Squeezing Church's Thesis Again

Peter Smith

March 24, 2010

In the very last chapter of my *Introduction to Gödel Theorems*, I rashly claimed that there is a sense in which we can informally *prove* Church's Thesis. This sort of claim isn't novel to me: but it certainly is still very much the minority line. So maybe it is worth rehearsing some of the arguments again. Even if I don't substantially add to the arguments in the book, it might help to approach things in a different order, with some different emphases, to make the issue as clear as possible.

1 Kreisel's squeezing argument

1.1 Squeezing arguments, the very idea

This much, I hope, is reasonably uncontentious: Church's Thesis is a claim in the domain of reference, not in the domain of sense. It isn't a claim about conceptual analysis, but about which functions fall under certain concepts. At a first pass, the claim is that the informal concept of an *algorithmically computable* function has the same extension as the formal concept of a *recursive* function.

So let's start by thinking about one type of argument that we might be able to use to show that a given informal concept is indeed coextensive with some precisely defined concept.

I begin very schematically. Take some given informally characterized concept I. And suppose firstly that we can find some precisely defined concept S such that falling under concept S is certainly and uncontroversially a *sufficient* condition for falling under the concept I. So, when e is some entity of the appropriate kind for the predications to make sense, we have

K1. If e is S, then e is I.

Now suppose secondly that we can find another precisely defined concept N such that falling under concept N is this time certainly and uncontroversially a *necessary* condition for falling under the concept I. So we also have

K2. If e is I, then e is N.

In terms of extensions, then, we have

Ki. $|S| \subseteq |I| \subseteq |N|$

where $|X|$ is the extension of X. So the extension of I – indeterminately bounded though that might be – is at least sandwiched between the determinately bounded extensions of S and N.

Now so far, so uninteresting. It is no news at all that even paradigmatically vague concepts can be so sandwiched between more sharply bounded concepts. Being tall is

sandwiched between being at least seven foot six and being at least four foot six. Nothing exciting there!

But now suppose, just suppose, that in a particular case our informal concept I gets sandwiched between a sharply defined concept S giving a sufficient condition and a sharply defined concept N giving a necessary condition, but we can *also* show that

K3. If e is N, then e is S.

In the sort of cases we are going to be interested in, I will be an informal logical or mathematical concept, and S and N will be precisely defined concepts from some rigorous theory. So in principle, the possibility is on the cards that the result K3 could actually be a *theorem* of the relevant mathematical theory. But in that case, we'd have

Kii. $|S| \subseteq |I| \subseteq |N| \subseteq |S|$

so the inclusions can't be proper. What's happened is that the theorem K3 squeezes together the extensions $|S|$ and $|N|$ which are sandwiching the extension I, and we have to conclude

Kiii. $|S| = |I| = |N|$

In sum, the extension of the informally characterized concept I is revealed to be just the same as the extensions of the sharply circumscribed concepts S and N.

So far, however, that's entirely schematic. The next question is: are there any plausible cases of informal concepts I where this sort of squeezing argument *can* be mounted, and we *can* show in this way that the extension of I is indeed the same as that of some sharply defined concept?

1.2 Kreisel's argument

Well, there's certainly one familiar example – due to Georg Kreisel, to whom the general idea of such a squeezing argument is due.

Take the entities being talked about to be *arguments couched in a given regimented first-order language*, meaning of course a language with the usual quantifiers and core truth-functional connectives. And let the concept I_L be the classical notion of *being valid-in-virtue-of-form* for such arguments – where we informally explicate this concept by saying that an argument α is valid in this sense if, whatever we take the relevant non-logical vocabulary to mean, and however the world might then fix truth-values, it's never the case that α's premisses are true and its conclusion is false. This intuitive notion might well be counted as pretty 'vague', given that it isn't sharply specified just what counts as permissible reinterpretation of the non-logical vocabulary (for example, how big a set-theoretic universe can be call upon?). However, a bit vague and arm-waving though it is, this informal explication does in fact suffice to pin down a unique extension for I_L. Here's how.

Take S_L to be the property of having a proof for your favourite natural-deduction proof system for classical first-order logic. Then (for any argument α)

L1. If α is S_L, then α is I_L.

That is to say, the proof system is classically sound: if you can formally deduce φ from some bunch of premisses Σ, then the inference from Σ to φ is intuitively valid according to the *intuitive* conception of validity-by-form. That follows by an induction on the length of the proofs, given that the basic rules of inference are sound according to the

intuitive conception of validity-by-form. Their intuitive classical soundness is, after all, the principal reason why classical logicians accept the proof system's rules in the first place!

Second, let's take N_L to be the property of *having no countermodel in the natural numbers*. A countermodel for an argument is, of course, an interpretation that makes the premisses true and conclusion false; and a countermodel in the natural numbers is one whose domain of quantification is the natural numbers, where any constants refer to numbers, predicates have sets of numbers as their extensions, and so forth. Now, even if we are a bit foggy about the limits to what counts as an 'interpretation' in the sense used in the informal explication of the idea of validity, we must recognize at least this much: if an argument does have a countermodel in the natural numbers – i.e. if we can reconstrue the argument to be talking about natural numbers in such a way that actually makes the premisses are true and conclusion false – then the argument certainly can't be valid-in-virtue-of-its-form in the intuitive sense. Contraposing,

L2. If α is I_L, then α is N_L.

So the intuitive notion of validity-in-virtue-of-form (for inferences in our first-order language) is sandwiched between the notions of being provable in your favourite system, and having no arithmetical counter-model, and we have

Li. $|S_L| \subseteq |I_L| \subseteq |N_L|$

But now, of course, it's a standard theorem that

L3. If α is N_L, then α is S_L.

That is to say, if α has no countermodel in the natural numbers, then α can be deductively warranted in your favourite natural deduction system. That's just a corollary of the usual proof of the completeness theorem for first-order logic.

So L3 squeezes the sandwich together. We can conclude, therefore, that

Liii. $|S_L| = |I_L| = |N_L|$

In sum, take the relatively informal notion I of a first-order inference which is valid in virtue of its form: then our pre-theoretic assumptions about that notion constrain it to be coextensive with each of two sharply defined, mutually coextensive, formal concepts.

1.3 What Kreisel's argument shows

Now, let's not get overexcited! We haven't magically shown, by waving a techno-flash wand, that an argument (in a first-order language) is 'intuitively' valid if and only if it is valid on the usual post-Tarski definition. In fact, I'm not even sure there *is* an 'intuitive' notion of valid consequence. If you think that there is, start asking yourself questions like this. The inference 'The cup contains water; so it contains H_2O'? necessarily preserves truth, but is it valid in the intuitive sense? If not why not? Is the intuitive notion of consequence constrained by considerations of relevance (so that ex falso quodlibet is a fallacy of relevance)? If not, why not? When can you suppress necessarily true premisses and still have an intuitively valid inference? Why?

It seems to me that such questions have no clear answers: so I'm with Tim Smiley who remarks that the idea of a valid consequence is "an idea that comes with a history attached to it, and those who blithely appeal to an 'intuitive' or 'pre-theoretic' idea of consequence are likely to have got hold of just one strand in a string of diverse theories."

So, to repeat, I'm *not* claiming that a squeezing argument shows that our initial inchoate, shifting, intuitions about validity – such as they are – succeed in pinning down a unique extension (at least among arguments cast in a first-order vocabulary). No, the idea is as follows. *One* way of beginning to sharpen up our intuitive ideas – still informal, but pushing us in certain directions with respect to those questions we've just raised – is this. At a first stab, we say that an inference is valid in virtue of form if there's no case which respects the meaning of the logical constants where the premisses are true and conclusion false. But we need to say more about what 'cases' are. After all, an intuitionist might here start talking about cases in terms of warrants or constructions. So at a second stab, pushing things in a classical direction, we'll say that an inference is valid-in-virtue-of-form when if, whatever we take the relevant non-logical vocabulary to mean, and however the world turns out, it can't be that α's premisses are true and its conclusion is false. Given that 'water' and 'H_2O' are bits of non-logical vocabulary, that means that the inference 'The cup contains water; so it contains H_2O' is *not* valid in virtue of form. It also settles that ex falso quodlibet *is* valid (and that we can suppress at least certain necessary premisses). Then the claim is that, having got *this* far, although on the face of it we've still left things rather vague and unspecific, *in fact* we've done enough to fix a determinate extension for the notion of validity-in-virtue-of-form at least as applied to arguments cast in a first-order vocabulary. It is, by the squeezing argument, co-extensive with the souped-up post-Tarski definition of first-order validity.

Put it like this then. We start with a rather inchoate jumble of intuitions about validity (as Smiley suggests, there is no one 'intuitive' concept here). We can sort things out in various directions. Pushing some way along in one direction (and there are other ways we could go, equally well rooted), we get an informal, still rough-and-ready classical notion of validity-in-virtue-of-form. But – and *here* is where the squeezing argument bites – we don't have to sharpen things completely before (so to speak) the narrowing extension of validity snaps into place and we fix on the extension of the modern post-Tarski notion of validity.

2 What is Church's Thesis?

Now let's turn back to Church's Thesis. What are the prospects of running a squeezing argument on the notion of a computable function?

None at all. Or at least, none at all if we really do mean to start from a very inchoate notion of computability guided just by some initial sample of paradigms. Ask yourself: before our ideas are too touched by theorizing, what kind of 'can' is involved in the idea of a function that 'can be computed'? Can be computed by us, by machines? By us (or machines) as in fact we or as we could be? Constrained by what laws, the laws as they are or as they could be in some near enough possible world? Is the idea of computability tied to ideas of feasibility at all? I take that such questions have no determinate answers any more than the comparable questions we had about a supposed intuitive notion of validity.

So, as with the notion of validity, if we are going to do any serious work with a notion of computability, we need to start sharpening up our ideas. And as with the notion of validity, there are various ways to go. But *one* line of development takes us to the sharper though still informal notion of an *algorithmic symbolic computation* (as I'll call it).

For convenience, and to fix ideas, let's concentrate throughout on total one-place functions from numbers to numbers. Then we'll say that such a such a function is

computable by an algorithmic symbolic computation if we can mechanically compute it using a symbolic algorithm, where we put no bound on the number of steps in the computation. And what's an algorithm?

> Roughly, this is the name given in mathematics to a mechanical procedure which, when applied to a number ... terminates after a finite number of steps .... Such a procedure has to be specifiable as a finite sequence of totally explicit, simple, and deterministic instructions. The instructions must tell you what is to be done at each step of the procedure, so that no creativity, ingenuity, or free choice is required.

But recall, of course, that numbers are abstract objects. Procedures we can execute must operate on symbolic representations of them. And what is it for a procedure's instructions for symbol-shuffling to be 'simple'? That obviously must mean 'simple to execute' (roughly speaking, *idiot-proof*). Which is still pretty vague, of course: but it surely means at least this much:

> For a particular algorithm, there is to be a fixed finite bound on the capacity or ability of the computing agent needed to execute a single instruction.

That is to say, the bound on what it takes to follow a step stays fixed throughout the execution: the computing agent mustn't be required to get smarter and smarter as the execution progresses, but just to keep on dumbly plugging away. This requirement is quite explicit in e.g. Hartley Rogers's explication of the idea of an algorithmic computation in his classic 1967 text *Theory of Recursive Functions and Effective Computability*.

In practice, of course, what we can do by way of symbol-shuffling will be subject to temporal and spatial constraints. But in characterizing algorithmic computability – computability in principle, if you like – such global constraints of time, memory resources, etc. are entirely set aside: we just require finitude. Otherwise, what matters is local; what is to be done at each local step is entirely fixed and requires no insight or ingenuity to implement; each step should be executable by a finite agent with fixed finite resources; and what it is also entirely fixed what the next step should be. (So the *absence* of a limit on the size of the workspace or how long a computation is to take and the *presence* of a limit on 'attention span' of the computing agent are both natural concomitants of the same fundamental assumption that we are dealing with a computing agent of fixed cognitive capacity, doing limited things locally.)

The first quotation I've just given is from a very recent (and actually, not particularly good) book by Francesco Berto, *There's Something about Gödel*. I've chosen it in part just because Berto's main concerns are elsewhere, and he is evidently trying to give very quickly what he takes to be a standard story without thinking about it too hard. I could, of course, have chosen perhaps more extensive but in the end very similar passages giving the same standard story from any one of dozens of books. However, as Berto says,

> One could describe the notions of [an algorithmically] computable function ... in more verbose and exhaustive ways. But when the chips are down, it remains the case that we are dealing with intuitive and, in this sense, slightly vague notions.

So, the observation is that the notion of algorithmic computability, though significantly sharpened compared with our initial inchoate concept, is still informal and far from precisely formulated.

Church's Thesis however, is that the notion of an algorithmically computable function – informally and somewhat vaguely given though it is – is in fact co-extensive with the precisely circumscribed notion of a recursive function (and so, of course, co-extensive with the notion of a Turing computable function, etc.).

So the picture is this. We really need to distinguish *three* levels of concepts which can be in play hereabouts:

1. We start with an initial, inchoate, 'unrefined', idea of *computability* – a concept fixed, insofar as it *is* fixed, by reference to some paradigms of common-or-garden real-world computation, and perhaps some arm-waving explanations (like 'what some machine might compute').

2. Next there is our idealized though still informal and vaguely framed notion of computability using a symbolic algorithm (also, of course, known as 'effective computability'). Here we require that the computational steps as we shuffle symbols follow an algorithm in the sense we've just briefly indicated, deterministic steps accessible to a limited agent. But we abstract away from 'practical' considerations of how long a computation will take or how much memory will be needed to execute it.

3. Then, thirdly, there are the formal concepts of *recursiveness*, *Turing computability*, and so on.

Now, the move from the first of these notions to the second involves a certain exercise in conceptual sharpening. And there is no doubt an interesting story to be told about the conceptual dynamics involved in reducing the amount of 'open-texture', getting rid of some of the imprecision, in our initial inchoate concept – for this exercise isn't just an *arbitrary* one. However, it plainly would be over-ambitious to claim that in refining our inchoate concept and homing in on the idea of effective computability we are just explaining what we were talking about all along. There's too much slack in our initial position; we can develop it in different directions.

Rather, Church's Thesis – as I understand it – kicks in at the *next* stage. The claim is that, once we have arrived at the second, more refined but still somewhat vague, concept of an algorithmic computable function, *then* we've got a concept which has as its extension just the same unique class of functions as the third-level concepts.

Now, I don't think that there is any question that what I am calling Church's Thesis is exactly what many others from Kleene and other founding fathers, and then through Hartley Rogers right on to Berto, call Church's Thesis. Unfortunately not everyone uses the label that way. But it would be very boring to dwell on *that*. What I'm interested in is the issue whether our somewhat rough-and-ready notion of an algorithmically computable function is in fact sufficiently constrained to fix on the set of recursive functions. I certainly think history is on my side in saying, for shorthand, that this is the issue of whether Church's Thesis is true. But if you balk at my use of the shorthand label, I'll perhaps raise an eyebrow but I'm not going to be particularly upset.

3 The shape of a squeezing argument for Church's Thesis

If we are to warrant Church's Thesis by a plausible squeezing argument, we need three premisses. We'll use f to range over one-place numerical functions (keeping to that

restriction for convenience), let C be the still-informal notion of being algorithmic computable. Then what we need to find is a pair of formally defined concepts S and N where we can show

C1. If f is S, then f is C;

C2. If f is C, then f is N;

C3. If f is N, then f is S.

Even though our concept of algorithmic computability is not fully 'clear and distinct', the aim is to pick an S such that it is clearly compelling that if a function falls under the concept S then it is algorithmically computable. We also aim to pick an N such that it is equally compelling that if a function *doesn't* fall under the concept N it *isn't* algorithmically computable. And then we need to be able to prove that whatever functions falls under N falls under S.

Well, we know what to put for S, given that after the squeeze, S, C and N are all to have the recursive functions as their extension. Choose S to *be* the concept of a recursive function. It is agreed on all sides that a recursive function is computable in the informal sense. If you need an argument, just note that a recursive function f is Turing-computable, so there is a Turing machine program which computes f; and that program gives us entirely determinate algorithmic instructions so – given world enough and time – we can compute the value of f by trivially simple steps for any given input. So a recursive function is algorithmically computable.

Obviously, then, the hard work is going to be finding an N which is weak enough to feature in a compelling necessary condition for being algorithmically computable, but strong enough to give us the third premiss. Can we pull off the trick?

4 Finding a formal necessary condition for computability

4.1 Starting with Turing

In his epoch-making 1936 paper, Turing notes various features that would *prevent* a procedure from counting as algorithmic in the intuitive sense. For example, an algorithmic procedure can't require an infinity of distinct fundamental symbols: 'if we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent' (given they have to be inscribed in finite cells), and then the difference between symbols wouldn't be recognizable by a limited computing agent. For similar reasons, the computing agent can't 'observe' more than a limited amount of the workspace at one go. And computations shouldn't involve arbitrarily large jumps around the workspace which can't be reduced to a series of smaller jumps – a bound is set by cognitive resources of the computing agent, who needs to be able to recognize where to jump to next. All in all, remember, we want individual steps in the computation to be 'simple' as Berto puts it, accessible to one of Hartley Rogers's computing agents with some fixed finite bound on his or her capactiies. So the 'size' of steps in the algorithmic procedure needs to be kept under control.

What we want to do, then, is put together the requirements of a finite alphabet, restricted local action on and movement in the workspace, etc. with whatever other similar constraints we can muster, and so build up to a formally specifiable composite necessary condition N for being calculable by an acceptably algorithmic procedure. The hope is to get a condition strong enough to be able to complete the squeezing argument.

With a bit of regimentation, we can think of Turing as gesturing towards a suitable condition N and giving us the beginnings of a defence of the second and third premisses for a squeezing argument. The bad news, however, is that Turing's way of spelling out N – if we think of what he is up to like that – is surely *too* strong. For example, when we do real-life computations by hand, we often insert temporary pages as and where we need them and equally often throw away temporary working done earlier. In other words, our workspace doesn't have a fixed form: so when we are trying to characterize intuitive constraints on computation we shouldn't assume straight out – as Turing in effect does – that we are dealing with computations where the 'shape' of the workspace stays fixed once and for all (a fixed geometry tape). We need to relax Turing's analysis of an algorithmic computation. So that's what we'll try to do.

4.2 Conditions on algorithmic computation

If we are going to be talking about conditions for being an algorithmic procedure for the step-by-step manipulation of symbols, then there are four sorts of thing we need to consider:

1. The symbols and the data-space (as I'll call it) in which they are to live.
2. The sort of manipulation that counts as a step in a computation.
3. The kind of instructions that take us from step to step.
4. How we input data into a computation and extract data at the end.

Let's take these in turn.

First then, *the symbols and the data-space*. For Turing's reasons, let's agree that

i. We require the basic alphabet of symbols which any particular algorithm uses has to be finite.

Now, in Turing's model of computation, individual symbols live in cells on a linear tape. Let's continue to say that symbols live in 'cells', but we now start relaxing Turing's assumptions about the shape of the dataspace, to get the most general story possible. Since we are going to allow the extent of the dataspace to change as the computation goes along, we can take it without loss of generality to consist of a finite network of 'cells' at any stage, adding more cells as we need them. So,

ii. The dataspace at any stage of the computation must consist of a finite collection of 'cells' into which individual symbols are written. But now generalizing radically from the 'tape' picture, we'll allow cells to be arranged in any network you like, with the only restriction being that there is some fixed upper limit (which can be different for different algorithms) on the number of immediate 'neighbours' we can reach from any given cell. And we won't require that being 'neighbours' is a matter of physical contiguity: we just require that a cell needs to carry *some* kind of 'pointers' or 'arrows' to zero or more other cell(s). Since the algorithm will need to instruct us to operate on cells and certain neighbours and/or move from one cell (or patch of cells) to some particular neighbour(s), we'll need some system for differentially labelling the 'pointers' from a cell to its various neighbours.

Why put an upper bound on the number of different arrows leading out from a given cell? Well, consider a finite computing agent with finite 'on board' cognitive resources

who's trying to follow program instructions of the kind that require recognizing and then operating on or moving around a group of cells linked up by particular arrows. There will be a bound on the number of discriminations the agent can make 'in a single step'. But subject to that constraint, the dataspace can now be structured in any way you like: we only require that the network of cells is locally navigable by a computing agent with fixed cognitive resources which bound what it can do in a single step.

Next, let's consider constraints how we manipulate symbols and/or the dataspace in a single step of a computation if it is to remain algorithmic. We've said that an algorithmic procedure should proceed by 'simple' steps. So, we can certainly make the weak requirements that

iii. At every stage in the computation of a particular algorithm, a patch of the dataspace of some maximum finite size is 'active'.

iv. The next step of the computation operates only on the active area, and leaves the rest of the dataspace untouched.

Without loss of generality once more, we can think of the 'active' area of dataspace at any point in the computation to be the set of cells that are no more than n arrows away from some current focal vertex, where for a given algorithmic procedure n again stays fixed throughout the computation. Why keep n finitely bounded? Because, recalling Hartley Rogers, we require the maximum attention span of a limited cognitive agent making simple single steps to stay fixed as he runs the algorithm. However, we will otherwise be ultra-liberal and allow the bound n to be as large as you like.

Now for perhaps the crucial radical relaxation of the Turing paradigm:

v. A single computing step allows us to replace the active patch of cells in the active area of the dataspace with particular contents and a particular pattern of internal 'arrows', by a new collection of cells (of bounded size) with new contents and new internal 'arrows'.

So, at least internally to the active area of the dataspace, we can not only fiddle with the contents of the cells at vertices, but change the local arrangement of arrows. As announced, then, the shape of the dataspace itself is changeable – and the dataspace can grow as needed as we replace a small patch by a larger patch with extra cells and new interlinkings. Again, to keep procedural steps counting as 'simple', for any algorithm we fix some bound (though it can be as large as you like) on just how much new organized data space can be created in a single step for that algorithm.

Having worked on one patch of the dataspace, our algorithm needs to tell us which patch to work on next. And, for Turing's reasons,

vi. For any algorithm, there is a fixed bound on how far along the current network of cells the focal vertex of the active patch shifts from one step of the algorithm to the next.

But again, we will be ultra-relaxed about the size of that bound, and once more allow it to be arbitrarily large for any given algorithm.

Thus far, then, we have put some exceedingly weak constraints on the symbols (finite alphabet) and dataspace (navigability by a finite agent); and we've put incredibly weak constraints on what happens an any step of executing an algorithmic procedure (do whatever you like as long as you keep it local and bounded). Anything that can count as an algorithmic procedure should surely meet these constraints. We now need to define the character of the algorithm itself.

vii. The body of an algorithm which dictates the step-by-step procedures consists in a finite consistent set of instructions for changing clumps of cells (both their contents and interlinkings) within the active patch and then for jumping to the next active patch. Without loss of generality, we can follow Turing in taking these instructions as in effect being given as labelled lines, each line giving a finite bunch of conditional commands of the form 'if the active patch is of type P, then change it into a patch of type P'/move to make a different patch P'' the new active patch; then go on to execute line q_j/halt'. What we going to require for an algorithm is only that each instruction is finite, and there is only a finite number of them.

Finally, we need to say something about feeding initial data to an algorithm and extracting a result after a run. For current purposes our focus, we said, is on algorithms for computing monadic functions from natural numbers to natural numbers. So, in that sort of case,

ix. We need a finite initial set of instructions which sets up and structures an initial patch of dataspace, and tells us how to write in some representation of the input argument n

x. And we will need to specify how we read off numerical output $f(n)$ from the configuration of cells in the active patch if and when we receive a 'halt' instruction after a finite number of steps.

OK, that's all a bit rough and informal. But it should be clear enough – if you have a taste for this kind of thing – how to go develop these sketched conditions into a formal abstract characterization of conditions on an algorithm: giving such abstract characterizations is just the sort of thing mathematicians are good at! And, fine details apart, the work was in fact done fifty years ago by Kolmogorov and Uspenskii. Let's say that any procedure satisfying our conditions is KU-algorithmic. We can then offer the following natural definition

> A (monadic, total) function $f(n)$ is then *KU-computable* if there is some KU-algorithmic procedure which, when it operates on n as input, delivers $f(n)$ as output.

Now, such is the wild generality of the KU conditions, we might suppose that they cover *far* too many procedures to count as giving an analysis of the intuitive notion of an algorithm. For what we ordinarily think of as algorithms proceed, we said, by simple steps and KU-algorithms can proceed by arbitrarily large operations on arbitrarily large chunks of dataspace (galaxy-sized chunks, if you will). But let's just not worry about that. The only question we need to focus on is: could the KU story possibly cover *too little*? Well, how could a proposed algorithmic procedure for calculating some function *fail* to be covered by the KU specification?

The KU specification involves a conjunction of requirements (finite alphabet, logically navigable workspace, etc.). So for a proposed algorithmic procedure to fail to be covered, it must falsify one of the conjuncts. But how? By having (and using) an infinite number of primitive symbols? Then it isn't usable by a finite computing agent like us (and we are trying to characterize the idea of an algorithmic procedure of the general type that cognitively limited agents could at least in principle deploy). By making use of a different sort of dataspace? But the KU specification only requires that the space has *some* structure which enables the data to be locally navigable by a limited agent. By not keeping the size of active patch of dataspace bounded? But algorithms are supposed

to proceed by the repetition of local operations which are surveyable by limited agents (whose limitations remain stable as they plod onwards). By not keeping the jumps from one active patch of dataspace to the next active patch limited? But again, a limited agent couldn't then always jump to the next patch 'in one go' and still know where he was going. By the program that governs the updating of the dataspace having a different form? But KU-algorithms are entirely freeform; there is no more generality to be had.

In sum: the idea of a KU-algorithmic procedure is so very generous and so all-embracing that Kolmogorov and Uspenskii are surely right that any procedure that counts as algorithmic in the informal sense certainly falls under an abstract formal characterization of a KU-algorithm. But that means we have a formal necessary condition on being an algorithmically computable function. We can put

2. If f is C (algorithmically computable), then f is N (KU-computable).

4.3 Completing the squeeze

So far then, we have

2. If f is S (recursive), then f is C (algorithmically computable).

3. If f is C (algorithmically computable), then f is N (KU-computable).

The last premiss of the squeezing argument – you won't be surprised to hear – is the technical result

3. If f is N (KU-computable), then f is S (recursive).

We won't go into details here. But the proof uses an entirely standard technique, generalizing the usual proof that Turing-computable functions are recursive (see the Appendix).

So the squeeze is complete. The claim is that once we've elaborated the notion of a computable function enough to get the idea of a function computable by step-by-step algorithmic process where the steps have to be 'simple' enough to be negotiable by a cognitively limited agent (but without global constraint of time or memory), the algorithmically computable functions have to be the KU-computable ones. And that forces them to be the recursive ones.

5 Conclusion

Is that a 'proof' of Church's Thesis? Well, what's in a word? The argument is a priori compelling, and it doesn't depend on (as it were) quasi-empirical considerations – as in the conventional defence of the Thesis on the grounds that no one has found a decent counterexample.

But still, consider the familiar one-direction conditional that if function is recursive, it is algorithmically computable. That is often enough said to be provable (indeed we sketched the proof), even though it does relate an informal concept to a formal one. Here's another familiar result: there are algorithmically computable functions that aren't primitive recursive. This is routinely established by a diagonal argument, which is again standardly taken as a proof, even though it again concerns the relation between an informal concept and a formal one. If we count *those* familiar arguments as 'proofs' – and by ordinary standards they are – then the squeezing argument equally counts indeed as a proof of Church's Thesis.

6 Appendix

Everything in an KU-algorithmic set up is finite, so we can arithmetize by some kind of Gödel numbering, and then come up with a code number for the state of play in the data space. Again since everything is finitely bounded, the function *code* that gives us the code number for the state of play at step j of the computation for algorithm A starting with input n will be primitive recursive in j and n. And similarly the function *decode* that gives the numerical output from the computation using A from the state of play code at halting will be primitive recursive. Suppose we set things up so that the state-of-play code number defaults to zero at the step-number after the computation halts. Then a run of algorithm A halts, if at all, at the step number

$$\mu j[code(n, Sj) = 0].$$

If there is a total function computed by A, it will therefore be

$$f(n) = decode(code(n, \mu j[code(n, Sj) = 0])$$

which must be recursive giving the coding and decoding functions are primitive recursive.

What that line of argument actually shows is that what matters is that we can keep the state-of-play function *code* primitive recursive. And for that reason, it isn't in fact necessary that the various finite bounds that appear in the characterization of a KU algorithm are actually fixed – it is enough that they are bounded by functions primitive recursive in the initial data and the step number. That techie observation allows us to weaken even further the idea of a KU-algorithmic procedure to the notion of a KURG-algorithm (KU-with-recursive-growth-on-bounds) while still running the squeezing argument. But really that's unnecessary overkill, since any ordinary algorithm satisfying e.g. Hartley Rogers's explication of the notion is a KU-algorithm.