IRDL: An IR Definition Language for SSA Compilers

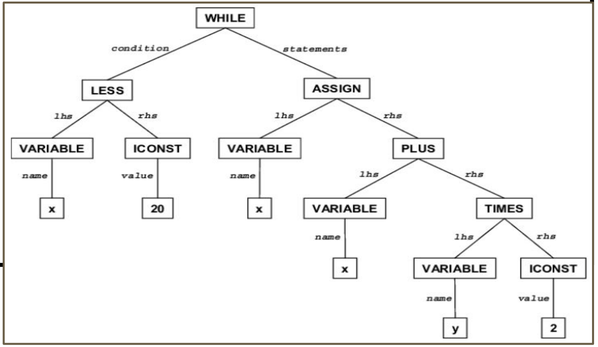
Gökay Gülsoy 270201072

IR Design for Compilers

- Developers generally use general-purpose programming languages to design IRs. As a result, IR implementations are verbose and manual modifications are expensive.

- While compilers relied historically on a few slowly evolving IRs, domain specific optimizations and specialized hardware motivate compilers to use and evolve many IRs.

```
; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** %argv) #0 {
entry:
  %retval = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  %argc.addr = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  store i32 %argc, i32* %argc.addr, align 4
  ret i32 0
}
```



What is IRDL and Why it exists ?

- Domain-specific language to define IRs, and it facilitates the implementation of SSA-based IRs.
- It is capable of expressing 28 domain-specific IRs developed as a part of LLVM's MLIR project while only rarely falling back to IRDL's support for generic C++ extensions.
- Aim is to enable concise and explicit specification of IRs and provide foundations for developing effective tooling to automate compiler construction process.

Need for IR design Languages

- LLVM, as a compiler infrastructure, has not only its user-facing LLVM IR but additionally uses various internal ones that are typically not visible to its users. Some of them are SelectionDAG, MachineInst, and MCInst.

- All of these IRs are deeply embedded into their respective compilers. Hence modifications require detailed compiler-specific knowledge, and even specialists are very hesitant to evolve existing IRs.

- While there exists approaches for generating parts of compiler (parsers, backends, code-generators, etc.), we lack a solution that streamlines the design of IRs themselves.

General Properties of IRDL

- MLIR does not provide predefined set of operations but relies on the concept of extensibility, with few built-in constructs leaving most of the IR customizable.

- Operations,types, and attributes are grouped into dialects, similar to namespaces or modular libraries. Each dialect sits at a abstraction level. For example, Linalg dialect in MLIR models linear algebra operations on either tensor or buffer operands.

```
func @conorm(%p: !cmath.complex<!f32>, %q: !cmath.complex<!f32>) -> !f32
    %norm_p = cmath.norm(%p) : !f32
    %norme_q = cmath.norm(%q) : !f32
    %pq = std.mulf %norm_p, % norm_q : !f32
    return %pq : !f32
```

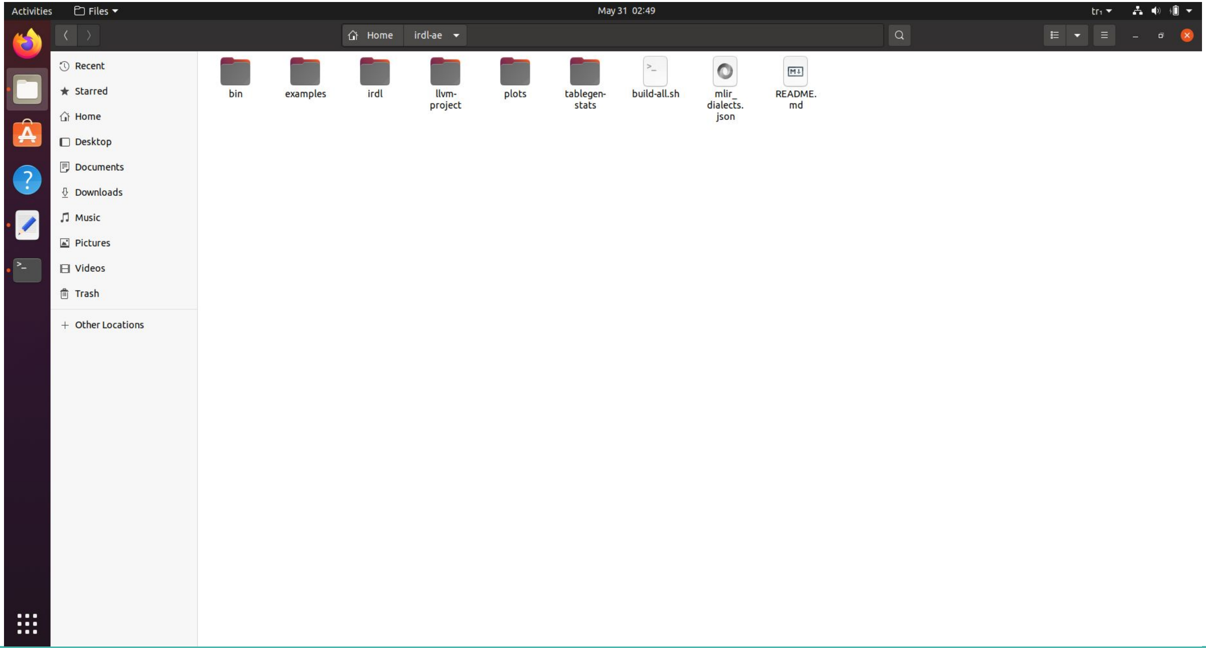
Self-Contained IRDL specification

```
Dialect cmath {
	Alias !FloatType = !AnyOf<!f32,!f64>

	Type complex {
		Parameters (elementType: !FloatType)

	Summary “A complex number”
	}

	Operation mul {
	  ConstraintVar (!T: !complex<FloatType>)
	  Operands (lhs: !T, rhs: !T)
	  Results (res: !T)

	  Format “$lhs, $rhs: $T.elementType”
	  Summary “Multiply two complex numbers”
	}
```

```
	Operation norm {
		ConstraintVar (!T: !FloatType)
		Operands(c: !complex<!T>)
		Results (res: !T)

		Format “$c: $T”
		Summary “norm of complex number”
	}
}
```

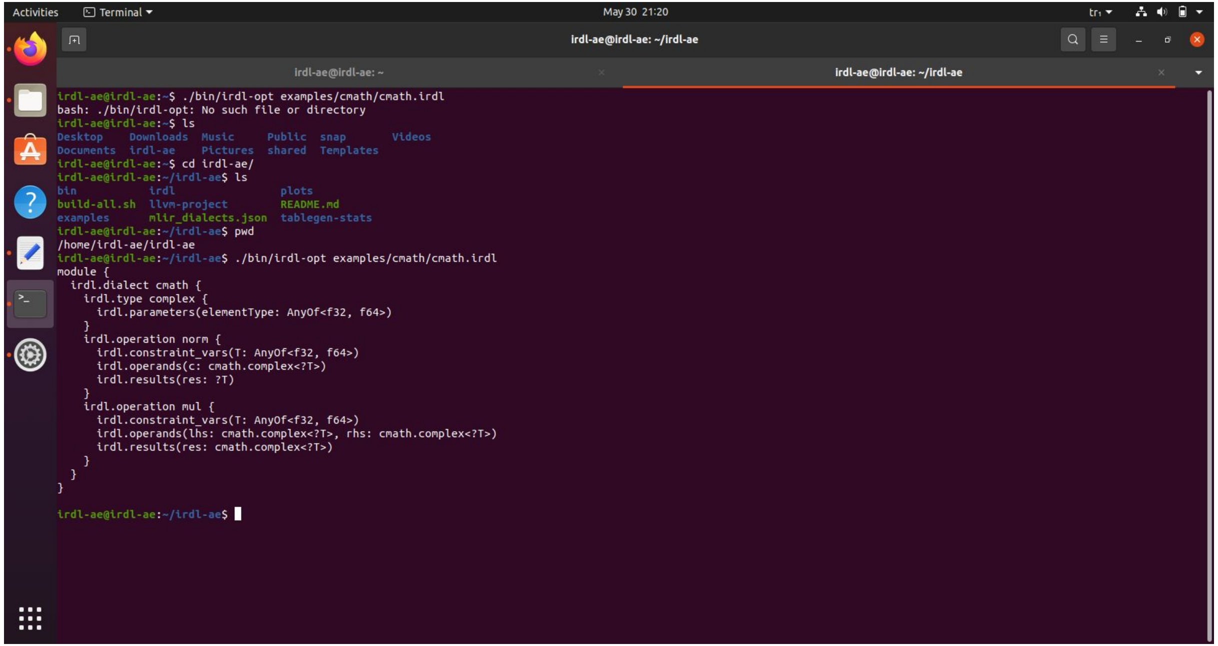
Reproduction of Artifact

- General layout of the project



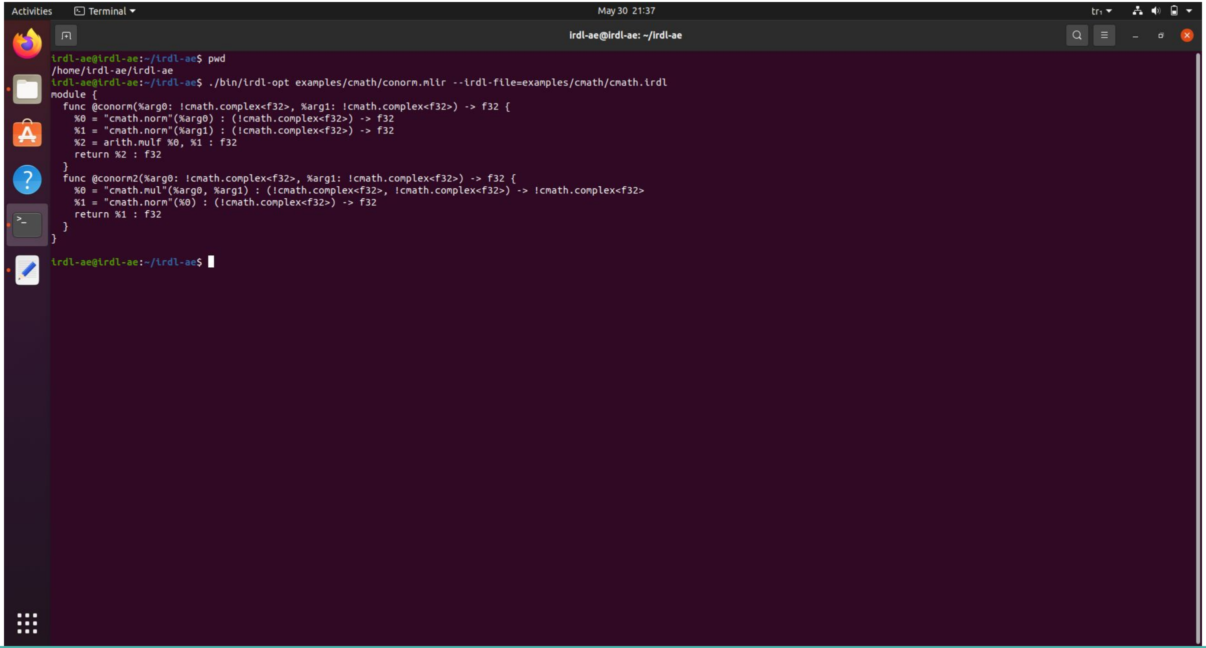
Reproduction of Artifact

- Parsing cmath module with irdl-opt



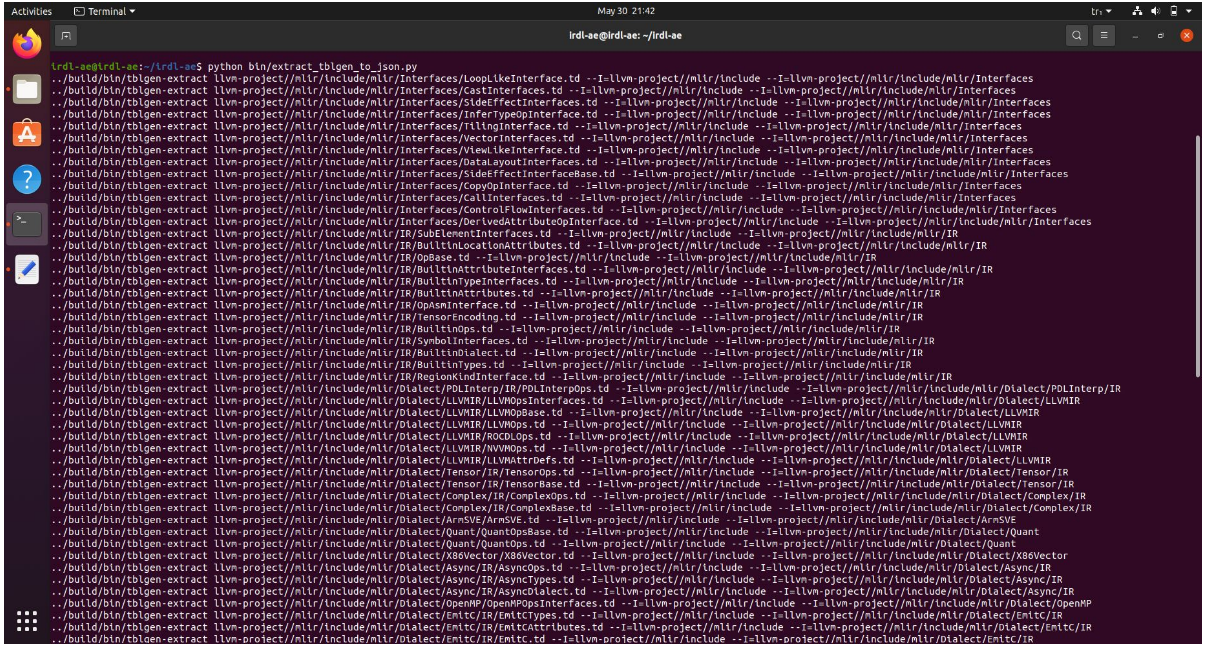
Reproduction of Artifact

- Registering cmath dialect and parsing the conorm.mlir example



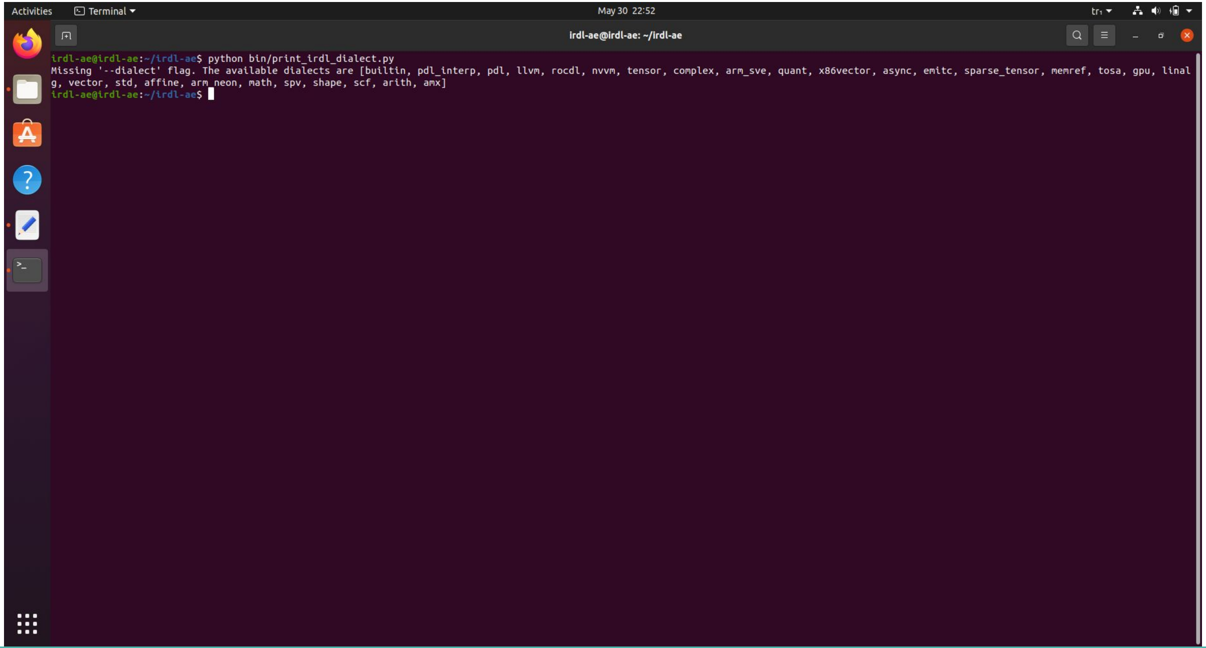
Reproduction of Artifact

- Extracting MLIR dialects in order to convert them into IRDL format



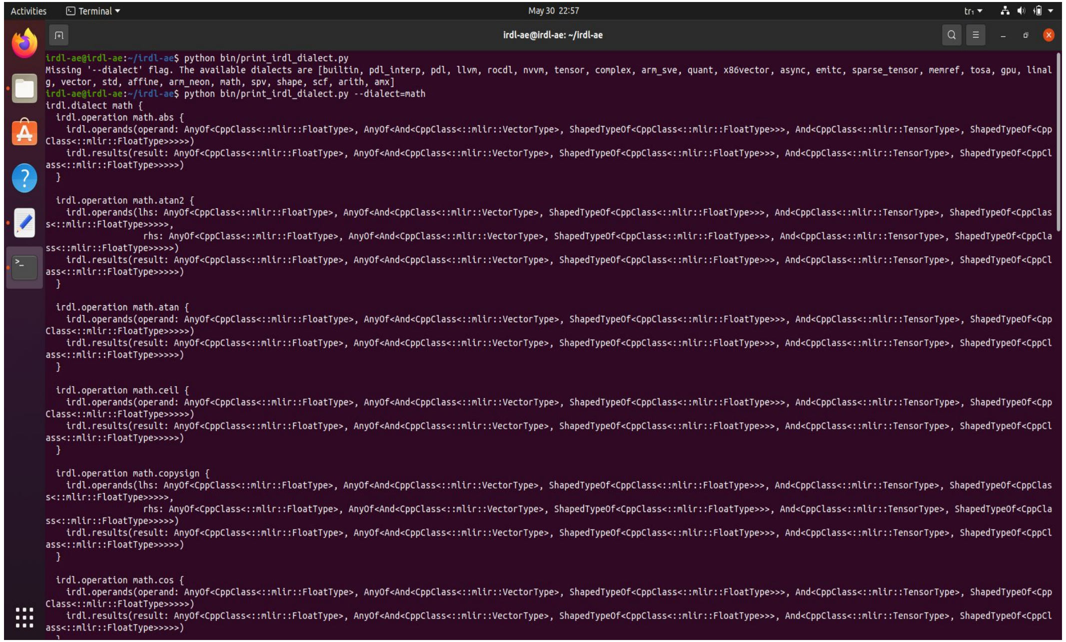
Reproduction of Artifact

- Querying the list of available dialects



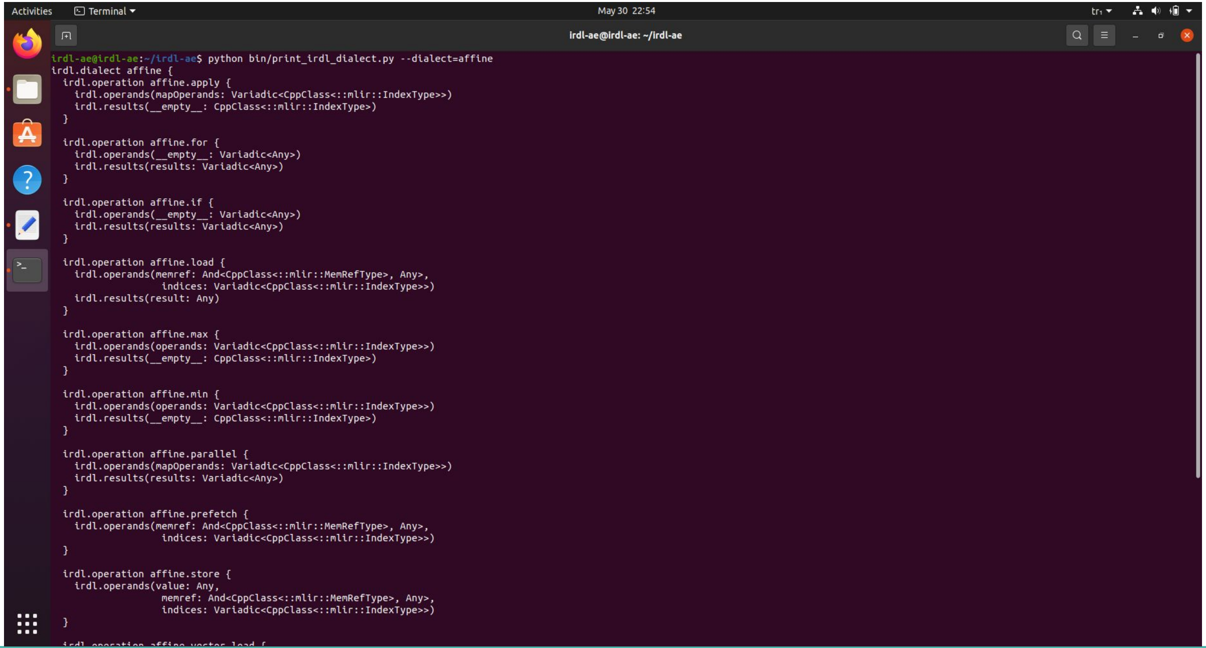
Reproduction of Artifact

- Converting math dialect to IRDL format



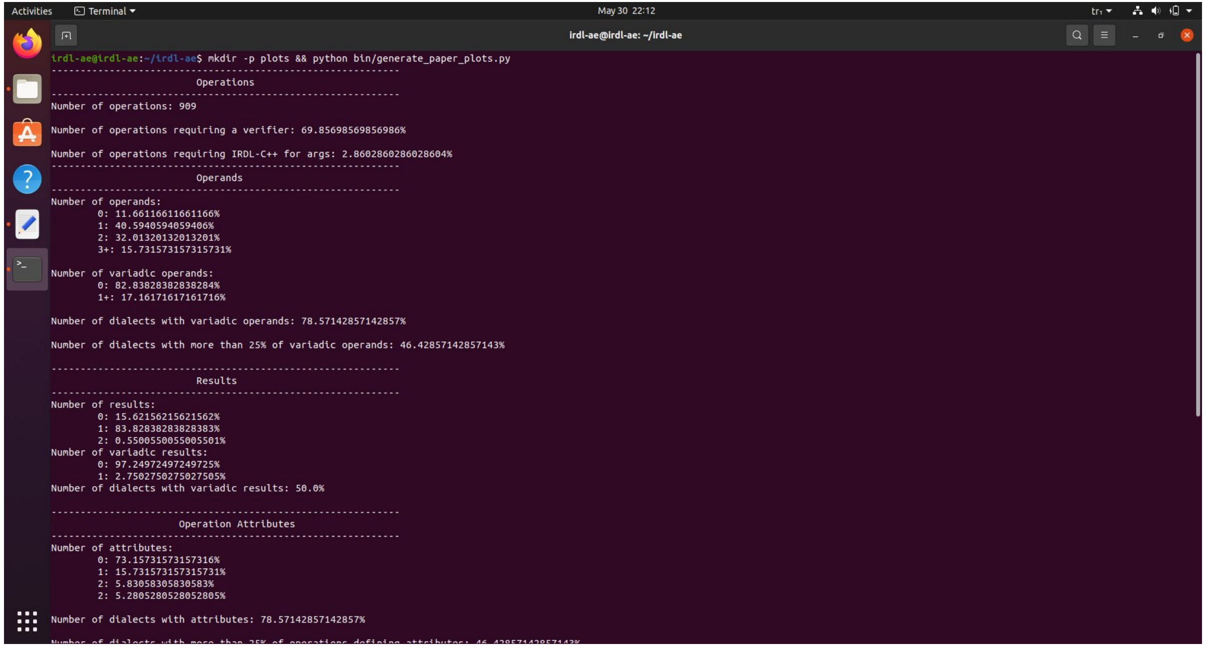
Reproduction of Artifact

- Converting math affine dialect to IRDL format



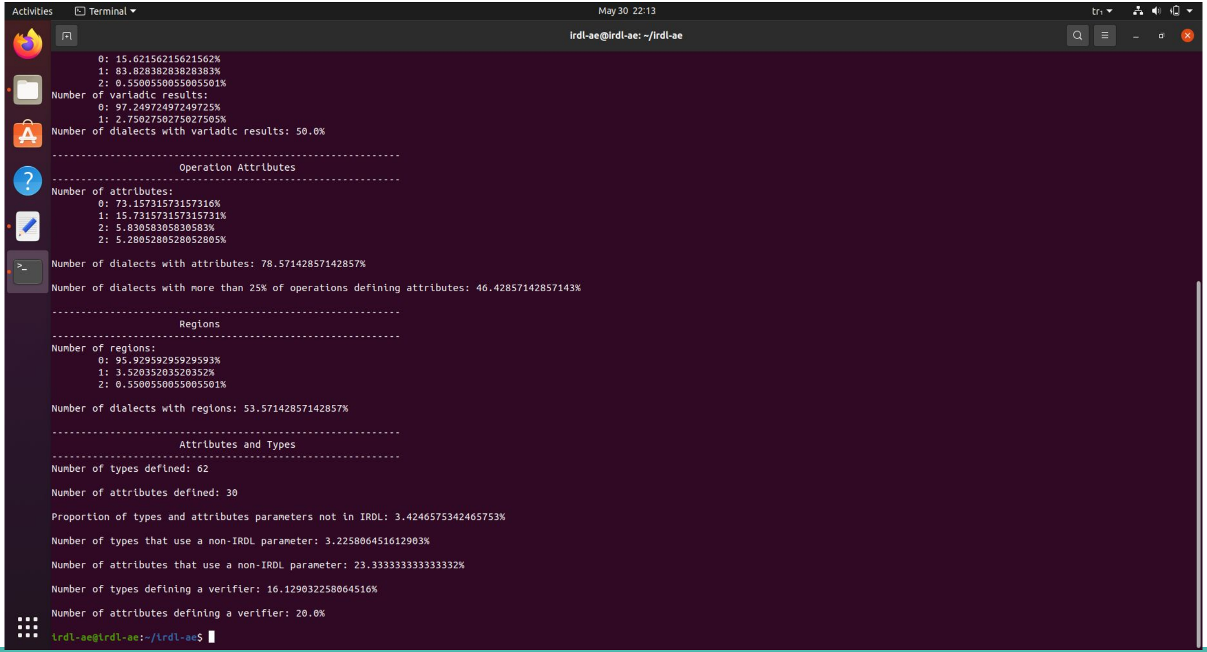
Reproduction of Artifact

- Reproduced table statistics for Operations, Operands, Results, Operation Attributes



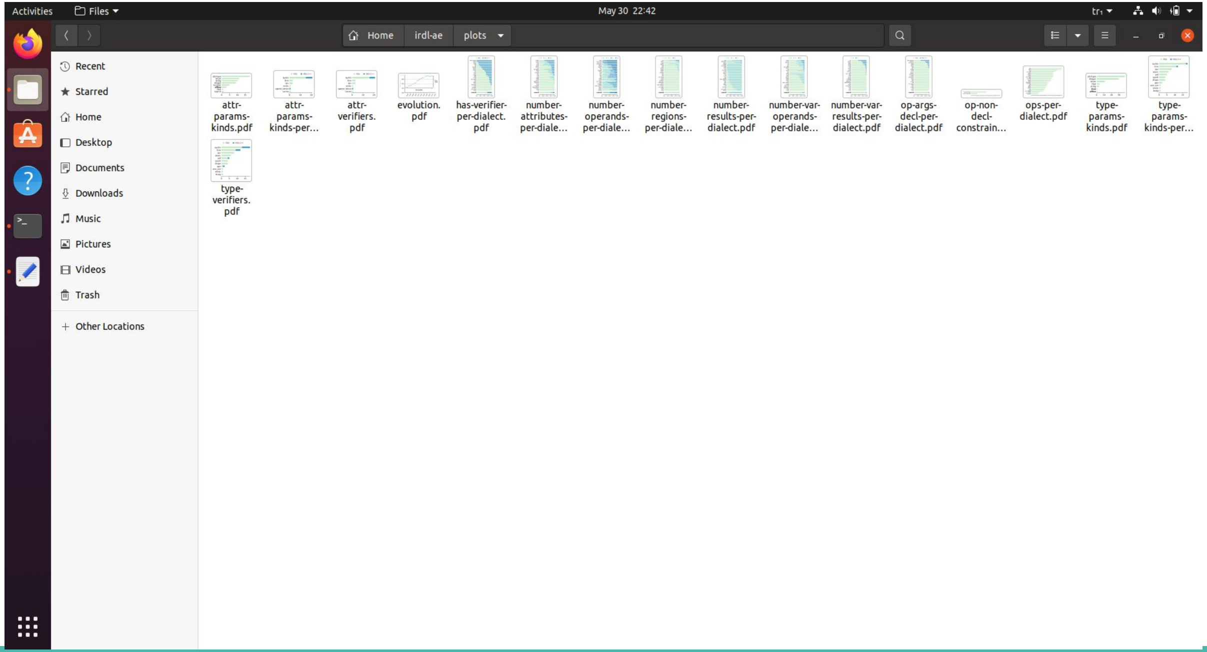
Reproduction of Artifact

- Reproduced table statistics for Regions and Number info for Types and Attributes



Reproduction of Artifact

- Plots are reproduced with command: mkdir -p plots && python bin/generate_paper_plots.py



References

- https://www.researchgate.net/profile/Peter-Fritzson/publication/228792639/figure/fig1/AS:393782852898820@1470896556105/Abstract-syntax-tree-of-the-while-loop.png
- https://raw.githubusercontent.com/Naios/notepad_llvm/master/preview.png
- https://dl.acm.org/doi/pdf/10.1145/3519939.3523700