Graphics in Coq for use in ...

Nathan St. Amour and ...

Ohio University, Athens, OH 45701

Abstract. We present a method for modeling graphics in coq that provides an intuitive way to certify and generate graphical programs by using the interactive theorem prover Coq. Our uses a set of graphics primitives that can be used to implement typical functions that a user would expect in a *nix like standard library for graphics. We show proof of concept and ease of extraction from the Coq code to OCaml. Need to add more This all needs to be a lot clearer and more organized maybe talk about the primitives here? There are two components of our implementation that provide it with flexibility and congruence between any instance. talk about what that means before this There is a type class that lives at the highest level for our implementation. The primary function of the type class is to provide functions that use the graphics primitives to affect a state that must be given when providing an instance of the type class. The next key component is the ability to be flexible with an instance of this graphics type class. We have been able to write two instances that hold as a proof of concept for the graphics implementation. The first instance uses finite maps to store the state of what has been drawn to the screen. We have been successful in proving properties about the correctness of drawing lines, particularly in the vertical and horizontal directions, and regular rectangles drawn with lines that are either vertical or horizontal. The corresponding functions for filling rectangles has also been proven. The other instance of the graphics class is an axiomatic way of implementing the graphics in OCaml. This is achieved by exporting the draw_pixel command to the OCaml graphics plot command which is then called by the interp function in our graphics type class to draw to the screen. Probably add some of the abstract to the introduction.

1 Introduction

Creating graphical programs may not be the first thing on the minds of the bold people using a purely functional proof assistant such as Coq. You should be easily convinced that this is the case by the resound lack of any attempt at modeling graphics in Coq. However there are people who have put work into the formalization of geometry in Coq. Talk about the geograbra people and the constructive geometry paper We propose that with some work our graphics implementation

1.1 The Graphics Type Class

```
Section interp.
 Context {T : Type} `{graphics_prims T}.

 Fixpoint interpolate (t : T) (i : nat) (p1 p2 V: point) (num_points : Z) c : T :=
 match i with
  | O → draw_pixel t p1 c
  | 1%nat → draw_pixel t p1 c
  | S i' → let p1' :=
  (((fst p1) + (fst V) ∗ (Z.of_nat i) / num_points),
  ((snd p1) + (snd V) ∗ (Z.of_nat i) / num_points))
  in
  draw_pixel (interpolate t i' p1 p2 V num_points c) p1' c
 end.

 Fixpoint draw_vline (t : T) (p : point) (c : color) (h : nat) : T :=
 match h,p with
  | O, _ → t
  | S h', (x,y) → draw_pixel (draw_vline t p c h') (x,y+(Z.of_nat h')) c
 end.

 Fixpoint draw_hline (t : T) (p : point) (c : color) (w : nat) : T :=
 match w,p with
  | O, _ → t
  | S w', (x,y) → draw_pixel (draw_hline t p c w') (x+(Z.of_nat w'),y) c
 end.

 Definition interp_draw_line (t : T) (c : color) (p1 p2 : point) : T :=
 interpolate t (Z.to_nat (distance p1 p2) − 1) p1 p2 ((fst p2) − (fst p1), (snd p2) − (snd p1) ) (distance p1 p

 Fixpoint interp (t : T) (e : g_com) : T :=
  match e with
   | draw_pix p c → draw_pixel t p c
   | open_graph s_size → update_state t s_size
   | resize_window s_size → update_state t s_size
   | lineto p1 p2 c → let st := interp_draw_line t c p1 p2 in
    let st' := draw_pixel st p1 c in draw_pixel st' p2 c
   | draw_rect (x,y) (w,h) c →
      let w' := Z.to_nat w in
      let h' := Z.to_nat h in
      let t1 := draw_hline t (x,y) c w' in
      let t2 := draw_hline t1 (x,y+h) c w' in
      let t3 := draw_vline t2 (x,y) c h' in
                draw_vline t3 (x+w,y) c h'
   | fill_rect p (w,h) c → fill_rect_rc t p (Z.to_nat w) (Z.to_nat h) c
   | seq g1 g2 → let st := (interp t g1) in (interp st g2)
  end.

 Definition run (e : g_com) : T :=
 interp (init_state tt) e.
End interp.
```

1.2 Subsection2

This is a second subsection.

This is a citation. [1]

This is a chunk of code:

```
Definition f(x : nat) := S x.
Definition g(y : nat) := f y.
```

This is inline code `Fixpoint f(x : nat) := ...` typeset within a line of text.

Para1. This is a paragraph, or subsubsection.

2 Conclusion

References

1. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, pages 465–482. Springer-Verlag, 2010.