Structuring GUIs using Containers

Knut Anders Stokke [1] Mikhail Barash [1] Jaakko Järvi [2] Elisabeth Stenholm [1] HØakon Robbestad Gylterud [1]
Yan Passeniouk [2]

[1]University of Bergen [2]University of Turku

UNIVERSITY OF TURKU

Introduction

Frameworks for graphical user interfaces (GUIs) are plagued by issues:

- Constant iteration on which framework is currently favoured
- An ad-hoc approach to their construction
- Brittleness with respect to complicated data flows and relationships
- Few guarantees of correctness beyond extensive testing
- Reusability of widgets is either disregarded or severely restrictive

We propose a formal account of GUIs, including a mechanically verified implementation, to further the theoretical understanding of GUIs and to improve their reliability and ease of design by combining two seemingly disparate topics in the literature: Multiway Dataflow Constraint Systems and Containers.

Theoretical Underpinnings

We consider GUIs through the lens of:

Model-View-Update

The *model* describes the functionality of the GUI and its widgets, which is displayed to the user through the *view*. The user interacts with the GUI, initiating an *update* to the *model*.
This approach does not provide an explicit method for handling the interdepence of data (e.g. circumference to radius) for which we turn to . . .

Multiway Dataflow Constraint Systems

Is defined as a tuple (V, C) of variables V and constraints C. Each constraint $c \in C$ is equipped with *methods* M where $m \in M \subseteq S \times T$ and $S, T \subseteq V$, which, along with an indexing of the variable set, is used as an input to a constraint solver, producing a plan of changes to V to satisfy the constraints.

By associating the variables to widgets in interfaces, multivariate relationships between them and the data they contain are conserved, with the change being reflected in the view.

The solver assumes a static constraint system (no differences in variables or constraints), while structural operations (updates to the model) require a consistent method of changing the underlying set of variables while maintaining the methods/constraints, which necessitates the use of . . .

Containers

emerged within type theory as a construction to generalise algebraic data types, defined as combination of a shape S (S being an arbitrary set/type) and a function from S into a type family indexed by I:

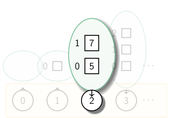
```
record Container (I : Set) : Set₁ where
  constructor _◁_
  field
    Shape : Set
    Position : Shape → I → Set
```

Intuitively, one can imagine S as giving a number of boxes where for each box $s \in S$, $P(s)$ gives the possible types the box accepts, which is suggested by the notation S ◁ P.
Containers form a category, with morphisms $SP \to TQ$ given by a covariant function $S \to T$ on shapes and contravariant function on P, the shape S maps to T, and for each $t : T$, Qt has a corresponding position Ps.

Containers encode a notion of *structure*, which we concretise using the *realisation* into a type X:

$$[\![SP]\!] = \Sigma_{s:S}\ (\lambda\ s\ \to \forall i\ \to P\ s\ i\ \to\ X)$$

As an example, we can realise the List container $\mathbb{N} \lhd \lambda\{n * \mapsto \texttt{Fin}\ n\}$ into a list of integers:



We replace the set of variables in the definition of MSC with the position component of Containers, the shapes giving *structure* of the GUI.

Containers as Structure

Containers can be combined using categorical constructions, which, in the context of GUIs, enable us to insert, remove, copy or reorder the widgets:

Functionality	Property	Operator
Relation/Tuple	Cartesian Product	$SP \times QR$
Choice	Coproduct	$SP + QR$
Reordering	(Consequence of) Derivative	δSP
Copy	Parallel Product	$SP \otimes QR$
Nesting	Composition	$SP \circ QR$

The most important of these being composition, as all other operations can be considered as special cases of it (by giving specific containers which correspond to the definitions of the operators). We define it using $[\![\ \ ]\!]$ and a function $I \to \mathit{ContainerJ}$:

```
_∘c_ :  {I J} → Container I → (I → Container J) → Container J
_∘c_ {I} {J} outer inner = ⟦ outer ⟧ (λ i → Shape (inner i))
  λ (s , innerShapes) j →  I λ i →  (Position outer s i) λ p → Position (inner i) (innerShapes i p) j
```

Operations on containers over a set I are defined by a function from the shape of C to a container over I (encoding the operations on the structure), and morphism from the dependent sum over the aforementioned function, allowing for the encoding of all permitted operations on a GUI. Operations are lifted to ordered operations (by an isomorphism with a finite type), which forms the basis of the constraint solver.

Mechanisation and Implementation

Using the dependently typed language/proof assistant Agda, we guarantee the soundness of our approach by providing proofs of all structural operations and theoretical claims, and allows for type-level encoding of invariants (equality, subset).

Constraint systems are implemented using record types encoding the data outlined in their definition and parametrized over the :

- A Method type holds Input and Output sets, along with data witnessing their ordering and subset relation (with the set of variables)
- Constraints are specified (ConstraintSpec) by a list of these methods and the ordering of all variables, along with witnesses of the non-emptiness of the method set
- And realised into a concrete type with ConstraintInst, giving a variable assignment for the specification.

ConstraintSystem is simply a list of ConstraintInst, to be handed to the solver. The setup also allows for reasoning about constraints without specifying the types, permitting for code reuse.
Views lift the computation into IO and are parametrised by a container (using its realisation as a base) and a GUIToolkit, which is assumed to have some concept of a widget. They specialise into components, which are combined into structures and acted on by actions, which contain the ordered operations mentioned above, and all of which hold a variable ordering.

The main loop waits for user input, adds the corresponding action to a queue, which resolves by applying an ordered operation on the underlying container and the ordering, invoking the constraint system solver on the result of the previous application, and updating the relevant variables in the view.

Most of the theoretical results in our formalisation concern the transferability of container operations into orderings (represented by the finite type), reordering appropriately when elements are added/removed, which in turn allows for the transfer of containers into the types mentioned above.

Conclusion

- A startpoint for a theoretical treatment of GUIs that joins two seemingly disparate fields.
- A prototype along with a number of examples has been implemented using the Haskell backend of Agda.
- All of our theoretical results are mechanised in Agda, yielding a measure of certainty (at the cost of practicality).

Further Work

- A front-end agnostic implementation.
- Exploring how results from the literature can be implemented to improve efficiency and expressivity.
- Automating the proof obligations necessitated by Agda/Providing a DSL specialised for the framework.
- A reconstruction of MDCS semantics using containers, to avoid the use of a separate types for the ordering

References

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, September 2005.

[2] John Freeman, Jaakko Järvi, Wonseok Kim, Mat Marcus, and Sean Parent. Helping programmers help users. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 177–184, New York, NY, USA, October 2011. Association for Computing Machinery.

[3] Magne Haveraaen and Jaakko Järvi. Semantics of multiway dataflow constraint systems. *Journal of Logical and Algebraic Methods in Programming*, 121:100634, June 2021.

[4] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: from incidental algorithms to reusable components. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, page 89–98, New York, NY, USA, 2008. Association for Computing Machinery.

[5] Jaakko Järvi, Gabriel Foust, and Magne Haveraaen. Specializing planners for hierarchical multi-way dataflow constraint systems. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 1–10, New York, NY, USA, September 2014. Association for Computing Machinery.

[6] Rudi Blaha Svartveit. Multithreaded multiway constraint systems with Rust and WebAssembly. Master's thesis, The University of Bergen, August 2021. Accepted: 2021-08-21.

[7] Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 18(1):30–72, January 1996.

cmss