Safeness of Make-Based Incremental Recompilation

Niels Jørgensen

Department of Computer Science, Roskilde University
P.O. Box 260, DK-4000 Roskilde, Denmark
nielsj@ruc.dk

Abstract. The make program is widely used in large software projects to reduce compilation time. make skips source files that would have compiled to the same result as in the previous build. (Or so it is hoped.) The crucial issue of safeness of omitting a brute-force build is addressed by defining a semantic model for make. Safeness is shown to hold if a set of criteria are satisfied, including soundness, fairness, and completeness of makefile rules. Conditions are established under which a makefile can safely be modified by deleting, adding, or rewriting rules.

Keywords. Make, incremental recompilation, semantic model.

1 Introduction

The make program reads a makefile consisting of rules with the following meaning: “If file $G$ is older than one or more of the files $D_1$, $D_2$, etc., then execute command $C$”, where $D_1$, $D_2$, etc., are source files that $G$ depends on, and the execution of $C$ creates $G$ by compiling the sources. This is characterized in [1] as cascading incremental recompilation, because recompilation spreads to other files along chains of dependency.

Safeness of make-based incremental compilation, the key result of this paper, can be stated as follows: Suppose we build a program brute-force, and then edit the source files, and possibly the makefile as well. Then under certain assumptions about the makefile rules and the kind of editing performed, the result of make-based incremental recompilation is equivalent to the result of a (second) brute force build. The result also applies to repeated cycles of editing and incremental recompilation.

The required properties of makefile rules are intuitively reasonable, for example, a fair rule may only create or update its own derived target. In confirming intuition about make, the safeness result provides formal justification for the existing practice of using make. Moreover, the result establishes that one may rely on make for incremental recompilation in situations where this is not obvious, for example upon certain modifications of the makefile.
Comparison with related work:

Historically, make originated \cite{Feldman77} within the Unix/C community. It is the most useful with languages such as C that allow for splitting source files into implementation and interface (header) files, because then make’s cascading recompilation can be instrumented as follows: Recompile a file either if the file itself changes, or an interface file on which it depends changes – but not if there is merely a change in the implementation of what is declared in the interface. Of course, the scheme is still extremely simple, and many files will be recompiled redundantly, for example, if a comment in a header file is modified.

A number of techniques exist for incremental recompilation which require knowledge about the syntax and semantics of the programming language being compiled. Tichy \cite{Tichy78} coined the notion of smart recompilation to describe a scheme for recompilation based on analysis of the modifications made to a file, to determine whether recompilation of a file that depended on the file would be redundant, and applied the scheme to a variant of Pascal. A variant of smart recompilation was proposed by Elsmann \cite{Elsman81} to supplement various other techniques for compiling Standard ML programs, and incorporated into the ML Kit with Regions compiler. Syntax directed editors \cite{Gries86,Stallman84} have been developed which perform compilation-as-you-type, and where the unit of granularity may be language constructs such as an individual assignment.

The level of granularity in make-based incremental recompilation is the file, and make controls the recompilation of files merely on the basis of their time stamps. Indeed, make is useful for tasks involving file dependencies in general (see \cite{Walden85} for some interesting examples) not just those that arise in the compilation of programming languages. An analysis of make must take a similar “blackbox” approach to files. The analysis framework comprises a small formal machinery for reasoning about execution of commands that appear in makefile rules. The machinery allows for proving the equivalence of certain command sequences, comprising the same commands but in a different order and possibly duplicated, representing brute-force vs. incremental recompilation.

The analysis framework also comprises a semantic definition for make which is in some ways similar to the semantic definition for (constraint) logic programs given in \cite{Noll86}. Makefile execution resembles logic program execution because it is query-driven and does not assign values to global variables.

Despite the widespread use of make, there are only few scientific or other publications on make. They include presentation or analysis of tools and methods \cite{Feldman77,Walden85,Brown85}, standardization \cite{Feldman84}, and tutorials \cite{Brown85,Walden85,Feldman84} on make and makefile generators such as mkmf and makedepend. In retrospect it can be seen that Stuart Feldman’s original paper on make \cite{Feldman77} tacitly assumed that makefile rules satisfy properties that guarantee safeness. Walden’s \cite{Walden85} analysis revealed errors in makefile generators for C. In the terminology of this paper, he showed that they did not generate complete rules for targets whose dependencies were derived targets. The framework supplements work such as Walden’s because the notion of rule completeness is defined formally and independently of C.