Skeletal Semantics

Martin Bodin[1] Philippa Gardner[1] Thomas Jensen[2] Alan Schmitt[2]
[1] Imperial College London [2] Inria

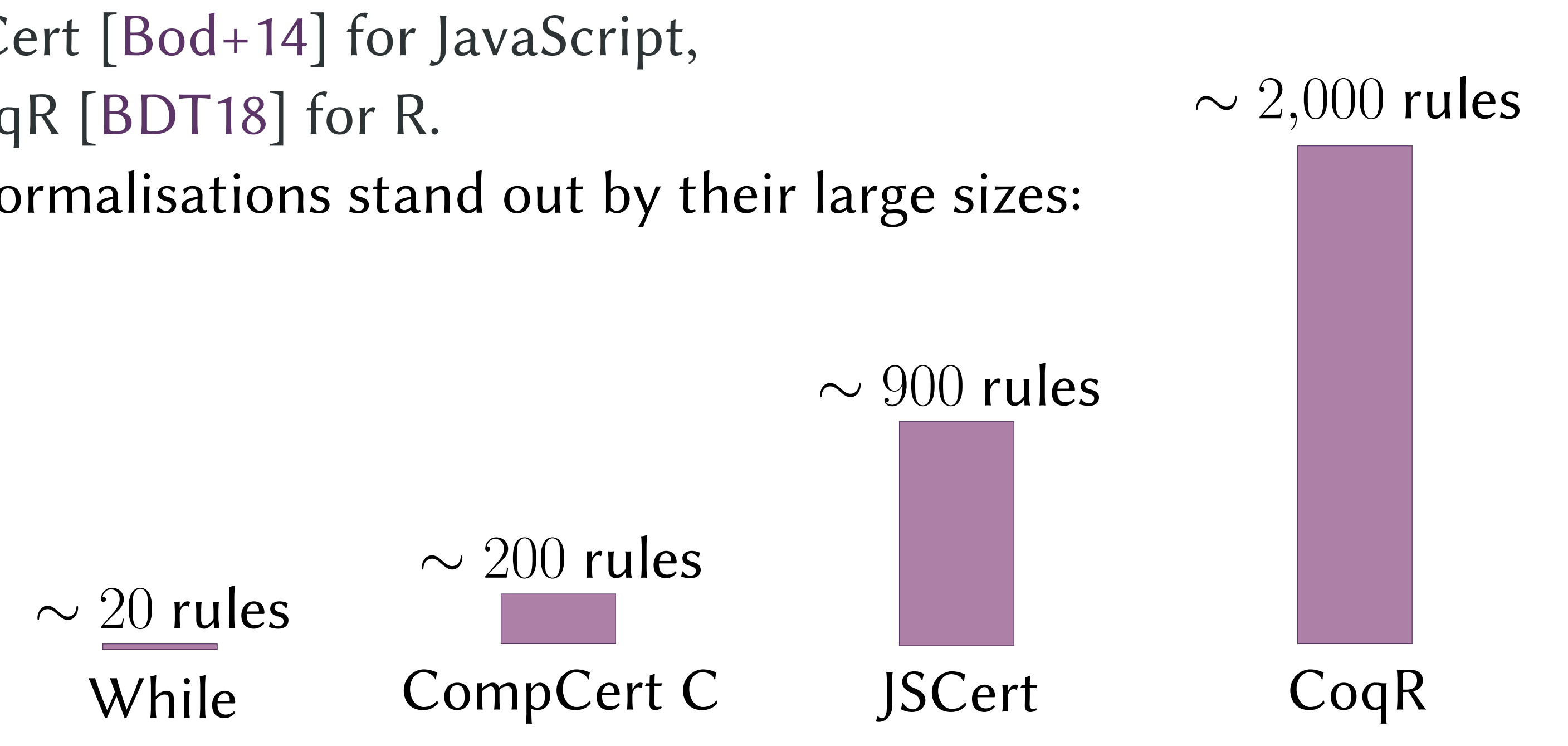
This research has been partially supported by the ANR projects AJACS ANR-14-CE28-0008 and CISC ANR-17-CE25-0014-01, as well as the EPSRC programme grant 'REMS: Rigorous Engineering of Mainstream Systems', EP/K008528/1.

Motivation

A lot of large-scale *Coq formalisations of real-world languages* have been published in the last years. Instances include:

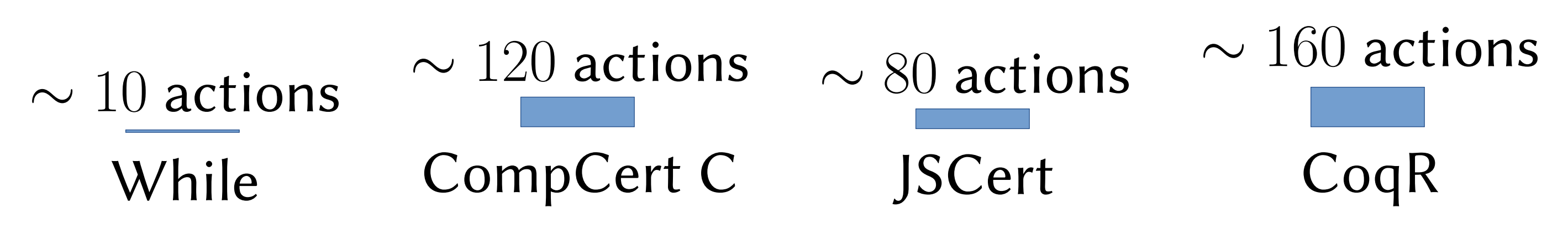
- *CH_2O* [KW11] for C (which can be linked to CompCert [Ler09; KLW14]),
- JSCert [Bod+14] for JavaScript,
- CoqR [BDT18] for R.

These formalisations stand out by their large sizes:



These formalisations are written in different styles and are thus difficult to compare. This graph tries to compensate for that but it is to be taken with care.

The size of these formalisations is however not due to the complexity of the objects that they manipulate. This complexity can be measured by the number of basic actions used in the language's specification, pictured below.



Instead, the large sizes of these formalisations come from *semantic exceptions*: special behaviours for some values, leading to additional rules in the formalisation. For instance, most JavaScript operations on numbers are actually defined on all sorts of objects, not only numbers. Convoluted heuristics convert these objects to get a result, and these heuristics are fully part of the language specification, and thus of its formalisation. There is for instance no string involved in the following example, yet changing `toString` has an effect somehow:

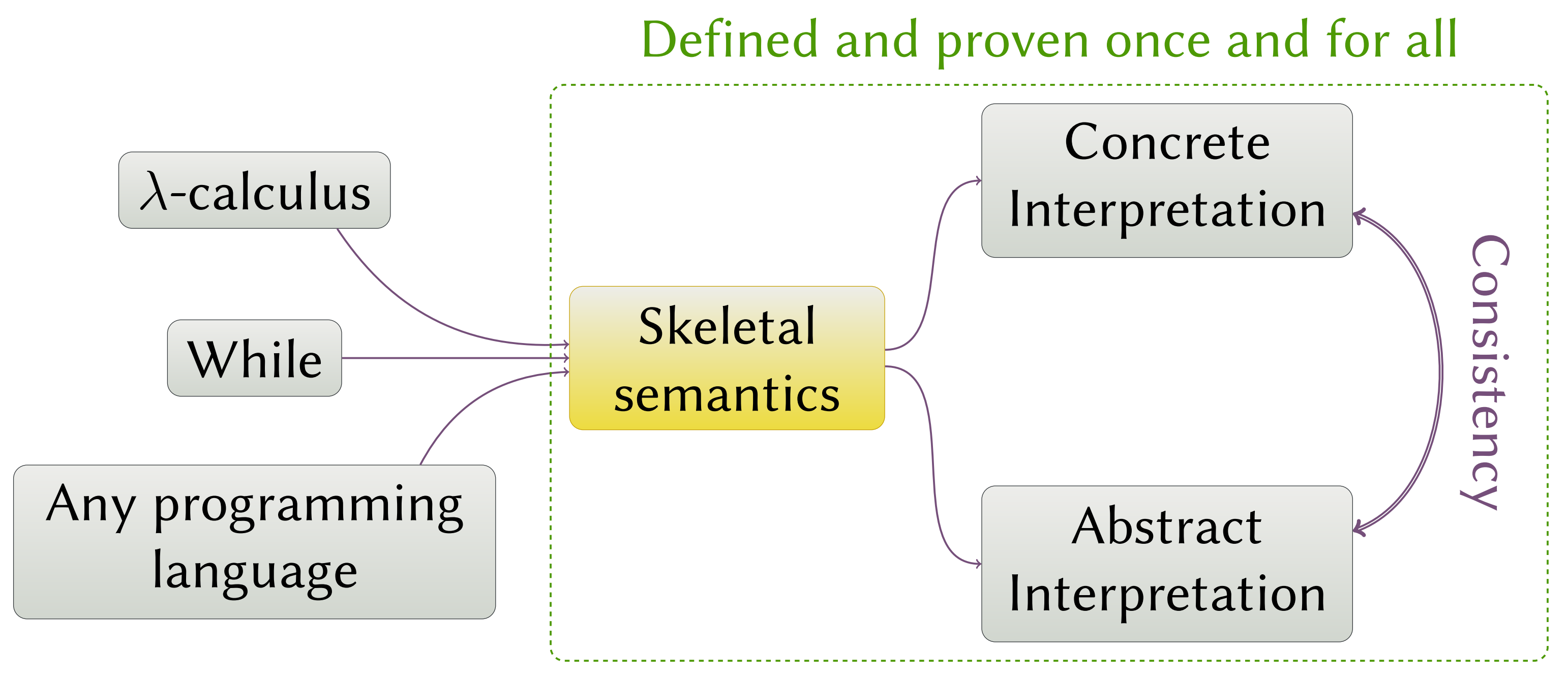
```
console.log (+[18]) ; // Prints 18.
Array.prototype.toString = function (){ return 42 } ;
console.log (+[18]) // Prints 42.
```

The size of these semantics has two implications:

- Mistakes are easy to be made when writing an interpreter or an analyser for these languages. There is thus a *need to formally prove such programs.*
- Such proofs are themselves impractical to be built as they are based on huge language specifications.

Our Solution

Our solution is to step away from the corner-cases of the specification by *formalising the specification language itself.* Interpreters and analysers can then be written and proven *generically*, for any such specification. These generic interpreters, analysers, and proofs rely on the language-specific parts: basic actions. These actions have to be implemented for each language.



We first formalised how JSCert is specified [BJS15], then generalised it: *skeletal semantics* and skeletons [Bod+19] are now much simpler and more general. Skeletons have been designed to facilitate the definition of interpreters, analysers, but also semantics. Skeletons are indeed associated with a general notion of *interpretation*, of which the usual concrete and abstract interpretations are instances. We were able to relate the concrete and abstract interpretations *in general*, only assuming small lemmas about each basic action.

Skeletons thus enable users to only focus on what matters: how computations are performed at the level of basic actions. All the semantic exceptions are faded away at this level, leading to much simpler lemmas to prove to get a particular theorem about the considered programming language.

Skeletons also enable formalisations to scale with specification changes: only lemmas about basic actions that have been changed need to be proven again to get a result about the new specification.

Skeletons and Skeletal Semantics

Skeletal Semantics [Bod+19] is a format to specify the semantics of a programming language. A skeletal semantics is guided by syntax: a skeleton is defined for each term constructor of the language. A skeleton is a sequence of declarations involving either (language-specific) basic actions, inductive calls to the skeletal semantics, or non-deterministic branchings. This enables to express a wide range of programming languages. Here follows an example of skeletal semantics for a small While language. (See full example in the artefact.)

```
type stmt =
| Skip
| Assign of ident * expr
| While of expr * stmt
```

Declaration of terms and their constructors.

```
val write : ident * value * state -> state
val isTrue : bool -> unit
val isFalse : bool -> unit
val isBool : value -> bool
```

Declaration of basic actions. Note that some are partial.

```
eval (s : state) : stmt -> state =
| Skip -> s
| Assign (x, e) ->
  let v = eval s e in
  write x v s
| While (e, t) ->
  let v = eval s e in
  let b = isBool v in
  branch
    let () = isTrue b in
    let s' = eval s t in
    eval s' (While (e, t))
  or
    let () = isFalse b in s
```

Declaration of the skeletal semantics of statements. (`eval (s : state) : stmt -> state =`)

Inductive call to the skeletal semantics. (`let v = eval s e in`)

Call to a basic action. (`let b = isBool v in`)

Non-deterministical branching. (In this particular case, it is actually deterministic thanks to the partiality of `isTrue` and `isFalse`.)

This skeletal semantics looks like an interpreter, but it is not. Indeed, one can extract from this skeletal semantics *both a concrete and an abstract semantics.* More generally, a wide range of interpretations can be derived from a skeletal semantics. Crucially, given a set of basic actions, these interpretations are defined and related one with each other *once and for all.* This makes skeletons scale with large specification sizes, but also with specification changes.

The interpretations that we provide in our paper [Bod+19] are only examples. The *simplicity of skeletons* makes it easy to define new interpretations, forcing us to focus on the interesting parts (typically the memory model and its associated operations) instead of being lost in a sea of semantic exceptions. Future works include exploring specific interpretations for separation logic, certified compilation, etc., as well as building real-world skeletal semantics.

Artefact

The artefact can be found at `skeletons.inria.fr`:

- *Coq formalisation* of skeletons,
- *Proof of correctness* of the abstract interpretation with respect to the concrete interpretation,
- *OCaml tool* to manipulate skeletons,
- *Various examples* of skeletal semantics.

References

[Bod+19] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. Skeletal Semantics and their Interpretations. *POPL.* 2019.

[BDT18] Martin Bodin, Tomás Diaz, and Éric Tanter. A Trustworthy Mechanized Formalization of R. *DLS.* 2018.

[BJS15] Martin Bodin, Thomas Jensen, and Alan Schmitt. Certified Abstract Interpretation with Pretty-Big-Step Semantics. *CPP.* 2015.

[Bod+14] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification. *POPL.* 2014.

[KLW14] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. Formal C semantics: CompCert and the C standard. *ITP.* 2014.

[KW11] Robbert Krebbers and Freek Wiedijk. A Formalization of the C99 Standard in HOL, Isabelle and Coq. *Calculemus/MKM.* 2011.

[Ler09] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM* (2009).