EPISODE 1365

[INTRODUCTION]

[00:00:00] KP: The Notebook paradigm of coding is relatively new in comparison to Repple and IDEs. Notebooks run in your browser and give you discrete cells for running segments of code. All of the code in a single cell runs at once, but cells run independently, cells can be rerun, which is a blessing and a curse. The ability to run cells out of order can make it difficult for users to have a clear understanding of what they also might want to recompute.

The NBSafety project is an easy to install tool for automated management of notebook state, which can help you catch bugs early. Stephen Macke is a PhD student at the Data and Information Systems Laboratory at University of Illinois, Urbana Champaign. In this episode, we discuss Jupyter Notebooks, the development of a custom kernel, and how NBSafety can help notebook users. Stephen, welcome to Software Engineering Daily.

[00:00:59] SM: Hi, Kyle. Thank you so much for having me. It's really great to be here.

[00:01:02] KP: Can you tell me a little bit about the work you do?

[00:01:06] SM: I'm a software engineer full time. But on the side, I've been working on this project related to safer computational notebooks. It's called NBSafety. The idea is it's really difficult to manage state in a computational notebook, it's hard to keep track of all the variables you've defined and what's changed, and what cells you should rerun when variable changes.

[00:01:32] KP: For listeners who haven't maybe messed around in a Jupyter Notebook before, I can be kind of a different paradigm to work in. If I'm used to programming in you know, console, Repple, or some IDE, can you speak a little bit to what the notebook experience is like?

[00:01:47] SM: The way I think of a notebook, which, it's evolved over time. But the way I model a notebook is it's kind of like a Repple on steroids. With a Repple, you are issuing commands to an interpreter and that is how your program runs. It runs one statement at a time. But oftentimes, you want to go back and you want to change some statements that you previously executed to do something else. Notebooks kind of captured the idea that, “Oh, maybe I don't want to just change a particular statement. Maybe I want to change an entire set of sub statements. And maybe I want to very quickly and easily go back and edit an entire set of sub statements, and run those.”

So, notebooks are a way to kind of group together related program statements and execute those interactively, and make it really easy to you know, edit stuff you've done in the past. And maybe you want to also reorder cells so that you can get kind of a logical program flow that fits the way you think about how your program should flow. So, correct me if I'm wrong, but I think Mathematica was actually the first framework to popularize the notion of a computational notebook.

[00:02:58] KP: I never used it. But I've heard that as well. I believe that's really the origin.

[00:03:01] SM: Nowadays, notebooks are used not primarily, but a large number of notebook use cases are for data science, because kind of the notebook programming paradigm fits the data science use case and very well, where you don't necessarily know in advance what code you want to write, that kind of evolves as you look at your data and mess around with it. So, notebooks capture this use case particularly well, and so that's one of the major applications today.

[00:03:30] KP: So, it's a very powerful framework that anybody who's used it for even a little while, I think, understands a lot of the benefits and immediate feedback they get, not to mention that we didn't even talk about yet that you can have visualizations of the data right there in line. So, it's very easy to kind of work through some of these data problems. So obviously, notebooks are a benefit. Are there any drawbacks or bad habits that you see emerge when people start to develop their code in this way?

[00:03:55] SM: Well, I mean, I certainly won't speak for others. But for myself, in particular, typically, when I'm in a notebook, I am focused purely on those advantages. You mentioned that immediate feedback, kind of getting that adrenaline and dopamine rush from being able to see things immediately, without too much intermediate debugging, so I tend to be very aggressive in a notebook and I tend to kind of write very hacky code very quickly, just trying to get something working as quickly as possible.

Oftentimes, I will create, like thousands of cells that I don't know what they're doing anymore, after five minutes or so. It becomes very unmanageable very quickly for me personally. So, I believe my experience is not isolated to myself. This has been a piece of feedback that others have expressed as well. In fact, there was a very, very famous talk at JupyterCon 2018 called, “I Don't Like Notebooks” by Joel Grus, which talked about some of these issues, and the difficulty of kind of keeping track of what you've done and what your current program state is in a notebook.

[00:05:11] KP: One of the great promises that was introduced to me as notebooks were becoming more prominent was that it could contribute to reproducible analysis that we're left with is, I guess, as long as we kind of behave well, you can leave behind a very nice trail of exactly what steps you did and that can be published. It seems it takes quite a bit of curation to do that effectively. What are your thoughts about putting some guardrails in place to help people achieve better reproducibility?

[00:05:37] SM: Absolutely. So, that was the major motivation behind NBSafety. To give a little bit of context, a lot of this curation that occurs in the notebook is to try and eliminate all of the stuff that you don't need and to try and put it in order such that when you execute the notebook from top to bottom, you get the exact same thing every single time. Because if I give a notebook to a random person, my expectation is that person will run it from top to bottom. They don't have the same context that I did when I developed it. And maybe that person is me in the future where I forgotten what happened in that notebook.

NBSafety is kind of a way to signal to you, “Hey, you change this thing, maybe you want to rerun this cell now.” If you're seeing that when I change a particular variable, it suggests that I rerun some cell that's like above, where I change the variable, “Hey, maybe I should move that cell below where I changed the variable.” So, it just kind of encourages you to reorder the notebook in the order that someone is going to execute it later on, it kind of provides to you, that hint, that, “Hey, if you want this to behave, the way that it will behave, when someone gets it later and executes it from top to bottom, you probably want to go ahead and rerun the cells that depend on the variable that you just changed."

So, it just provides a little bit of encouragement to put the notebook in that form, without having to think too much manually about how to go about doing that. So, yeah, I think I'll stop there. I think that's a reasonable description of the guardrails that NBSafety provides, and that I think, are a pretty good compromise between giving you the flexibility that you want in a notebook. Wow, still encouraging some best practices.

[00:07:27] KP: So, in order for NBSafety to deliver on that, it seems to me it either has to be running as the state of the notebook evolves and doing some bookkeeping of some kind, where it attracts things. Or at the time a cell is run, it can maybe start from first principles, look backwards and figure some things out. How does it do its reasoning?

[00:07:46] SM: NBSafety in particular does kind of a strange approach compared to what other systems that have attempted to make this happen perform. NBSafety uses kind of two main components. One is a tracer in order to detect what variables have changed. So, as the program is running, it's looking at all of the statements that execute and figuring out, "Okay, when this statement executes, x equals 5, x changed. So, let's mark x as having a new version." And then once a cell has finished executing, it says, "Okay, now let's take all of those variables that have changed and let's look for live references to those variables in other cells."

A live reference just means that the current value of a variable could be used in some other cell, and that means that that other cell depends possibly on the value of the variable that has changed. So, this is the static component of NBSafety, which says, "Hey, let's look at the code without executing anything, of course, to figure out what cells maybe need to rerun, based on the cell that has been changed." So, in summary, two components dynamic to figure out what has changed, and static to figure out what I should rerun.

[00:09:04] KP: In pretty much any programming language, I can write obfuscated code, maybe like, have a weird way of assembling a string, that's a statement and do some system.execute statement sort of thing. No one would hold NBSafety responsible for tracking through that sort of hidden mess. That's the extreme case. Are there any anti patterns you're seeing? Any sort of typical behaviors that make it difficult for NBSafety to do this effectively?

[00:09:30] SM: Oh, yeah, that's a great question. So, as you mentioned, if someone is using exec or eval, that's going to cause some problems. That's like the kind of extreme case of very, very dynamic behavior.

[00:09:44] KP: I'm trying to hide from you at that point.

[00:09:46] SM: Exactly. These are all corner cases, in some sense, because in principle, I could develop a system that is completely capable of reasoning about which variables have changed. It just depends on you know how deep I go. So, another example of where there are going to be some issues are, let's say I've monkey patched something. If I monkey patch something like, let's say I set collection.ordered.dict, to my own implementation, this is going to cause some issues. Because, well, I won't go into the details, but this is going to cause some issues.

So, there are some corner cases where it's just going to be really difficult to infer what changed. But we're slowly, very gradually eliminating those. So, for example, one of the corner cases that used to be really difficult was, let's suppose, I insert a value into a list, like into the middle of a list, all of the cells that depend on the latter part of the list, now might need to change. If I insert something at position two, now lists sub two is a different thing. So, as lists sub three and lists sub four. But lists sub zero and lists of sub one, those are the same as before. I need to rerun the cells that reference lists sub two, lists sub three, lists sub four, so on and so forth. But I don't need to rerun the cells that reference lists sub one or lists sub zero, because those refer to the same things as before.

So, this is one corner case where we would just kind of threw in the towel previously and said, “Okay, everything that reference this list, you need to rerun that.” Now, we've gotten a little bit better and are able to understand the finer details here about what should actually be rerun. So, that's just one example. And there are other places where there'll be some behavior that's maybe overly conservative, and we're slowly getting rid of that. So, it's a very gradual process, but it's pretty good now, actually. It's done what I expected to do more often than not.

[00:11:46] KP: If we look deeply into a lot of these edge cases, I have a feeling at some point, we're going to bump into the halting problem. So, the NBSafety can't be expected to be perfect, but it doesn't have to be perfect to be useful. As you work with it, maybe you find different ways to improve it to learn some of the common mistakes you make. Are there any ways to broaden that and see if it can learn from in more of a crowdsource kind of fashion?

[00:12:09] SM: That's a great question, because I had someone ask me a very similar question the other day, which is like, if I have some libraries that are calling that expose some API's, for example, Pandas. When I execute certain data frame operations, maybe you take some of the columns, but not others. Or maybe they return an object that were some of the columns of that object depend on some of the columns of the previous object, but not others. So, for example, suppose I have a data frame, and I group by Column C, all of the columns in the group by object depend on C as well as themselves. Without that kind of like domain specific knowledge, we're going to just say, all of the fields of the group by object depend on all of the fields of the previous object. One approach to kind of getting that kind of more detailed lineage information is attempt to infer these kinds of rules automatically without having to like type them all in.

So, one way to do that would be like, let's try running this group by method after deleting various subsets of columns and figuring out what breaks, and then based on what breaks, try to infer some rule about which columns in the group by object depends on which columns from the previous object. And this could be used in general, like just replace column with like attribute. You have a general recipe for other libraries as well. But this is like kind of very far future reaching stuff. I think, probably the conservative approach is going to be what we use for the time being. But in principle, it does seem like something like this could be done, but it's very hard.

[00:13:56] KP: Along those lines and thinking about where research might go, what are your thoughts on, at least when I think about the problem, the notebook doesn't necessarily contain my full state, right? It doesn't know which version of Pandas I have installed and some of the data behind and all these kinds of things. And that's not the notebooks fault, but there's maybe a way we could attach a container to this or some other, I don't know, hashing process that could be added to augment this. Do you have any thoughts on ways that we might want to extend or grow beyond just the notebook environment with some auxiliary or plugin or something like that?

[00:14:30] SM: That's a very excellent question. Because like you said, if I'm just looking at a notebook, the notebook is just a set of program statement. It doesn't contain any information about the environment. People are going to be very opinionated on this, but it would be nice to know from a reproducibility standpoint, what was the version of Pandas I was using when this notebook ran? What was the version of Python that I was using when this notebook ran? This state could be captured in you know, a YAML file somewhere, for a container. And that's the approach that a lot of frameworks and companies are going with.

But with a custom kernel, like NBSafety, for example, we could in principle actually extract this information at runtime while the notebook is running and stuff it in a JSON object that get serialized as part of the NB format. So, that when you're actually looking at the messy JSON structure that specifies the, you know, .ipynb, you could actually figure out just based on that exactly what the notebook is running on. I think there's some appeal to that approach. I don't think anybody does this at the moment. But this is definitely an approach that could be taken once you have this ability to kind of dynamically inspect the runtime values of objects. I can look at what this PD dot, underscore, underscore, version, underscore, underscore. I can look at, what sys.version info is, and I can get all of that information.

[00:15:59] KP: What is the getting started story? How would someone install or get set up and working with NBSafety?

[00:16:05] SM: Oh, yes. My favorite question. All you have to do is get tip which is easier said than done and then to install NBSafety. And you know, I also hear that you're not supposed to write PIP install anymore, you're supposed to do, what is it Python dash and then the module –

[00:16:22] KP: Honest answer, I don't know this one.

[00:16:24] SM: Okay. Yeah, you're supposed to not because PIP, it might be configured on your path to something that you don't expect it to be or something. So, you're supposed to run Python dash, and then the module path to PIP and then install. But yeah, sorry, this is like –

[00:16:40] KP: Thank you for the note.

[00:16:42] SM: It's just PIP install NBSafety. Eventually, we're going to try and get Conda to work as well. But yeah, that's all it is. And then you can just fire up Jupyter and the only difference is you pick the NBSafety kernel instead of the normal I-pipe kernel.

[00:16:56] KP: From there, what's the user experience, like in contrast to the existing one?

[00:17:00] SM: In theory, you might actually see no difference. That's kind of the extreme case where you're using a notebook, it's like you know exactly in advance what you want to write, and you're not iterating on it at all. But in practice, what will happen is, you'll go back and you'll edit a cell and when you edit the cell and rerun it, now you'll see highlights in front of the dependent cells suggesting you to rerun those dependent cells. So, the idea is to just kind of reduce some of that mental overhead. But it keeps all the same flexible, any order execution semantics that a normal Python notebook has, which is something that was very important to us. We don't want to deviate too far from the typical notebook experience.

[00:17:44] KP: It seems to me a lot of people would make fewer errors if they installed this. Is there any way you can track or measure that?

[00:17:51] SM: Oh, I mean, we could. We could collect those kinds of metrics, like how many exceptions occur for people running NBSafety versus without NBSafety? This is kind of like an invasion of privacy, so we don't collect any information.

[00:18:07] KP: Can we talk a little bit about the empirical study from the paper?

[00:18:10] SM: This was a little bit interesting, because behind the scenes, you never hear about behind the scenes of papers. But behind the scenes, I was kind of in a rush to graduate. Ideally, this would have actually been a real user study with real people trying out the software and getting feedback from them. But we didn't have the luxury of time in that case. So actually, what we did is we scraped execution logs from GitHub. So, the history dot SQLite files that

IPython, which, backs Jupyter behind the scenes, uses to record history of which cells are executed.

So, we had the execution logs from a bunch of users and what we found out is that the choices of which cell to re execute based on these execution logs actually correlated very strongly with the suggestions made by NBSafety. So, in some sense, it agreed with what real users were trying to do. And similarly, if there's a cell that was highlighted by NBA safety as being stale or unsafe to re-execute, because it has some dependency that still needs to be refreshed. These cells, it turns out, were actually avoided by real users. So, the long and short of it is it looks like it agrees with what real users tend to do, and the idea is that, if you can just kind of delegate some of that responsibility to this kernel, you don't have to think as hard about what you need to rerun in order to make sure that your cells are referencing the freshest data.

[00:19:44] KP: So, that execution log is that embedded in my .ipynb file?

[00:19:49] SM: No actually. So, the .ipynb file is just the kind of static snapshot of the most recent code that has been executed in each cell. The execution log actually tracks across all of the executions what is the code that you asked to be executed for each cell? It's a little more high fidelity, because we actually needed that data. It wasn't enough to just look at the static snapshot of a notebook in order to infer like whether a cell was rerun or not, for example, because we just didn't have that data available, or like, how did the cell change over time, that kind of thing. As you know, that is something that is very common for notebooks, you'll iterate on a cell several times before you get it right. So, we actually needed that kind of more fine-grained data, which is not available in the notebook JSON, by default. You actually need to look at the history dot SQLite, that is part of IPython.

[00:20:49] KP: So, that's going to be alongside my executables, that's why I've never seen that.

[00:20:53] SM: That's right. I think it's in one of those hidden dot files in your home folder. It's in dot IPython, or dot Jupyter or something, probably that IPython.

[00:21:04] KP: Well, I imagine most Jupyter users are kind of blissfully unaware that that's sitting there. Could you talk a little bit about what kind of data is captured in it?

[00:21:04] SM: Basically, every time you submit a cell to be executed in Jupyter, the timestamp of when you submitted the cell for execution, as well as the code will be recorded. I think, probably the execution counter, so each cell, as you've probably remember, has like an execution counter next to it. There are a few other things as well, that I can't remember. But those are the important ones. Interestingly enough, they actually do not record the ID of the cell that you're submitting. So, when we were actually trying to figure out when a cell has been edited, we had to use a little bit of a heuristic for a cell that had been edited, there's no way to actually link the before and after state of the cell. What we did is we used like a text similarity heuristic and said, “Okay, if the text is more than 90% similar, then it was probably an edit of a previously submitted cell.”

[00:22:06] KP: So that's not a file, I think, many users are committing to their GitHub repositories. How did you get access to that?

[00:22:12] SM: Yeah, most people are not committing that to GitHub. I think probably most of the ones that we scraped were accidents, users who basically accidentally committed the history dot SQLite file. There's not too many of them out there, we basically had to use GitHub API to search across all of the GitHub repositories, which one had a file called history dot SQLite. I think there was something like 50,000 or so. But by the time we actually filtered out all of the stuff that wasn't obviously, kind of just garbage data, I think we were left with something like 600 notebook sessions that looked reasonably like what a real notebook user would be running.

[00:22:56] KP: And do you have any criteria for what that would be? I'm sure it's got to be more than one cell, more than two cells executed, that kind of thing? Was there any cut off?

[00:23:04] SM: Yeah. I don't remember the exact details. I think we talked about it the paper, but I think it was something like at least 20 cell executions that didn't result in exceptions. And at least 50% of the executions had to not be exceptions, as well. There are some people who are still learning Python, maybe that kind of thing. So, we wanted to at least, like make sure that the sessions we were evaluating against were people who had a firm grasp of the Python language and who were doing real data tasks. So, I think one of the things I tended to look for was whether Pandas was getting imported. There were a few other heuristics. A lot of manual inspection, actually, to make sure that it just kind of looked okay, eyeballing it, but no cherry picking, if that's what you're wondering. It was mainly just to make sure that they looked reasonable in terms of like, we could actually replay them and get the same thing that the actual user saw, because a lot of them were actually like downloading data that we didn't have access to anymore. So, those cells wouldn't be re-playable, for example, or maybe using some obscure library that isn't in PyPy anymore, that sort of thing.

[00:24:13] KP: Yeah, it's a really interesting natural experiment. I'm curious if you spent any time sort of spelunking in that data. Are there any general trends you see about the longevity of notebooks or usability observations?

[00:24:26] SM: At least for these particular notebooks, they were very messy. Definitely like scratch pads, just trying to get something prototyped quickly or working quickly, and then kind of throw it away afterwards, which is not the way everybody uses notebooks, definitely. But in this case, it was definitely get something working once, and then throw it away once the idea has been tried out.

[00:24:53] KP: Yeah, I relate to that use case very strongly. That's a typical Jupyter Notebook experience for me.

[00:24:59] SM: Me too.

[00:25:00] KP: Well, I'd love to delve into the process of integrating it or building up your own IPython kernel. I think people are aware that these I kernels exist, maybe they've installed a couple additional ones beyond Python to use other languages. But there's something a little bit mystical there for many developers. What are these kernels and how do you go about creating one?

[00:25:21] SM: My understanding is, broadly speaking, there are two kinds of kernels. There is a full blown, custom kernel that basically implements various hooks for the Jupyter kernel protocol. And then there are wrapper kernels, which maybe override certain hooks, but for the most part delegate most of them. NBSafety is a wrapper kernel. For the most part, it is just overriding think the execute function, and doing a little bit of stuff behind the scenes. There are other kernels like Xavius that are actually like full blown C++ implementations of the Jupiter kernel protocol.

For a wrapper kernel, in particular, it's quite easy to get started. You basically just create a class that overrides kernel imported from I-pipe kernel or something, and you've already got a wrapper kernel. You need to do some extra stuff to register it. But by default, it's just going to delegate to its superclass methods, which are already providing all the functionality that you need. So, you can then override some of those methods. For example, NBSafety overrides the do execute method to do a little bit of extra stuff. One of the things that it does is it rewrites the abstract syntax tree of the code that it's being requested to execute to make it a little bit easier to see what's going on.

So, for example, if I have an attribute A dot B, I want to know exactly what part of memory that points to in order to know like, what is the piece of data that I'm referencing, or that I'm changing in order to properly set up the data flow. So, this is why NBSafety is actually like using a lot of dynamic analysis, as opposed to a purely static data flow approach that a lot of other kernels would take that approach instead. For example, observable uses static data flow to implement reactivity. But yeah, it's actually like a lot simpler than you would expect. It's just override the kernel superclass. In NBSafety case, it was just override the do execute method to do some extra stuff.

[00:27:37] KP: Well, you make the implementation sound easy, but I imagine there's layers of packaging as well. How do you go from that innovation to something that shipped and installable by other people?

[00:27:47] SM: Sure. Most of that is handled by like setup tools and Python disk tools, which again, there's not super great documentation, to be honest. The best approach there is, honestly to look at somebody else's package, and try and reverse engineer what do all of the various things in setup.pi mean. That's what I ended up doing for NBSafety and the dirty little secret is that that's what everybody does, whenever they want to release a package to PyPy.

The other dirty little secret about PyPy in particular, is there's like no validation process for packages that get uploaded there. Anybody can upload a package under any name that's not already taken. So, actually, you'll get a lot of people that have an idea for a package. But maybe it's already taken, or maybe it's not, but they want to make sure they get something uploaded there, so they can squat on it kind of like a URL.

For a language as mature as Python, it is very wild west getting a package onto PyPy, like there's basically no rules. You can kind of do whatever you want, and I think the commands once you have everything ready to go to upload is like twine upload, dist slash star, where dist slash star is where your built Python egg goes, basically. But yeah, it's all just setup tools and betas tools to actually create that egg. And then once you've got that, anybody can just upload that to PyPy.

[00:29:20] KP: Well, can you describe a little bit more about the user experience? Once I've got this installed? We talked about a little bit earlier and as you'd commented, if I have very linear code, it's a traditional experience. But if I start going and running things out of order, how does NBSafety jump in and alert me and let me know that it has some insights for me?

[00:29:39] SM: If you start running things out of order, NBSafety is going to highlight the cells that reference variables that have been updated from your out of order execution and suggest to you, “Hey, maybe you want to rerun the cell now.” However, there are cases where the order in which you kind of refresh yourselves matters. Let's say I have three cells, x equals zero, y equals x plus one, print y. If I change x, now, I probably want to update y before I print it. So, the print y cell will have a red highlight, it's kind of alerting you, “Hey, maybe this one is unsafe. You probably want to run y equals x plus one again, before you reprint y.” This is kind of a contrived example. But these notebooks can get very, very large. So, it might not be obvious for you been hacking your notebook for a little while that this dependency structure is even present in your notebook. It just makes it easier to reason about, okay, I probably want to rerun the cells because they reference symbols that have been updated, and I probably want to do that before I rerun these other cells, which refer to symbols that I still need to update.

[00:30:56] KP: Well, I'm thinking back now to all of those history files and if you were in a notebook, let's say like this one with three cells that you described earlier, and they've run them all, they run x equals one, y equals x plus one, then they've run print y, and then gone back and changed x equal to some other number, let's say three. A couple of things might happen in the history, I guess it could go stale there, that could have been the last change ever. They could go and rerun those cells in order, the y equals and then the print y. Or they could run the print y straightaway ignoring the y equal step. Or I guess they could go off and do things entirely different and never touch those two cells. Are those like four categories? Is that aligned with your analysis? Or is that a proper way of looking at it?

[00:31:43] SM: No, that's a great way of looking at it. What we observed is more often than not, people would go back and run the y equals x plus one cell, and then the print y itself. It was not super common for people to skip the y equals x plus one cell and the print y cell. It was even less common for people to go straight to the print y cell. Based on our analysis. That was sort of the inspiration for why we you decided, it didn't go in that order, actually. It wasn't like we designed the highlights based on the study. It was more like we validated that the highlights were useful based on the study. But yeah, that's exactly right. More often than not, it looks to me, at least, like people want to rerun all of the cells in their dependency chains, and do so in the correct order.

[00:32:31] KP: Do you envision this as an educational tool, or something that an industry practitioner is using in their day to day work like a linter?

[00:32:39] SM: I think it has applications for both actually. One of the confusing aspects of notebooks for someone who's just learning how to program is, a lot of times people expect reactive semantics. So, when I run a cell, all of the dependent cell should automatically be rerun, kind of like a spreadsheet, for example. NBSafety doesn't have reactivity by default, but it does have opt in reactivity. So, that's one potential application, is to make it easier for beginners to kind of get the semantics that they expect. Let's see, there's actually a great Giulia notebook or Giulia kernel called Pluto that has this exact kind of reactive functionality.

The other cases, I do think that there are more traditional industry applications. I saw someone actually asking on Twitter once, is there a linter that can fail to validate my notebook, if I've run stuff out of order? I think this is one application. If you think about it, it would be fairly simple to write a linter that just fails to validate a notebook that has had some out of order execution, but it's so common to have out of order execution. That this is almost like overly restrictive. So, by actually understanding the dependency structure of the cells, and instead, failing to validate the notebook if that dependency structure has been violated somehow, I think that's a real possible application in industry. In general, I think it just makes it easier to reason about the notebook.

[00:34:11] KP: No, I think that's just in part evangelization. If this were built in, no one would complain, right?

[00:34:16] SM: I hope not. But it's hard to say.

[00:34:19] KP: Are there any similar projects that you're following?

[00:34:21] SM: Absolutely. So, there's a few different kernels notebook, kind of inspired pieces of software that I'm following very closely. I think I mentioned pluto.jl, which is a Giulia notebook that implements reactivity. A lot of people have probably heard of observable, which is a JavaScript notebook with reactivity. So, NBSafety, I mentioned doesn't have reactivity, but it kind of suggests the cells that would get rerun if you had reactive semantics. So, I consider it similar.

Then there's also this tool called Nodebook which uses memorization and serialization of the cell inputs and outputs to kind of enforce in order execution semantics in your notebooks so that you always get what you would expect to see if you run a notebook from top to bottom. Let's see, there's also another kernel called DF kernel for data flow notebooks. The main difference there is you need to manually specify the cell inputs and outputs, which is probably in some cases, what you want anyway. But yeah, there's a few different solutions already existing.

One I want to draw particular attention to is there's a software startup called Hex, which is doing some really cool stuff with reactive notebooks. They actually let you visualize the dependency structure of which cells depend on which other cells and that kind of eliminates surprises from reactive notebook semantics. So, you know exactly what's going to get re-executed when you change a cell and rerun it. But yeah, it's an exciting space and it's still very nascent. So, I'm sure we're going to see a lot of exciting developments in the future.

[00:36:06] KP: Well, it's interesting that there's a startup. Do you see a lot of commercialization opportunity here?

[00:36:10] SM: That's a great question. I have no idea. I've talked to the founder of this startup. And apparently, this is something that people want is these kinds of reactive notebook semantics. I'm sort of skeptical, which is why it's not on by default in NBSafety. I suspect that it's one of those features that people ask for, but then when you give it to them, maybe they're like, "Oh, this is actually kind of making it even more difficult to reason about." What do I know, right? Maybe people are asking for it, maybe this is really what people want.

So, for reactive semantics, in particular, maybe there's some commercialization. Certainly, there's a lot of commercialization opportunities in the notebook space in general, for various features, like collaboration, or integration with your data pipelines at your company, or, you know, built in SQL execution engines and that kind of stuff. Reactive semantics, in particular, not so sure about the ability to highlight cells to rerun, or cells that are unsafe to rerun. Also, not so sure about in terms of commercial viability. Certainly though, there's a lot of exciting features that are kind of being explored in the space of notebook startups. Maybe reactivity is one of them, maybe NBSafety type functionality is one of them. But I maybe don't have enough courage to do that experiment. We'll see.

[00:37:30] KP: Well, is NBSafety effectively a completed project, or do you have a roadmap for it?

[00:37:36] SM: It is definitely under active development. It is usable. It's, I would say, kind of pre alpha to alpha level stability. I do have a roadmap contains entirely in my brain.

[00:37:52] KP: Well, let's get it in podcast form.

[00:37:54] SM: Sure. Okay. I mentioned that reactivity is an opt in feature. This is actually not documented at the moment, the most pressing matter is to like update the documentation to show how to turn this on, and show how to turn on and off various other features. So that's like the hot priority on the NBSafety roadmap is improve the documentation.

Past that, if there are cases where you might not want to opt in to reactivity globally, but you might want to do it on like a symbol by symbol basis. So, one of the things I'm working on now is let's say I prefix a variable with dollar sign. Anytime that variable updates, all of the cells that reference that variable with dollar sign in front of it will automatically re-execute. That's on the roadmap. It hasn't been implemented yet. That's one thing I'm very excited about is kind of getting these kinds of spreadsheet style variable references into NBSafety.

Let's see what else, the other thing that I'm really excited about is because we have all of this dynamic data flow information, we can actually construct really high fidelity and accurate dynamic program slices. So, let's say I have a cell, I've been messing around in my notebook for a really long time, I have all of this cruft that's accumulated. But I finally have the one cell that shows the output that I care about and I just want to figure out what is the code that I need to reconstruct the output for this one cell. So, we have a really nice backward slicer that can actually look at all of this data flow information and try and get the minimal programs sliced necessary to reconstruct this output. And the idea is to kind of bridge that gap from notebook code to production code a little more easily.

There's a similar project actually called nbgather, which does this using static data flow, but when you have this kind of dynamic data flow information available, it turns out that you can make the slices kind of smaller and more accurate.

[00:39:56] KP: Stephen, where's the best place for people to keep up with the project if they want to Install and follow the future of NBSafety?

[00:40:03] SM: Oh, that's a good question. Probably needs to do a better job advertising and refreshing the documentation. But let's see, the website is nbsafety.org. It hasn't been updated in a while, but I'm sure, try and do a better job of keeping it up to date. Also, I mean, the unfortunate reality is if you do want like the most up to date stuff on this project, probably the best thing to do is to just follow my Twitter, which is Stephen_Macke. I'll post future updates on there.

[00:40:36] KP: Well, Stephen, thank you so much for sharing your project and taking the time to come on Software Engineering Daily.

[00:40:42] SM: Well, it's been an absolute pleasure, Kyle. I was honestly really astounded when I got contacted in the first place. It's quite an honor to be here and thank you so much for taking the time to chat with me about this. It's something I'm very excited about and it's been a pleasure to share it.

[00:40:57] KP: Same here.

[END]