Jonni Bidwell learns all about Mozilla's recently released language with veteran Rustafarian **Jim Blandy**.

Diamonds and Rust

Jim Blandy (aka jimb) cut his teeth working on *Emacs*, The GNU Project Debugger (gdb) and various other bits of the GNU Project. He's a founder of Red Bean, a company set up to never to make any money, but also to never go away. These days he's a software developer at Mozilla, and an ardent Rust proponent. We caught up with him at OSCON to learn all about the new language, as well as the idiosyncrasies of the old ones. Plus the various challenges of teaching a new generation of rustaceans.

Linux Format: So Rust 1.0 (this interview took place in July 2015) has just been released. I have a sort of pointed question to begin with: All these new program languages from major players – Go, Swift, Rust – why do we need them?

Jim Blandy: That's a great question. Especially when you're learning programming languages and once you get enough experience you realise that, for the most part, they're all pretty much the same. Then once you've got the gist of one you can pick up another pretty quickly. That's why languages like Haskell are a real joy because they're definitely not easy to learn. Prolog's a good one too, because you can run your programs backwards.

LXF: Even as a mathematician I don't really believe functional programming works. I still don't understand monads.

JB: They're just monoids in the category of endofunctors.

LXF: Thanks, that really clears that up. Let's go back to Rust.

JB: The defining characteristics of C and C++ are their attitude towards undefined behaviour. In a language like Python, if you reference an element off the end of an array, it throws an exception. That exception is described in the documentation, it says that that's what happens. So even when you do bad things, the language specifies what the response is.

JavaScript is much the same. In a sense, those languages try to be total: every program that you could possibly give them has some meaning – it might not be useful, it might just be throwing an error – but everything has meaning. In C and C++, what they say is "Well there's some errors that we can detect efficiently or at compile time and we'll tell you about those. But basically everything that would cost us even the smallest amount of overhead to detect is up to you to avoid." In fact, if your program does any of these undefined things the compiler is within its rights to produce a program that does anything at all. So the demo that I opened up my talk with is a simple three-line C program that declares an array and one element, assigns a value to its third element (which it doesn't have) and it returns. When you run this it displays a weird error message saying that your password is exposed and things like that. What's happened is that the program has overwritten the return address for main() so when main() returns it dumps into some poor little C library and it just falls apart.

LXF: And that's not due to any bug in the compiler or anything?

JB: Nope. This is completely legitimate behaviour according to C and C++; basically their attitude is that it's the programmer's responsibility to avoid undefined behaviour. We now have 30 years of history testifying how well that works out. In 1988, the Morris virus exploited a buffer overrun to break into people's computers through the finger protocol. Since then, there's been a steady flow of those kinds of exploits. If you look at the open-source vulnerability database they have a little chart up there and it's a consistent 10%. These days we have SQL injections and PHP, so there's a lot of competition, but they just keep coming and it's not a surprise: The jury is in, the experiment has been run, humans can't write that code, they can't be trusted.

The Google security blog had a post recently about some integer-size based vulnerabilities. Even the innocent things like 'Well, I'm just gonna cast this 32-bit integer into a 16-bit integer, I don't wanna waste time just take the bottom 16 bits, I know what I'm doing'. You don't. Or rather, you might, but the frequency with which you don't is often enough that we power the Russian mafia. It's bad.

So what we've got is the situation where the systems programming languages can be trusted with low-level stuff, kernels, crypto and implementing VMs for other languages. All those systems languages are unsafe, it's up to you to do this thing that humans can't do.

ON 30 YEARS OF BUFFER OVERFLOWS

"The jury is in, the experiment has been run, humans can't write that code…"

Basically every other language is well-defined: Python, JavaScript, Haskell, Ruby and everything else tries to be complete. So there's this weird dichotomy where the languages we trust with all kinds of untrusted data are the ones where it's a sword dance on an ice skating rink. Rust exists to bridge that chasm. It's a systems programming language where it tells you when you break the rules. In Rust, when I declare a structure with two 32-bit integers, that is a 64-bit value with nothing else. It's just those two words, there's no metadata or dynamic anything, it's just a simple data structure. We have implemented a garbage collector for »

» Servo (a project to port *Firefox* to Rust), but that's not part of the language itself. When you write a Rust program you know exactly when each value gets freed, you don't have to wait for a garbage collector to know when it's gone.

So the storage management is very deterministic and easy to control, data representations are simple and direct – they're just exactly what the machine needs to do to represent those values, and operations that look cheap in the code are cheap. In C++ when you do an assignment, if that assignment happens to be a vector, that's copying that vector over to the destination. And if that vector happens to be a vector of strings, it's copying each string. So you can end up accidentally writing code that is incredibly inefficient, it's allocating vast amounts of memory. This isn't usually a problem, but it's not a characteristic you'd like in a systems programming language whose whole selling point is that it gives you control over the machine.

LXF: What is it that Rust does differently?
JB: Rust takes a different approach to those things. It uses moves for big expensive values: assignment will move a value from the source to the destination and then leave the source de-initialised. The consequence of that is when you have a big structure like a vector or a hash table, at any given time that value has exactly one owner. You can pass it to a function and then maybe the function takes ownership of it, but you the caller have lost access to it – you moved the owner, but there is only ever one. You could take that big value and store it in another table, now that other table owns it and again you've lost access to it. Having only one owner makes it very clear when that value is going to go away. That's basically Rust's storage management story.

Of course, it's very restrictive to have only one owner, there's a reason why people write ownership-ambiguous programs the way they do. So Rust has a thing called borrowed pointers, which means that you're using something for a little while, but you're going to give it back to its owner eventually. A borrowed pointer gives you access to a value without changing its ownership. You can compute on it or modify it, but you have to give up your borrowed references to it in a way that the compiler can tell that you're doing it. The compiler has to be able to see that all of your borrows end at a well-defined time. So there's two kinds of borrows, you can have shared references, you can have millions of those ones, you can hand them out to everyone so long as they all come back—they all have to expire. Or you can have a mutable borrow, you're only allowed one of these, it's a multiple reader single writer pattern. If you have a mutable borrow then that's the only thing that can access that item at all, you can't even call it by its original variable name. The mutable borrow is now the sole point of access to the underlying object.

By strictly segregating access that can be shared from access that can mutate, Rust can actually prove at compile time that you never have share and mutation at the same time. If you read the Java spec and look at the hash table interface, there is a rule that says if you are iterating over a hash table, the only thing that's allowed to modify that hash table is that iterator. If somebody else modifies that hash table your iterator will throw an exception.

In classic C++ style their response to that situation is if you ever modify this hash table while you've got iterators on it all of those iterators are invalid, it's undefined behaviour – it just throws it back in your face and you're responsible for maintaining this whole program invariant, which as I've said before you can't be trusted with. So at least Java throws an exception, and other languages recognise this too. Rust goes one better than Java, where it will tell you at compile time that it's possible that this situation could arise.

So we're throwing away a lot of programs that you could write in other languages, and some of those programs will be fine and correct. But the nature of any static analysis, any analysis that happens at compile time and doesn't have the running program to look at, is that if it permits all correct programs, if it allows you to write all the programs that are actually OK, then it must also permit some unsafe programs. You can't exactly match the boundary between the OK programs and the not-OK programs – it's not computable. So the Rust analysis is conservative, it rejects correct programs and it always will, but it turns out that it's not that bad. Once you get used to it and the way it's seeing the world, it's actually entirely comfortable and it doesn't really forbid you from writing much at all.

LXF: I guess there's an analogy here with Gödel's Incompleteness theorem here, in that you can have completeness or consistency, but not both.
JB: That is exactly what it is. But the borrow checker is something that will improve over time, any time that we can see a way to improve that analysis so that it will allow more correct programs then we'll do that, but we have to be sure that it's sound. The experience of working in Rust is freaking amazing, though. Once you get past that, I should say learning curve but it's more like a period of suffering, right, maybe purgatory is a better word. Anyway, you have to climb that mountain for a few weeks, but once you get there the view is good. A friend of mine came to me showed me this algorithm to insert a value into a binary tree, he said 'I haven't checked it, but it doesn't crash'. And you see this code, it's finding values and replacing nodes and walking down and things like that. So in C or C++ you'd be thinking: 'There's gotta be a dangling pointer here somewhere', you'd have to really read it thoroughly to check.

LXF: What about multithreaded programs?
JB: There it's really exciting. Matthias Felleisen – who is a professor at Northeastern University,

he's one of the big people in the PLT group that produced Racket, DrScheme and a bunch of other things, like a really cool functional-reactive system called Father Time. His other big focus is computer science pedagogy and he's done a lot of work in how to actually teach Computer Science to people so that they get it. He was in a panel just two weeks ago with Gilad Bracha which was about 'What should we be studying, where's computer science going?'. There was a lot of dumping on types. Anyway Matthias told this story about a class that he taught: He took a whole bunch of students that had never written parallel code before and he taught them how to do just that in Rust. Matthias is a serious iconoclast, he's so mean, so he'll put down anything that he doesn't think is good—he's sincere, but he's brutal. He said that people found it difficult for the first two weeks because progress was slow and students found the error message confusing. But after those two weeks they didn't have any problems, their programs compiled and ran and did exactly what they were supposed to do. That I find is really exciting. The idea that you can use concurrency as a technique of first resort instead of a technique of last resort. Which is the polar opposite of how we do things now, where you optimise your single-threaded code to within an inch of its life and then, when you can't squeeze out another second, then turn to concurrency.

LXF: Quite often people like to divide programming languages into fast languages and slow ones. The fast ones are C, C++ and Java, and the slow ones seem to be everything else. On which side does Rust live?

ON RUST'S THREAD-SAFETY

"You can use concurrency as a technique of first resort instead of last resort"

JB: Rust is a fast language. Obviously we're building on all the amazing work that the Clang people have done. Clang itself is a great front-end, and any C++ front-end that's usable is an accomplishment, but Clang is especially good and it is supported by the optimisation infrastructure that LLVM has. Not only that, the LLVM people are just fanatics about software architecture. It would be very easy to have a back-end that's specific to your front-end, I think that's probably a common case, but LLVM is very nicely isolated, which means that languages like Rust benefit from all that work.

LXF: How hard would it be for an amateur programmer, say someone familiar with things like Python and PHP, to pick up Rust?

JB: It's tricky to say, my sort of workflow looks like this. So I start a Rust program, it looks really simple and clean but it doesn't compile. Then I argue with the compiler for fifteen minutes or so and it goes through a process of being really hairy and code starts to look awkward. Then I realise that I've rearranged things so I start to take some of this complexity out and picking the hair out, and then I'm done and it's actually beautiful. So the end result often looks like a nice Python program. It's like I just made a dictionary and then stuck some stuff in a dictionary and then I iterated over it and got it back out. It does type inference so you only have to use types at function boundaries, you don't have to use them inside functions, generally, it just figures them out.

So in that sense I think it'd be great for people coming from the dynamic languages. At the same time I would worry about that intermediate step where everything's in pieces on the floor. Getting through that I'm using everything that I know, and that might be a sizeable obstacle for the less experienced. The difficulty of learning the language might turn out to be Rust's biggest weakness. For all the C++ meta-template programmers out there and the Haskell hackers I think it'll be no problem, but that's a very small population.

I am working on how best to explain how Rust works. In the presentation yesterday there were some parts that didn't go so well, but the parts that did go well were the hardest parts. In particular the ownership moving and borrowing step, I think I found a good way to explain that. There will be a book coming out with O'Reilly at the end of this year, so if all goes well, hopefully it will have solid explanations.

What Rust shares with Python is safety, when you're writing a Python program you don't get weird corruption where the system starts to behave in strange ways you can't understand, at worst you get an exception. It's very friendly, it just tells you what went wrong. And Rust shares that quality, it tells you what's going on—you don't end up in Tombolia. That phrase comes from Gödel, Escher, Bach, where he says 'You've violated the rules and suddenly you're in Tombolia and you have no idea what anything means anymore'. So there's no Tombolia in Rust. That I think will be very welcoming to people. Thinking about types and writing them out, some people already think that way and they'll be fine. Some people never think that way, I don't know how they program, but they're going to find it difficult.

The great thing is that although it's a low-level, close to the metal language, you don't end up worrying about the bits and bytes. It's not like you think, 'Oh, I've overflowed and now my size is wrong and I've crashed the program'. You get exceptions when you convert a 64-bit value to a 32-bit value and it doesn't fit. So in some circumstances it will be welcoming, and in other senses there will be serious challenges. I don't want to say something's hard, as a flat statement, because it's about how everything's taught, so we just have to wait for there to be good teachers, that's what I want to do. LXF