Containers for GUI Models

Knut Anders Stokke
University of Bergen
Bergen, Norway
knut.stokke@uib.no

Mikhail Barash
University of Bergen
Bergen, Norway
mikhail.barash@uib.no

Jaakko Järvi
University of Turku
Turku, Finland
jaakko.jarvi@utu.fi

Elisabeth Stenholm
University of Bergen
Bergen, Norway
elisabeth.stenholm@uib.no

Håkon Robbestad Gylterud
University of Bergen
Bergen, Norway
hakon.gylterud@uib.no

ABSTRACT

We present an ongoing work towards a programming approach for Graphical User Interfaces (GUIs), where structural operations on GUI data structures (such as inserting, removing, or reorganizing) can be declaratively specified and their implementations automatically generated. Concretely, we investigate how the type-theoretical notion of containers, an abstract formalism for specifying data structures, can be used for defining manipulatable GUI structures that have (multi-way) dependencies between their elements.

CCS CONCEPTS

• **Theory of computation** → **Type theory**; • **Software and its engineering** → *Graphical user interface languages.*

KEYWORDS

Containers, GUI specification, GUI structures, reuse

ACM Reference Format:
Knut Anders Stokke, Mikhail Barash, Jaakko Järvi, Elisabeth Stenholm, and Håkon Robbestad Gylterud. 2024. Containers for GUI Models. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming (‹Programming›Companion '24), March 11–15, 2024, Lund, Sweden.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3660829.3660830

1 INTRODUCTION

Graphical User Interfaces (GUIs) let users observe and edit variables in (possibly complicated) data structures. Users can interact with these variables through widgets (such as input text fields, checkboxes and sliders), but they can also manipulate the data structures through *structural operations*, such as inserting, removing and reorganising widgets [9]. Implementing GUIs that allow for such operations can be difficult, in particular when variables in the data structures are somehow *related* to one other, such that interactions with one widget may cause changes in others—structural operations do not only add or remove variables, they also modify the relations between them.

We describe an ongoing work towards a GUI programming approach where *structural operations can be declaratively specified and their implementation automatically generated.* The specification contains both the data structure underlying the GUI and the relationships between the variables within this structure. The goal is that the generated code will keep the critical relationships between GUI variables up-to-date, even as the structure is changed. We build on two lines of our past work, applying new theoretical tools to tackle new challenges. First, we rely on dataflow constraint systems for maintaining relations between GUI variables [4, 5]; they provide us with a systematic way of expressing relations between variables. Importantly, they can concisely describe *multi-way* dependencies, where the direction of the dataflow between variables depends on in which order users interacts with different widgets. This is in contrast to contemporary GUI programming, where relations between GUI variables are often maintained in *ad hoc* manner (e.g., by code in widgets' event handlers).

Second, we take as a starting point our prior work on a DSL for specifying GUI structures [7–9], and investigate a new conceptual basis for it: we turn to the type theoretical notion of *containers* [1] for defining GUI structures, reusable structural operations for them, and (multi-way) constraints between the elements of the structures.

Containers are an abstract formalism for specifying a large class of data structures, including algebraic data types [2]. In particular, all the usual data structures used in GUIs [8] can be represented by containers. The theory of containers includes operations for combining containers, which represent building GUIs from smaller components, and for mapping data from one container to another, which can be used to represent structural operations. Furthermore, containers abstract over "physical" implementation details of data structures (e.g., whether grids are lists of rows or lists of columns). Containers are therefore promising as a basis for a natural and convenient framework for representing GUIs.

2 CONTAINERS

To understand how we apply containers to GUI programming, we first give a brief introduction to containers in general.

A container consists of a (possibly infinite) set of *shapes*, and for each shape, a set of *positions* in which values can be stored. We use $S \triangleleft P$ to denote a container where S is the set of shapes and P is a family of positions, indexed by S, i.e., for each shape $s \in S$, the set $P(s)$ specifies the positions of s which can be filled with data [1]. One often uses functional notation for P, such as λ-abstraction and pattern-matching, to specify the positions in a given shape.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
‹Programming›Companion '24, March 11–15, 2024, Lund, Sweden
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0634-9/24/03
https://doi.org/10.1145/3660829.3660830

As an example of a container, consider the list data structure. The shape of a list is its length, and for a list of length n, there are n positions to place data in the list. The container representing the list data structure is therefore $\mathsf{List} := \mathbb{N} \lhd \mathsf{Fin}$, where $\mathbb{N}$ denotes the natural numbers and the family Fin denotes a canonical set of n elements, i.e.: $\mathsf{Fin}\, n = \{0, \ldots, n-1\}$.

Another basic container is $\mathsf{Id} := \mathsf{Fin}\, 1 \lhd \lambda x.\, \mathsf{Fin}\, 1$, which consists of one shape with a single position. This represents the data structure with a single field of a generic type.

There is also a container for optional values. It has two shapes: the first one has exactly one position for placing data, and the second one has no positions for placing data. This data structure is thus represented by the container $\mathsf{Fin}\, 2 \lhd \{0 \mapsto \mathsf{Fin}\, 1; 1 \mapsto \mathsf{Fin}\, 0\}$.

One can similarly define containers for grids (two-dimensional lists), different variations of trees, and other data structures commonly used in GUIs.

A container $S \lhd P$ is a specification of a data structure and does not store data itself. An *instance* of $S \lhd P$ for a data type X, denoted $[\![S \lhd P]\!]\, X$, is a data structure specified by $S \lhd P$ and filled with values from X. For instance, $[\![\mathsf{List}]\!]\, \mathsf{Int}$ is the type of lists of integers. Specifically, an instance of a container is (i) one of the container's shapes, together with (ii) a lookup function that maps every position in the shape to the value that is stored at that position. To get the value at a position, one must provide the position as argument to the lookup function.

3 CONTAINERS FOR GUIS

The correspondence between GUI models and containers can be outlined as follows: the data structure of the GUI model is represented by a container, GUI variables are represented by positions in the container, and relationships between GUI variables are expressed as extra structure on the underlying container. A GUI model consisting of several components is constructed from the containers of each component using well-known [2] operations for combining containers, such as products and compositions.

Simple combinations of components in the GUI are represented by *products* of containers. For example, consider a GUI model consisting of two components, represented by the containers $S \lhd P$ and $T \lhd Q$, respectively. These two must be combined into one container, underlying the complete GUI model. The shapes of this larger container are pairs of shapes from the first and the second container, and the set of positions is the disjoint union of the positions of each container with the given shape. This product operation on containers is denoted by $\_ \times \_$ and defined as:

$$(S \lhd P) \times (T \lhd Q) := (S \times T) \lhd (\lambda(s,t) \to P(s) + Q(t))$$

A GUI model consisting of components placed inside a larger component is represented by composition of containers. Natural examples are tables for tabular data where each cell of the table is an individual GUI component. This construction assumes two containers: the outer structure $S \lhd P$ and the inner structure $T \lhd Q$. The composition, denoted $(S \lhd P) \circ (T \lhd Q)$, is then a large container which has instances of the inner structure placed within the positions of the outer structure. The shapes of this large container are pairs of a shape of the outer component and for each position in this shape a shape of the inner component. The positions are then the disjoint union of the positions of all the inner components. Formally, the definition of composition is:

$$(S \lhd P) \circ (T \lhd Q) := \sum_{s \in S} (P(s) \to T) \lhd \left(\lambda(s, v) \to \sum_{p \in P(s)} Q(v(p)) \right)$$

We have also found natural interpretations for other operators for combining containers [2].

Structural operations on GUIs, such as inserting, deleting or reordering values, can be specified using linear container morphisms, a class of functions between containers which gives control over insertion and deletion, and prohibits data duplication. For example, an operation which inserts a single element into $S \lhd P$ can be described as a linear container morphism from $(S \lhd P) \times \mathsf{Id}$ to $S \lhd P$. Dually, deletion of a single element is described by a linear morphism in the other direction: namely, from $S \lhd P$ to $(S \lhd P) \times \mathsf{Id}$.

When a GUI specification is constructed by composition of several components, these structural operations can be specified for each of the individual components and then *lifted* to the larger structure, providing reusability [8].

Containers also enable us to define dynamic constraint systems which depend on the shape of a GUI's model. A constraint in a multi-way dataflow constraint system [4, 5] represents a relation over GUI variables and has a set of *satisfaction methods*: procedures that enforce the relation. Each method specifies which of the variables in the constraint are *inputs* and *outputs*, and the procedure enforces the relation by computing values for the output variables from values of the input variables. The constraint system solver uses the inputs and outputs of the methods to *plan* [4] a dataflow, that is, picking *one* method from every constraint.

We represent a method procedure as a function from one container instance to another; the positions of the input and output containers are, respectively, the input variables and output variables of the method. A constraint is a function that for every shape in the GUI model computes the set of positions that are related, together with a set of satisfaction methods on these positions.

4 CONTRIBUTIONS

We explore a connection between two seemingly unrelated areas of programming, graphical user interfaces and type theory, and provide a basis for formalising aspects of GUI programming, which may eventually improve how GUIs are programmed. In particular, container operations enable using containers as core building blocks for GUI models, and structural operations in GUIs can be expressed as linear container morphisms. GUI models with explicit shapes and positions enable specifications of dynamic constraint systems [7].

We are currently formalising these concepts using the proof assistant *Agda* [6]. Using the formalisation, we plan to develop an (extensible) set of base structures with structural operations and patterns for relating GUI variables. A future goal is to make this way of building GUIs easily accessible to programmers (possibly unfamiliar with dependent types) through a library or a DSL.

The (simplified) definition of containers above only supports GUI variables of the same data type. In our Agda formalisation, containers are specified over a *family of types* [3]—this is necessary for GUIs where variables can be of different data types.

Future work also includes connecting these specifications to GUI views, so that models and views are synchronously updated [7].

REFERENCES

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2003. Categories of Containers. In *Foundations of Software Science and Computation Structures*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–38.

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theoretical Computer Science* 342, 1 (2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002 Applied Semantics: Selected Topics.

[3] Thorsten Altenkirch and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* 25 (2015), e5. https://doi.org/10.1017/S095679681500009X

[4] John Freeman, Jaakko Järvi, and Gabriel Foust. 2012. HotDrink: a library for web user interfaces. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE '12)*. Association for Computing Machinery, New York, NY, USA, 80–83. https://doi.org/10.1145/2371401.2371413

[5] Magne Haveraaen and Jaakko Järvi. 2021. Semantics of multiway dataflow constraint systems. *Journal of Logical and Algebraic Methods in Programming* 121 (2021), 100634. https://doi.org/10.1016/j.jlamp.2020.100634

[6] Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of the 4th international workshop on Types in language design and implementation*. 1–2.

[7] Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. 2023. A domain-specific language for structure manipulation in constraint system-based GUIs. *Journal of Computer Languages* 74 (Jan. 2023), 101175. https://doi.org/10.1016/j.cola.2022.101175

[8] Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. 2023. Towards Reusable GUI Structures. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2023, Cascais, Portugal, October 22-27, 2023*, Vasco Thudichum Vasconcelos (Ed.). ACM, 68–69. https://doi.org/10.1145/3618305.3623611

[9] Knut Anders Stokke, Mikhail Barash, and Jaakko Järvi. 2023. The Ultimate GUI Framework: Are We There Yet?. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASIcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:9. https://doi.org/10.4230/OASICS.EVCS.2023.25

Received 2024-02-08; accepted 2024-02-20