A Constructive Method of Generating Proofs in Existential Graphs

Daniel Wojcik

Using the Alpha system of Existential Graphs, it is possible to constructively generate a proof of ψ from ϕ iff ϕ ⊢ ψ. This is done intuitively via a proof by contradiction, and more formally as follows:

Start with ϕ as the current scene:	ϕ
Use Double-Cut (DC) to add two cuts around ϕ:	[[ϕ]]
Use Insertion (INS) to add the negation of ψ, [ψ], inside the outer cut:	[[ψ] [ϕ]]
Use Iteration (ITR) to add the negation of ψ to the inner cut:	[[ψ] [[ψ] ϕ]]
Derive a contradiction, [ ], inside the inner cut:	[[ψ] [[ ]]]
Use DC to remove the inner cut and contradiction:	[[ψ]]
Use DC to remove the cuts around ψ:	ψ

Thus, as long as the contradiction can be derived from the subgraphs ϕ and [ψ], this method will always yield ψ as the scene. If the contradiction cannot be derived, then no such proof exists and ψ is not a consequence of ϕ. The validity of this construction relies on the validity of E.G. itself -- since everything done is an inference in E.G., then as long as E.G. is sound then it cannot result in a proof of something which is not a logical consequence of the premises.

This, however, requires a constructive method of generating a contradiction that will also catch failure. Consider the following method:

Given a series of statements Γ in E.G. (Alpha system):
Remove all Double-Cuts (DC).
If any literal (statement or cut containing only one statement) is at level 0:
    Use Deiteration to remove all copies of that literal that are at level 1 (DEIT).
    Remove all Double-Cuts (DC).
    Repeat for a different literal.
Once a contradiction is derived (an empty cut at level 0), Erase all other subgraphs (ER), and end successfully.
If, however, there are no literals that can be deiterated, end and fail.

This is very similar to Resolution methods, with a few key differences. First, the subgraphs in E.G. are not necessarily in clausal form (essentially CNF) and cannot be rewritten as such within the system (due to the limited number of logical operators preventing graphs from applying equivalences to reposition or distribute the operators). As a result, it is possible to have graphs in forms, such as [ϕ [ψ χ]], which are not clausal. However, the nature of E.G. does not require the clausal form, for exactly the same reason. The graph [ϕ [ψ χ]] is equivalent to the expression ϕ → (ψ ^ χ), which can only result in a contradiction if ϕ and either [ψ] or [χ] are both true. Thus, in order to garner a contradiction, ϕ must already exist at level 0, allowing the DEIT and DC to

provide ψ and χ at level 0 to form a contradiction with either [ψ] or [χ]. Again, due to the nature of E.G., all such non-clausal forms must reside within a cut (they would otherwise be a literal or clause), and can be expressed in terms of the implication operator, thus requiring methodology such as the one above in order to result in a contradiction.

In other words, as long as the algorithm provided above does not erase literals when it finishes, it will be able to catch such non-clauses properly. Since this will enable it to check for DEIT of a literal repeatedly, then it will be able to catch an instance where there was no applicable instance in one case, but after "satisfying the antecedent", so to speak, such an instance became available.

The key then becomes ensuring that the algorithm does not hit an infinite loop when checking literals. This is very simple to do: Have the program go through all current literals in a list, adding new ones to the end as they are formed. Upon exhausting all the ones in the list, it determines if it was able to apply any DEIT successfully. If so, it goes through it again, otherwise it knows that no further steps can be made and it fails, deciding that a contradiction does not follow from the graph.

It is also worth noting that, although the algorithm makes references to levels being '0' or '1', in terms of the larger proof they would both be increased by 2. This makes no difference to the decidability of it, since Erasure and Insertion only need the level to be even or odd, which is preserved in this.

Unfortunately, I did not end up having enough time to implement this into the Peirce system. I spent too much time fixing up the existing code (it does, at last, handle the full Insertion now, which makes it a functional piece of software, though). However, it would be entirely possible to do so, just requiring some additional support code that is tricky to write -- namely, getting the program to be able to add new objects to the graph itself without breaking anything. The plan was to have the user define the "conclusion" graph, letting the program copy it to form the necessary assumptions, but it stills leaves the matter of placement (twice) and forming the cuts around everything properly. There was also the issue of keeping track of all the steps, since the program has limited memory and, indeed, a finite number of "steps" it will store before dumping them. If not done properly, it could delete the starting premises, find it is not a consequence, and then have nothing to 'revert' to. Finally, finding and removing all double-cuts would also be tricky. Though it is made simpler by knowing you only need to check around anything that was just deiterated, there are still a few potential cases to check (since Peirce does its own backtracking to form double-cuts from selected cuts, however, this would be slightly easier).

As a result, examples of the method in action were done by hand (in Peirce) with enough screenshots to provide a good idea of the steps taken, rather than the result of an action in the software. Once the issues with storage and automated placement are worked out, however, adding it in as an actual ATP within Peirce is quite feasible.

The examples provided are of the following proofs:

LPL 6.41 (Tautology):

(A ^ B) v ~A v ~B

LPL 8.32:

Horned(c) → (Elusive(c) ^ Dangerous (c)
(Elusive(c) v Mythical(c)) → Rare(c)
Mammal(c) → ~Rare(c)

Horned(c) → ~Mammal(c)

LPL 8.47 (Invalid):

Small(a) ^ (Medium(b) v Large(c))
Medium(b) → FrontOf(a,b)
Large(c) → Tet(c)

~Tet(c) → FrontOf(c,b)

LPL 8.48:

Small(a) ^ (Medium(b) v Large(c))
Medium(b) → FrontOf(a,b)
Large(c) → Tet(c)

~Tet(c) → FrontOf(a,b)

LPL 8.53:

Small(a) → Small(b)
Small(b) → (SameSize(b,c) → Small(c))
~Small(a) → (Large(a) ^ Large(c))

SameSize(b,c) → (Large(c) v Small(c))

Non-Clausal Test:

P → (Q v R)
R → (Q ^ S)

P → (Q v S)

A few notes on the changes to Peirce.

First, Insertion now works fully. It is still possible to copy / move objects via Insertion as before, but you can now select cuts (which are themselves at even levels) and hit Enter, switching to Insertion mode. This functions similarly to Premise mode (it actually triggers Premise mode as well, and there was a bug that made it say you were in Premise mode at the time I was making the animations), but enforces additional rules that you can't modify anything you didn't just make, and can only add things to the inside of those cuts. Like most implementations in Peirce, you can Insert into multiple cuts in one 'step', and add an arbitrary amount of things to it in that step. You can also still move and resize existing things (to make room, etc.) as long as they don't change the structure of the graph (what it actually means). Finishing the Insertion is done via ALT+r, like exiting Premise mode, and has no additional checks -- since the system enforces valid Insertions with each action taken in that mode, it already knows it's valid.

Second, the screen scrolls when you get close to the edges of the screen. Simple, but useful.

Collision routines for enforcing valid graph structures work a lot better now, and the code is cleaner and better organized. From a user standpoint, this means that you can't make invalid graphs anymore, at least as far I as was able to test, which is pretty important.

I think there might still be some bugs and issues with selection, but it's better than it was before, at least.

Lastly, it's still Windows only at this point. I tried porting it over to Mac OS X, but hit a lot of weird problems and decided it would be best to clean it up on the existing platform so it was easier to tell when something was the result of my changes rather than OS issues.