} & arr\ (f \times id) \\
first\ (f \ggg g) & \overset{(first\ \ preserves\ \ \ggg)}{=} & first\ f \ggg first\ g \\
first\ f \ggg arr\ (id \times g) & \overset{(first\_swap)}{=} & arr\ (id \times g) \ggg first\ f \\
first\ f \ggg arr\ fst & \overset{(fst\ \ eliminates\ \ first\ )}{=} & arr\ fst\ \ \ggg f \\
first\ (first\ f) \ggg arr\ assoc & \overset{(assoc\ \ eliminates\ \ first\ )}{=} & arr\ assoc \ggg first\ f \\
 & & \\
left\ (arr\ f) & \overset{(left\ \ extension)}{=} & arr\ \ (f \oplus id) \\
left\ (f \ggg g) & \overset{(left\ \ functor)}{=} & left\ f \ggg left\ g \\
left\ f \ggg arr\ (id \oplus g) & \overset{(left\ \ exchange)}{=} & arr\ (id \oplus g) \ggg left\ f \\
arr\ Left \ggg left\ f & \overset{(left\_unit)}{=} & f \ggg arr\ Left \\
left\ (left\ f) \ggg arr\ assocsum & \overset{(left\ \ association)}{=} & arr\ assocsum \ggg left\ f \\
\textbf{where} & & \\
fst\ (a,b) & = a & \\
f \times g\ (a,b) & = (f\ a, g\ b) & \\
f \oplus g\ (Left\ a) & = Left\ (f\ a) & \\
f \oplus g\ (Right\ b) & = Right\ (g\ b) & \\
assoc\ ((a,b),c) & = (a,(b,c)) & \\
assocsum\ (Left\ (Left\ a)) & = Left\ a & \\
assocsum\ (Left\ (Right\ b)) & = Right\ (Left\ b) & \\
assocsum\ (Right\ c) & = Right\ (Right\ c) &
\end{array}
}
$$

3.1 *Iterate laws*

Definition 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 *iterateN* can use this number to count the iterations.

The $iterateN - base$ law expresses the fact that the argument is applied zero times. The $iterateN - next$ law expresses the fact that the argument is applied $m+1$ times consecutively with decreasing values starting with $m+1$.

Definition 2 (Iterate Laws)

$$
\boxed{
\begin{array}{rcl}
iterateN\ f \odot 0 & \overset{(iterateN-base)}{=} & arr\ \ snd \odot 0 \\
iterateN\ f \odot (m+1) & \overset{(iterateN-next)}{=} & f \odot (m+1) \ggg iterateN\ f \odot m \\
 & & \\
\textbf{where} & & \\
 & f \odot a = arr\ (\lambda x \rightarrow (a,x)) \ggg f &
\end{array}
}
$$

Definition 3 (Eventloop Properties)

$$
\boxed{
\begin{array}{lcl}
eventloop\ f\ (s,a)\ [\,] & \overset{(eventloop-end)}{=} & s \\
eventloop\ f\ (s,a)\ [e:es] & \overset{(eventloop-next)}{=} & eventloop\ f \\
 & & (drop\ ((arr\ dupl \ggg first\ f \ggg arr\ snd) \\
 & & \qquad\qquad e\ (s,a,Pending) \\
 & & \qquad\qquad\quad) \\
 & &)\ es \\
\textbf{where} & & \\
 & dupl = \lambda x \rightarrow (x,x) & \\
 & drop = \lambda(s,a,p) \rightarrow (s,a) &
\end{array}
}
$$

3.2 Eventloop properties

The properties of the eventloop are given in Definition 3. The property *eventloop − end* expresses that in the absence of events, the store is the result of the program. The *eventloop − 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.

3.3 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 large number of equivalences "for free" when the edit combinators are involved.

In addition, we introduce 10 laws that are specific to uses of *editread* and *editset*. They are given as Definition 4. We express that *editset* and *editread* expect a proper identification value *id* with the auxiliary function $\odot$ to put the *id* at the right place for *editset* and another auxiliary function $\boxdot$ to put the *id* at the right place for *editread*.

- We distinguish four *edit elimination* laws (one for each combination of the two *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*).
- 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 $***$ 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 to the value that is edited by a user and store its result for this editor. In this way, editors can control the values

Definition 4 (Editor Laws)

$$
\begin{aligned}
editread \boxdot i \ggg editread \boxdot i &\overset{(read-read\_elimination)}{=} editread \boxdot i \\
editread \boxdot i \ggg editset \odot i &\overset{(read-set\_elimination)}{=} editread \boxdot i \\
editset \odot i \ggg editread \boxdot i &\overset{(set-read\_elimination)}{=} editset \odot i \\
editset \odot i \ggg editset \odot i &\overset{(set-set\_elimination)}{=} editset \odot i \\[1em]
editread \boxdot i *** editread \boxdot j &\overset{(read-read\_swap)}{=} editread \boxdot j \star\star\star editread \boxdot i \\
editread \boxdot i *** editset \odot j &\overset{(read-set\_swap)}{=} editset \odot j \star\star\star editread \boxdot i \\
editset \odot i *** editread \boxdot j &\overset{(set-read\_swap)}{=} editread \boxdot j \star\star\star editset \odot i \\
editset \odot i *** editset \odot j &\overset{(set-set\_swap)}{=} editset \odot j \star\star\star editset \odot i \\[1em]
self\ f\ i \ggg self\ g\ i &\overset{(self\_composition)}{=} self\ (g \circ f)\ i \\
feedback\ i\ j &\overset{(feedback\_swap)}{=} feedback\ j\ i^{1}
\end{aligned}
$$

where

$$
\begin{aligned}
f \boxdot a &= arr\ (\lambda x \rightarrow a) \ggg f \\
f \odot a &= arr\ (\lambda x \rightarrow (a, x)) \ggg f \\
f *** g &= first\ f \ggg second\ g \\
f \star\star\star g &= second\ f \ggg first\ g \\
self\ f\ i &= editread \boxdot i \ggg arr\ f \ggg editset \odot i \\
feedback\ i\ j &= editread \boxdot i \ggg editset \odot j \ggg editset \odot i
\end{aligned}
$$

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.

3.4 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 consequences for the entire model, which were left implicit in Section 2. In this section, these consequences are made explicit by means of formulating definedness laws.

[1] The “feedback swap” law is only valid if the editors i and j hold the same value initially, which is the case when $read\ \ i\ s = read\ \ j\ s$ for the input editable data s.

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 $\perp$ in it. When values are constructed with tuples or eithers, multiple lifts may even be necessary. The input-output signatures of editor arrows are 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 transforms A to B, g transforms B to C
$first\ f$	$(A \times C)_\perp$	$(B \times C)_\perp$	f transforms A to B
$left\ f$	$(Either\ \ A\ \ C)_\perp$	$(Either\ \ B\ \ C)_\perp$	f transforms A to B
$iterateN\ \ f$	$(\mathbb{N}_\perp \times A)_\perp$	A	f transforms $(\mathbb{N}_\perp \times A)$ to A
$editread$	$ID_\perp$	$Value$	–
$editset$	$(ID_\perp \times A)_\perp$	$Value$	–

Here, $A_\perp$ denotes $A \cup \{\perp\}$, and a "f transforms A to B" on the right is a recursive input/output condition on the editor arrow f. For instance, if f transforms A to A and $a \in A$, then $(0, a)$, $\perp$ and $(\perp, a)$ are all valid input for $iterateN\ \ f$. 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 Sections 2.2 and 2.3, and is the same for the denotational and operational semantics. In the case of $\perp$ values, this behavior can be summarized as follows:

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

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

Case 3: A $\perp$ occurs where structural information is required, and it is not possible to continue. This case covers the following situations:
$left\ f$ on $\perp$ (cannot decide whether to apply f or not);
$iterateN\ \ f$ on $\perp$ and $(\perp, x)$ (cannot decide how many times to apply f);
$editset$ on $\perp$ (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)

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

Definition 7 (Definedness Laws for Editor Arrows)

$$\boxed{\begin{array}{lclll}
f \in A \to B & \overset{(arr\ def)}{\Rightarrow} & Def(arr\ f, & A, & B) \\
Def(f, A, B), Def(g, B, C) & \overset{(\ggg\ def)}{\Rightarrow} & Def(f \ggg g, & A, & C) \\
Def(f, A, B) & \overset{(first\ def\ 1)}{\Rightarrow} & Def(first\ f, & A \times C, & B \times C) \\
Def(f, \{\bot\}, B) & \overset{(first\ def\ 2)}{\Rightarrow} & Def(first\ f, & \{\bot\}, & B \times \{\bot\}) \\
Def(f, A, B) & \overset{(left\ def)}{\Rightarrow} & Def(left\ f, & Either\ A\ C, & Either\ B\ C) \\
Def(f, \mathbb{N} \times A, A) & \overset{(iterate\ def)}{\Rightarrow} & Def(iterateN\ f, & \mathbb{N} \times A, & A) \\
 & {\scriptstyle (editread\ def)} & Def(editread, & ID, & Value) \\
 & {\scriptstyle (editset\ def)} & Def(editset, & ID \times Value, & Value) \\
Def(f, A, B), Def(f, C, D) & \overset{(combine\ def)}{\Rightarrow} & Def(f, & A \cup C, & B \cup D)
\end{array}}$$

(editset-def has been simplified: we must also 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.

4 Programming with editor arrows

In this section, we build a direct implementation of the semantic *EditorArrow* model that was described in Section 2. The implementation is realized by means of a library in *Clean* and is named "*EditorArrowCore* ". The library serves two purposes. First, 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. Second, it is a basis for formal reasoning, because it allows the laws of Section 3 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 Section 4.1. Then, we define composed arrow operations in Section 4.2, which are used to make programming with arrows easier. In Section 4.3, we give a number of example programs. Finally, in Section 4.4 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.

4.1 Base editor arrows in the EditorArrowCore *library*

The *EditorArrowCore* library is a direct implementation of the *EditorArrow* model of Section 2. 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)
:: EState a          :== (EditableData, a, EventStatus)
:: EditableData      :== [(EditorId, SerializedValue)]
:: EventStatus       =   Processed | Pending

:: EditorId          :== (EditorName, InitialValue)
:: EditorName        :== String
:: Event             :== (EditorId, SerializedValue)
:: InitialValue      :== SerializedValue
:: SerializedValue   :== String
```

With respect to the *EditorArrow* model, there are only two differences. First, 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. Second, 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 the problem of 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
where
    serialize       :: a → String
    deserialize     :: String → a
```

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:

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

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 Section 2.2, the grammar *EdArrow* was introduced for editor arrows 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.[2]

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

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

disadvantage that argument editor arrows can only be typed by means of `EDET` as well, and are therefore no longer restricted to well-formed 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:[3]

```
(⋙) :: (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 Section 2. In line with 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 is explained further in Section 4.4.

Next, the operations `left` and `iterateN` are defined.

```
:: 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.
    = (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 $n \leqslant 0$ (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

[3] For reasons of clarity, we simplify the types. In *Clean* the number of type arguments is the number of function arguments, resulting in for instance: `>>> :: (EDET a b) (EDET b c) Event (EState a) -> EState c`

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* are defined, which are 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.
  In *Clean*, partial functions can be modeled by producing $\perp$ for the input values that are not in its domain. In *EditorArrowCore*, *read* is 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 defined identifiers and values.
  In *EditorArrowCore*, *write* is 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
evalString s
    = True

evalEditorId :: EditorId → Bool
evalEditorId (name, value)
    = evalString name && evalString value

read :: EditorId EditableData → SerializedValue
read id data
     | not (evalEditorId id)          = ⊥
    = read‘ id data
     where
         read‘ id [record:data]
             | fst record == id       = snd record
             | otherwise              = read‘ id data
         read‘ id []
             = snd id

write :: EditorId SerializedValue EditableData → EditableData
write id value data
     | not (evalEditorId id)          = ⊥
     | not (evalString value)         = ⊥
    = write‘ id value data
     where
         write‘ id value [record: data]
             | fst record == id       = [(id,value):data]
             | otherwise              = [record: write‘ id value data]
         write‘ id value []
```

```
          = [(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 an `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. Recall that 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 are used for transferring information from one event to the next. It also requires an initial editable data.

```
:: Scenario        :== [Event]

eventloop :: (EDET a b) (EditableData, a) Scenario → EditableData
eventloop f (data, a) [event:events]
    # (data, _, _)      = f event (data, a, Pending)
    = eventloop f (data, a) events
eventloop f (data, a) []
    = data
```

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 Figure 1 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). In lines 4 and 5 we model a scenario of six events. In event 1, the user enters the number 2 to create two more input fields. In events 2 and 3, she enters the values 30 and 12 in the first and second created input field. In event 4, she reduces the number of input fields to 1, thus removing the second input field from appearing (but not its state!). In event 5, she adds two more input fields. Note that this re-creates the second input field with value 12. Entering value 58 in the third input field, event 6, results in displaying the sum 100 (which was shown in the lower screenshot of Figure 1).

4.2 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, we define a layer of derived arrow operations 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 Section 4.3, we use them to construct example programs more easily.

We define derived operations for branching, choice, and mapping. First, however, a number of useful abbreviations are introduced:

```
dupl        :== λx → (x,x)                                              1.
set a       :== λx → a                                                  2.
add1 a      :== λb → (a,b)                                              3.
add2 b      :== λa → (a,b)                                              4.
arr2 f      :== arr (λ(a,b) → f a b)                                    5.
(@) f g     :== arr g ⋙ f                                               6.
skip        :== arr id                                                  7.
```

The function `dupl` (line 1) duplicates an arrow state, `set`, `add1` and `add2` (lines 2–4) introduce a constant into the arrow state, and `arr2`, `@`, and `skip` are special applications of `arr`. The infix operation `@` is useful for providing ids to `editread` and `editset` by means of `(editread @ set id)` and `(editset @ add1 id)`, which in Section 3.3 were even abbreviated further to *editread* $\boxdot$ *id* and *editset* $\odot$ *id*.

Arrows often require separate computations to be carried out independently, after which the results are combined again. Using the standard `first` and its dual `second`, this desired behavior can be achieved as follows:

```
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 function `branch` (lines 4 and 5) duplicates its input value, which in fact creates two separate branches, and then executes its first argument on the first branch and its second argument on the second branch. Combining the values afterward must be performed separately.

For programming purposes, it is also important that an arrow operation is available that chooses between computations based on the contents of the arrow state. This standard arrow extension can be defined in terms of `left` and its dual `right`:

```
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. The standard `Either` type handles the domain and range values.

```
mapAB :: (EDET a b) → EDET [a] [b]                                    1.
mapAB f =  arr       (λas → (length as, map Left as))                 2.
       >>> iterateN (   arr    (λ(_, [Left a:las]) → (a, las))        3.
                    >>> first f                                       4.
```

```
                     >>> arr    (λ(b,las) → las ++ [Right b])
                     )
        >>> arr      (map (λ(Right b) → b))
```

Using the *Arrow* laws, one can prove the following properties of `mapAB`:

$$\begin{aligned}
arr\ (\lambda x \rightarrow [\,])\ &\ggg mapAB\ f \overset{(mapAB[])}{=} arr\ (\lambda x \rightarrow [\,]) \\
arr\ (\lambda x \rightarrow [c : cs]) \ggg mapAB\ f &\overset{(mapAB[:])}{=}\ f @ (\lambda x \rightarrow c) \\
&\qquad \ggg first\ (mapAB\ f) @ (\lambda b \rightarrow (cs, b)) \\
&\qquad \ggg arr\ (\lambda(bs, b) \rightarrow [b : bs])
\end{aligned}$$

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.

4.3 Some small editor arrows programs

The example of Figure 1 created an editor arrow that repeatedly computes the sum of the values of a varying number of editable integer values. Using the derived operations of *EditorArrowCore*, this editor arrow can now be expressed as follows:

```
variable_sum_arrow :: EDET Int Int
variable_sum_arrow
    = editread @ (set nrId)
    >>> iterateN (first (editread @ argId) >>> arr2 (+)) @ (add2 0)
    >>> editset  @ (add1 sumId)
```

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 use the scenario that was presented at the very end of Section 4.1, 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*:

```
[]
→ Event(nr, 2)
[nr = 2; sum = 0]
→ Event(arg 1, 30)
[nr = 2; arg 1 = 30; sum = 30]
→ Event(arg 2, 12)
[nr = 2; arg 1 = 30; arg 2 = 12; sum = 42]
→ Event(nr, 1)
[nr = 1; arg 1 = 30; arg 2 = 12; sum = 30]
→ Event(nr, 3)
[nr = 3; arg 1 = 30; arg 2 = 12; sum = 42]
→ Event(arg 3, 58)
[nr = 3; arg 1 = 30; arg 2 = 12; arg 3 = 58; sum = 100]
```

The incoming events are shown on the even-numbered lines. The editable data, which contain the current values of the editors, are shown on the odd-numbered lines. 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 line 1 and 13 correspond with the screenshots in Figure 1.

Another interesting example is a convertor between euros 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
convert_arrow
    = editread @ (set euroId)
   >>> arr      toDollar
   >>> editset  @ (add1 dollarId)
   >>> arr      toEuro
   >>> editset  @ (add1 euroId)
    where
        toDollar euro   = euro * 1.592
        toEuro dollar   = dollar / 1.592
```

The following editor arrow changes indicated values in a list. It consists of two editors, one to input the index of the element and another 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
list_editor
    = branch (editread @ set indexId) skip
   >>> arr    (λ(i,list) → (list!!i, (i,list)))
   >>> first  (editset @ (add1 fieldId))
   >>> arr    (λ(n,(i,list)) → updateAt i n list)
```

The final example is inspired by the “bounded counter” case study that has been described by Courtney (2004) in which a GUI contains a counter element that counts the number of occurrences of some external event. At some given maximum value, counting should no longer increment the counter value. We adapt the example to count button presses. In order to do that, we first model a button GUI element as an editor of `Boolean` values: pressing the button is the same as “editing” its value to `True`, and when it is not pressed, its value is `False`. After pressing the button, its state should be reset to `False`. We obtain the following implementation:

```
button :: EditorId → Arrow a Bool                          1.
button buttonId                                            2.
    = editread       @ (set    buttonId)                   3.
   >>> first editset @ (add1 (buttonId,False))             4.
   >>> arr snd                                             5.
```

First, the editing state of the button is retrieved (line 3), thus handling a button press event if present. Hence, if the button is pressed, the value `True` is returned, and `False` otherwise. In any case, the button state is reset to `False` (line 4) to enable next button presses. Finally, the retrieved button value is returned (line 5).

With this element, we create the bounded counter that keeps track of the number of button presses of a button identified by `buttonId` in a storage identified by `counterId` (this only makes sense if `buttonId` identifies a different storage than `counterId`) up to a given limit `n`.

```
bounded_counter :: EditorId EditorId Int → Arrow a Int          1.
bounded_counter buttonId counterId n                            2.
    = button buttonId                                           3.
    ⋙ ifthenelse id                                             4.
            (self ((min n) o ((+) 1)) counterId)                5.
            (editread @ (set counterId))                        6.
```

The bounded counter first checks whether the button has been pressed. In that case, the value of the counter store is incremented, but up to the given maximum value (line 5). This is an example of a self-correcting editor, and is therefore expressed with the `self` combinator. If the button was not pressed, then the current value of the counter is returned (line 6).

To illustrate this example, suppose we have a GUI that contains (`bounded_counter button counter` 3). Then the following scenario illustrates that the counter is incremented after the first three button presses (at lines 2–3, 8–9, and 12–13) but after that remains bounded by the value 3 (lines 14–15). (Note that `other` refers to some other editor identification that is unequal to either `button` or `counter`.)

```
 []                                         1.
→ Event(button, True)                       2.
 [button = False; counter = 1]              3.
→ Event(other, 30)                          4.
 [button = False; counter = 1]              5.
→ Event(other, 12)                          6.
 [button = False; counter = 1]              7.
→ Event(button, True)                       8.
 [button = False; counter = 2]              9.
→ Event(other, 42)                          10.
 [button = False; counter = 2]              11.
→ Event(button, True)                       12.
 [button = False; counter = 3]              13.
→ Event(button, True)                       14.
 [button = False; counter = 3]              15.
```

4.4 Arrow laws for Sparkle

Implementing the *EditorArrow* model in *Clean* as *EditorArrowCore* allows us to use the integrated proof assistant *Sparkle* (Mol *et al.*, 2002, 2008; Kesteren *et al.*, 2004) and its proof tool support for strictness (Eekelen & Mol, 2005) to verify the correctness of the laws of Section 3 when translated to *EditorArrowCore*.

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 Section 3.

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 Section 4.1. 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 Section 3.4.

The laws of Section 3 can now be transferred to *Sparkle* directly. We demonstrate this for the following three laws:

Law '⋙ 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': $\mathit{first}\ (\mathit{first}\ f) \ggg \mathit{assoc} = \mathit{arr}\ \mathit{assoc} \ggg \mathit{first}\ f$

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.

All the laws in this paper (see Definitions 1–4 and 7) have been transferred to *Sparkle* and have been proved correct with it. The proofs were relatively straightforward to construct, but the amount of work was considerable due to the many different cases that had to be examined explicitly. In the end, it took an experienced *Sparkle* user about 30 hours to build the proofs. The proofs have a size on disk of about 156KB and consist of approximately 3500 lines of proof code.

By successfully building proofs for all laws, we have shown that our framework is indeed correct: the classic arrow laws still hold in our extended setting, and the additional operators (in particular *editread* and *editset*) behave as intended. Note that as expected, many conditions had to be added to the formalized laws in *Sparkle* because of the different domains (for instance, `Int` is used to represent $\mathbb{N}$, leading to $n \neq \perp$ and $n \geqslant 0$) and the propagation of definedness (for instance, $f \succ (g \succ h) = (f \succ g) \succ h$ requires h to maintain definedness).

The use of *Sparkle* did bring the following two inaccuracies to light:

- In *EditorArrowCore*, lists are used to represent editable data, which are really sets. Permutations of such a list represent the same set, and will therefore result in the same behavior of editor arrows. Consequently, if editor arrows produce editable data that are permutations of each other (and an identical value and event status), then they should be considered equal as well.
  This extended view of equality is required to prove the validity of the editor swapping laws (see Definition 4). In case no entry exists in the initial editable data for the used editors, namely, permutations of the same editable data will be produced. If this situation is disallowed, the normal equality is proved easily. Alternatively, the laws can be proved as well by formally incorporating the extended view of equality.
  Note that this issue does not occur on the level of *EditorArrow*, because it uses real sets to represent editable data.
- Initially, the feedback law (see Definition 4) did not have the condition that i and j should hold the same value initially. The need for this additional condition was discovered in the process of building the proof with *Sparkle*. Note that the underlying idea of the feedback law is to *maintain* equality of editors during the process of handling an event; therefore, the condition does not invalidate the use of the law.

5 Reasoning with editor arrows

In this section, we give three examples of reasoning with the resulting *EditorArrowCore* framework. In Section 5.1, we generalize the *read-read elimination* law, by phrasing it as an iterated version. In Section 5.2, we use the *Arrow* and *EditorArrow* laws to prove the equivalence of the example variable sum list with a variant that uses the `mapAB` combinator. In Section 5.3, we use the operational semantics to show that the *EditorArrow* version of the bounded counter-example never exceeds the given upper bound.

5.1 Generalizing read-read elimination

The *read-read elimination* law states that it is pointless to read the same editor twice. We use the `iterateN` combinator to generalize this to arbitrary many times for an editor that is identified by some defined value i:

$$\forall m > 0 : \texttt{iterateN (editread @ (set } i\texttt{)) @ (add1 } m\texttt{)} = \texttt{editread @ (set } i\texttt{)}$$

The proof proceeds by induction on m. We start with the base case, $m = 1$, and apply the *iterateN-next* law:

$$\begin{aligned}
& \texttt{iterateN (editread @ (set } i\texttt{)) @ (add1 } m\texttt{)} \\
= \; & \texttt{iterateN (editread @ (set } i\texttt{)) @ (add1 1)} \\
= \; & \texttt{(editread @ (set } i\texttt{)) @ (add1 1)} \ggg \texttt{iterateN (editread @ (set } i\texttt{)) @ (add1 0)}
\end{aligned}$$

We simplify the first part of the expression by unfolding `@`, (`add1 1`), and (`set`i).

$$\begin{aligned}
= \; & \texttt{arr } (\lambda \texttt{x} \to (1,\texttt{x})) \ggg \texttt{arr } (\lambda \texttt{x} \to i) \ggg \texttt{editread} \\
& \qquad\qquad\qquad\qquad \ggg \texttt{iterateN (editread @ (set } i\texttt{)) @ (add1 0)}
\end{aligned}$$

which can be folded back again to the `editread` combinator after combining the two function arguments of the first two occurrences of *arr* :

$$= \texttt{editread @ (set } i\texttt{)} \ggg \texttt{iterateN (editread @ (set } i\texttt{)) @ (add1 0)}$$

The second half of this expression can be simplified using the *iterateN-base* law:

$$= \texttt{editread @ (set } i\texttt{)} \ggg \texttt{arr snd @ (add1 0)}$$

which, after unfolding `@` and (`add1 0`) amounts to `arr id`, which is the *Right unit* of any arrow expression.

$$= \texttt{editread @ (set } i\texttt{)}$$

This completes the proof of the base case. The induction step also starts with the *iterateN-next* law:

$$\begin{aligned}
& \texttt{iterateN (editread @ (set } i\texttt{)) @ (add1 (}m\texttt{+1))} \\
= \; & \texttt{(editread (set } i\texttt{)) @ (add1 (}m\texttt{+1))} \ggg \texttt{iterateN (editread @ (set } i\texttt{)) @ (add1 } m\texttt{)}
\end{aligned}$$

The induction hypothesis can be applied immediately:

$$= \texttt{(editread @ (set } i\texttt{)) @ (add1 (}m\texttt{+1))} \ggg \texttt{editread @ (set } i\texttt{)}$$

We bring (`add1 (`m`+1)`) to the front of the expression by unfolding `@`:

$$= \texttt{add1 (}m\texttt{+1)} \ggg \texttt{editread @ (set } i\texttt{)} \ggg \texttt{editread @ (set } i\texttt{)}$$

and see that we can apply the *read-read elimination* law:

```
= add1 (m+1) >>> editread @ (set i)
```

The last part is to unfold `add1 (m+1)`, `@`, and `(set (i))`:

```
= arr (λx → (m+1,x)) >>> arr (λx → i) >>> editread
```

which is recombined to the correct end-result:

```
= editread @ (set i)
```

This completes the proof.

Similarly, the *set-set elimination* law can be generalized for any editor identified by i:

```
∀m > 0 : iterateN (editset @ (set1 i)) @ (add1 m) = editset @ (add1 i)
           where
               set1 a :== λ(b,x) → (a,x)
```

Although this property is formulated in a slightly asymmetric way due to the use of `(set1 i)`, the proof can be carried out in the same way as the proof above.

5.2 Variable sum list

In Section 4.3 we have shown how to define the variable sum list example with the derived combinators. Alternatively, the same program can also be expressed with the `mapAB` combinator that was introduced in Section 4.2:

```
variable_sum_arrow2 :: EDET Int Int
variable_sum_arrow2
    =  editread @ (set nrId)
   >>> arr       (λn → map argId (reverse [1..n]))
   >>> mapAB     editread
   >>> arr       sum
   >>> editset   @ (add1 sumId)
```

The key difference is that this version first reads all values of the integer editors and collects them in a list, and afterward applies the standard sum function on this list. In order to prove the equivalence of the two programs, it suffices to prove:

```
∀m ⩾ 0 :
       arr       (λx → m)
   >>> iterateN (first (editread @ argId) >>> arr2 (+)) @ (add2 0)
    =
       arr       (λx → m)
   >>> arr       (λn → map argId (reverse [1..n]))
   >>> mapAB     editread
   >>> arr       sum
```

Hence, we abstract from the very first input editor and the very last output editor. We proceed by induction on m. The base case ($m = 0$) starts by rewriting the left-hand side:

```
    arr (λx → 0)
>>> iterateN (first (editread @ argId) >>> arr2 (+)) @ (add2 0)
```

to (using `arr (λx → 0) >>> arr (add2 0) = arr (λx → (0,0)) = arr (λx → 0) >>> arr (λx → (0,x))`:

```
=   arr (λx → 0)
>>> arr (λx → (0,x))
>>> iterateN (first (editread @ argId) >>> arr2 (+))
```

which allows us to apply the *iterateN-base* law:

```
=   arr (λx → 0)
>>> arr (λx → (0,x))
>>> arr snd
```

which simplifies to:

```
=   arr (λx → 0)
```

This expression can be transformed to sum the elements of an empty list:

```
=   arr (λx → [])
>>> arr sum
```

which, by the *mapAB[ ]* law, is equivalent to:

```
=   arr (λx → [])
>>> mapAB editread
>>> arr sum
```

and, by splitting the first function, also to:

```
=   arr (λx → 0)
>>> arr (λn → map argId (reverse [1..n]))
>>> mapAB editread
>>> arr sum
```

which is the right-hand side of the expression in case of $m = 0$.

The induction step starts in a similar way, but now prepares the left-hand side in order to apply the *iterateN-next* law:

```
    arr (λx → 0)
>>> arr (λx → (1+m,x))
>>> iterateN (first (editread @ argId) >>> arr2 (+))
```

which is rewritten as:

```
=   arr   (λx → 0)
>>> arr   (λx → (1+m,x))
>>> first (editread @ argId)
>>> arr2  (+)
>>> arr   (λx → (m,x))
>>> iterateN (first (editread @ argId) >>> arr2 (+))
```

We make sure that `editread @ argId` is applied to `1+m` and swap it with the insertion of the zero constant:

```
=   editread @ (set (argId (1+m)))
>>> arr   (λx → (x,0))
>>> arr2  (+)
>>> arr   (λx → (m,x))
>>> iterateN (first (editread @ argId) >>> arr2 (+))
```

This simplifies to:

```
=  editread @ (set (argId (1+m)))
>>> arr (λx → (m,x))
>>> iterateN (first (editread @ argId) >>> arr2 (+))
```

Because of their occurrence within `first`, the `editread` actions only depend on the counter value of the `iterateN` combinator. This implies that it makes no difference whether they read and add editor values starting with the result of reading editor `argId (1+m)`, or whether they read and add the same order of editor values starting with zero, and add the result of reading editor `argId (1+m)` afterward. This is expressed by the *Sum-iterate-distribution* lemma:

```
∀m ⩾ 0 :
        arr    (λx → (m,x))
    >>> iterateN (first f >>> arr2 (+))
     =
        arr    (λx → ((m,0),x))
    >>> first (iterateN (first f >>> arr2 (+)))
    >>> arr2   (+)
```

If we apply this lemma to the above expression, we obtain:

```
=  editread @ (set (argId (1+m)))
>>> arr    (λx → ((m,0),x))
>>> first (iterateN (first (editread @ argId) >>> arr2 (+)))
>>> arr2   (+)
```

In order to prepare the argument of `first` for the induction hypothesis, we need to push in the `λx → (m,0)` component of the `arr` function:

```
=  editread @ (set (argId (1+m)))
>>> arr    (λx → (x,x))
>>> first (    arr (λx → m)
           >>> arr (add2 0)
           >>> iterateN (first (editread @ argId) >>> arr2 (+))
          )
>>> arr2   (+)
```

After moving `arr (add2 0)` to the right, using the definition of `@`, we can apply the induction hypothesis which yields:

```
=  editread @ (set (argId (1+m)))
>>> arr    (λx → (x,x))
>>> first (    arr (λx → m)
           >>> arr (λn → map argId (reverse [1..n]))
           >>> mapAB editread
           >>> arr sum
          )
>>> arr2   (+)
```

The next step is to move the program fragments before and after `mapAB editread` out of the `first` combinator. For this purpose we "break" it into three parts:

```
=  editread @ (set (argId (1+m)))
>>> arr    (λx → (x,x))
>>> first (arr (λx → m) >>> arr (λn → map argId (reverse [1..n])))
>>> first (mapAB editread)
>>> first (arr sum)
>>> arr2  (+)
```

We merge the three functions before `first (mapAB editread)` and recombine the `arr sum` component with `arr2 (+)`:

```
=   editread @ (set (argId (1+m)))
>>> arr    (λx → (map argId (reverse [1..m]),x))
>>> first (mapAB editread)
>>> arr    (λ(bs,b) → [b:bs])
>>> arr    sum
```

This allows us to apply the *mapAB[:]* law (after moving the first `arr` component behind `first` using the definition of `@`):

```
=   arr    (λx → [argId (1+m) : map argId (reverse [1..m])])
>>> mapAB editread
>>> arr    sum
```

Finally, we move the first element of this list to the back of the reversed list, and split the function in order to obtain the right-hand side of the expression:

```
    arr    (λx → 1+m)
>>> arr    (λn → map argId (reverse [1..n]))
>>> mapAB editread
>>> arr    sum
```

This concludes the proof of equivalence between the two programs.

5.3 Bounded counter

In this section, we show that `bounded_counter` (Section 4.3) indeed has the property that it never increases the value of its counter store beyond a given bound value `N`. In order to prove that this example implements a bounded counter, we show that it maintains the invariant that `bounded_counter BID CID N` (for any `BID` $\neq$ `CID`) keeps the value of the counter store below a given bound `N`. This should hold for any event that does not touch the counter store (because it is supposed to be local to the bounded counter). In addition, it should hold for any event status (*Pending* or *Processed*) in order to abstract from the position of `bounded_counter BID CID N` (including multiple occurrences of itself) within an arrow program. This amounts to proving the following property:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall\ \mathtt{BID}, \mathtt{CID}, \mathtt{N}, (eid, e), (s_0, a_0, p_0)\ :} \\
\quad s_0 & \supseteq \{(\mathtt{BID}, \mathtt{serialize\ False}), (\mathtt{CID}, c_0)\} & \wedge \\
\quad eid & \neq \mathtt{CID} & \wedge \\
\quad \mathtt{N} & \geqslant \mathtt{deserialize}\ c_0 & \\
\Rightarrow (s_1, a_1, p_1) & = \mathtt{bounded\_counter\ BID\ CID\ N}\ (eid, e)\ (s_0, a_0, p_0) & \wedge \\
\quad s_1 & \supseteq \{(\mathtt{BID}, \mathtt{serialize\ False}), (\mathtt{CID}, c_1)\} & \wedge \\
\quad a_1 & = \mathtt{deserialize}\ c_1 & \wedge \\
\quad \mathtt{N} & \geqslant \mathtt{deserialize}\ c_1 &
\end{array}
$$

The proof proceeds by rewriting using the *Arrow* laws and combinator definitions. Below we give a comprised overview of the proof. The initial expression:

```
bounded_counter BID CID N (eid,e) (s0,a0,p0)
```

is rewritten as:

```
ifthenelse id (self ((min N) o ((+) 1)) CID)
              (arr (set CID) >>> editread)
                            (eid,e)
(arr   snd                  (eid,e)
(first editset              (eid,e)
(arr   (add1 (BID,False))   (eid,e)
(editread                   (eid,e)
                            (s0,BID,p0)
))))
```

We distinguish two cases for `p0`.

- p_0 = `Processed`: the assumption $s_0 \supseteq \{(\texttt{BID}, \texttt{serialize False}), \ldots\}$ causes rewriting of `editread` to result in:

  ```
  ifthenelse id (self ((min N) o ((+) 1)) CID)
                (arr (set CID) >>> editread)
                              (eid,e)
  (arr   snd                  (eid,e)
  (first editset              (eid,e)
  (arr   (add1 (BID,False))   (eid,e)
                              (s0,False,Processed)
  ))))
  ```

  The application of `editset` stores the value `False` in the `BID` store. Subsequently, the `False` value causes selection of the second branch of `ifthenelse`:

  ```
  arr (set CID) >>> editread (eid,e)
   (write BID (serialize False) s0,Right False,Processed)
  ```

  which, in turn, is rewritten as:

  ```
  editread (eid,e) (write BID (serialize False) s0,CID,Processed)
  ```

  Here, `editread` is set to read the `CID` store. Because $s_0 \supseteq \{\ldots, (\texttt{CID}, c_0)\}$, `editread` yields the final result:

  ```
  ( write BID (serialize False) s0, deserialize c0, Processed )
  ```

  The `BID` store contains `False`, the `CID` store has not changed and hence its value is still bound by `N`.

- p_0 = `Pending`: we distinguish two cases for `eid`:

  — *eid* $\neq$ `BID`: this case proceeds analogously to the above case because *eid* $\neq$ `BID` causes the same branches to be chosen for `editset` and `ifthenelse`. Again, the `CID` store remains unaltered, and the final result is:

  ```
  ( write BID (serialize False) s0, deserialize c0, Pending )
  ```

— *eid* = `BID`: In this case we have a pending event for the button GUI:

```
ifthenelse id (self ((min N) o ((+) 1)) CID)
              (arr (set CID) >>> editread)
                            (BID,e)
(arr   snd                  (BID,e)
(first editset              (BID,e)
(arr   (add1 (BID,False))  (BID,e)
(editread                   (BID,e)
                            (s0,BID,Pending)
))))
```

which causes `editread` to store value `e` in the `BID` store, and emit its deserialized value:

```
ifthenelse id (self ((min N) o ((+) 1)) CID)
              (arr (set CID) >>> editread)
                            (BID,e)
(arr   snd                  (BID,e)
(first editset              (BID,e)
(arr   (add1 (BID,False))  (BID,e)
                            (write BID e s0,deserialize e,Processed)
)))
```

The button press event is immediately "neutralized" by `editset` which resets the `BID` store value to `False`:

```
ifthenelse id (self ((min N) o ((+) 1)) CID)
              (arr (set CID) >>> editread)
              (BID,e)
              (write BID (serialize False) s0,deserialize e, Processed)
```

We need to distinguish between two further cases:

- *e* = `False`: actually, this case cannot occur in an implementation because you cannot unpress an unpressed button. Because *e* = `False`, `ifthenelse` chooses the second branch, and we end with the final value:

  ```
  ( write BID (serialize False) s0, deserialize c0, Processed )
  ```

  which is identical to the very first case.
- *e* = `True`: rewriting of `ifthenelse` now selects the first branch:

  ```
  self ((min N) o ((+) 1)) CID
       (BID,True)
       (write BID (serialize False) s0, deserialize True, Processed)
  ```

  Expanding the `self` definition, and the subsequent occurrences of `@` and >>> gives:

  ```
  editset         (BID,True)
  (arr (add1 CID) (BID,True)
  ```

```
(arr ((min N) o ((+) 1))
                (BID,True)
(editread       (BID,True)
(arr (set CID)  (BID,True)
                ( write BID (serialize False) s0
                , deserialize True
                , Processed
                )
))))
```

First, the value of the CID store is obtained:

```
editset         (BID,True)
(arr (add1 CID) (BID,True)
(arr ((min N) o ((+) 1))
                (BID,True)
                ( write BID (serialize False) s0
                , deserialize c0
                , Processed
                )
))
```

This value is incremented with upper limit N:

```
editset         (BID,True)
(arr (add1 CID) (BID,True)
                ( write BID (serialize False) s0
                , min N (1 + deserialize c0)
                , Processed
                )
)
```

and stored in the CID store by editset. This gives the final result:

```
( write CID (serialize (min N (1 + deserialize c0)))
 (write BID (serialize False) s0)
, min N (1 + deserialize c0)
, Processed
)
```

Again, the BID store is reset to False. The emitted value is equal to the value that is placed in the CID store. Moreover, its value is an increment of the original value, except when the upper bound N has been reached.

This concludes the proof.

6 Related work

We have presented the semantic *EditorArrow* model that uses value-editors as elementary interactive components, and *EditorArrow* combinators to glue these elements. The inspiration for this work was based on the high-level *GEC* and *iData* toolkits. We discuss two other *Arrow* based approaches in detail: *Fudgets* and *Fruit/Yampa*.

Fudgets (Carlsson & Hallgren, 1993, 1998) is a seminal GUI toolkit. The base element in *Fudgets* is a *fudget*, which is a typed unit (of type `F a b`) that processes two input streams and produces two output streams. One input stream contains low-level GUI events (such as mouse movements, screen update requests, and keyboard presses). These are transformed to low-level GUI commands (such as rendering commands, widget movement, and size control) but also high-level program values of type `b` (such as letting the remainder of the program know that a button has been pressed, and that text has been entered). High-level program values of predecessor fudgets are received as values of type `a` on the high-level input stream. In order to integrate computation with GUI handling, *abstract fudgets* are introduced. An abstract fudget neither consumes low-level GUI events nor produces low-level GUI commands; hence, it processes only high-level program values. The original *Fudgets* toolkit comes with a comprehensive set of combinator functions. This library is in fact an extended *Arrow* library (Hughes, 2000). By construction, communication between fudgets follows the combinator structure. In order to allow "later" fudgets to manipulate "earlier" fudgets, feedback is created by means of `loop`-like combinators.

We believe that our proposed framework is sufficiently expressive to model *Fudgets*. When a fudget of type (`F a b`) handles low-level GUI events and decides to emit a high-level value, then we consider this as an editing event of a new value `v` of type `b`. This is the same as sending it the event `(id,v)` where `id` identifies the fudget. (In Carlsson & Hallgren, 1998,, Section 21.1, events are tagged with the path in the fudget expression where the fudget can be found. Hence, these paths serve well as identifiers.) Using high-level values to control fudget behavior is the same as setting a new value in an editor using `editset`. The editor adapts its rendering accordingly. Feedback using a loop is implemented in *Fudgets* by maintaining a local queue (Carlsson & Hallgren, 1998, Section 20.4.2) that needs to be consumed entirely before new "external" high-level values are consumed. As observed in Carlsson & Hallgren (1993, Section 3.2.4), this may lead to non-termination: if `textF :: F String String` does not discern "edited" values from "received" values, then (`loopAll (textF >==< textF)`) loops forever. Because our model has been set up to prevent non-termination as much as possible, loops cannot be modeled in a direct manner. However, our model supports identified state; hence, we can also maintain a local queue for a feedback construct that is written to when producing values. An editor within a loop construct must first read the local queue if there are still pending values. If that is the case, then the new value is appended to the queue, and the oldest pending value is removed and passed on to the editor. If the local queue is empty, then the editor uses the new value. In this way, the system becomes similarly non-productive when editors continue filling the local queue even though at each event the system terminates.

Fruit/Yampa (Courtney, 2004) is an exponent of the school of *reactive programming*. In reactive programming, interactive programs are constructed by composing and transforming typed units that represent time-varying values. For an in-depth discussion of the various approaches within this school, we refer to Courtney (2004, Chapter 3). *Yampa* is an *Arrow* based language embedded in *Haskell* for describing reactive systems. *Fruit* is the GUI toolkit that is founded on *Yampa*. *Fruit* and *Yampa* have been developed to enable formal reasoning about GUI programs. Although we also wish to reason formally about GUI programs, and use *Arrows* to structure our programs, the model used by *Yampa* and *Fruit* differs materially from our model. In *Yampa*, programs are constructed as combinations of signals

```
:: Signal a :== Time → a
```

and signal functions

```
:: SF a b :== (Signal a) → Signal b
```

A *Fruit* program is a signal function that transforms GUI-event signals (of type `GUIInput`) to rendering signals (of type `Picture`), and hence, a program has type

```
:: GUI a b :== SF (GUIInput,a) (Picture,b)
```

When constructing a GUI program in *Fruit*, a programmer keeps composing and transforming signal (functions) to reach the desired behavior. This is clearly analogous to the functional programming style. The evaluation of such a program is "within" the signal function: elements consume and produce events and values. This also motivates the need for a `loop` construct.

The above features make programming and reasoning about *Fruit* programs radically different from our model. Here, the evaluation of a program is only required to compute the answer to one single event. This is much more analogous to programming traditional GUI systems (see Section 1). We can afford to restrict ourselves to primitive recursion because there are no events that need to be consumed during the program. Primitive recursion is required to guarantee that the program yields a GUI. Nevertheless, we have shown that our system can be structured as an *Arrow*, that it satisfies the *Arrow* laws, and that equational reasoning can be applied.

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 last 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.

7 Discussion and conclusions

We have introduced the formal *EditorArrow* semantic model for the *GEC* and the *iData* toolkits. This model is based on the *Arrow* framework and extends it with the primitive combinator functions *editread* and *editset* for creating editors with shared state. Furthermore, to guarantee the defined termination of editors, we have replaced arbitrary loops with controlled *iteration*.

The unified, *Arrow* based framework provides an *API* that is highly platform independent: the very same *Arrow* expression can be compiled and executed as a widget-based application as well as a thin-client web application. The sharp distinction between desktop and web applications is blurring. The framework that we have presented here can be a first step toward formalization of such applications.

In Section 6 we have argued informally that the seminal *Fudgets* system can be modeled within *EditorArrow*. It is interesting to answer the question to what extent the formal *EditorArrow* framework can be used to structure or model GUI toolkits other than *GEC* and *iData*. The application domain is typically dialog based: the core interface component is that of a dialog (or form) and manipulating that dialog may create new dialogs. These dialogs have resident state, and that state can both be read and altered. We think that for these application domains, the framework can be used as well.

The long-term goal that initiated this project is to reason formally about real-world GUI programs. In the past we have performed case studies: a conference management system (Plasmeijer & Achten, 2006a) and a project adminstration system (Plasmeijer & Achten, 2006b) in *iData*. Neither is written in *Arrow* style. In case of the conference management system, we conjecture that it can be captured with the *Arrow* combinators, as the program uses chiefly sequencing and choice. There is only one place that uses recursion (`guestHomePage`), but there it is only used to implement the three stages of a user first creating an account, then provide the data for that account, and finally submit a paper with that account. The project administration system has a sequential structure in which feedback between the editors takes care that after each user event, the administration is consistently updated. So we expect that this example can also be transformed to an *Arrow* structure.

We have proved the classis *Arrow* laws for our framework, as well as a number of additional laws for iteration and 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. If *Arrows* are used for domains that contain undefined values, then the definition of the *Arrow* combinators needs to handle undefined values as well. This experiment confirms the general insight that reasoning about *Arrows* is relatively easy.

We have formally proved all laws with *Sparkle*. Working with this proof assistant helped us to identify issues that escaped our attention in the process of specifying the model and its expected properties. One consequence is that proofs need to be reconstructed when the model has been adapted to repair these issues. Unfortunately, *Sparkle* has no means of refactoring (partial) proofs; hence, this was a time-consuming effort. Nevertheless, having such a tool available that allows reasoning about source code is an invaluable asset because it greatly increases confidence in the correctness of the proofs.

Acknowledgments

The authors thank the reviewers for their valuable feedback.

References

Achten, P., Eekelen, M. van & Plasmeijer, R. (2003) Generic Graphical User Interfaces. In *Selected Papers of the 15th International Workshop on the Implementation of Functional Languages, IFL03, Edinburgh, UK*. Michaelson, G. & Trinder, P. (eds), LNCS, vol. 3145. Springer Verlag.

Achten, P., Eekelen, M. van & Plasmeijer, R. (2004a) Compositional model-views with generic Graphical User Interfaces. In *Practical Aspects of Declarative Programming, PADL04*. LNCS, vol. 3057. Springer Verlag, pp. 39–55.

Achten, P., Eekelen, M. van, Plasmeijer, R. & van Weelden, A. (2004b) Automatic Generation of editors for higher-order data structures. In *Second Asian Symposium on Programming Languages and Systems (APLAS 2004),* Wei-Ngan C. (ed), LNCS, vol. 3302. Springer Verlag, pp. 262–279.

Achten, P., Eekelen, M. van, Plasmeijer, R. & van Weelden, A. (2004c) GEC: A toolkit for Generic Rapid Prototyping of type safe interactive applications. In *Proceedings of the 5th International Summer School on Advanced Functional Programming (AFP 2004)*. LNCS, vol. 3622. Springer Verlag, pp. 210–244.

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

Carlsson, M. & Hallgren, T. (1993) Fudgets – a graphical user interface in a lazy functional language. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture (FPCA '93)*.

Carlsson, M. & Hallgren, T. (1998) *Fudgets – Purely Functional Processes with Applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, Göteborg University, Sweden. ISBN 91-7197-611-6; ISSN 0346-718X.

Cartwright, R. & Donahue, J. (1982) The semantics of lazy (and industrious) evaluation. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming (LFP '82)*. New York: ACM, pp. 253–264.

Courtney, A. A. (2004 May) *Modeling User Interfaces in a Functional Language*. PhD thesis, Yale University.

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

Courtney, A., Nilsson, H. & Peterson, J. (2003) The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*. Uppsala, Sweden: ACM Press, pp. 7–18.

Dowse, M., Butterfield, A. & Eekelen, M. van. (2004) Reasoning about deterministic concurrent functional i/o. In *Ifl*, Grelck, C., Huch, F., Michaelson, G. & Trinder, Philip W. (eds), Lecture Notes in Computer Science, vol. 3474. Springer, pp. 177–194.

Eekelen, M. van & Mol, Ma. de. (2005) Proof tool support for explicit strictness. In *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Dublin, Ireland, September 19–21, 2005, Revised Selected Papers*, Butterfield, A., Grelck, C. & Huch, F. (eds), LNCS, vol. 4015. Springer Verlag, pp. 37–54.

Eekelen, M. van, Smetsers, S. & Plasmeijer, R. (1997) Graph rewriting semantics for functional programming languages. *Computer Science Logic*, 106–128.

Elliott, C. & Hudak, P. (1997, June). Functional reactive animation. In *International Conference on Functional Programming*, pp. 163–173.

Groningen, J. van, Noort, T. van, Achten, P., Koopman, P. & Plasmeijer, R. (2010) Exchanging sources between Clean and Haskell – A double-edged front end for the Clean compiler. *Proceedings of the Haskell Symposium, Haskell'10, Baltimore, MD, USA*, Gibbons, J. (ed), ACM Press, pp. 49–60.

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

Hanus, M. (2006) Type-oriented construction of Web User Interfaces. In *Proceedings of the 8th International ACM Sigplan Conference on Principle and Practice of Declarative Programming (PPDP'06)*. ACM Press, pp. 27–38.

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

Hudak, P., Courtney, A., Nilsson, H. & Peterson, J. (2003) Arrows, robots, and functional reactive programming. In *Advanced Functional Programming, 4th International School, Oxford*. Jeuring, J. & Peyton Jones, S. (eds), LNCS, vol. 2638. Springer Verlag, pp. 159–187.

Hudak, P., Hughes, J., Peyton Jones, S. & Wadler, P. (2007) A history of haskell: Being lazy with class. In *Proceedings of the Third ACM Sigplan Conference on History of Programming Languages*. HOPL III. New York: ACM, pp. 12-1–12-55.

Hughes, J. (2000) Generalising monads to arrows. *Sci. Comput. Program.* **37**(May), 67–111.

Kesteren, R. van, Eekelen, M. van & Mol, M. de. (2004) Proof support for general type classes. In *Trends in Functional Programming 5: Selected Papers from the 5th International Symposium on Trends in Functional Programming (TFP04)*. Loidl, H.-W. (ed), Intellect, pp. 1–16.

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

Mol, M. de. (2009 March 4) *Reasoning about Functional Programs – Sparkle: A Proof Assistant for Clean*. PhD thesis, University of Nijmegen, The Netherlands. ISBN 978-90-9023885-2.

Mol, M. de, Eekelen, M. van & Plasmeijer, R. (2002) Theorem proving for functional programmers – Sparkle: A functional theorem prover. In *Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL 2001, Stockholm, Sweden, Selected Papers*. Arts, T. & Mohnen, M. (eds), LNCS, vol. 2312. Springer Verlag, pp. 55–72.

Mol, M. de, Eekelen, M. van & Plasmeijer, R. (2008) Proving properties of lazy functional programs with SPARKLE. In *Proceedings of the 2nd Central-European Functional Programming School, CEFP 2007, Cluj-Napoca, Romania*. Horváth, Z. (ed), LNCS, vol. 5161. Springer Verlag, pp. 41–86.

Paterson, R. (2001) A new notation for arrows. In *International Conference on Functional Programming*. ACM Press, pp. 229–240.

Plasmeijer, R. & Achten, P. (2005) Generic editors for the World Wide Web. In *Central-European Functional Programming School, Eötvös Loránd University, Budapest, Hungary – Revised Selected Lectures*. LNCS, vol. 4164. Springer Verlag, pp. 1–34.

Plasmeijer, R. & Achten, P. (2006a) A conference management system based on the iData toolkit. In *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL'06*. Horváth, Z. & Zsók, V. (eds), LNCS, vol. 4449. Budapest, Hungary, Eötvös Loránd University, Faculty of Informatics, Department of Programming Languages and Compilers: Springer Verlag, pp. 108–125.

Plasmeijer, R. & Achten, P. (2006b) iData for the World Wide Web – Programming interconnected web forms. In *Proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006)*. LNCS, vol. 3945. Fuji Susone, Japan: Springer Verlag, pp. 242–258.

Plasmeijer, R. & Achten, P. (2006c) The implementation of iData – A case study in generic programming. In *Implementation and Application of Functional Languages, 17th International Workshop, ifl 2005, Dublin, Ireland, September 19–21, 2005, revised selected papers*, Butterfield, A., Grelck, C. & Huch, F. (eds), LNCS, vol. 4015. Department of Computer Science, Trinity College, University of Dublin: Springer Verlag, pp. 106–123.

Plasmeijer, R. & Eekelen, M. van. (1999) Keep it clean: A unique approach to functional programming. *ACM SIGPLAN Noti.* **34**(6), 23–31.

Plasmeijer, R. & Eekelen, M. van. (2002) Clean language report version 2.1. Department of Software Technology, University of Nijmegen.

Shneiderman, B. (1992) *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 2nd ed. Addison Wesley.

Thiemann, P. (2002) WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages: 4th International Symposium, PADL 2002*, Krishnamurthi, S. & Ramakrishnan, C. R. (eds), LNCS, vol. 2257. Portland, OR: Springer-Verlag, pp. 192–208.