ib.org

# positions as Filenames, Builds as Proofs: The Essence of ke

make program is a widely used tool for building files from existing files, according to a f build rules specified by the user. It is usually used to compile executable programs source code, but can also be used for many other jobs where a bunch of things are rated from other things, like this website, for example.

y alternatives to make have been proposed. Motivations for replacing make range from a e to replace make's very Unix-philosophy inspired domain-specific language for ribing build rules (Unix-philosophy in the sense that it often works by coincidence, but over if you do something exotic, like have filenames with spaces in, or have an onment variable with the “wrong” name), or make's slowness at some tasks, or a eption that make doesn't treat the make-alternative implementor's favourite programming uage with the special treatment it so obviously deserves.

rtheless, I think that make (or at least the GNU variant I am most familiar with) has a essence that can be profitably extracted and analysed.

## Essence of make

**essence of make is this**: make is an implementation of *constructive logic programming*, the following instantiation of the “Propositions-as-X” paradigm:

Atomic propositions are *filenames*. The filenames main.c, main.o and myprogram are all examples of atomic propositions in make's logic. For make, the idea of “well-formed ormula” from traditional logic means “doesn't have spaces in”.

Compound propositions are *build rules*. A build rule that states that myprogram can be

built from `main.o` and `module.o` is a statement that the atomic propositions `main.o` and `module.o` imply `myprogram`. Pattern rules like `%.o: %.c; gcc -o $@ -c $<` are *universally quantified* compound propositions: this says that, for all *x*, the atomic proposition *x*`.c` implies the atomic proposition *x*`.o`. Static pattern rules are essentially a form of bounded quantification.

that the form of compound propositions allowed is extremely restricted, even by the dards of logic programming: we are allowed at most one universal quantifier, which tifies over space-less strings, the rest of proposition must be of the form "`f1` and `f2` and d `fn` implies `g`", *and* if there is a quantifier, the variable must appear in the goal formula his format corresponds to a restricted form of Horn Clauses, as used in normal logic ramming.

stopped here, then `make` would not be any more than an extremely restricted form of g. But what makes `make` special is that it implements a *constructive* logic: it generates fs, or evidence, for the propositions it proves.

Proof, or evidence, of an atomic proposition `somefile` is the *content of an actual file `somefile` in the filesystem*. Some evidence is provided by the user, in the form of source files. Evidence for deduced atomic propositions, e.g., `.o` files, is generated by the proofs for compound propositions:

Proof of a compound proposition "`x` and `y` implies `z`" is a *command* to run that will generate the proof of the atomic proposition `z` from the proofs of the atomic propositions `x` and `y`. For pattern rules, this proof is parameterised by the instantiation of the universally quantified variable. For some reason, in `make`, the universally quantified variable is written as "`%`" in the proposition, and "`$*`" in the proof.

### t `make` does

g this mapping between logic and `make`, I see `make` as conceptually performing three s when it is told to use `Makefile` to generate the target `myprogram`.

t executes the `Makefile`, expanding out variables. This generates a collection of build ules.

t constructs a proof of the atomic proposition `myprogram` using backward-chaining roof search from the goal, via the build rules (aka compound propositions), back to the evidence for atomic propositions provided by the user in the file system. In traditional ogic, this proof would be represented using a tree, but obvious efficiency gains can be ad by exploiting sharing and representing it as a directed acyclic graph.

t *executes* the proof to generate the evidence of the atomic proposition `myprogram`. The evidence for the provability of `myprogram` is a file `myprogram` in the filesystem, generated by the proofs of the build rules and source files it's proof depends on. This tep can often be made more efficient by reusing existing pieces of evidence if the evidence they were built from hasn't changed.

GNU `make` manual's description combines the last two steps into one “run-the-build” and in practice this is what an realistic implementation ought to do. (And the first step GNU `make`'s reality, more complex because `make` can rebuild included files and restart , but I'm glossing over that for now.)

### vhat?

k that there are real benefits to seeing `make`-like systems as implementations of tructive logic programming:

believe that seeing `make`-like systems as a form of constructive logic programming elucidates the differences between some of the `make` alternatives that have been roposed. For instance, I think that the Ninja system essentially gets its speed ups by caching the some of results of the proof search step by storing the expansions of all of he universally quantified build rules that are needed. The OMake system allows for argets to dynamically depend on dependencies listed in generated files, via “scanner” dependencies. I think this corresponds to proof search in a *dependently-typed* logic that allows propositions to depend on the generated evidence of other propositions.

We can start to look at `makes`'s restrictions through the lens of logic programming, and start to think about more expressive build tools:

- Why are build rules restricted to at most one universal quantifier? What would we gain my allowing unrestricted Horn clauses? What if the universally quantified variables didn't have to appear in the goal formula, as in most other logic programming languages?

- Build rules that generate multiple files are Horn clauses with conclusions that are conjunctions (ands) of atomic propositions. GNU `make` and others make multiple targets difficult, but from a logical point-of-view, there is no problem.

- What rules does `make` use to resolve the choice between multiple proofs of the one atomic proposition? Could we have build systems that produce sets of proofs for each proposition? Can this be used to do multi-platform builds? Can we assign weightings to build rules so that `make` picks the overall “best” proof/build strategy?

- `make` implements a “top-down” approach to evaluating its logic program. Why not also implement a “bottom-up” evaluation too? This would enable us to ask questions like “what can be built using these rules and these source files?”. This might finally enable decent TAB-completion for `make` at the command line, and IDE introspection capabilities.

- Can logics that incorporate forms of (sound) circular reasoning be used to do build jobs that require iteration until a fixpoint. Can we use fixpoint logic to work around the tragedy of LaTeX's “Label(s) may have changed. Rerun to get cross-references right.”?

- Do all atomic propositions have to be filenames? Why not URLs? Why not proof-irrelevant ephemeral propositions?

- Can the connection between logic programming and relational databases be used? Can we use a `make`-like tool to query a database, and generate reports?

- Can we automatically augment the proof graphs that `make` implicitly generates to add provenance information?

esting stuff, I think.