Contracts for Rust

Felix S Klock II

he/they

We have some problems

Problems

Too much invention: Every Rust verification tool invents its own contract dialect

Safe code is not the safe bet: Verification tools often focus on safe code alone, but validating/verifying Rust's unsafe code is critically important!

No project support: Neither Rust language nor std library offer formal contracts

- Workaround: Tool supplies an alternative std lib
- Workaround: Inline the contract-free code *as* the specified behavior (e.g. Kani)
- Workaround: Attach contracts in post-hoc fashion to existing std lib

Problems

Too much invention

Safe code is not the safe bet

No project support

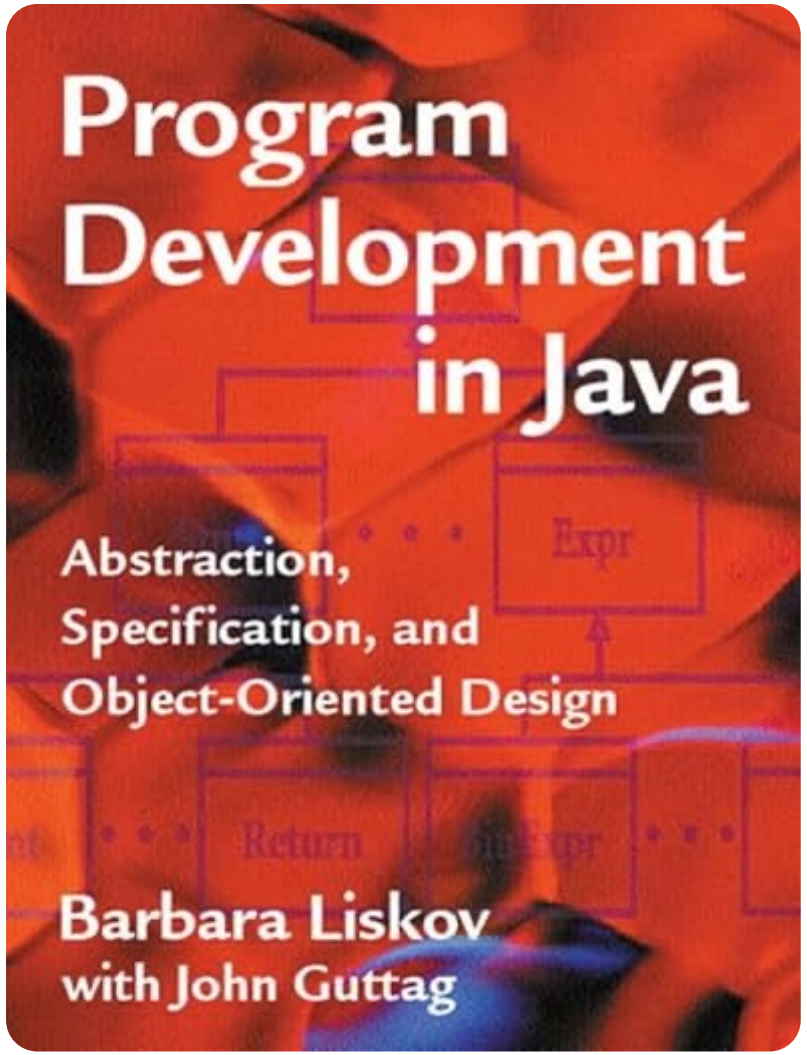
The dream: resolve all of these via one common contract language provided in the Rust project itself

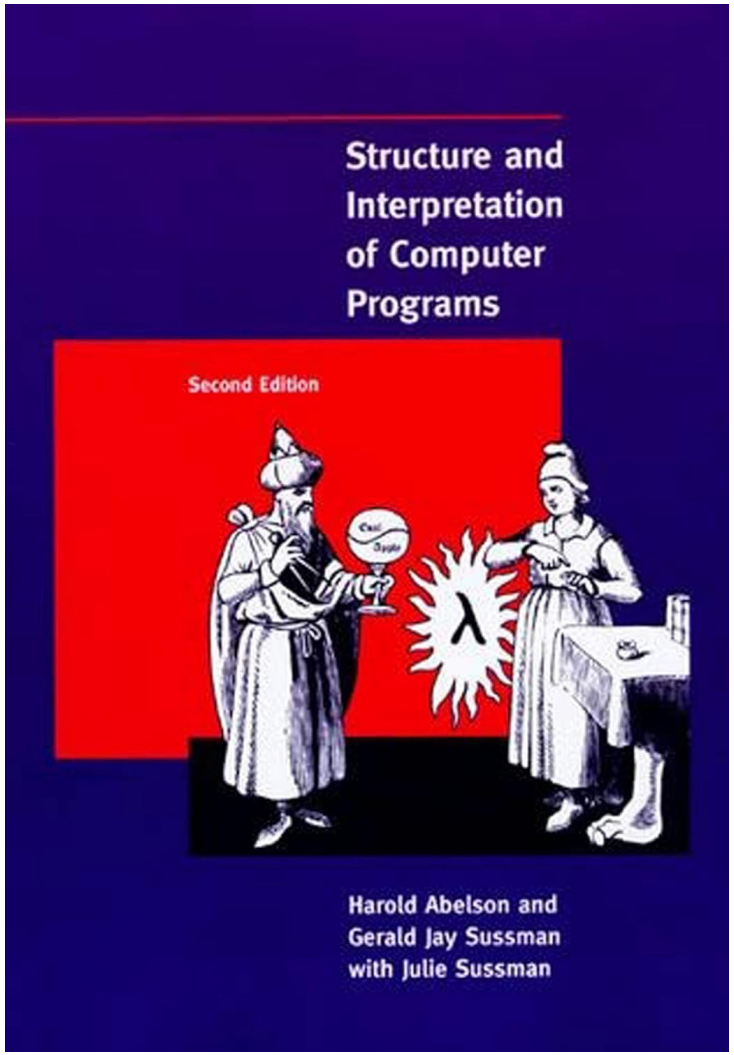
How do we get there?

What does my *fantasy* contract system aim to accomplish?

My weird background

My weird background





Background

Static Rules

- Long fascination with static reasoning
- Java ESC (now JML)
- Type systems (Haskell, FX)
- Model checking
- Proof construction+search+checking; ACL2

Background

Static Rules

- Long fascination with static reasoning
- Java ESC (now JML)
- Type systems (Haskell, FX)
- Model checking
- Proof construction+search+checking; ACL2

versus Dynamic Power

- First "real" PL was *Scheme*
- Grad school at Northeastern
- Lots of exposure to Racket Contracts a la Findler+Felleisen

Problems (revisited)

Too much invention

Safe code is not the safe bet

No project support

The dream: resolve all of these via one common contract language provided in the Rust project itself

How do we get there?

Contracts for Rust: How to get there, *together*?

Contracts for Rust:
First, establish shared values

Contracts for Rust:
First, establish shared values

If you disagree with something, note it!

Let's all argue after I'm done talking!

Tenet 1: The purpose of contracts is …

Specification Mechanism, first

- Design by Contract
- pre + post + frame conditions (aka “requires”, “ensures”, “modifies”)

Tenet 1: The purpose of contracts is …

Specification Mechanism, first

- Design by Contract
- pre + post + frame conditions (aka “requires”, “ensures”, “modifies”)

Verification Mechanism, second

- Encode formal correctness arguments
- Representation (aka Type) Invariants
- Loop invariants
- Termination measure (aka “decrementing/decreasing function”)

Tenet 2: (Semi) Useful out-of-the-box

Anyone can eat

- Can turn on "contract checking" without changing toolchain nor installing 3rd party tool, and get *some* utility.
  - Why? Because: without above, Rust Project unlikely to adopt *any* contracts in lang + stdlib.
- Caveat: Contracts might become *more useful* when used in concert with 3rd party automated reasoning tools.

Anyone can cook

- Can add contracts to your own code without changing toolchain.

Contracts should validate code even without awesome static verification technology

An immediate implication of 2nd tenet

Require some utility without other tool

⇒

Majority of contracts *must* have some dynamic interpretation

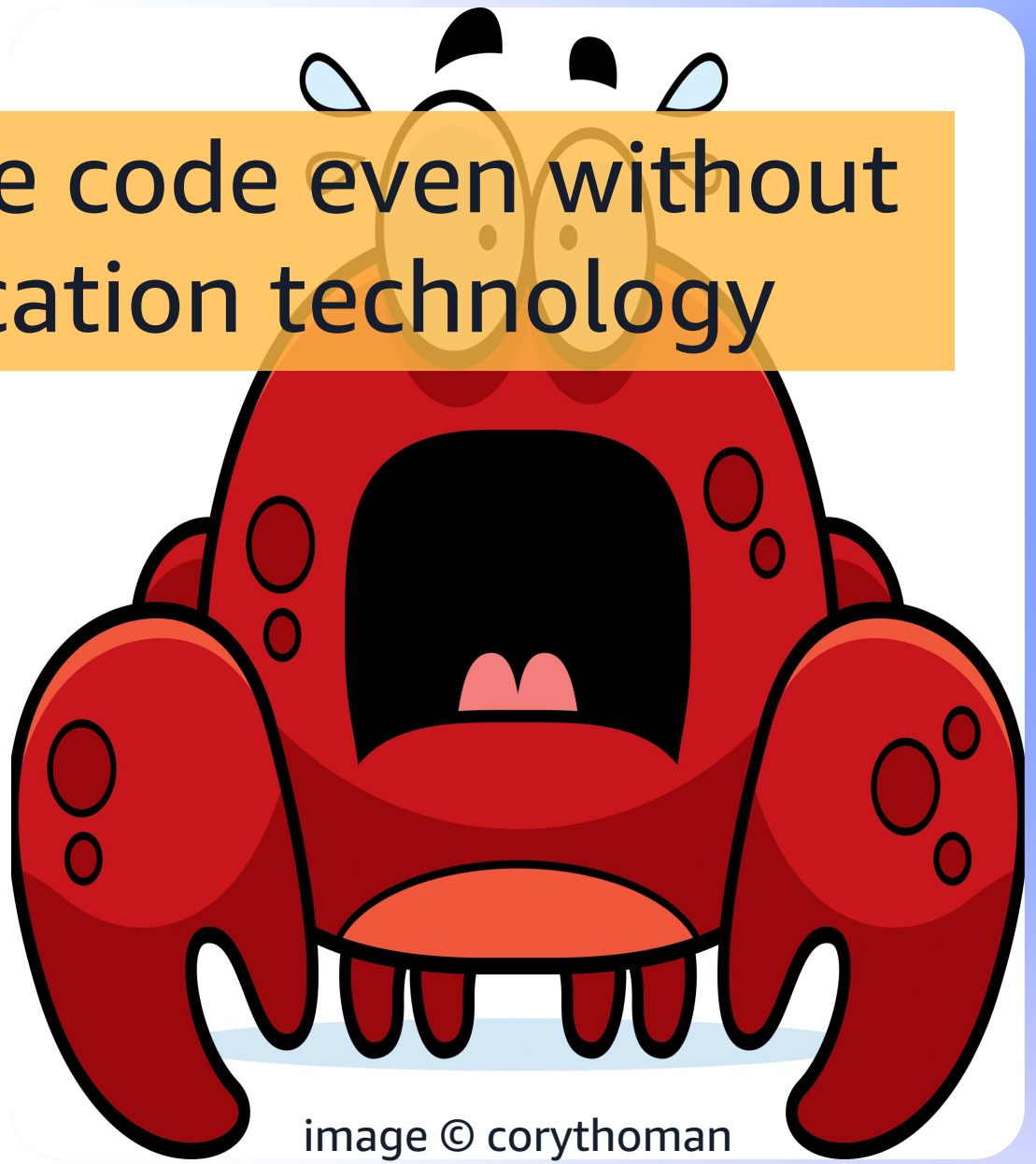


image © corythoman
used with permission via Adobe Stock

Tenet 3: Contracts are not just assertions

Dynamic semantics

- Contracts enable modular reasoning
- A broken contract identifies *which component* is at fault.
- Precise blame assignment becomes non-trivial with higher-order functions (aka OOP, Traits, dynamic dispatch, etc)

Static semantics

- Contracts enable modular reasoning
- Instead of reasoning about F(G), a contract allows independent proofs for F(□) and G.

Tenet 4: Balance accessibility over power

Accessible

- Rust contracts should strive for a syntax that is, or closely matches, the syntax of Rust code itself
- Any variation is potential hurdle to use and adoption
- Changes to syntax *or semantics* must meet high bar

Expressive Power

- Contracts may *need* forms that are not valid Rust code
- E.g. forall x: Type { pred(x) }
  - Of course, `forall(|x: Type| { pred(x) })` is valid syntax adopted by many tools.
- Unavailable to executables (intentionally)
- E.g. May want intrinisic predicates that can query memory model internals (such contracts would be similarly restricted to miri)

Tenet 5: Accept Incompleteness

Dynamic Limitations

- Not all properties of interest can be fully checked at runtime
- E.g. forall a, b: Integer, a + b = b + a
- Devise useful approximations!

Static Limitations

- Full functional correctness specs still often lie outside realm of economic feasibility.
- An impoverished contract system may still be useful for specifying more conservative functional properties (e.g. invariant maintenance, memory safety, panic-freedom, decrementing functions).

Tenet 6: Embrace tool diversity

- Different static verification systems will require or support differing levels of linguistic expressiveness.
- Same holds for dynamic checking!
  - E.g. injecting assertions into object code (versus miri or valgrind)
- An ideal contract system needs to account for this in some way
- e.g. perhaps by allowing third-party tools to swap in different contracts (with more expressive formulae) attached to std library procedures.

Tenet 7: Verification cannot be bolted on, but…

- In general, code must be written with verification in mind as one of its design criteria.
- We cannot expect to add contracts to arbitrary code and be able to get it to pass a static verifier.
- This does not imply that contracts must be useless for arbitrary code.
  - Dynamic contract checks have proven useful for the Racket community.
  - Racket development style: add more contracts to the code when debugging (including, but not limited to, contract failures)
  - A *validation* mechanism *can* be bolted-on after the fact.

Tenets, repeated

1. Specification mechanism first; Verification mechanism second
2. (Semi) Useful out of the box: Anyone can eat, and Anyone can cook
3. Contracts are not just assertions: contracts enable modular reasoning
4. Balance accessibility over power
5. Accept Incompleteness
6. Embrace tool diversity
7. Verification cannot be bolted on, but… validation ≠ verification

Even if we all agreed, where would this leave us?

Nagging questions; naïve ideas

Nag: How will stuff this help my tool again?

Answer: Once we have a contract language built into rustc, we can include its expressions as part of the compilation pipeline, turning them into HIR, THIR, MIR, *et cetera*.

> For example, we could add contract-specific intrinsics that map to new MIR instructions. Then tools can decide to interpret those instructions. rustc, on its own, can decide whether it wants to map them to LLVM, or into valgrind calls, *et cetera*.
>
> (Or compiler could throw them away; but: unused = untested = unmaintained)

This ties into the Stable-MIR project; stay tuned for the talk tomorrow.

Nag: Dynamically check arbitrary contracts?

Example: a dynamic `forall(|x:T| { … })` sounds problematic for most T of interest

Potential solution: `forall!(|x:T| suchas: [x_expr1, x_expr2, …] { … })`

(Static tools can ignore the sample population, and dynamic tools can use them directly, or feed them into a fuzzer, etc)

Nag: Isn’t proper blame hard?

Answer: Contracts with proper blame, as implemented in Racket, can be very expensive. (Source: “Is Sound Gradual Typing Dead?”, Takikawa et al., POPL 2016)

But: Do not have to implement blame the same way.

More importantly: Do not have to provide strong blame guarantees out-of-the-box for contracts to be *useful*.

I just want proper blame assignment in back of our collective mind.

Nag: I want math without bounds!

Example: Some specifications benefit from using constructs like unbounded integers, or sequences, or sets. (Especially important for devising abstraction functions/relations to describe the meaning of a given type.)

Is this in conflict with “Balance accessibility over power”?

Answer A: Indeed, Rust ≠ Haskell..

Nag: I want math without bounds!

Example: Some specifications benefit from using constructs like unbounded integers, or sequences, or sets. (Especially important for devising abstraction functions/relations to describe the meaning of a given type.)

Is this in conflict with “Balance accessibility over power”?

Answer B: Two main problems to resolve:

1. Dynamic interpretation may incur unacceptably high overhead
2. Freely copying terms *is* useful.

Na

Exa
inte
fun

Is th

Ans

1. I
2. Freely copying terms *is* useful.

Imagine a spec for `Vec::push`.

How do you *dynamically* check a generic spec for `v.push(x)` …

e.g. ⌈post(v)⌉ = ⌈pre(v)⌉ ⧺ [x]

… without copying `x`?

(This is my personal independent justification for the choice of Verus and Pearlite to *allow* free copying in their spec functions!)

Nag: I want math without bounds!

Example: Some specifications benefit from using constructs like unbounded integers, or sequences, or sets. (Especially important for devising abstraction functions/relations to describe the meaning of a given type.)

Is this in conflict with “Balance accessibility over power”?

Answer B: Two main problems to resolve:

1. Dynamic interpretation may incur unacceptably high overhead.
2. Freely copying terms *is* useful.

… maybe *some* forms simply cannot be interpreted via the Rust abstract machine

Nag: What about unsafe code?

Way back on slide 3, we said:

> *Safe code is not the safe bet*: Verification tools often focus on safe code alone, but validating/verifying Rust's unsafe code is more important!

I don't know the complete answer here.

Some dynamic checks would benefit from access to memory model internals.

But in general, checking the correctness of an unsafe abstraction *needs* type-specific ghost state (to model permissions, etc). I'm leaving this for future work!

Nag: Is it embracing tools, or keeping distance?

The Embrace Tool Diversity discussion noted

> “e.g. perhaps by allowing third-party tools to swap in different contracts (with more expressive formulae) attached to std library procedures.”

but at the outset that was already discounted as a mere “workaround”:

> “Workaround: Attach contracts in post-hoc fashion to existing std lib”

Is above embracing those tools? Or merely blessing existing practice, reluctantly?

Answer: Allowing some contracts to be swapped in is better than forcing them all to be specified in that manner. My hope is for Rust project to work with verification tool community to ensure most of our contracts are useful to you.

Concluding thoughts

Get those notes ready!

Tenets, repeated

1. Specification mechanism first; Verification mechanism second
2. (Semi) Useful out of the box: Anyone can eat, and Anyone can cook
3. Contracts are not just assertions: contracts enable modular reasoning
4. Balance accessibility over power
5. Accept Incompleteness
6. Embrace tool diversity
7. Verification cannot be bolted on, but… validation ≠ verification

Thanks for listening