Arrows for
Generic Graphical Editor Components

Peter Achten, Marko van Eekelen, Rinus Plasmeijer, Arjen van Weelden
peter88@cs.kun.nl, marko@cs.kun.nl, rinus@cs.kun.nl, arjenw@cs.kun.nl

Department of Software Technology, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

Abstract. GUI programming is hard, even for prototyping purposes. In this paper we present the Graphical Editor Component toolkit in which GUIs can be created in an abstract and compositional way. The basic building blocks are (Abstract) Graphical Editor Components ((A)GEC) with which the programmer can create GUIs by specification of the data models only. No low-level GUI programming is required. We show how these building blocks can be glued together conveniently using a combinator library based on the arrow combinators that have been introduced by John Hughes. The proofs of the associated arrow laws can be done with standard reasoning techniques without resorting to a dedicated semantic model.

1 Introduction

In the last decade, Graphical User Interfaces (GUIs) have become the de facto standard. Programming these interfaces can be done without much effort when the interface is rather static. For many of these situations excellent tools are available. However, when there is more interaction between interface and application logic, programming such applications is hard, in any programming language. Programmers need to be skilled in the use of a large programming toolkit.

One direction to reduce the complexity of GUI programming is to use a User Interface Management System (UIMS). With these systems, software designers construct UI components visually. These UI components can be stored and loaded in the running application. The main advantage of UIMSs is that UI designers can create quality user interface components with a minimum of programming knowledge. The main disadvantages are that the application code needs to synchronize its logic with these resources, and that these solutions do not work well when the UI depends on the run-time state of the application.

The other direction that can be taken to overcome this problem, is to create a programming toolkit that offers a sufficient level of abstraction and compositionality. Abstraction is required to reduce the size of the toolkit, whereas compositionality reduces the effort of putting together, or altering, GUI code. This is what the Graphical Editor project is about. Programming toolkits do offer the required expressive power when GUIs depend on the run-time state of the application. Creating GUI components in code has the additional advantage that this
code can be type-checked statically, just as conventional non-interactive code. We conjecture that having an abstract and compositional programming toolkit also eases the development of a UIMS, because this enables their implementation to map visually created GUI components to more abstract and compositional destination code.

In the Graphical Editor project, we have developed a universal building block for constructing GUls on a high level of abstraction and in a compositional way. This building block is the Graphical Editor Component (GEC) [4]. A GEC is an interactive editor for values of type t. It is universal because it works for all concrete types, including function types. This has been achieved using generic programming techniques [6, 12, 11]. Both the user and the program in which it is embedded can change the current value of a GEC, provided that it is of type t. If the user modifies the value, the program is notified of this event via a callback function. Furthermore, the program is able to retrieve the current value from a GEC.

GECs satisfy our requirement of abstraction and compositionality. They abstract from all conventional GUI programming knowledge because they only define which values are edited and not how they are edited. Compositionality is obtained because GECs are constructed automatically via the generic decomposition of the type structure whose values are edited. Creating an editor of a composite type is therefore as easy as composing the type itself.

As argued above, compositional systems facilitate modifications of existing code. Within our framework, this can be done by abstract GECs, or AGEcs [5]. An AGEc works externally as a GEC, but is implemented internally as a GEC for some type u. This means that code that is defined on the external interface, does not need to alter when the programmer experiments with different internal implementations.

From the discussion above it should be clear that the composition of GECs is within the GECs. In order to obtain an editor for values of type (a, b) one creates a GEC(a, b) editor. The goal in this paper can be stated as: suppose we have a GECa and a GECb, how can we compose them? With GECs, this can be done by using the callback functions of GECa and GECb. In general combining GECs in this way is cumbersome, can easily lead to errors, and can be very hard to reason about because there are no restrictions on the actual functions. Instead, we want to take the standard approach in functional programming to develop a small library of combinator functions. It turns out that we can base this combinator library on Hughes' arrows [14].

Finally, a note on the implementation. The project has been realized in Clean [16]. The GUI code is mapped to Object I/O [3]. The generic support of Clean is used to construct a GEC for any Clean type t, including function types. The implementation for function types reuses the Esther system [17] which relies on Clean's support for dynamics [18]. GECs have been designed not to be a replacement for Object I/O programs, but rather an additional layer on top. Given sufficient support for generic programming, this project could also have
been carried out in *Generic Haskell* [9], using the *Haskell* [15] port of Object I/O [2].

Contributions of this paper are:

- we turn GECs into basic arrow elements,
- we show that these elements are indeed arrows,
- we show that they satisfy the required laws,
- we show that the proofs of the arrow laws can be done using standard reasoning techniques for functional programs without the need to resort to a dedicated semantic model.

This paper is structured as follows. We first give an overview of GECs in Sect. 2. Sect. 3 introduces GEC arrows. We discuss the implementation of the required arrow combinators, and show how to prove the basic arrow laws. Related work is presented in Sect. 4. Finally, we conclude in Sect. 5.

2 Graphical Editor Components

In [4] we introduced the concept of a Graphical Editor Component, a GEC\textsubscript{t}. A GEC\textsubscript{t} is an editor for values of type \( t \). It is provided with an initial value of type \( t \) and it is guaranteed that an application user can only use the editor to create values of type \( t \). A GEC\textsubscript{t} always contains a value of type \( t \).

A GEC\textsubscript{t} is generated with a generic function [11, 6]. A generic function is a meta description on the structure of types. For any concrete type \( t \), the compiler is able to automatically derive an instance function of this meta description for the given type. The power of a generic scheme is that we obtain an editor for free for any data type. This makes the approach particularly suited for rapid prototyping.

Before explaining GECs in more detail, we need to point out that *Clean* uses an explicit multiple environment passing style [1] for I/O programming. Because GECs are integrated with *Clean* Object I/O, the I/O functions that are presented in this paper are state transition functions on the program state (PSt ps). The program state represents the external world of an interactive program, tailored for GUI operations. In this paper the identifier env is a value of this type. In the Haskell variant of Object I/O [2], a state monad is used instead. The uniqueness type system [7] of *Clean* ensures single threaded use of the environment. Uniqueness type attributes that actually appear in the type signatures are not shown in this paper, in order to simplify the presentation.

2.1 Creating GECs

GECs are created with the generic function gGEC. This function takes a definition (GECDef \( t \) env) of a GEC\textsubscript{t} and creates the GEC\textsubscript{t} object in the environment. It returns an interface (GECInterface \( t \) env) to that GEC\textsubscript{t} object. It is a (PSt ps) transition function because gGEC modifies the environment.
A $GEC_t$ is defined by a $GECDef t env$ which consists of three elements. The first is a string that identifies the top-level Object I/O element (window or dialog) in which the editor must be created. The second is a value of type $t$ which will be the initial value of the editor. The third is a callback function of type $t ightarrow env ightarrow env$. This callback function is provided by the context of the editor, and tells it which parts of the program need to be informed of user-edit actions. The editor uses this function when the user has changed the current value of the editor.

The $GECInterface t env$ is a record that contains all methods that the ‘context’ can use to employ the newly created $GEC_t$.

Let $gec_t :: GECInterface t env$ be such an interface to a $GEC_t$ with callback function $f$. Using the explicit environment passing style of Clean, a program can obtain the current value by:

$$\#(v, env) = gec.gecGetValue env$$

and change it to $v'$ with:

$$\# env = gec.gecSetValue ... v' env$$

The $\#$ notation of Clean has a special scope rule such that the same variable name can be used for subsequent non-recursive $\#$-definitions. It is particularly suited for the explicit environment passing style of Clean. In this paper we use this notation in order to emphasize the ‘natural’ threading of environments. At some points we need to deviate from this style, because there are recursive dependencies between local definitions. In those cases, we will annotate the environments $env$ with numbers, in order to indicate their relative threading (so we use $env1, env2, ...$).

The first argument of the $gecSetValue$ method is of type $IncludeUpdate$, which is a simple algebraic data type:

$$:: IncludeUpdate = NoUpdate | YesUpdate$$

This argument controls the flow of information. If the argument of $gecSetValue$ is $NoUpdate$, then its effect is simply to set the new value of $gec$ to $v'$. If the

---

1. $::$ introduces a synonym type.
2. Record types have exactly one alternative.
3. $r.f$ denotes the record field selection of $f$ from $r$. 
argument of gecSetValue is YesUpdate, then its effect is that immediately after
the new value of gec is set to v', its callback function f is evaluated with argument
v' (as if the user had edited the current value to v'). Put in other words, it has
the same effect as:

\[
\begin{align*}
\text{env} &= \text{gec}.\text{gecSetValue } \text{NoUpdate } v' \text{ env} \\
\text{env} &= f(v') \text{ env}
\end{align*}
\]

Additionally, GECInterface contains several other useful methods for a pro-
gram that are not shown above. These are methods to open and close the created
GEC, and to show or hide its visual appearance.

The appearance of a standard GEC is illustrated by the following complete
program that creates an editor for the well-known Tree type:

```haskell
module TreeEditor

import StdEnv, StdIO, StdGEC

Start :: *World → *World // Entry of Clean program
Start world
  = startIO // Entry of Object I/O program
  SDI // Request single window
  Void // Empty application state
  myEditor // Create GEC
  world

myEditor = snd o4 gGEC ("Tree",Node Leaf 1 Leaf,const id)

:: Tree a = Node (Tree a) a (Tree a) | Leaf
```

Note that the only things that need to be specified by the programmer are
the initial value of the desired type, and the callback function. In the remainder
of this paper, we will only modify the myEditor definition in order to produce a
wide range of examples.

In this particular example, we create a GECTreeInt which displays the indi-
cated initial value (see Fig. 1). The application user can manipulate this value
in any desired order thus producing new values of type TreeInt. Each time a
new value is created, the callback function is applied automatically. The callback
function of this first example (const id) has no effect. The shape and lay-out of
the tree being displayed adjusts itself automatically. Default values are generated
by the editor when needed.

2.2 Semantics of GECs

The example program above illustrates that GECs can be created in an Object
I/O program. If we want to explain the meaning of GECs, we first have to explain
the meaning of Object I/O programs. We do this by presenting an abstract

\[ o \] is the standard function composition operator.
version of the actual Clean code in which Object I/O has been written. We want to show the essence of Object I/O, rather than a stripped down version of Object I/O in Clean. The reason is that even a stripped down version must be type correct and complete. This is not our goal. Therefore, we do not use pure Clean syntax, but deviate where useful.

Every interactive program is a function that manipulates the external world. For our purposes, it is sufficient that this world, represented by the data type World, contains an infinite event stream, and an infinite identification value stream:

$$\text{World} = \delta ([\text{Event}], [\text{Id}], \ldots)$$

The exact nature of events or identification values is not important, we only require them to be comparable. Of course, the identification value stream contains no duplicate elements.

An Object I/O process is a state-transition system. It manipulates a process state (PSt ps) that consists of a program state (ps) and an I/O state (IOSI ps). The first is defined by the program, the second contains all information required to handle GUIs (GUI ps) and the external world (World).

$$\text{PSt ps} = \delta (ps, \text{IOSI ps})$$
$$\text{IOSI ps} = \delta (\text{GUI ps}, \text{World})$$

Again, the exact representation of GUI ps is irrelevant. It is parameterized with the program state only because it contains all callback functions of all GUI components. We assume that we can store these functions with standard set operations, and retrieve them via their associated events, using the function getCallBackFun :: Event → (GUI ps) → (PSt ps) → (PSt ps).

The Object I/O function startIO turns the World into an initialized PSt ps, for any program state and initialization function. Then, as usual with event driven applications, it enters the event-loop until termination (a 'quit' event).

$$\text{startIO} :: \text{ps} \rightarrow ((\text{PSt ps}) \rightarrow (\text{PSt ps})) \rightarrow \text{World} \rightarrow \text{World}$$

$$\text{startIO ps initIO w}$$
= eventloop (initIO (ps, initializeIOSt w))

where eventloop :: (PSt ps) → (PSt ps)
    eventloop (ps, (∅, w)) = (ps, (∅, w))
    eventloop (ps, (gui, (∥:es, ids)))
    = eventloop (f (ps, (gui, (∥:es, ids))))

where f = getCallBackFun e gui

In Sect. 2.1, we have introduced the GEC creation function, gGEC, that is parameterized with a string s, an initial value v :: t and callback function f :: CallBackFunction t (PSt ps). In essence, gGEC (s, v, f) is an action that creates the GEC and returns its interface i :: GECInterface t (PSt ps). The creation of the GEC is represented by storing it in the GUI ps after tagging it with a fresh identification value.

gGEC (s, v, f) (ps, (gui, (∥:es, ids)))
= (i, (ps, (gui ∪ {(id, v, f)}, (∥:es, ids)}))

where
i = (get’ id, set’ id)

Interface i is actually a record of the two functions, gecGetValue and gecSetValue. These are modeled via the functions get’ id and set’ id respectively, which are parameterized with the proper identification value for retrieval purposes. The method get’ id returns the currently stored value of the editor indicated by id, and set’ id replaces the currently stored value. If it is also applied to YesUpdate, it evaluates the associated callback function of the editor.

gecGetValue (ps, (gui, w))
= v if (id, v, f) ∈ gui
= ⊥ otherwise

gecSetValue (ps, (gui, w))
= (ps, (gui’, w)) if (id, v, f) ∈ gui ∧ iu = NoUpdate
= f (ps, (gui’, w)) if (id, v, f) ∈ gui
= ⊥ otherwise

where
gui’ = gui \ {(id, v, f)} ∪ {(id, v’, f)}

We assume that getCallBackFun is able to track down the callback function f whenever the user edits the corresponding GEC.

2.3 Manual composition of GECs

In this section we present a number of examples to show how GECs can be combined relying on the callback mechanism and method invocation. In Sect. 3.6 we show how these examples can be expressed using arrow combinators.

The first example establishes a functional dependency of type a → b between a source editor GEC_a and destination editor GEC_b:
applyGECs :: (String, String) (a → b) a (PSt ps) → PSt ps

applyGECs (sa, sb) f va env
% (gec_b, env) = gGEC (sb, f va, const id) env
% (gec_a, env) = gGEC (sa, va, set gec_b f) env
  = env

set :: (GEC b (PSt ps)) (a → b) a (PSt ps) → (PSt ps)
set gec f va env = gec.gecSetValue NoUpdate (f va) env

The callback function of \( gec_a \) uses the gecSetValue interface method of \( gec_b \) to update the current \( b \) value whenever the user modifies the \( a \) value. As a simple example, one can construct an interactive editor for lists that are mapped to balanced trees by the following single change of definition of the example program shown in Sect. 2.1 (see Fig. 2):

myEditor = applyGECs ("List", "Balanced Tree")
  balancedTree [1, 5, 2]

with balancedTree :: [Int] → Tree Int.

Fig. 2. Turning lists into balanced binary trees.

Of course, the same can be done for binary functions with slightly more effort:

apply2GECs :: (String, String, String) (a → b → c)
  a b (PSt ps) → (PSt ps) | gGEC[k4] a, b, c
apply2GECs (sa, sb, sc) f va vb env = env3
where
  (gc, env1) = gGEC (sc, f va vb, const id) env
  (gb, env2) = gGEC (sb, vb, combine ga gc (flip f)) env1
  (ga, env3) = gGEC (sa, va, combine gb gc f) env2

5 Types of function definition separate arguments with whitespace instead of →.
6 Class restrictions appear at the end of a type.
7 Use the generic instance of kind * of gGEC.
combine :: (GEC y (PSt ps)) (GEC z (PSt ps))
        (x -> y -> z) x (PSt ps) -> PSt ps
combine gy gz f x env
    % (y,env) = gy.gecGetValue env
    % env = gz.gecSetValue NoUpdate (f x y) env
    = env

Notice that, due to the explicit environment passing style, it is trivial in Clean
to connect $GEC_b$ with $GEC_a$ and vice versa. In Haskell's monadic I/O one needs
to tie the knot with fixI0.

As an example, one can construct two interactive list editors, that are merged
and put into a balanced tree:

myEditor = apply2GECs ("List1","List2","Balanced Tree")
            makeBalancedTree [] []
where
    makeBalancedTree 11 12 = balancedTree (11 ++ 12)

with ++ :: [a] [a] -> [a] the Clean list concatenation operator. Fig. 3 shows
the result.

Fig. 3. Merging two lists into a balanced binary tree.

The final example is that of self-correcting editors. These are editors that
update themselves in response to user edit operations. The function definition is
concise:

selfGEC :: String (a -> a) a (PSt ps) -> (PSt ps)
        | gGEC |*| a
selfGEC sa f va env = envi
where
    (thisGEC,env1) = gGEC (sa,f va,set thisGEC f) env

As an example, one can now construct a self-balancing tree with (see Fig. 4):
myEditor = selfGEC "Self Balancing Tree"
  (balancedTree o toList) Leaf

  with toList :: (Tree a) → [a]. This means that it is impossible for a user of
  this editor to create a stable non-balanced tree value.

Fig. 4. Self balancing binary tree.

2.4 Customizing GECs

The generic function gGEC creates a default editor for arbitrary values of given

type. This makes it universally applicable to all data domains. In order to make

it a flexible tool, one needs to be able to deviate from the default when required

by the application under construction. In this section we show that this can be

done for all values of a given type (Sect. 2.4) and even for specific values (Sect.

2.4), using AGECs.

Customizing Types Clean allows generic functions to be overruled by cus­
tom definitions for arbitrary types. gGEC is no exception to this rule. The left

screenshot in Fig. 5 shows the default interface of the definition below for the

ubiquitous counter example, when created by:

myExample = selfGEC "Counter" updCntr (0,Neutral)

updCntr :: Counter → Counter

updCntr (n,Up) = (n+1,Neutral)

updCntr (n,Down) = (n-1,Neutral)

updCntr any = any

:: Counter ::= (Int,UpDown)

:: UpDown = Up | Down | Neutral

Although the definition of the counter is a sensible one, its visual interface

clearly is not. In [4] we show how to change the representation of all values

type Counter to the screenshot shown at the right in Fig. 5. Because it has

been explained in detail in [4], we will not repeat the code, but point out the

important points:
— In this particular example, only the definitions of (, ) (hide the constructor and place its arguments next to each other) and UpDown (display \(\pm 1\) instead of \(\text{Neutral} \rightarrow \text{Neutral} \rightarrow \text{Neutral}\)) need to be changed.
— Normally ggEC creates the required logical (value passing) and visual infrastructure (GUI components). The programmer, when customizing ggEC, only needs to define the visual infrastructure. The programmer must be knowledgeable about Object I/O programming.
— The overruled instance works not only at the top-level. Every nested occurrence of the Counter type is now represented as shown right in Fig. 5.

Customizing Values Above we have shown how the programmer can change the editor interface for any type, at all occurrences. This is in some cases much too rigid. One can not use different visual appearances of the same type within a program. An approximation is to give a different type to each different occurrence, at the expense of flexibility: changing the visual appearance via a change of type requires modification of the code. What is needed is the same level of abstraction for editors as multiple implementation abstract data types do for ‘conventional’ data types. This has resulted in abstract GECs (AGEC\(_a\)) [5].

In the following example we use three AGEC\(_{int}\) editors in combination with an AGEC\(_{Int \rightarrow Int} \rightarrow \text{Int}\) editor to construct a GUI in which the user can enter integer values (using counterAGEC which have the counters described above as internal implementation) and function definitions of type Int Int \(\rightarrow\) Int (using dynamicAGEC which offers a strongly typed, textual editor for arbitrary types [17]). At each edit operation, the current function is applied to the current arguments. Because AGEC editors are abstract types themselves, their current value is obtained via the prefix operator \(^{\text{new}}\). Note that the programmer can freely experiment with abstract editors in the definition of toAGEC without changing any other piece of code. The result of this particular editor is shown in Fig. 6.

```plaintext
myEditor = selfAGEC "test operator"
            (toAGEC o updFun o fromAGEC) (toAGEC funTest)
updFun x
  = { x \&^6 result = x.f x.arg1 x.arg2 } // apply f to args
toAGEC x
  = { arg1 = counterAGEC x.arg1 } // counter editor
```

\(^6\{r \& ^f = v\}\) denotes a new record value, that is equal to \(r\), but with value \(v\) for field \(f\).
Again, we summarize the main results:

- We can define arbitrarily many editors \( gec_1 :: AGEC_t \) that have a private implementation of type \( GEC_u \).
- For every \( GEC_t \), there is an \( AGEC_t \) that has the former as its implementation. This is the identity \( AGEC_t \).
- Code that has been written for editors that manipulates some (type containing) \( AGEC_t \), does not change when the value of type \( AGEC_u \) is changed for another \( AGEC_t \). This facilitates experimenting with various designs for an interface without changing its code.
- In contrast with customizing types, when customizing values the programmer does not have to be knowledgeable about Object I/O programming. The only things that need to be defined when creating an \( AGEC_t \) that has a \( GEC_u \) implementation are an initial value of type \( t \), conversion functions \( (t \rightarrow (\text{Maybe} \ u) \rightarrow u \text{ and } u \rightarrow t) \), and a self-correcting function for the implementation \( (u \rightarrow u) \). These are all expressed at the data domain level.

3 Combining GECs using Arrows

The examples in Sect. 2.3 show that GECs can be composed by writing appropriate callback functions that use the GECInterface methods \( gecGetValue \) (get the value of a GEC) and \( gecSetValue \) (set its value). This explicit plumbing can
become cumbersome when larger and more complex situations must be specified. What is needed, is a disciplined, and more abstract way of combining components. Monads [19] and arrows [14] are the main candidates for such a discipline. Monads abstract from computations that produce a value, whereas arrows abstract from computations that, given certain input, produce values. Because GECs also have input and produce values, arrows are the best match. In this section we show how arrows can be used successfully for the composition of GECs, resulting in structures that resemble circuits of GECs (GecCircuit a b).

In Sect. 3.1 we show that GecCircuit is an instance of the Arrow class by providing implementations of the basic arrow combinators. Given these circuit-like structures, we show how to embed them properly in Object I/O (Sect. 3.2), and, conversely, how arbitrary Object I/O functions can be embedded in circuits themselves (Sect. 3.3). In Sect. 3.4 special combinators are presented that abstract from recursion and looping. In order to be a full member of the Arrow class, the corresponding arrow laws [14] have to hold. We show in Sect. 3.5 that these laws can be proven in a surprisingly straightforward manner. Finally, Sect. 3.6 concludes with redefinitions of the examples in Sect. 2.3 using the GEC arrows. In order to illustrate the expressive power, a more complex example is also presented, namely that of an editor-editor.

3.1 Definition of GEC-Arrows

The arrow class definition for which we need to provide implementation for our GEC arrows of type GecCircuit is given below. This class describes the basic combinators >>> (serial composition), arr (function lifting), and first (saving values across computations). The other definitions below can all be derived in the standard way from these basic arrow combinators. They are repeated here because we use them in our examples (Sect. 3.6).

class Arrow arr where
   arr :: (a -> b) -> arr a b
   (>>>) :: (arr a b) -> (arr b c) -> arr a c
   first :: (arr a b) -> arr (a,c) (b,c)

/* Combinators for free: */
second :: (arr a b) -> arr (c, a) (c, b)
second gec = arr swap » arr first gec » arr swap
where
   swap t = (snd t ,fst t)

returnA :: arr a a
returnA = arr id

(<<<) infixr 1 :: (arr b c) (arr a b) -> arr a c
(<<<) 1 r = r <<< 1

(<<<) infixr 3 :: (arr a b) (arr c d) -> arr (a,c) (b,d)
(<<<) 1 r = first l <<< second r

13
\[ \text{infixr 3 :: (arr a b) (arr a c) \rightarrow arr a (b, c)} \]
\[ \text{l r = arr (\lambda x \rightarrow (x,x)) \triangleright \triangleright (l *** r)} \]

The GEC-Arrow type

It is the task of our arrow model to introduce a standardized way of combining GECs. As explained in Sect. 2.1, one uses a \( \text{GEC}_t \) through its interface of type \( \text{GECInterface } t \text{ env} \). Method \( \text{gecSetValue :: GecSet } t \text{ env} \) sets a new value of type \( t \) in the associated \( \text{GEC}_t \), and \( \text{gecGetValue :: GecGet } t \text{ env} \) reads its current value of type \( t \).

If we generalize these types, then we can regard a \( \text{GEC-to-be-combined} \) as a component that has input \( a \) and output \( b \) (where \( a = b = t \) in case of a 'pure' \( \text{GEC}_t \)). This generalization of a \( \text{GEC-to-be-combined} \) has type \( \text{GecCircuit } a \ b \) because of its resemblance with electronic circuits. Consequently, this \( \text{GecCircuit } a \ b \) has a slightly more general interface, namely a method to set values of type \( \text{GecSet } a \text{ env} \), and a method to get values of type \( \text{GecGet } b \text{ env} \). This generalized flow of control of a circuit is visualized in Fig. 7.

![Fig. 7. A GEC Circuit (external view).](image)

When circuits are combined this will yield a double connection (one forward \( \text{set} \) and one backward \( \text{get} \) for each circuit). It is essential to realize that usage of the \( \text{set} \) method is restricted to the circuit that produces that input, and, likewise, usage of the \( \text{get} \) method is restricted to the circuit that needs that output.

Moreover, a \( \text{GEC-to-be-combined} \) of type \( \text{GecCircuit } a \ b \) needs to know where to send its output to, and where to obtain its input from. More precisely, it is only completely defined if it is provided with a corresponding \( \text{set} \) method (of type \( \text{GecSet } b \text{ env} \)) and a \( \text{get} \) method (of type \( \text{GecGet } a \text{ env} \)). These methods correspond exactly with the 'missing' methods in Fig. 7. Put in other words, a \( \text{GecCircuit } a \ b \) behaves as a \emph{function}. Indeed, the way we obtain the restricted communication is by passing \emph{continuation functions}. Through these continuations values are passed and set throughout the circuit. Each \( \text{GecCircuit } a \ b \) is a function that takes two continuations as arguments (one for the input and one for the output) and produces two continuations. The way a circuit takes its continuation arguments, creates a circuit and produces new continuations, can be visualized with the internal view of a circuit (see Fig. 8).

A \( \text{GecCircuit} \) is not only a continuation pair transformation function but it also transforms an Object I/O environment since it also has to be able to incorporate the environment functions for the creation of graphical editor components. These environment functions are of type \( \text{(PSt ps)} \rightarrow \text{(PSt ps)} \).
The global idea sketched above motivates the following full definition of the \texttt{GecCircuit} \(a\ b\) type:

\[
\texttt{:: GecCircuit} \ a\ b = \texttt{GecCircuit} \  (\forall \ ps:\ (\texttt{GecSet} \ b \ (PSt \ ps), \texttt{GecGet} \ a \ (PSt \ ps), PSt \ ps) \\
\quad \quad \quad \quad  \rightarrow (\texttt{GecSet} \ a \ (PSt \ ps), \texttt{GecGet} \ b \ (PSt \ ps), PSt \ ps))
\]

The circuits do not depend on the program state \(ps\). This is expressed elegantly using a rank-2 polymorphic function type.

**Lifting a GEC to a GEC arrow** Before implementing the arrow combinators we first explain how we lift a \textit{GEC} to a circuit. This is done by the function \texttt{edit}. Its overloaded type conveniently expresses that for every \textit{GEC}_a, created by the \* indexed instance of \textit{gGEC}, there also exists a \texttt{GecCircuit} \(a\ a\). The outside view of an edit circuit is illustrated in Fig. 9.

\[
\texttt{edit} :: \text{String} \rightarrow \text{GecCircuit} \ a\ a \mid \text{gGEC}[^\*] \ a
\]

\texttt{edit} \(s\) = \texttt{GecCircuit} \(k\)

where

\[
k \ (\texttt{seta}, \texttt{geta}, \texttt{env})
\]

\[
= (\texttt{seta}, \texttt{geta}, \texttt{env})
\]

A \textit{GEC}_a is created which, when it is initialized, fetches its initial value using the result of the get-function of its input argument. Furthermore, its callback function is defined such that editing this \textit{GEC}_a will result in calling the set-function of the circuit’s output connection with the new edited value. Finally, the \textit{GEC}-interface which is the result of creating the \textit{GEC}_a consists of a get function.
and a set function. These functions are exactly its `GecInterface a env` methods, obtained from `gGEC`. The internal view of the edit circuit is shown in Fig. 10.

![Fig. 10. A GEC edit circuit (internal view).](image)

**Basic GEC-Arrow combinators** In this section we implement the three basic `Arrow` class combinators `>>>`, `arr`, and `first`.

The **arrow combinator `>>>`** The external view of composition of circuits is given in Fig. 11.

![Fig. 11. Composition of two GEC circuits, external view.](image)

The basic serial composition of arrows applies the circuit functions of its arguments to the appropriate continuations, yielding a new component. It is defined as:

\[
\text{>>>} :: (\text{GecCircuit } a \ b) (\text{GecCircuit } b \ c) \rightarrow \text{GecCircuit } a \ c
\]

\[
\text{>>>} (\text{GecCircuit } l) (\text{GecCircuit } r) = \text{GecCircuit } k
\]

where

\[
k (\text{setc}, \text{geta}, \text{env}) = (\text{seta}, \text{getc}, \text{env2})
\]

where

\[
(\text{seta}, \text{getb}, \text{env1}) = l (\text{setb}, \text{geta}, \text{env})
\]

\[
(\text{setb}, \text{getc}, \text{env2}) = r (\text{setc}, \text{getb}, \text{env1})
\]

The definition of `>>>` comes naturally when you consider the types of each circuit. The circuit `l` is of type `GecCircuit a b`. Hence, `l` is applied to a `setb` and a `geta` function and produces a `seta` and a `getb` function.

It may be surprising that filling in the natural applications of `l` and `r` yields mutually recursive definitions. Due to laziness the actual dependencies can be resolved, because they are not circular dependent upon execution. There is one
exception which may cause an unwanted runaway computation as is the case in
looping circuits [13]. In Sect. 3.4 we treat solutions to this problem.

In Fig. 12, it is illustrated how the connections are made when two compo-
nents are composed.

![Fig. 12. Composition of two GEC circuits, internal view.](image)

The arrow combinator arr Lifting a function to a GEC circuit can be done
without creating an editor. The external view of a lifted function circuit is given
in Fig. 13.

![Fig. 13. The arr combinator, external view.](image)

arr :: (a -> b) -> GecCircuit a b
arr f = GecCircuit k
where
  k (setb, geta, env) = (seta, getb, env)
  where
    getb env
    = (a, env) = geta env
    = f a, env
    seta u a env = setb u (f a) env

The function f is simply applied inside the seta function as well as inside the
getb function. Both the set and the get functions are chained together through
application.

In Fig. 14, it is illustrated how the connections are made in order to create
the arr combinator.
The arrow combinator first. The external view of the first combinator is given in Fig. 15.

The first combinator is defined as:

\[
\text{first} :: \text{GecCircuit} \ a \ b \rightarrow \text{GecCircuit} \ (a, c) \ (b, c)
\]

\[
\text{first} \ (\text{GecCircuit} \ g) = \text{GecCircuit} \ k
\]

where

\[
k \ (\text{setbc, getac, env}) = (\text{setac, getbc, env})
\]

where

\[
(\text{seta, getb, env}) = g \ (\text{setb, geta, env})
\]

\[
\text{setac} \ u \ (a, c) \ \text{env} = \text{setac} \ u \ a \ \text{env}
\]

\[
\text{setac} \ u \ (a, c) \ \text{env} = \text{setac} \ u \ (a, c) \ \text{env}
\]

\[
\text{getbc} \ \text{env} = \text{getbc} \ \text{env}
\]

\[
\text{setb} \ u \ b \ \text{env} = \text{setb} \ u \ b \ \text{env}
\]

\[
\text{geta} \ \text{env} = \text{geta} \ \text{env}
\]
As was the case for composition and lifting, the first combinator is defined straightforwardly considering the types of the circuits. Producing a \( \text{setac} \) and a \( \text{getbc} \) function requires a \( \text{seta} \) and a \( \text{getb} \) function. A \( \text{seta} \) and a \( \text{getb} \) function can be produced by applying the circuit function \( g \) on a \( \text{setb} \) and \( \text{geta} \) function which in turn can be produced with \( k \)'s argument functions \( \text{setbc} \) and \( \text{getac} \).

In Fig. 16 it is illustrated how the connections are made in order to create the first combinator.

![Fig. 16. The first combinator, internal view.](image)

This completes the basic set of definitions required.

### 3.2 GEC Arrows in Object I/O

Now that we have shown how to lift every \( \text{GEC}_t \) to \( \text{GecCircuit} \; t \; t \), and know how to glue circuits with arrows, we need to show how such a circuit comes to life in Object I/O. This is done with the function \( \text{startCircuit} \) which basically turns a circuit into an Object I/O state transition function. As such it can be used in the \( \text{myEditor} \) function of Sect. 2.3.

\[
\text{startCircuit} :: (\text{GecCircuit} \; a \; b) \; a \; (\text{PSt} \; ps) \rightarrow \text{PSt} \; ps
\]

\[
\text{startCircuit} \; (\text{GecCircuit} \; k) \; a \; \text{env} =
\]

\[
\langle \_ , \_ , \text{env} \rangle = k \; (\text{setb} , \text{geta} , \text{env})
\]

where

\[
\text{geta} \; \text{env} = (a , \text{env})
\]

\[
\text{setb} \; \_ \; \_ \; \text{env} = \text{env}
\]

Upon creation, the circuit function is applied to a \( \text{geta} \) function producing the initial argument and a dummy \( \text{set} \) function that just passes the environment.

### 3.3 Object I/O in GEC Arrows

The \( \text{startCircuit} \) function can be used just as any other Object I/O environment function. It is also possible to promote an Object I/O environment function to a \( \text{GEC} \) circuit. This is done with the function \( \text{gecIO} \) which enables the programmer embed all functionality offered by the large Object I/O library. It has the following definition:
gecIO :: \( \forall \) ps: \( a \rightarrow (\text{PSt} \ ps) \rightarrow (b, \text{PSt} \ ps) \)
\( \rightarrow \) GecCircuit a b

gecIO \( f = \) GecCircuit \( k \)
where
\( k \) (setb, geta, env) = (seta, getb, env)
where
getb env
\( \downarrow \) (a.env) = geta env
= \( f \) a env

seta u a env
\( \downarrow \) (b.env) = \( f \) a env
= setb u b env

A warning is at its place here. With gecIO it is possible to embed all kinds of Object I/O environment functions within a GEC circuit. Although it is simply not possible to violate the properties proven in Sect. 3.5 such environment functions can include all kinds of interactive actions. Of course it is up to the programmer to make sure that the overall program still behaves in the intended way.

3.4 Feedback

Feedback to a circuit can be given using the following definition of the feedback function. Of course the input and the output must be of the same type (see its external view in Fig. 17).

Fig. 17. Feedback of a GEC circuit, external view.

feedback :: (GecCircuit a a) \( \rightarrow \) GecCircuit a a
feedback (GecCircuit g) = GecCircuit \( k \)
where
\( k \) (seta, geta, env)
\( \downarrow \) (a.env1) = geta'. env1
\( \downarrow \) env1 = seta'. NoUpdate a env1
= (seta', geta', env1)

where
(seta', geta', env1) = g (seta'', geta, env)
seta'' u a env = seta u a (seta'. NoUpdate a env)
The way the feedback combinator constructs a feedback circuit is by taking the value of the circuit and feeding it back again into the circuit (see also Fig. 18 for its internal view). This is done in such a way that it will not be propagated further when it arrives at a GEC editor. This is achieved using NoUpdate (see also Sect. 2.1 in which NoUpdate is introduced).

![Fig. 18. Feedback of a GEC circuit, internal view.](image)

When a feedback circuit contains no editor at all, the meaning of the circuit is undefined since in that case the calculation of the result would depend on itself in a circular way. A feedback circuit in which each path of the circuit contains an editor, is called well-formed. It is easy to check syntactically whether feedback circuits are well-formed. Consider the following examples of non-well-formed and well-formed feedback circuits.

```plaintext
nonWellFormed1 = feedback (arr id >>> arr ((+ 1))
nonWellFormed2 = feedback (arr id &&& edit "Int" >>>
                        arr (λ(x, y) → x + y) )
wellFormed = feedback (edit "Int" >>> arr ((+ 1))
```

### 3.5 Properties

A GecCircuit `a b` is *not* a pure function from `a` to `b` (i.e. without side effects). Instead, a GecCircuit `a b` is a pure function from a `get/set` pair and a PST `ps` to a `get/set` pair and a PST `ps`. Its definition is fully specified in Clean. All the arrow combinators are also fully specified in Clean. Clean functions may use Object I/O functions. In Object I/O side effects are modelled via the abstract polymorphic type PST `ps` of which the single threaded use is guaranteed by uniqueness typing. Apart from the assumption that all side-effects are modelled in this way within Object I/O, no other assumption is necessary in order to reason about GecCircuits.

The fact that the basic definitions are fully given within the programming language is quite different from the arrows defined in the Yampa [13] and the Fruit system [10] where the basic definitions rely on special semantic functions. In that case proofs of arrow laws have to be done on the level of the introduced semantics using appropriate reasoning techniques for that semantic level.

In our case proofs can be done using the actual function definitions and standard reasoning techniques for functional programs. The main techniques we will use are unfolding and extensionality.
Proving Arrow Laws  To illustrate the process of proving arrow properties we will show how to prove the basic arrow laws that are stated by John Hughes in [14]. They correspond roughly to the monad laws.

- \( \text{arr id} \circ \circ \ l = l = l \circ \circ \ \text{arr id} \)
- \( (l \circ \circ \ r) \circ \circ \ s = l \circ \circ \ (r \circ \circ \ s) \)
- \( \text{arr} \ (g \circ f) = \text{arr} \ f \circ \circ \ \text{arr} \ g \)

In expressing the laws above as well as in the rest of this section, we consistently use \( f, g, \) and \( h \) for functions and \( l, r, \) and \( s \) for circuits. We prove partial correctness only. Hence, we will assume throughout this section that no values or functions are undefined.

The first law states that the lifting combinator \( \text{arr} \) and the composition combinator \( \circ \circ \) are consistent with the identity function.

\[
\text{arr id} \circ \circ \ l = l = l \circ \circ \ \text{arr id}
\]

**Proof.** There are two statements to prove: \( \text{arr id} \circ \circ \ l = l \) and \( l = l \circ \circ \ \text{arr id} \).

Take the first statement. The left-hand side can be transformed by subsequent transformations to the right-hand side. This proof is given in full detail in the appendix.

The proof of the other statement (\( l = l \circ \circ \ \text{arr id} \)) is analogous to the previous one.

Due to the length of the proofs we merely sketch other proofs leaving it to the reader to work out the precise proof steps. The second law states that composition of circuits is left-associative.

\[
(l \circ \circ \ r) \circ \circ \ s = l \circ \circ \ (r \circ \circ \ s)
\]

**Proof.** The proof of this statement can be given easily using the same techniques as with the previous proof. In this case it is convenient to transform both the left-hand side and the right-hand side of the property to a common equivalent function. This equivalent function is \( \text{gec} \, \text{composition} \):

\[
\text{gec} \, \text{composition} \ s = (l \circ \circ \ r) \circ \circ \ s = l \circ \circ \ (r \circ \circ \ s)
\]

**Note:** That \( \text{gec} \, \text{composition} \) naturally expresses the fact that it is the composition of three circuit functions.

The third law states that lifting functions to circuits distributes over function composition into circuit composition.

\[
\text{arr} \ (g \circ f) = \text{arr} \ f \circ \circ \ \text{arr} \ g
\]
Proof. We proceed as in the previous proof. We can prove both sides of the
property to be equivalent to the single function `arr_distribution`:

```plaintext
arr_distribution /= arr (g o f) = arr f >>= arr g */
= GecCircuit k
```

where

```plaintext
k (setb,geta,env) = (seta,getb,env)
```

where

```plaintext
getb env
  \$ (a,env) = geta env
  = (g (f a),env)
```

```plaintext
seta u a env = setb u (g (f a)) env
```

Note that `arr_distribution` naturally expresses composition of functions
within a circuit.

This completes the proofs of the basic GEC arrow laws. Other properties can
be proven in a similar way.

Other properties of GEC Arrows

As is common with arrows, duplicating
an arrow makes a semantic difference, e.g.

```
l >>= (r &&& s) \neq (l >>= r) &&& (l >>= s)
```

For circuits the difference lies in the fact that duplicating a circuit may mean
applying an environment function twice. Clearly, the circuits `edit >>= (arr id
&&& arr id)` and `(edit >>= arr id) &&& (edit >>= arr id)` are not equivalent since
the first contains only one editor and the latter contains two editors (also on­
screen).

Propagation of edited values

If a circuit contains a GEC component, then a
change of the value of that component will always propagate to the end of
the circuit starting at the edit component. In the case of a well-formed feedback
circuit, it will also propagate to the beginning of the feedback circuit and to
each path from that point on up to the first GEC editor on that path. An initial
value propagates from the beginning of the circuit to the end including possible
feedbacks.

```plaintext
propagationexample = arr ((+) 1) >>= edit "Propagation" >>= arr ((*) 2)
```

If `propagationexample` is created with initial value 0 then the initial result of
the circuit will be 2. When the value of the edit component is changed by a user
into 10, then this value is propagated through the circuit to the end. Therefore,
the first lifted function is not applied and the result is 20 and not 21. Now,
extend the example with a feedback combinator:

```plaintext
propagationexample2 = feedback propagationexample
```

If `propagationexample2` is created with initial value 0 then the initial result of
the circuit will be 2. However, the value is also propagated through the feedback
up to the edit component. Therefore, the edit component will display 3.
When the value of the edit component is edited by a user and changed into 10, then this value is propagated through the circuit to the end. The result is 20. However, the value is also propagated through the feedback up to the edit component and the edit component will display 21.

The propagation mechanism achieves a natural behavior from the user's point of view.

### 3.6 Examples

We use the arrow combinator definitions from Sect. 3.1 in the examples that are given below. For each example of Sect. 2.3, we give the definition using arrow combinators, and some of the circuit structures as figures.

The first example (of which the external view is given in Fig. 19) shows the arrow combinator version of the applyGECs example of Sect. 2.3.

![Fig. 19. applyGECs using arrows, external view.](image)

myEditor = startCircuit applyGECs [1,5,2]

applyGECs :: GecCircuit [Int] (Tree Int)
applyGECs = edit "List" >>>
            arr balancedTree >>>
            edit "Balanced Tree"

Again, two visual editors are shown. The first allows the user to edit the (initial) list, and the second shows (and allows the user to edit) the resulting balanced tree. In the hand coded examples, the initial value of a GEC was specified at the time of its creation. Using the arrow combinators to construct a GecCircuit, we specify the initial values for all GECs when we start the circuit.

Particularly interesting is the use of the edit combinator (see Sect. 3.1 for its definition). The example above has two occurrences of edit. However, the occurrences do not yield the same result since they are of different type. Due to the way edit is defined with a generic function, it will produce a circuit with an \( GEC_{\text{[Int]}} \) editor component if the inferred type is [Int]. However, if the inferred type for edit is Tree Int then the resulting circuit contains a \( GEC_{\text{Tree Int}} \) editor component.

myEditor = startCircuit apply2GECs ([],[])

apply2GECs :: GecCircuit ([Int], [Int]) (Tree Int)
apply2GECs = edit "list1" *** edit "list2" >>>
    arr makeBalancedTree >>>
    edit "Balanced Tree"

where

makeBalancedTree (l1,l2) = balancedTree (l1 ++ l2)

The example above (see Fig. 20 for its external view) shows the arrow combinator version of the apply2GECs example. The initial values for the input lists are paired, to allow the delayed initialization using startCircuit. The example clearly shows that combining GECs using arrow combinators is much more readable than the (often) recursive handwritten functions. The linear flow of information between GECs, using the >>> combinator, corresponds directly with the code. Although splitting points in flow of information, using the *** combinator, is less clear, it is still easier on the eyes than the examples of Sect. 2.3.

The example below shows the arrow combinator version of the first selfGEC example (see its external view in Fig. 21). This example makes use of feedback, and is obviously well-formed.

myEditor = startCircuit selfGEC Leaf

selfGEC :: GecCircuit (Tree Int) (Tree Int)
selfGEC = feedback (arr (balancedTree o toList) >>>
    edit "Self Balancing Tree"

The counter and function examples below are also conveniently, and concisely, expressed using arr and >>>.

myEditor = startCircuit selfGEC (0,Neutral)
selfGEC :: GecCircuit Counter Counter
selfGEC = feedback (arr updCntr >>> edit "Counter")

myEditor = startCircuit selfGEC (toGEC funTest)

selfGEC = arr (toGEC o updFun o fromGEC) >>>
         edit "test operator"

This completes the arrow combinator versions of the examples of Sect. 2.3.

As a somewhat larger, and more tantalizing, example we show the basic
structure of a GEC for GECs below. We use quite complex GECs that allow the
user to edit the type and visual appearance of another GEC. These editors are
not shown because we want to emphasize on GEC circuits here, not the internal
workings of the editors themselves. The information flow between these editors
can, again, nicely be expressed using the arrow combinators.

Both the editor for designing an GEC, as well as the editor that displays,
and allows the designer to interact with, the designed GEC use an well-formed
feedback loop. Auxiliary conversion, and state carrying, functions are lifted using
the arr combinator. Both editors are combined (without feedback) using the >>>
combinator.

editorEditor = startCircuit (designEditor >>>
           arr convert >>>
           applicationEditor)initvalue

designEditor :: GecCircuit DesignEditor DesignEditor
designEditor = feedback {
    toDesignEditor >>>
    edit "design" >>>
    arr (updateDesign o fromDesignEditor)}

applicationEditor :: GecCircuit ApplicationEditor
applicationEditor = feedback {
    arr (toApplicEditor o updateApplication) >>>
    edit "application" >>>
    arr fromApplicEditor )

4 Related Work

In [14] John Hughes introduces arrows as a structuring tool for combinator li-
braries that is more general than using monads [19] for combinator libraries. Examples
of the use of arrows are given for various application areas such as parsers,
interpreters, stream processors, and CGI programming. The stream processors
are basically the same as those presented in the Fudgets library [8].

Other authors have applied the arrow concept for compositional program-
ning of Functional Reactive Programs (the Yampa system [13] for mobile robots),
and, again, GUI programming (the Fruit system [10]). In both cases arrows compose signal transformers, which are functions that transform one continuous time-varying value of some type a to another of some type b. Both systems handle discrete events, by modeling event streams as continuous time-varying Maybe values.

The system of combining GECs with arrows, as proposed in this paper, bears the most resemblance with the above mentioned Fudgets. Both systems are collections of event-driven components that can trigger events autonomously, and synchronize with each other (using streams with Fudgets, and arrow combinators with GECs). However, the main difference is that in our system, the programmer does not have to use arrow combinators. For more complex synchronization behaviour the programmer can always use the callback mechanism of GECs.

Finally, the GEC system differs from all of these systems in its level of abstraction from GUI programming. The programmer concentrates on the model (this includes the model of GUI) instead of working with GUI elements such as buttons, counters, windows, and so on, that can only be connected with a restricted set of operations. Moreover, GEC code can be mixed with Object I/O code, as explained in Sect. 2.2 and 3.3. To our knowledge there is no other functional system for describing general purpose GUIs that achieves the same level of abstraction with such a complete separation of model and GUI without loss of flexibility because it is integrated seamlessly with Object I/O.

5 Conclusions

In this paper we have presented the Graphical Editor programming toolkit for constructing and composing GUI components on a high level of abstraction and in a fully compositional way. The programmer does not construct GUI components in the 'traditional' way by managing widget-like entities, but instead concentrates on the data model of his application. The system automatically derives the intended GUI from concrete values of this data model, using generic programming techniques. Therefore, programming GUI components is as easy and as compositional as programming functional data structures. We have founded a library of GUI component combinators on arrow combinators. This facilitates the composition of components.

As a result, we have obtained a system that has three distinctive features. The programmer constructs arbitrarily large GUI components using (abstract) GECs. These components can be glued together using the arrow combinator library. (We expect that these circuits tend to be small when compared to the size and complexity of the components.) The programmer can still use Object I/O code where needed without effort.

We have shown how to prove the corresponding arrow laws for our system. It turns out that these proofs can be carried out using standard reasoning techniques for functional programs. In particular, we do not have to resort to some underlying semantic model for GECs.
References


Appendix

arr id >>> 1
= /* unfolding arr, moving def. of k to top-level */
(GecCircuit k) >>> 1
  where
  k (setb,geta,env) = (seta,getb,env)
    where
    getb env
    = /* unfolding id (two occurrences) */
      (GecCircuit k) >>> 1
  where
    k (setb,geta,env) = (seta,getb,env)
      where
      getb env
      = /* by unfolding the \( \lambda \) inside the definition of getb */
        (GecCircuit k) >>> 1
      where
        k (setb,geta,env) = (seta,getb,env)
          where
          getb env = geta env
          seta u a env = setb u a env
    seta u a env = setb u a env
  = /* extensionality */
(GecCircuit k) >>> 1
  where
  k (setb,geta,env) = (seta,getb,env)
    where
    getb = geta
    seta = setb
= /* unfolding getb and seta */
(GecCircuit k) >>= l
where
k (setb,geta,env) = (setb,geta,env)
= /* assuming l = GecCircuit lk (definedness) */
(GecCircuit k) >>= GecCircuit lk
where
k (setb,geta,env) = (setb,geta,env)
= /* unfolding >>= */
GecCircuit k'
where
k' (setc,geta,env) = (seta,getc,env2)
where
(seta,getb,env1) = k (setb,geta,env)
(setb,getc,env2) = lk (setc,getb,env1)
where
k (setb,geta,env) = (setb,geta,env)
= /* unfolding k */
GecCircuit k'
where
k' (setc,geta,env) = (seta,getc,env2)
where
(seta,getb,env1) = (setb,geta,env)
(setb,getc,env2) = lk (setc,getb,env1)
= /* injectivity of the first tuple definition */
GecCircuit k'
where
k' (setc,geta,env) = (seta,getc,env2)
where
seta = setb
getb = geta
env1 = env
(setb,getc,env2) = lk (setc,getb,env1)
= /* unfolding seta, getb and env1 */
GecCircuit k'
where
k' (setc,geta,env) = (setb,getc,env2)
where
(setb,getc,env2) = lk (setc,geta,env)
= /* unfolding the inner where definition */
GecCircuit k'
where
k' (setc,geta,env) = lk (setc,geta,env)
= /* extensionality */
GecCircuit lk
= /* by definition */
1