Automatic Graphical User Interface Form Generation Using Template Haskell

Gracjan Polak[⋆], Janusz Jarosz[⋆⋆]

Department of Computer Science
University of Science and Technology in Kraków

Abstract

This paper presents *AutoGUI*, a *Template Haskell* library for automatic form generation. A form is a part of graphical user interface (GUI) restricted to displaying a value and allowing the user to modify it and then either accept changes or abandon them. The library is built on top of medium-level GUI library *wxHaskell*. The *Template Haskell* and Haskell type system allow the forms to be built fully automatically, but manual intervention is not excluded.

1 INTRODUCTION

In many programs, some of the graphical user interface parts can be considered *forms*: they show a set of values and allow the user to update them. For example the dialogs *Settings*, *Properties* or *Info* are most likely forms in many applications. In our meaning of this word, forms do not only present values to the user, they also allow modification. Once required changes are made, the user can accept new data or refute it. In any way this is treated as an atomic action; no intermediate data is ever seen. Forms are fairly common GUI (graphics user interface) elements and provide clean and intuitive way of data presentation and input.

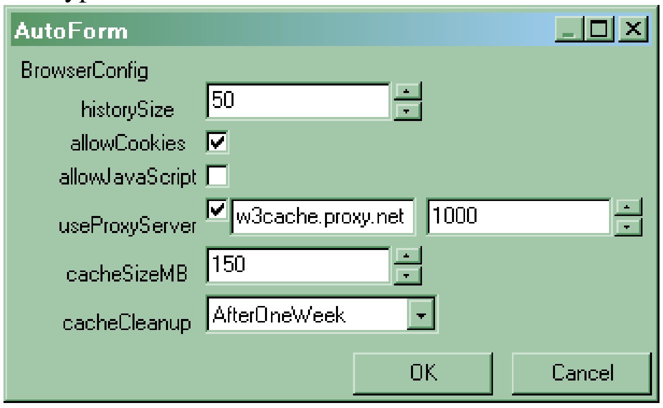
In this paper we will use the definition of a *form* from [1]. A *form* is a GUI part, residing somewhere within a dialog with *OK* and *Cancel* buttons, which is only able to display and alter a certain value. When the dialog appears, the form has an *initial value* which is provided by its environment; subsequently the user can read and alter this value; at the end, the user closes the dialog with one of the buttons, and the form passes the *final value* back to the environment.

Manual form creation is inherently very tedious work. A large amount of code is written just to convert values between internal and screen representations. A lot of boilerplate code is spent on appearance problems like positioning, layout and shape. The structure of the data is typically quite hard to reflect well in presentation. Of course this presents application maintaince problems in the event of data type changes. Programs typically contain many dialogue forms so easing this part of development can bring serious benefits.

[⋆]gpolak@agh.edu.pl
[⋆⋆]jjarosz@agh.edu.pl

As research conducted by Adobe shows, GUI boilerplate code can amount up to 30% of total code in serious, enterprise level applications. Optimizing programming effort in this area can have a large impact on overall project cost reduction. This is the main motivation of many libraries and frameworks, including our work presented here.

As an example, the following screenshot presents the configuration dialog of an archetypical Internet browser:



AutoForm

BrowserConfig

historySize 50

allowCookies ✓

allowJavaScript ☐

useProxyServer ✓ w3cache.proxy.net 1000

cacheSizeMB 150

cacheCleanup AfterOneWeek

The above dialogue box was fully automatically generated using *AutoGUI*. All the user has to do is to declare the type of configuration data and issue the *Template Haskell instanceAutoForm* command to generate all appropriate classes and instances.

$(*instanceAutoForm* ''*BrowserConfig*)

This paper describes *AutoGUI*, an automatic GUI generation library. It is built on top of wxHaskell[4] graphics interface library, but general ideas presented could be equally well used in another setting.

We take special care to use as much information as available in Haskell source code, including semantic part, which is not always written explicitly, but can be inferred from usage context. This happens when polymorphic data structures come into play.

What follows is a step by step introduction of concepts present in the language with its mapping to screen controls. First, we show the basic organization of the library, then go on with atomic controls, simple structural types like tuples, optionals and enumerations, lastly we show how to tackle user defined disjoint unions and record types. Then implementation details are presented. We end with related work, conclusions and future directions.

2 REPRESENTATION DETAILS

The functionality in *AutoGUI* library resolves around one type class *AutoFormClass*. The types declared as instances of this class are expected to have on screen representation, present values to the user and take interactive input. When (and if) the user accepts changes, a new value is read and returned as a dialogue result.

The main *AutoGUI* type class is defined as follows:

```
class AutoFormClass a
  where
    autoForm ::Maybe a - >        -- initial value
               Window b - >       -- parent window
               IO (AutoForm a)    -- result form
```

Instances of *AutoFormClass* are all those data types that have screen representation and can be used in forms. The function *autoForm* can be hand written or autogenerated. Many basic types are custom instances of *AutoFormClass*. This functionality is internal to the library and is already implemented in the package.

What is *AutoFrom a*? It is a parametrized data type that contains a screen representation of values of type *a*. It is a record type; its fields have the information necessary to compose controls on screen to form larger structures and to read value entered by the user.

Basically, the record is declared as:

```
data AutoForm a =AutoForm
                  { af_layout  ::Layout
                  , af_valueio ::IO a
                  , af_vertical ::Bool
                  }
```

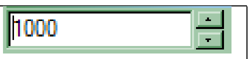
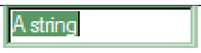
The meaning of the fields is as follows:

- The *af_layout* field contains a reference to *wxWindows Layout* data, which is used to compose controls in window.

- The *af_vertical* boolean flag indicates when it is better to lay out them vertically or horizontally. The framework decides using its value (among other things) if controls should be stacked below each other or make a row. This flag is aesthetically meaningful.

- The most important field is *af_valueio*. This is the IO action, that reads form value from controls when the user accepts changes. In case of basic controls, it just reads their values and converts them to appropriate types as necessary. Composite types need more treatment; in such case values are read from subparts and then composed into the final value. This mechanism is treated in the latter part of this paper.

TABLE 1. Basic types

Integer	 1000
String	 A string
Bool	

2.1 Primitive values

Many of the data types defined in Haskell should be treated atomically by the *AutoGUI* framework. This includes not only basic types like *Integer*, *Double*, *Bool*. Some structures, like *String* for example, are also treated atomically by *AutoGUI* library. Those are represented as native controls in graphical interface. The wxHaskell library has text input field for *String* values, spin control for *Int*, check box for *Bool*. Table 1 has screen shots of these controls. Of course, more can be added by deriving *AutoFormClass* and implementing *autoForm* function.

Direct mapping from a data type to a screen control does not require *Template Haskell* facilities: standard Haskell type classes are enough[1].

2.2 Optional values (*Maybe*)

The first extension problem we tackle is optional input control. Optional values in Haskell are modeled with *Maybe a* data type where *a* is the type parameter of value that may be present or absent. A *Just a* models a present value, otherwise *Nothing*.

$$data\ Maybe\ a \ = \ Just\ a \mid Nothing$$

Certain values are meaningful only when some feature is enabled. When such a situation happens, the *Maybe* data type is a very natural model with *Nothing* representing the disabled state and *Just a* when feature is enabled and has specified parameters. The representation of an optional value is quite obvious: a check box with a set of additional controls. But there are two behavioral possibilities:

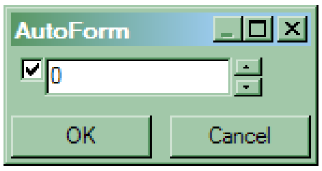
- When the check box is disabled, controls are put into the disabled state, typically grayed out.
- When the check box is disabled, controls are hidden.

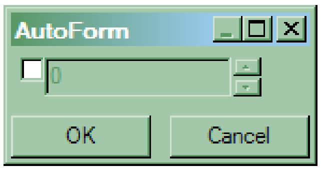
In any case, input in the control is allowed only when the check box is checked. We have chosen the second way as we thought it is more clear to the user. ”Disappearing controls” can confuse users as they are probably going to look for them somewhere else. Contrary, disabled controls paired with check boxes clearly indicate how features are enabled.

[1]Haskell98 is not enough, undecidable instances are required. As we already use the Glasgow Haskell specific extension, we do not think this is any serious problem.

Consider the following code and its representation in two states: *enabled* and *disabled*:

$$maybeInt :: Maybe\ Int$$
$$maybeInt = Nothing$$





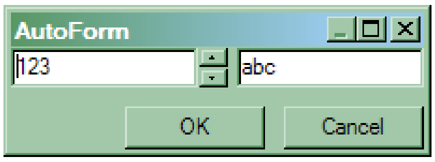
Optional values seem to be a quite common construct. Clean presentation is important and we claim that we have achieved this goal.

2.3 Tuples

Tuples are aggregate data types that consist of many values. Tuple elements are identified by their position in tuple definition. The simplest example is a pair, that has two components. The number of elements in a tuple is unlimited [2] and, unlike in the case of lists, it is reflected in the type of a tuple. Elements can have different types, independent of types of other elements. In Haskell tuples are written inside parentheses with elements separated with commas. Almost the same syntax applies to tuple values and tuple types.

Consider the following example:

$$intString :: (Int, String)$$
$$intString = (123, "abc")$$



Tuple represents a row of values, here *Integer* and a *String*. There is no relation between them, any integer can be paired with any string. Editing such a value in form means editing each of its components independently. *AutoGUI* simply composes together edit controls for each value.

There are some presentation issues involved here. For simple values the most natural composition is linear, side by side in a row. If any of fields are more complicated, all controls would be stacked above each other and form a vertical column. This is the point where *af_vertical* field of *AutoForm* record comes into play.

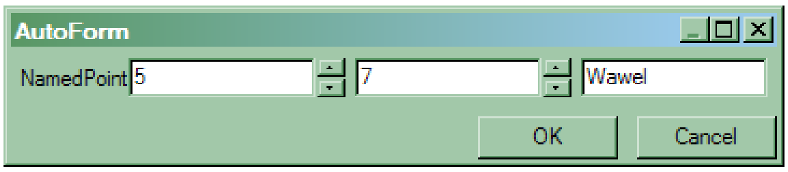
2.4 Product types

Product types, from the perspective of *AutoGUI* library, are very similar to tuples. They are composed of none, one or many components. The important feature

[2] Actually, there is a practical limit. As of now for GHC it is 66.

is that they are named, so the *AutoGUI* library can make use of this additional information. See this simple example:

```
data NamedPoint = NamedPoint Int Int String
namedPoint = NamedPoint 5 7 "Wawel"
```



NamedPoint 5 7 Wawel

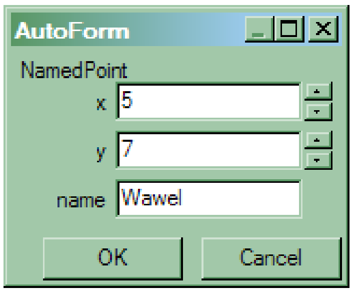
For screen presentation purposes the product type is treated in the same way as a tuple of the same arity, but with a name. If all fields are of horizontal kind, the whole type is represented as a row of controls with the name of the type in front of them. Otherwise, a vertical column is created. Such a representation leans itself to extension for disjoint union types, which we cover later, in Section 2.6.

2.5 Records

Tuples and product types have one drawback: their compounds are unnamed. When there are not too many values, their meaning can be obvious from context. For other cases, Haskell provides a record syntax. This is in many respects equivalent to products, except that fields are named. The *AutoGUI* tries to use this additional information and presents field names to the user. Consider the following example:

```
data NamedPoint =NamedPoint
                  { x :: Int
                  , y :: Int
                  , name :: String
                  }


namedPoint =NamedPoint
              { x = 5
              , y = 7,
              name = "Wawel"
              }
```



NamedPoint
x 5
y 7
name Wawel

From Haskell's point of view this does not present any significant advantages, but *AutoGUI* could generate a much more readable form.

2.6 Disjoint union types

The disjoint union types representation is the main achievement of this library. We built on the knowledge from [1], but choose a bit different path.

A variable of disjoint union type can have one of many enumerated values. Here we have to choose one of two possible representations:

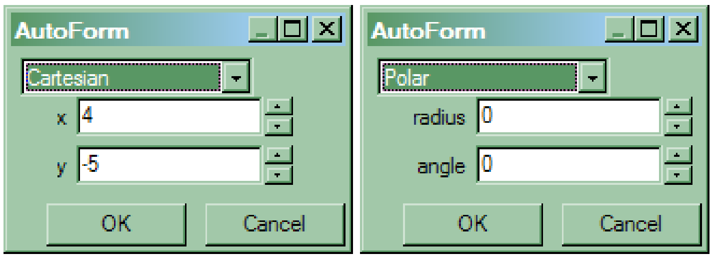
- As a group of radio buttons. This is the way chosen by authors of Disjoint Forms paper.
- As a drop down list of options.

We argue for the second representation. There are some reasons for this. First of all, this is compatible with product type representation where we have only one constructor. In such case, only the static text control containing only the constructor name is created on front of the list of controls representing parameters. When there are more constructors, the static text control is changed to a drop down list control.

Parameters to disjoint union data type constructors are used as alternative views. At any point in time, only fields of selected constructor are visible. Depending on the current selection only relevant configuration options are shown to the user, all others are stacked under them and hidden. Important from the usability point of view is that such a presentation spares screen real estate and allows users to concentrate on important input fields.

In the below example, we see two alternative representations of complex numbers:

```
data ComplexNumber = Cartesian { x :: Int, y :: Int }
                   | Polar { radius :: Int, angle :: Int }
```



Cartesian
x 4
y -5

Polar
radius 0
angle 0

The drop down list is responsible for constructor selection. In the above example the user can choose one of the two available complex number representations. In any case she is presented only with the controls relevant to the selected constructor. Such layered design reduces screen clutter and adds to clean human interaction.

3 IMPLEMENTATION DETAILS

The goal of the library was to use as much information present in types as possible and require minimal manual intervention. Generated forms are allowed to look a bit rough but their behavior should be pleasant enough for most users and considered relevant for prototyping purposes. The logic of the forms should directly follow from the structure of the types fo data for which the dialogue is constructed. All user visible texts should be directly derived from names of types, data and fields.

To achieve our goals, first we turned to data type generic programming as defined in papers [9] and [6]. The recursive definitions of folds defined over types represented data structures very well. Therefore we could create forms, but only for data presentation, because changes would require data creation and information about data construction could not be obtained from data type generics. But even if this would be possible, there is still one missing point: neither Haskell nor data type generics provide enough information to know about type names, data constructor names and record field names, which are essential for fully automatic form generation. Introspection facilities are just not there yet.

Finally we decided to use *Template Haskell* as, although being a bit heavy, it provides all the things we needed to complete our project.

The simple data types were implemented manually as instances of *AutoFormClass*. Our library provides an implementation for *Bool*, *Int* and *String*, but nothing prevents users from creating instances for their own types by hand. For example some applications may treat color picker control as one of the basic ones. If there is a proper instance of *AutoFormClass* declared then such control could be embedded in autogenerated forms without any problems.

The *Maybe* data type is also implemented manually, as it needed a special treatment in screen representation. Instances for all other types are autogenerated.

3.1 Tuples

The *AutoFormClass* instances for tuples are autogenerated. We create instances for tuples that depend on all elements' types to be also instances of *AutoFormClass*. In the pseudo-Haskell code this could be written as:

```
instance(AutoFormClass a1,...,AutoFormClass an)
        => AutoFormClass (a1,...,an)
  where
    autoForm = ... ⟨autogenerated⟩
```

All the users have to do is to issue *Template Haskell* command to create an appropriate instance for *n*-arity tuple:

```
$(instanceAutoFormTuple n)
```

The *AutoGUI*library comes with pregenerated *AutoFormClass* instances for tuples with up to 10 elements. If needed, more instances can be generated in the user code.

While implementing the instances for tuples we found the Idiom [7] idiom very useful. The screen representation of each element of the tuple must be generated in the IO monad, but otherwise those elements are independent of each other.

3.2 Enumerations, disjoint unions, products and records

The most interesting part of our library is that the autogenerated instances of *AutoFormClass* for all user types falling into any of the categories from the title of this section are created using one *Template Haskell* invocation. This single macro is able to create forms for enumerations, products, records and disjoint unions, provided that the types of all the used fields are also instances of *AutoFormClass*. See the following code fragment:

```
data UserData = ⟨constructors⟩
$(instanceAutoForm ''UserData)
```

generates *AutoFormClass* instance:

```
instance AutoFormClass UserData
  where
    autoForm = ⟨autogenerated⟩
```

The introspection facilities offered by *Template Haskell* are put to good work here. The *AutoFormClass* instance creation algorithm is quite obvious and easy. It iterates over all constructors, for each one constructs appropriate form for it's parameters or fields in case of record constructor is generated. Those forms are stacked on top of each other so that, at any given time, only one is visible. The constructor is selected with a drop down list control; when there is only one constructor, the drop down list is changed into a static text control, as there is no point in the drop down with only one option.

4 RELATED WORK

Simplifying form creation has been tackled by many researchers and developers with varying degrees of success.

Most of graphical database front ends provide means to create default record views. This is accomplished with information available in database, which includes field name, field type and some other information dependent on a specific database engine used, relations with other fields, records and tables.

Ruby on Rails [2] is one of such frameworks. Based on database description, the Rails generate default view, which is a HTML form. Although it does lack on the artistical side, all of expected functionality is already in place. These forms present database content to the user, but not only. Typical operations like record addition, deletion or update are available. Database integrity is automatically ensured. Table navigation presented as HTML links is generated whenever possible. Default (autogenerated) forms are very helpful in an early development stage but with the wonders of *cascading style sheets* technology they could fulfill the needs of a final product. Possibilities are bound by SQL database and the amount of information available in dynamically typed Ruby language.

As already noted a GUI code can amount up to 30% of total code in serious applications. This is a main motivation for Adobe Open Source Library (ASL), which, in the concept of its authors, should simplify GUI form creation. ASL has much more features than *AutoGUI*, but it has its additional costs. Forms are specified in two languages, one for presentation, second for logic and interaction, called Eve and Adam, respectively. Form is populated and after user acceptance values are read in an untyped fashion with the help of an intermediate name-value map. Showing such a dialogue requires a specialised interpreter. For many programs this is overkill.

On the academic side of problem we would like to mention Disjoint Forms[1] as an important step in the direction of simplifing the form creation. The authors of this paper managed to decouple presentation from content at a very high level. A vast amount of boilerplate code is removed. All that remains is mapping from values to controls, which itself is still very tedious and Disjoint Forms leave this problem unsolved. This was the main motivation for our work.

5 CONCLUSION AND FUTURE DIRECTIONS

We consider the *AutoGUI* library as a serious step forward to make Haskell graphical user interfaces more programmer friendly. Although generated forms look a bit rough, they are very usable. Form follows a rigid specification and this guarantees coherence between the internal data and a screen representation. Their expected use is mostly in the application prototyping, but we think that many final programs have less user friendly forms in the production releases.

Our main contribution is a mapping from the Haskell type space to GUI controls using as much information as possible.

Future directions of our research include creating controls for set like types, for example *Data.List*, *Data.Set* or *Data.Map*. The natural representation for them seems to be a list control. As there are many unanswered questions in this area as of the current version of *AutoGUI* library does not support automatic form generation for such types.

Another possible extension direction is support for recursive data structures. The current *AutoGUI* engine generates all controls in the first place, then fills them with values. This works only for data structures with bounded depth. In the other case the library would try to create an infinite tree of windows and this of course fails. A possible extension of this library is to create windows only on demand. Research paths here remain open.

6 ACKNOWLEDGMENTS

We would like to thank Wlodzimierz Funika for encouragement and the editorial work. Also the Haskell's community was very helpful in solving many of our problems. Thank you for answering our emails!

REFERENCES

[1] S. Evers, P.M. Achten, and M.J. Plasmeijer. Disjoint forms in graphical user interfaces. In Loidl [5].

[2] David Heinemeier Hansson. Ruby on rails.

[3] Simon Peyton Jones, editor. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, December 2002.

[4] Daan Leijen. wxhaskell, a portable and concise gui library for haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, September 2004.

[5] H.W. Loidl, editor. *Proceedings of the Fifth Symposium on Trends in Functional Programming (TFP 2004)*. Ludwig Maximilians Universität München, 2005.

[6] Ian Lynagh. Template haskell: A report from the field. May 2003.

[7] Conor McBride and Ross Paterson. Functional pearl. applicative programming with effects.

[8] T. Sheard and S. Jones. *Template meta-programming for haskell.* 2002.

[9] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.