Proving "`make`" Correct: I/O Proofs in Two Functional Languages

Malcolm Dowse, Glenn Strong and Andrew Butterfield

January 29, 2003

Contents

1 Introduction

In [BS01] we presented some preliminary work describing reasoning about the I/O-related properties of programs written in functional programming languages. Only tentative conclusions could be drawn from that study because of the relatively simple nature of the program being considered. The program considered in this case study combines I/O and computation in essential and non-trivial ways. The results were encouraging regarding the ease of reasoning about I/O operations on functional languages, but more work was required. There are, therefore, a number of issues to be addressed:

> How do the reasoning techniques used in [BS01] scale when applied to more complex programs which perform I/O in unpredictable ways?

Our aim is to reason about the side-effects of a program on its environment. It is therefore an essential property of our reasoning system that it enables us to discuss I/O actions in a functional program in terms of their side effects. Since we are interested in both Haskell[HPJWe92] and Clean[PvE01] we require I/O system models which can accommodate both languages. The apparent differences between the two I/O systems raises another question:

> Do the different I/O systems used by Haskell and Clean lead to any significant differences in the proofs developed for each program, and if so do these differences correspond to differences in ease of reasoning?

2 Methodology

The aim of the case-study is to attempt to reason about a complex I/O applications in two different languages by building a practical simplification of the workings of an operating system. The two specific languages, Haskell and Clean, both have separate mechanisms for dealing with I/O. The two different paradigms are as follows:

- **Uniqueness types:** Referentially transparent side-effects are achieved by extending the type-system so it handles uniqueness attributes. [PvE01].
- **Monadic:** Referentially transparent side-effects are achieved as a result of embedding all side-effecting aspects of the language in a monad structure [HPJWe92].

In order to carry out the case-study, the following steps were performed.

1. Create an informal description of the task to be performed by the program.
2. Develop and run real applications in the specified languages.
3. Develop a standardised I/O model, deciding upon what aspects of I/O are considered important.
4. Encode the standardised I/O model within each language.
5. Re-write the programs so they operate in the semantic domain, with the standardised I/O model.
6. State properties to be proven.
7. Attempt proofs.

3 An informal description of `make`

The task chosen to be performed was a simplified version of the standard programming tool make [Fel79], which automates certain parts of program compilation.

The essential features of `make` are:

1. It controls all necessary compilation to ensure that the most up-to-date sources are used in the target program,
2. It ensures that no *unnecessary* compilations are performed where source files have not changed.

The program's input, commonly known as a "make-file", contains all necessary information about the file-names, the commands required to update the files, and, for each file, a list of the other files on which it depends.

For this study we can observe that the I/O `make` performs is limited to:

1. Reading the description of the program dependencies from a text file (the make-file),
2. Checking these dependencies against the filesystem to determine what compilation work needs to be done,
3. Executing external commands as detailed in the make-file in order to bring the target program up to date.

The I/O performed in the first point is of little interest from a reasoning point of view (being essentially a parsing problem) and so we consider our make program only from the position where the dependencies have been read and examined. These dependencies can be represented by a tree-like data structure where there is a root node representing a goal (a file to be built) and a number of sub-trees representing dependencies. Each node in the tree has an associated command to be run when the goal at that node should be rebuilt. We can represent the tree of dependencies as follows:

$$\begin{aligned} n \in \mathit{Name} &= \mathbb{A}^\star \\ c \in \mathit{Command} &= \mathbb{A}^\star \\ \mathcal{T} \in \mathit{Target} &= \textsc{Target}\ \mathit{Name}\ \mathit{Command}\ \mathit{Target}^\star \mid \textsc{Leaf}\ \mathit{Name} \end{aligned}$$

This tree structure requires an additional pre-condition, before any reasonable properties may be proven about it. These pre-conditions (and all other issues to do with the make-file structure) are described in detail in proof section **HL.2**, starting on page 11.

An imperative-style summary of how `make` behaves is as follows:

- Making (Leaf n): Check for the existence of n in the file-system. If it doesn't exist, abort the program with a run-time error. Otherwise, return, doing nothing.
- Making (Target nm cmd $deps$): Rebuild $deps$, all recursive dependencies. Then find the modification time of the newest file in all the dependencies. If it is newer than the modification time of the file nm, rebuild the file by running command cmd.

To further refine the above imperative description, we assume that `make` returns a time-stamp representing the time-stamp of the newest file in its list of dependencies. This value is found using the (informal) assumption that there are no dependency cycles. The resultant informal specification is as follows:

- Making (Leaf nm): Check for the existence of nm in the file-system. If it doesn't exist, abort the program with a run-time error. Otherwise, return the time-stamp associated with nm.
- Making (Target nm cmd $deps$): Rebuild $deps$, all recursive dependencies, retaining all their returned time-stamps. If the newest of these times is newer than the modification time of the file nm, then rebuild the file by running command cmd. Either way, return the resultant modification time of nm.

4 Real applications

Two fully working applications were written which implemented the above informal description – one in Haskell, the other in Clean. They can be found in Appendix B (Haskell), and Appendix C (Clean).

Each program was written with the idiom of the language, and, as a result, the programs differed considerably. The most notable differences are as follows:

- **Execution of commands.** In Haskell, the `System` library provides a simple `exec` function which executes a command. There is no such function in Clean libraries, and an alternative had to be created.
- **File-times.** A specific `FileTime` data-type had to be constructed in Clean, which is actually a pair containing a `Date` and a `Time`. In Haskell, the standard `EpochTime` type sufficed.
- **Writing to the console.** In Clean, once the "standard output" has been opened, it must be maintained as a separate entity throughout the program.
- **Handling of side-effects.** As a result of the different paradigms used by the two languages, the semantics of the side-effect handling are, of course, notably different.

5 The I/O model

In order to attempt formal reasoning about the two programs it was necessary to first create an I/O model in IVDM ([aA90], [Hug00]). In this model, all unnecessary (or uninteresting) parts of the file-system and operating system were removed. Essentially, this means that only the operations used by `make` were included in the I/O system.

To facilitate the proofs we provide a model of I/O that covers all the operations used by the make implementations.

Times can be represented as integers (from some suitable zero-moment).

$$t \in \mathit{EpochTime} = \mathbb{Z}$$

Each file can have a time associated with it; we also provide a value to represent the lack of a file time (associated with a missing file). We also represent names, commands and the target dependency tree in the obvious way:

$$f \in \mathit{FileTime} \quad = \quad \textsc{FileTime}\ \mathit{EpochTime} \mid \textsc{NoFileTime}$$

The filesystem is represented as a map from (file)names to times. We can represent the complete world that the program operates in as the product of a filesystem and a universal clock:

$$\begin{aligned} \phi \in FS &= \mathit{Name} \xrightarrow{m} \mathit{EpochTime} \\ \mathcal{W}, (\phi, \tau) \in World &= FS \times \mathit{EpochTime} \end{aligned}$$

The level of abstraction chosen for this case study is high enough to eliminate any need to model the contents of files, or any filesystem operations other than touching a file to update the modification time. This operation corresponds to the notion of updating a file, without committing to any specific notion of what the update involves. The operation will ensure that after the action the named file exists with an "up to date" file time, regardless of the state of the file before the action was performed.

We provide an ordering on times where the "missing" time is older than all other possible times, and times are ordered sequentially otherwise. This is a convenient representation for make as it allows us to view missing files as being older than all other files and therefore eternally out of date.

$$\begin{aligned} \textsc{NoFileTime} \preceq f &= \mathit{True} \\ (\textsc{FileTime}\ t_1) \preceq (\textsc{FileTime}\ t_2) &= t_1 \leq t_2 \end{aligned}$$

We provide two operations on the filesystem. The first allows us to look up the value associated with a given file name, which will be the files modification time. We do not advance the clock in this operation.

$$\begin{aligned} \text{getFileTime} &: \mathit{Name} \rightarrow \mathit{World} \rightarrow \mathit{FileTime} \\ \text{getFileTime}[n](\phi, \tau) &\mathrel{\widehat{=}} n \in \mathtt{dom}\ \phi \ \rightarrow\ \textsc{FileTime}\ \phi(n)\ ,\ \textsc{NoFileTime} \end{aligned}$$

The second important operation is to execute command 'cmd'. We assume here that the execution of a command c with associated filename n will have exactly the effect of updating the associated file date in the filesystem and advancing the universal clock.

$$\begin{aligned} \text{exec} &: \mathit{Name} \rightarrow \mathit{Command} \rightarrow \mathit{World} \rightarrow \mathit{World} \\ \text{exec}\ n\ c\ (\phi, \tau) &= (\phi \dagger \{n \mapsto \tau + 1\}), \tau + 1) \end{aligned}$$

This assumption allows us to reason effectively about the "exec" operation which would otherwise be capable of performing any arbitrary transformation on the world model. The intention of this definition of "exec" is to model a particular case of program execution by `make`, which corresponds to running a simple compiler. We use the operator †, called override, to introduce and replace bindings in a map. The notation $\phi \dagger \{n \mapsto \alpha\}$ indicates the map ϕ changed in such a way that the value n is mapped to the value α, leaving all other values untouched.

It is clear from this model that we are only interested in modelling "reasonable" uses of `make`. This is because a full implementation of `make` (such as GNU Make [SM00]) can contain arbitrary system commands, shell scripts and calls to arbitrary programs which we do not attempt to model.

6 Encoding the semantic I/O models

Using the I/O model described above, the actual operating-system primitives were replaced with their logical equivalents in the languages. In the proofs, we use a natural semantics and apply re-write rules for reasoning about the programs, and include functional descriptions of the I/O operations; in effect the languages are expanded to include an explicit representation of the side effects caused by the I/O system.

In [BS01] both functional programs were rewritten in a common syntax to facilitate a comparison of the reasoning steps in the proofs, and the proofs were performed on that common syntax. This time we have chosen to work at a level closer to the original languages since there is no clear advantage to syntactically sugaring the programs into a neutral form in this case.

In Clean, the world is represented with the types:

```
:: FS :== [(Name,EpochTime)]
:: World :== (FS,EpochTime)
```

Implementations of the I/O operations used in the program are provided in terms of their effect on this `World` value. These implementations reflect the embedding of the I/O model of §5 into the semantics of the language. Note that in a Clean implementation the `World` value requires uniqueness attribution, which is not required here since we are safely in the domain of the language semantics. Indeed, the I/O model does not have a direct equivalent to this attribute. We note, however, that the use of the `World` value remains single-threaded.

In Haskell, the world is represented as follows:

```
type FS = [(Name,EpochTime)]
type World = (FS,EpochTime)
```

Because of the monadic style of I/O in Haskell, it was necessary to construct a new (logical) monad, in the style of [Wad92], to take the place of Haskell's normal `IO` monad. This new monad includes an explicit representation of the world in the program so that we can directly state the required properties. The World type is defined as above, and the `IO` type is wrapped around it to represent the I/O monad.

```
newtype IO a = IO (World -> (World, a))
```

The usual set of monadic operators ("bind", "seq" and "return") are provided, along with rules for a de-sugaring of Haskell's idiomatic `do` notation that will allow the Haskell program to be rewritten in terms of this IO monad definition.

We introduce the usual set of map manipulation operations (such as override) and give semantics to the necessary I/O operations, for instance:

```
exec nm cmd = IO (\(p,k) -> ((override nm (k+1) p, k+1),()))
```

This operation `override` corresponds to the † operator introduced earlier, and indicates that in the map `p`, `nm` maps to a different value (`(k+1)` in this case).

The Haskell and Clean semantic models, along with all the proof conditions, can be found in Appendices D and E respectively.

7 Properties Proven

There are six principal theorems relating to the implementation of `make`. Each of these theorems is true under the simplifying assumptions of the I/O model and program abstractions performed on the original programs.

- Theorem 1 states that files whose names do not appear in the make-file will not have their modification times changed by running make.
- Theorem 2 states that directly after executing the command for a file, that file will be newer than any other file in the file system.

- Theorem 3 states that after executing make the modification time of a file will be no earlier than it was before running make.
- Theorem 4 states that following an execution of make the topmost dependency in the tree will be newer than all of the dependencies under it.
- Theorem 5 states that if the top dependency in the tree has not changed following an execution of make, then all of the dependencies under it will also be unchanged,
- Theorem 6 states that following an execution of make that Theorem 4 holds recursively through the tree.

These are Theorems **H.1** to **H.6** (for Haskell) and **C.1** to **C.6** (for Clean). They form the specification of `make`'s expected behaviour, and are the basis for our choice of implementation.

8 The use of reasoning operators

In the Haskell proof, the use of the monadic operators (`>>=`, `>>` and `return`) presents a problem. These operators are used to enforce the single threading of the world value by disallowing any other function access of the explicit world value.

This single threading is a necessary property of any implementation, but when attempting to produce our proofs it is necessary to refer directly to that value and inspect it. This is necessary because the properties that we wish to establish via the proofs are properties of that world value, and it will be necessary to trace the transformations applied to the world in order to verify that the property holds.

One solution to this problem is to carefully unwrap the monadic value each time the world must be inspected, and re-wrap it again before the next proof step is taken. While possible, this approach requires an inconvenient degree of mechanical work. Instead, we provide a number of new operators related to the standard monadic combinators. These operators can be used to lift a world value out of a monadic computation so that it can be inspected and manipulated in a proof. The three most interesting of these operators are:

- `>=>`, called "after", an operator which applies a monadic `IO` action to a world value, effectively performing the requested action and producing a new world. The function can be trivially defined:

  ```
  (>=>) :: World -> (IO a) -> (World, a)
  w >=> (IO f) = (f w)
  ```

- `>->`, called "world-after", an operator which transforms a world value using an I/O action. The result of this operation is a new world value which represents the changes made.

  ```
  (>->) :: World -> (IO a) -> World
  w >-> act = fst (w >=> act)
  ```

- `>~>`, called "value-after", the corresponding operator to `>->`, which transforms a value but does not retain the new world value that was produced.

```
(>~>) :: World -> (IO a) -> a
w >~> act = snd (w >=> act)
```

For this case-study it was noticed that, even when written entirely in the style of the language, Clean's ability to "split" the world was of little use – treating the world as a single monolithic entity is quite adequate. For this reason, it was decided, in the semantic model, also to adopt these operators, and re-write the Clean functions with a type-signature style which would permit these same operators to be used (a small change, which has little or no consequences).

The operators are defined as follows in Clean:

```
(>=>) infixl 9 :: World (World -> (World,a)) -> (World,a)
(>=>) w f = f w

(>->) infixl 9 :: World (World -> (World,a)) -> World
(>->) w f = fst (w >=> f)

(>~>) infixl 9 :: World (World -> (World,a)) -> a
(>~>) w f = snd (w >=> f)
```

Note that we can safely define and use these operators in our *proof* since we are working with a type correct program, which is therefore safely single-threaded. These operators would not be safe if added into a functional language, but are appropriate for reasoning.

More information about these operators are to be found in proof sections **HL.3** and **CL.3**, where a number of useful properties are proven about them.

9 General information on proofs

The proofs in this document are performed by manipulating/rewriting expressions in an abstracted program language, using a natural deduction style.

9.1 Abstracted program notation

The notation used in the abstracted programs (and in the proofs) is kept as similar as possible to that of the languages. Although we are really working in the semantic domain, the original language syntax is retained for simplicity.

Extra notation is added for convenience when the existing syntax is either not powerful enough, or its use would be cumbersome. There are two particular examples of this:

- Sometimes, a `FORALL` is inserted where explicit universal quantification is required for a particular variable. This usually occurs in inductive hypotheses, in inductive proofs.
- Since Clean's `#`-let notation can only be used in function definitions, it is replaced with explicit let-before expressions using `letb`s.

9.2 Error-handling

Run-time errors which abort the program are, for the purposes of simplicity, not being considered in this case-study. The abstracted model removes the possibility of any abnormal program termination. This occurs in **HL.3.1.2** and **CL.3.1.2**.

9.3 Sets

At certain points throughout the proofs, it is useful to view a list with equality defined over its elements as a set. As a result a small library of standard set functions were defined especially for use in the proofs. These are documented in Appendix A.

9.4 Haskell and Clean

As a result of the "reasoning operators", many of the proofs are extremely similar. It is recommended that the reader reads the Haskell proofs first, since many Clean proofs refer directly to their Haskell counterparts.

The lemma/theorem naming schemes are as follows:

H	Haskell Theorem
HL	Haskell Lemma
C	Clean Theorem
CL	Clean Lemma

9.5 Variable naming policy

In general we try to keep to a consistent variable naming policy in the proofs:

```
t,t0,t1,t2  ::  Target
    ts,ts1  ::  [Target]
n,n0,n1,n2  ::  Name
    ns,ns1  ::  [Name]
w,w0,w1,w2  ::  World
    ws,ws1  ::  [World]
         m  ::  Command
         f  ::  FileSystem
   k,k1,k2  ::  FileTime
    ks,ks1  ::  [FileTime]
      p,p1  ::  Arbitrary predicate or parameter
  s1,s2,s3  ::  [a]
```

If certain variable names appear, they will invariably mean the same type of thing throughout the document. For example, a variable `ws` will almost always

be the list of intermediate worlds returned from an invocation of the `trace` function.

10 Haskell Make Proofs

The Haskell proofs begin here. This section is the main body of the document, containing all the proofs for the Haskell version of `make`.

10.1 Directed Acyclic Graph Tree Proofs

Overview

This section describes the make-file structure used (DAG-Trees), the reasons why that type of structure was so suitable, and a number of properties about it.

- **HL.2.3** shows that if a tree is a DAG-Tree, then so are all its subtrees.
- **HL.2.8.1** shows how the top file in a DAG-Tree can't occur anywhere else at the same time – there are no cyclic dependencies.
- **HL.2.12** shows some of special properties that arise when case-analysis is performed on the possible ways two compatible trees can share information.
- **HL.2.13** is proof of the validity of a special type of induction over the structure of DAG-Trees, used much later.

This section is self-contained, and is not actually required until much later on in the proofs.

The requirements of a make-file structure

In order to reason about `make` it is necessary to create a structure which represents what one would consider to be a "normal" make-file. Here are five statements which we believe capture what is and is not a make-file (at least for the purposes of a case-study of this nature).

1. No file can depend on itself.
2. No sequence of files can cyclically depend on each other.
3. Multiple different files can depend on one file.
4. A target file cannot have multiple different (conflicting) commands to update the target. (Similarly, the dependencies of a specific target are constant – they don't change arbitrarily through make-file)
5. Each make-file only has one single target on which no other file depends.

1 and 2 are obvious. It is impossible to build any target if it is necessary for that same target to have already been built. Points 1 and 2 suggest that there must be some sense of ordering on the building of files.

Point 3 is a standard feature one would expect of `make`. An implementation of `make` which didn't permit this would most-likely be of limited practical use.

Point 4 is also quite acceptable – if the commands or dependencies associated with a particular file were somehow ambiguously stated in the make-file, it would lead to serious problems.

5, however, is somewhat less standard. Normal implementations of make (GNU make, for example), permit multiple "top" targets. Nevertheless, this can be assumed, for this case-study, to be just a convenient way of merging multiple make-files into one.

Implementing a make-file data-structure

What is the best (functional) data-structure to use when representing a make-file with the structure described above?

There are a number of possible solutions, but the one decided upon was a tree structure with an added pre-condition. The tree structure of `Target` is as follows:

```
data Target = Target Name Command [Target] | Leaf Name
```

It is a standard recursive data-type, and the recursive nature of it proved ideal for reasoning, using inductive proofs.

This structure has a number of useful look-up functions defined, all of which are commonly used throughout this document:

```
-- get the name of a target
name :: Target -> Name
name (Leaf n) = n
name (Target n _ _) = n

-- get the immediate descendants of a target
deps :: Target -> [Target]
deps (Leaf _) = []
deps (Target _ _ ts) = ts

-- get the command of a target
cmd :: Target -> Command
cmd (Leaf _) = ""
cmd (Target _ m _) = m

-- get all the descendant nodes of a node, including the node itself
alldeps :: Target -> [Target]
alldeps t = [t] ++ (concatMap alldeps (deps t))
```

Continuing with the discussion about the representation of make-files, this particular tree structure, however, only captures two of the five properties described

above (numbers 3 and 5). There is a "top" target (the root of the tree), and there is no problem having two different files depend on one. Additionally, we can capture property 4 by adding the pre-condition `isDAGT` defined below.

```
isDAGT :: Target -> Bool
isDAGT t = prodall safe (alldeps t)

-- is (r a1 a2) true for all a's in the given list.
prodall :: (a -> a -> Bool) -> [a] -> Bool
prodall r lst = and [r a1 a2 | a1 <- lst, a2 <- lst]

-- if the two names are the same, are the commands and
-- dependencies also the same?
safe :: Target -> Target -> Bool
safe t1 t2 =
  ((name t1)==(name t2)) ==>
    ((cmd t1)==(cmd t2) && (deps t1)==(deps t2))
```

`(isDAGT t)` effectively states that `(safe t1 t2)` is always true, where `t1` and `t2` are any sub-targets in `t` (here, a sub-target of `t` could also be `t` itself.) What `(safe t1 t2)` states, is that if the names of `t1` and `t2` are identical, then both the commands and all the dependencies of `t1` and `t2` will also be identical.

Put together, `(isDAGT t)` states that if the name associated with a node is equal in any two arbitrary sub-targets, then, although those two sub-trees might be in different parts of the full tree, with different parent nodes etc., the nodes themselves will still be indistinguishable. In other words, when the file is the same, the command, and the dependencies will also be the same – Property 4.

Finally, to show that Properties 1 and 2 hold, it is first necessary to use the assumption that infinite make-trees are ruled out in this case-study. It is shown (in **HL.2.8.1**) that if any file was to, directly or indirectly, be dependent on itself, then an infinite tree would result.

A type of tree with this pre-condition shall be known as a *Directed Acyclic Graph Tree* (or DAG-Tree, for short). The reason for this choice is that once the tree is finite, and there are no dependency loops, the dependency graph that results is a directed acyclic graph.

Lemma HL.2.1

If relation `r :: a -> a -> Bool` yields `True` when applied to any pair of elements in set `s`, then it will also yield `True` when applied to any pair of elements in any subset of `s`.

```
((prodall r s) && (s2 `subset` s)) ==> (prodall r s2)
```

Proof: Simple, using boolean and list properties.

Lemma HL.2.2

If t is a DAG-Tree, so are all its immediate descendants.

```
(isDAGT t) ==> all isDAGT (deps t)
```

Proof: Application of **HL.2.1** to the definition of `isDAGT`, and `alldeps`.

⟨ assuming pre-condition ⟩
`(isDAGT t)`
= ⟨ `isDAGT` defn. ⟩
`(prodall safe (alldeps t))`
= ⟨ `alldeps` defn. ⟩
`(prodall safe ([t]++(concatMap alldeps (deps t))))`
⇒ ⟨ Adding additional fact, using set laws ⟩

```
(prodall safe ([t]++(concatMap alldeps (deps t)))) &&
  (all (\t1 ->
    (alldeps t1) `subset` ([t]++(concatMap alldeps (deps t)))
  ) (deps t))
```

⇒ ⟨ Applying **HL.2.1**, merging both expressions ⟩
`(all (\t1 -> prodall safe (alldeps t1)) (deps t))`
= ⟨ `isDAGT` defn. ⟩
`(all (\t1 -> isDAGT t1) (deps t))`
= ⟨ using currying, and adding pre-condition ⟩
`(isDAGT t) ==> (all isDAGT (deps t))`

Lemma HL.2.3

If t is a DAG-Tree, so are all its recursive descendants.

```
(isDAGT t) ==> all isDAGT (alldeps t)
```

Proof: Simple, using **HL.2.1** and **HL.2.2**

Lemma HL.2.4

Two targets are identical if and only if their recursive dependencies are also identical.

```
(t0==t1) == ((alldeps t0) == (alldeps t1))
```

Proof:

⟨ **Left-to-right implication** ⟩
`(t0==t1)`
⇒ ⟨ application ⟩
`((alldeps t0) == (alldeps t1))`
⟨ **Right-to-left implication** ⟩
`(alldeps t0)==(alldeps t1)`
⇒ ⟨ application ⟩
`(head (alldeps t0)) == (head (alldeps t1))`
⇒ ⟨ `alldeps` defn. ⟩
`(head (alldeps t0))==t0 && (head (alldeps t1))==t1`
⇒ ⟨ subs. ⟩
`(t1==t0)`

Definition of `allnames` functions

Sometimes it is necessary to retrieve from a target information about all the names in that target. The `allnames` function does that, in very similar style to the way `alldeps` retrieves all sub-targets (in fact, **HL.2.11** shows just how similar they are).

The `allnames2` function is just slightly different, since it excludes the top file. **HL.2.5** shows the relationship of the two.

```
-- get all names of descendant nodes of a target, including
-- that node itself
allnames :: Target -> [Name]
allnames t = [name t]++(concatMap allnames (deps t))

-- get all names of descendant nodes in a target, excluding
-- that node itself
allnames2 :: Target -> [Name]
allnames2 t = (map name (deps t))++(concatMap allnames2 (deps t))
```

Lemma HL.2.5

The relationship between `allnames` and `allnames2`.

```
(allnames t) `setEq` ((name t):(allnames2 t))
```

Proof: Induction of target structure. In the inductive case, certain set axioms must be used, showing that two lists in a different order are still the same *sets*.

Definition of `couldBeDAGT`

The function `couldBeDAGT` states that two targets could co-exist in different parts of a larger DAG-Tree structure without conflicts – i.e. both targets are themselves valid DAG-Trees, and all of their sub-targets are mutually compatible.

```
-- if target t1 and t2 were "put together" they would make
-- a valid DAGTree
couldBeDAGT :: Target -> Target -> Bool
couldBeDAGT t1 t2 = prodall safe ((alldeps t1)++(alldeps t2))
```

When `(couldBeDAGT t1 t2)`, it is sometimes said that `t1` and `t2` are *Compatible DAG-Trees*.

Lemma HL.2.6

Some simple facts about `couldBeDAGT`.

Treated as relation, `couldBeDAGT` is symmetric.

`(couldBeDAGT t1 t2) == (couldBeDAGT t2 t1)`

If a targets t1 and t2 could be successfully combined to make a DAG-Tree, then t1 and t2 must be DAG-Trees already.

```
(couldBeDAGT t1 t2) ==> (isDAGT t1) && (isDAGT t2)
```

A target is compatible with itself if and only if it is itself a valid DAG-Tree.

```
(couldBeDAGT t1 t1) == (isDAGT t1)
```

If `t` is a DAG-Tree, then all its sub-trees are compatible DAG-Trees.

```
(isDAGT t) ==> (prodall couldBeDAGT (alldeps t))
```

Proof: All relatively simple (or obvious), using properties of string concatenation, and boolean algebra.

Lemma HL.2.7

A target `t0` is a recursive dependency of target `t1` if and only if the recursive dependencies of `t0` is a proper subset of the recursive dependencies of `t1`.

```
(t0 `elem` alldeps t1) == ((alldeps t0) `psubset` (alldeps t1))
```

Proof: Split the equality of the form $A = B$ into the form $(A \Rightarrow B) \wedge (A \Leftarrow B)$. Proving the right-to-left implication is trivial. Showing the left-to-right implication requires induction.

Lemma HL.2.8.1

If `t` is a DAG-Tree, the filename at the root will not appear anywhere else in the tree.

```
(isDAGT t) ==>
  all (\t1 -> (name t) `notElem` (allnames t1)) (deps t)
```

Proof:

For this case-study we state informally that infinite make-trees are disallowed. The most sensible justification of this decision is that an infinite make-tree would be of no practical use, since rebuilding the target would take infinitely long, and it would be impossible to store such information on disc as a (normal) make-file.

To prove the above lemma, let us suppose that a tree `t` is a valid DAG-Tree, and `(name t)` *does* reappear elsewhere in the tree. Since it is a valid DAG-Tree, if two names appear twice, their dependencies will be equal. But, one dependency is *within* the other – therefore the structure will be infinite, and, as a result, disallowed.

An example of a DAG-Tree which is technically valid, but infinite, is:

```
let t = (Target n m [t,t])
in t
```

Lemma HL.2.8.2

If `t` is a DAG-Tree, the filename at the root will not appear anywhere else in the tree.

```
(isDAGT t) ==> (name t) `notElem` (allnames2 t)
```

Proof: Uses **HL.2.8.1**, showing how it relates to `allnames2`.

Lemma HL.2.10

If two targets could be merged to form a valid DAG-Tree, then the three statements "the targets are identical", "the targets have identical names", and "the dependencies of the targets are all equal" are logically equivalent.

```
(couldBeDAGT t1 t2) ==>
  ((t1==t2) == ((name t1)==(name t2))) &&
  ((t1==t2) == ((alldeps t1)==(alldeps t2)))
```

Proof: All relatively trivial.

⟨ Assume pre-condition: ⟩
`(couldBeDAGT t1 t2)`
= ⟨ defn. ⟩
`(prodall safe ((alldeps t1)++(alldeps t2)))`
⇒ ⟨ `prodall` properties ⟩
`(safe t1 t2)`
= ⟨ `safe` defn. ⟩

```
((name t1)==(name t2)) ==>
  ((cmd t1)==(cmd t2)) && (deps t1)==(deps t2)
```

⇒ ⟨ Adding defn. of equality on targets ⟩

```
((t1==t2)==((name t1)==(name t2) && (cmd t1)==(cmd t2)
  && (deps t1)&&(deps t2))) &&
(((name t1)==(name t2)) ==>
  ((cmd t1)==(cmd t2)) && (deps t1)==(deps t2))
```

⇒ ⟨ substitution ⟩

```
(t1==t2)==((name t1)==(name t2))
```

⇒ ⟨ **HL.2.4** ⟩

```
((t1==t2)==((name t1)==(name t2))) &&
  ((t1==t2)==((alldeps t1)==(alldeps t2))
```

⇒ ⟨ adding pre-condition ⟩

```
(couldBeDAGT t1 t2) ==>
  ((t1==t2)==((name t1)==(name t2))) &&
    ((t1==t2)==((alldeps t1)==(alldeps t2))
```

Lemma HL.2.11

```
(allnames t) == (map name (alldeps t))
```

Proof: Induction on target structure.

Lemma HL.2.12.1

If `t0` and `t1` are compatible DAG-Trees, then any statement about the how the sets `(allnames t0)` and `(allnames t1)` share information is logically equivalent to the same statement made about the sharing of information between `(alldeps t0)` and `(alldeps t1)`.

Some examples are as follows:

```
(couldBeDAGT t0 t1) ==>
  ((allnames t0) `setEq` (allnames t1)) ==
    ((alldeps t0) `setEq` (alldeps t1))

(couldBeDAGT t0 t1) ==>
  ((allnames t0) `subset` (allnames t1)) ==
    ((alldeps t0) `subset` (alldeps t1))

(couldBeDAGT t0 t1) ==>
  ((allnames t0) `disjoint` (allnames t1)) ==
    ((alldeps t0) `disjoint` (alldeps t1))
```

Proof:

It can be seen from **HL.2.11** that mapping the function `name` across every element of `(alldeps t)` yields `(allnames t)`. Since all statements about list or set membership are based around the ability to compare elements, it it necessary to show that this mapping preserves equality:

1. Will two identical elements of `(alldeps t)` be mapped onto two identical elements of `(allnames t)`?
2. Will two different elements of `(alldeps t)` be mapped onto two different elements of `(allnames t)`?

The first question is trivially true, because of the nature of function application. Additionally, if we assume that `t` is a DAG-Tree, the second question is also true, since (from **HL.2.10**) the equality of two elements of `(alldeps t)` is the same thing as the equality of the two names.

Now, if `(couldBeDAGT t0 t1)`, then `(alldeps t0)` and `(alldeps t1)` can be effectively viewed just as subsets of `(alldeps t2)`, where `t2` is a DAG-Tree containing `t0` and t1. Therefore, the above lemma is true: any statement about the overlapping of `(alldeps t0)` and `(alldeps t1)` is the same when talking about `(allnames t0)` and `(allnames t1)`, since member equality is preserved.

Definition of five tree cases

It is often advantageous, when working with DAG-Trees, to perform case-analysis on the different ways two trees can share data. It turns out that the most natural way of doing this is to use the five ways two sets can share information.

It can be shown easily that if S and T are sets, then exactly one of the following is true:

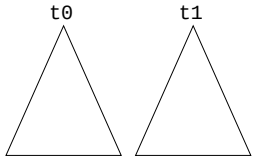
$$S = T$$
$$S \subset T$$
$$S \supset T$$
$$S \cap T = \emptyset$$
$$(S \cap T \neq \emptyset) \wedge (S \not\subseteq T) \wedge (S \not\supseteq T)$$

Therefore, if one assumes the existence of two targets, `t0` and `t1`, and substitute `(allnames t0)` and `(allnames t1)` for S and T, respectively, the following five predicates take shape:

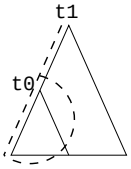
```
apartT, bwithinT, crushedT, dwithinT, equalT :: Target -> Target -> Bool
apartT t0 t1   = (allnames t0) `disjoint` (allnames t1)
bwithinT t0 t1 = (allnames t0) `psubset` (allnames t1)
crushedT t0 t1 =
  (allnames t0) `notdisjoint` (allnames t1) &&
  (not ((allnames t0) `subset` (allnames t1))) &&
  (not ((allnames t1) `subset` (allnames t0)))
dwithinT t0 t1 = (allnames t1) `psubset` (allnames t0)
equalT t0 t1   = (allnames t0) `setEq` (allnames t1)
```

The following diagrams go some way towards explaining the rationale behind the rather eclectic naming scheme:

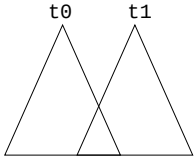
- **Apart** No filenames are shared between the two targets.



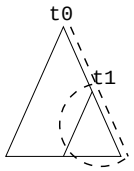
- **BWithin** Target `t0` is entirely within target `t1`.



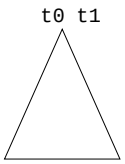
- **Crushed** Some of `t0` is in `t1`, and some of `t1` is in `t0`.



- **DWithin** Target `t1` is entirely within target `t0`.



- **Equal** Both targets are identical. Everything is shared.



Lemma HL.2.12.2

If two trees are compatible DAG-Trees, the definitions of the five tree cases are logically equivalent to the same statements where `allnames` is replaced by `alldeps`.

```
(couldBeDAGT t0 t1) ==> (
  (apartT t0 t1)==((alldeps t0) `disjoint` (alldeps t1)) &&
  (bwithinT t0 t1)==((alldeps t0) `psubset` (alldeps t1)) &&
  (crushedT t0 t1)==(
    ((alldeps t0) `notdisjoint` (alldeps t1)) &&
    (not ((alldeps t0) `subset` (alldeps t1))) &&
    (not ((alldeps t1) `subset` (alldeps t0)))) &&
  (dwithinT t0 t1)==((alldeps t1) `psubset` (alldeps t0)) &&
  (equalT t0 t1)==((alldeps t0) `setEq` (alldeps t1)))
```

Proof: A direct application of **HL.2.12.1** to the five definitions.

Lemma HL.2.12.3

If `t0` and `t1` are compatible DAG-Trees, then performing case-analysis on four of the five cases yields some additional facts of interest. These are stated below:

```
(couldBeDAGT t0 t1) ==> (
  (bwithinT t0 t1 ==> (t0 `elem` (alldeps t1))) &&
  (crushedT t0 t1 ==> (
    ((name t0) `notElem` (allnames t1)) &&
    ((name t1) `notElem` (allnames t0)) &&
    (name t0 /= name t1))) &&
  (dwithinT t0 t1 ==> (t1 `elem` (alldeps t0))) &&
  (equalT t0 t1 ==> (t0==t1)))
```

Proof: Perform case-analysis, and use additional lemmas to prove the specific facts.

B-Within Case:

⟨ `B-Within` defn. ⟩

```
  (allnames t0) `psubset` (allnames t1)
```

= ⟨ using **HL.2.12.2** ⟩

```
  (alldeps t0) `psubset` (alldeps t1)
```

= ⟨ `alldeps` defn. ⟩

```
  ([t0]++(concatMap alldeps (deps t0))) `psubset` (alldeps t1)
```

⇒ ⟨ set theory ⟩

```
  t0 `elem` (alldeps t1)
```

Crushed Case:

⟨ `Crushed` defn. ⟩

```
  ((allnames t0) `notdisjoint` (allnames t1)) &&
    (not ((allnames t0) `subset` (allnames t1))) &&
    (not ((allnames t1) `subset` (allnames t0)))
```

= ⟨ if `t2` and `t3` are compatible DAG-Trees... ⟩

```
  (((name t2) `elem` (allnames t3)) ==>
    ((allnames t2) `subset` (allnames t3))) &&
```

⇒ ⟨ therefore.. (using contrapositive) ⟩

```
  ((name t0) `notElem` (allnames t1)) &&
    ((name t1) `notElem` (allnames t0))
```

⇒ ⟨ using set theory ⟩

```
  ((name t0) /= (name t1))
```

D-Within Case:

⟨ `D-Within` defn. ⟩

```
  (allnames t1) `psubset` (allnames t0)
```

= ⟨ using **HL.2.12.2** ⟩

```
  (alldeps t1) `psubset` (alldeps t0)
```

= ⟨ `alldeps` defn. ⟩

```
  ([t1]++(concatMap alldeps (deps t1))) `psubset` (alldeps t0)
```

⇒ ⟨ set theory ⟩

```
  t1 `elem` (alldeps t0)
```

Equal Case:

⟨ `Equal` defn. ⟩

```
  (allnames t0) `setEq` (allnames t1)
```

= ⟨ using **HL.2.12.2** ⟩

```
  (alldeps t0) `setEq` (alldeps t1)
```

⇒ ⟨ since, in each, the full trees (`t0` and `t1`) are in both sets, they must be identical ⟩

```
  t0==t1
```

Lemma HL.2.13

Proof of the validity of a special form of induction over DAG-Trees.

This specific form of induction is used when proving a property about the interaction of two compatible DAG-Trees. It is different to a normal structural induction, where the induction is effectively an iterative process that eventually terminates at a base case. Instead, the sub-targets of target `t0` are continually examined recursively until either `t0` is entirely inside `t1` ("B-Within" or "Equal") or they are entirely outside each other ("Apart"). Target `t1` remains constant always.

```
p, bcase, icase :: Target -> Target -> Bool
bcase t0 t1 = (apartT t0 t1) || (bwithinT t0 t1) || (equalT t0 t1)
icase t0 t1 = (crushedT t0 t1) || (dwithinT t0 t1)

((couldBeDAGT t0 t1) && (bcase t0 t1) ==> (p t0 t1))
  &&
((couldBeDAGT t0 t1) && (icase t0 t1) &&
  (all (\t -> p t t1) (deps t0))  ==>  (p t0 t1))
  ==> ((couldBeDAGT t0 t1) ==> (p t0 t1))
```

Proof: Structural induction over target `t0`.

Base Case: `t0==(Leaf n)`

⟨ `allnames2` defn. ⟩

```
  (allnames2 t0) == []
```

⇒ ⟨ From set properties $S \not\subset \emptyset$, $S \cap \emptyset = \emptyset$ ⟩

```
  (not (dwithinT t0 t1)) && (not (crushedT t0 t1))
```

⇒ ⟨ `icase` defn. ⟩

```
  not (icase t0 t1)
```

⇒ ⟨ **HL.2.12**, and `bcase`,`icase` defn. ⟩

```
  (couldBeDAGT t0 t1) ==> (bcase t0 t1)
```

⟨ Assuming the following.. ⟩

```
  (couldBeDAGT t0 t1) && (bcase t0 t1) ==> (p t0 t1)
```

⟨ infer ⟩

```
  (couldBeDAGT t0 t1) ==> (p t0 t1)
```

Inductive Case: `t0==(Target n m ts)`

⟨ Inductive hypothesis ⟩

```
  (all (\t -> (couldBeDAGT t t1) ==> (p t t1)) ts)
```

If `(bcase t0 t1)`

⟨ Assuming the following.. ⟩

```
    (couldBeDAGT t0 t1) && (bcase t0 t1) ==> (p t0 t1)
```

⟨ infer ⟩

```
    (couldBeDAGT t0 t1) ==> (p t0 t1)
```

Else-If `(not (bcase t0 t1))`

⟨ **HL.2.12**, and `bcase`, `icase` defn. ⟩

```
    (couldBeDAGT t0 t1) ==> (icase t0 t1)
```
⟨ also, by **HL.2.6**, and `couldBeDAGT` defn. ⟩
```
    (couldBeDAGT t0 t1) ==> (all (\t -> couldBeDAGT t t1) (deps t0))
    (couldBeDAGT t0 t1) ==> (all (\t -> couldBeDAGT t t1) ts)
```
⟨ With inductive hypothesis: ⟩
```
    (couldBeDAGT t0 t1) ==> (all (\t -> p t t1) ts)
```
⟨ Then, from the following.. ⟩
```
    (couldBeDAGT t0 t1) && (icase t0 t1) &&
      (all (\t -> p t t1) (deps t0)) ==> (p t0 t1)
```
⟨ infer ⟩
```
    (couldBeDAGT t0 t1) ==> (p t0 t1)
```

10.2 Semantic Monad Proofs

Overview

This section deals with proofs relating to the reasoning operators `>=>`, `>->` and `>~>`, and the special `trace` function. These are all used to reason about all monadic actions (in Haskell), and is the corner-stone of most of the Haskell related I/O reasoning.

- **HL.3.1** shows how the three operators behave under the monadic `return` action.
- **HL.3.2.1** to **HL.3.2.3** show how the three operators interact with the monadic operator `>>`. These are often used to "dissect" an imperative style function into its constituent parts.
- **HL.3.3** to **HL.3.6** give important properties of the `trace` function. In this document, the `trace` function is used almost exclusively for reasoning about the `update_deps` function defined later on.

Monad definition

Our `IO` monad is defined in the following way (the structure being similar to that of [Wad92]):

```
newtype IO a = IO (World -> (World, a))

instance Monad IO where
  return v                  = retf
    where retf = IO (\w -> (w,v))
  (IO f1) >>= ac2    = IO bindf
    where
      bindf w =
        let (w1,v) = (f1 w)
            (IO f2) = (ac2 v)
        in  (f2 w1)
```

Reasoning operators definition

The three reasoning operators are defined as follows. `>=>` returns the world, and return value after executing an action on a world. The `>->` operator just returns the world. The `>~>` operator, on the other hand, only returns the value.

```
infix 1 >=>, >~>
infixl 1 >->

(>=>) :: World -> (IO a) -> (World, a)
w >=> (IO f) = (f w)

(>->) :: World -> (IO a) -> World
w >-> act = fst (w >=> act)

(>~>) :: World -> (IO a) -> a
w >~> act = snd (w >=> act)
```

Lemma HL.3.1

```
(w >=> return x) == (w,x)
```

Proof: Function expansion and substitution.

```
(w >=> return x)
```

= ⟨ `return` defn. ⟩

```
w >=> (IO (\w -> (w,x)))
```

= ⟨ `>=>` defn. ⟩

```
(w,x)
```

Corollaries:

```
(w >-> return x) == w
(w >~> return x) == x
```

Lemma HL.3.2.1

```
(w >=> (a1 >> a2)) == ((w >-> a1) >=> a2)
```

Proof: Function expansion and substitution.

```
let a1 = (IO f1)
    a2 = (IO f2)
```

⟨ LHS ⟩

```
  w >=> (a1 >> a2))
```

= ⟨ `>>` defn. ⟩

```
  w >=> (a1 >>= \_ -> a2)
```

= ⟨ a1,a2 expansion. ⟩
```
  w >=> ((IO f1) >>= \_ -> (IO f2))
```
= ⟨ >>= defn., in new IO monad. ⟩
```
  w >=> (IO (\w -> f2 (fst (f1 w))))
```
= ⟨ >=> defn. ⟩
```
  f2 (fst (f1 w))
```

⟨ RHS ⟩
```
  (w >-> a1) >=> a2
```
= ⟨ >-> defn. ⟩
```
  (fst (w >=> a1)) >=> a2
```
= ⟨ >=> defn. ⟩
```
  (fst (f1 w)) >=> (IO f2)
```
= ⟨ >=> defn. ⟩
```
  f2 (fst (f1 w))
```
= LHS

Lemma HL.3.2.2

>> is an associative dual of >->. (This terminology is taken from [BW82]).

```
(w >-> (a1 >> a2)) == ((w >-> a1) >-> a2)
```

Proof: Function expansion and substitution.

⟨ LHS ⟩
```
(w >-> (a1 >> a2))
```
= ⟨ >=> defn. ⟩
```
fst (w >=> (a1 >> a2))
```
= ⟨ Apply **HL.3.2.1** ⟩
```
fst ((w >-> a1) >=> a2)
```
= ⟨ >-> defn. ⟩
```
(w >-> a1) >-> a2
```
= RHS

Lemma HL.3.2.3

```
(w >~> (a1 >> a2)) == ((w >-> a1) >~> a2)
```

Proof: Function expansion and substitution.

⟨ LHS ⟩
```
(w >~> (a1 >> a2))
```
= ⟨ >=> defn. ⟩
```
snd (w >=> (a1 >> a2))
```
= ⟨ Apply **HL.3.2.1** ⟩
```
snd ((w >-> a1) >=> a2)
```
= ⟨>~> defn.⟩
```
(w >-> a1) >~> a2
```
= RHS

`trace` definition

The `trace` function takes an action and a list of parameters and applies the action sequentially to the world with the specific parameters, returning a list containing all intermediate worlds (including the first, and last).

It shares special properties with the `mapM` function in Haskell, as can been seen in **HL.3.4.2**

```
-- run mapM on a world, and return all intermediate worlds.
trace :: (a -> IO b) -> [a] -> World -> [World]
trace a [] w = [w]
trace a (p:ps) w = (w : (trace a ps (w >-> a p)))
```

Lemma HL.3.3

```
(length (trace a ps w)) == (length ps)+1
```

Proof: Induction on list of parameters.

Base Case: `ps = []`

⟨ LHS: ⟩

```
  (length (trace a ps w))
```

= ⟨ Replace `ps`. ⟩

```
  (length (trace a [] w))
```

= ⟨ `trace` defn. ⟩

```
  (length [w])
```

= ⟨ `length` defn. ⟩

```
  (length []) + 1
```

= ⟨ replacing `ps`. ⟩

```
  (length ps)+1
```

= RHS

Inductive Case: `ps = (p1:ps1)`

⟨ *Inductive hypothesis*: ⟩

```
  (length (trace a ps1 w1)) == (length ps1)+1
```

⟨ LHS ⟩

```
  (length (trace a ps w))
```

= ⟨ `trace` definition. ⟩

```
  (length (w:(trace a ps1 (w >-> a p1))))
```

= ⟨ letting `w1 = w >-> a p1`. ⟩

```
  (length (w:(trace a ps1 w1)))
```

= ⟨ using inductive hypothesis, algebra ⟩

```
  (length ps1)+2
```

= ⟨ relationship of `ps` to `ps1`. ⟩

```
  (length ps)+1
```

= RHS

Lemma HL.3.4.1

```
(head (trace a ps w)) == w
```

Proof: Trivial. (Expansion of `trace` function.)

Lemma HL.3.4.2

```
(last (trace a ps w)) == w >-> (mapM a ps)
```

Proof: Induction on list of targets.

Base Case: `ps = []`

⟨ LHS: ⟩
```
  (last (trace a ps w))
```
= ⟨ replacing `ps`. ⟩
```
  (last (trace a [] w))
```
= ⟨ `trace`, `last` defns. ⟩
```
  w
```
= ⟨ `mapM` property ⟩
```
  w >-> (mapM a [])
```
= ⟨ re-introducing `ps`. ⟩
```
  w >-> (mapM a ps)
```
= RHS

Inductive Case: `ps = (p1:ps1)`

⟨ *Inductive hypothesis*: ⟩
```
  (last (trace a ps1 w1)) == (w1 >-> mapM a ps1)
```
⟨ LHS ⟩
```
  (last (trace a ps w))
```
= ⟨ `trace` defn. ⟩
```
  (last (w:(trace a ps1 (w >-> a p1))))
```
= ⟨ letting `w1` = `w >-> a p1`. ⟩
```
  (last (w:(trace a ps1 w1)))
```
= ⟨ `last` semantics. ⟩
```
  (last (trace a ps1 w1))
```
= ⟨ using inductive hypothesis. ⟩
```
  (w1 >-> mapM a ps1)
```
= ⟨ replacing `w1`. ⟩
```
  (w >-> a p1) >-> (mapM a ps1)
```
= ⟨ Using **HL.3.2.2** ⟩
```
  (w >-> ((a p1) >> (mapM a ps1)))
```
= ⟨ Rewriting using do-notation. ⟩
```
  (w >-> do x <- a p1
            xs <- mapM a ps1
            return (x:xs))
```
= ⟨ `mapM` defn. ⟩
```
  (w >-> mapM a (p1:ps1))
```
= ⟨ replacing `ps`. ⟩
```
  (w >-> mapM a ps)
```
= RHS

Lemma HL.3.5

```
(i >= 0) && (i <= length ps) ==>
  (trace a ps w)!!i == w >-> mapM a (take i ps)
```

Proof:

Base Case: `i=0`

⟨ LHS ⟩

```
  (trace a ps w)!!0
```

= ⟨ `!!`, `trace` defn. ⟩

```
  w
```

= ⟨ `mapM` ⟩

```
  w >-> mapM a []
```

= ⟨ `trace` defn. ⟩

```
  w >-> mapM a (take 0 ps)
```

= RHS

⟨ adding pre-condition ⟩

```
  (i >= 0) && (i <= length ps) ==>
    ((trace a ps w)!!i ==(w >-> mapM a (take i ps)))
```

Inductive Case: `i=j`

⟨ *Inductive hypothesis*: ⟩

```
  ((j-1) >= 0) && ((j-1) <= length ps) ==>
    (trace a ps w)!!(j-1) == w >-> mapM a (take (j-1) ps)
```

⟨ LHS ⟩

```
  (trace a ps w)!!j
```

= ⟨ `trace` properties. (**HL.3.6**) ⟩

```
  (trace a ps w)!!(j-1) >-> a (ps!!(j-1))
```

= ⟨ From inductive hypothesis, assuming local pre-condition ⟩

```
  (w >-> mapM a (take (j-1) ps)) >-> a (ps!!(j-1))
```

= ⟨ **HL.3.2.2** ⟩

```
  w >-> ((mapM a (take (j-1) ps)) >> a (ps!!(j-1)))
```

= ⟨ `mapM` defn. ⟩

```
  w >-> mapM a (take j ps)
```

= RHS

⟨ adding pre-condition ⟩

```
  (i >= 0) && (i <= length ps) ==>
    ((trace a ps w)!!i ==(w >-> mapM a (take i ps)))
```

Lemma HL.3.6

The only difference between two adjacent worlds in a list returned by `trace` is the execution of one command.

```
(i >= 0) && (i < length ps) ==>
  ((trace a ps w)!!i >-> a (ps!!i))
   ==((trace a ps w)!!(i+1))
```

Proof: Induction over the value of `i`.

Base case: `i==0`

⟨ Expanding `trace`, and one of its recursive calls ⟩

```
  (trace a (p1:p2:ps1) w) ==
    (w : (w >-> a p1) : (trace a ps1 ((w >-> a p1) >-> a p2)))
```

⇒ ⟨ therefore.. ⟩

```
  ((trace a (p1:p2:ps1) w)!!0 == w) &&
    ((trace a (p1:p2:ps1) w)!!1 == (w >-> a p1)) &&
    (ps!!0 == p1)
```

⇒ ⟨ introducing `i`, and letting `ps` equal `(p1:p2:ps1)`. ⟩

```
  (i >= 0) && (i < length ps) ==>
    ((trace a ps w)!!i >-> a (ps!!i))
     ==((trace a ps w)!!(i+1))
```

Inductive case: `i==j`

⟨ *Inductive hypothesis* ⟩

```
  (j-1 >= 0) && (j-1 < length ps1) ==>
    ((trace a ps1 w)!!(j-1) >-> a (ps1!!(j-1)))
     ==((trace a ps1 w)!!j)
```

⇒ ⟨ list properties ⟩

```
  ((trace a (p1:ps1) w)!!j >-> a ((p1:ps1)!!j))
    ==((trace a (p1:ps1) w)!!(j+1))
```

⇒ ⟨ Let `ps = p1:ps1` ⟩

```
  (j >= 0) && (j < length ps) ==>
    ((trace a ps w)!!j >-> a (ps!!j))
     ==((trace a ps w)!!(j+1))
```

⇒ ⟨ Re-introducing `i`, and pre-conditions ⟩

```
  (i >= 0) && (i < length ps) ==>
    ((trace a ps w)!!i >-> a (ps!!i))
     ==((trace a ps w)!!(i+1))
```

10.3 Basic I/O Model Proofs

Overview

This section proves many simple properties associated with the I/O primitives `getFileTime` and `exec`, and from them proves some simple properties of `make`. The last lemma in this section states the important property that if `make` pre-condition `pre_make` is true when making a target, it will be true for all recursive calls to `make`.

In this section:

- The definitions of all the main I/O primitives are given.
- **HL.4.1.1** and **HL.4.1.2** state that getting a file modification time and making a leaf-node target don't change the world.
- **HL.4.1.3** is an essential property of `make`. It shows that the resultant world after `make` is equal to the resultant world after all the dependencies have been made, with the added possibility that the top file may or may not have been rebuilt.
- **HL.4.2.1** shows that if no file in the file-system is newer than that of the clock, then this will also be true after running `exec` on a file.
- **HL.4.4** is the most important proof in this section, and the last. It shows that if the `pre_make` pre-condition holds when making a particular target, then it will hold for all recursive calls as well.

Definition of `World`

In this case-study, the world contains two things – a clock time, and a file-system. The file-system itself is just modelled as a mapping from names to time-stamps.

```
type EpochTime = Integer
type Name = FilePath
type FS = [(Name,EpochTime)]
type World = (FS,EpochTime)
```

The two I/O primitives have the following type signatures:

```
getFileTime :: Name -> IO2 FileTime
exec :: Name -> Command -> IO2 ()
```

Firstly, the type `Command` is just a string:

```
type Command = String
```

In the case-study, it is important to notice the distinction between the types `EpochTime` and `FileTime`. The former is simply a number. The latter, however, can represent either an `EpochTime` or no time at all.

```
data FileTime = FileTime EpochTime | NoFileTime
```

On top of the actual data-type definition, equality and ordering is also defined on `FileTime`s. With regard to ordering, `NoFileTime < (FileTime k)`.

```
instance Eq FileTime where
  NoFileTime   == NoFileTime    = True
  NoFileTime   == _             = False
  _            == NoFileTime    = False
  (FileTime t) == (FileTime s)  = (t==s)
```

```
instance Ord FileTime where
  compare NoFileTime   NoFileTime   = EQ
  compare NoFileTime   _            = LT
  compare _            NoFileTime   = GT
  compare (FileTime t) (FileTime s) = (compare t s)
```

Definition of `getFileTime` I/O primitive

The `FileTime` data-type is the return type of the `getFileTime` action, which is defined as:

```
-- given a file name, return its modification time
getFileTime :: Name -> IO2 FileTime
getFileTime fn = IO2 (\(f,k) -> ((f,k), (td f)))
  where
    td f = case (lookup fn f) of
           Nothing  -> NoFileTime
           (Just t) -> FileTime t
```

In the case that the file doesn't exist, it will return `NoFileTime`.

Lemma HL.4.1.1

Getting the time of a file does not change the world.

```
(w >-> getFileTime n) == w
```

Proof: Trivial

Definition of `make`

The `make` call, around which the entire case-study is based, is now defined as follows:

```
-- make the target, and return the modification time of the
-- newest file of all recursive dependencies.
make :: Target -> IO2 FileTime
```

Firstly, leaf-nodes cannot be built as such. If they exist, their time-stamp is returned. Otherwise, a run-time error occurs.

```
make (Leaf nm) = do
  mtime <- getFileTime nm
  if (mtime==NoFileTime)
    then error ("can't make "++nm++"!")
    else return mtime
```

When building a target with dependencies, all the dependencies are updated, returning their respective time-stamps. If the modification time of the top file `n` is older than any dependency then it is necessary to rebuilt the top file. If not, it just returns.

```
make (Target nm cmd depends) = do
  -- get modification times of this file
  mtime <- getFileTime nm
  -- build and get the times of all children
  ctimes <- update_deps depends
  -- if its older than the newest child, rebuild, and return time
  if (mtime <= (newest ctimes))
    then do
      exec nm cmd     -- execute the command
      getFileTime nm  -- return the update time of the file
    else
      return mtime  -- just return mtime
```

The function `newest` just finds the maximum time-stamp from a list of `FileTime`.

```
-- given a list of times, find the newest
newest :: [FileTime] -> FileTime
newest times = foldl max NoFileTime times
```

The `update_deps` function makes a list of dependencies, returning their time-stamps. Quite often, throughout the proof, `update_deps` and `mapM make` are freely interchanged.

```
-- update all the targets, returning all their times
update_deps :: [Target] -> IO2 [FileTime]
update_deps = mapM make
```

Lemma HL.4.1.2

Making a leaf-node target does not change the world.

```
(w >-> make (Leaf n)) == w
```

Proof:

⟨ Definition of `make` on leaf targets ⟩

```
(w >-> make (Leaf n)) ==
  (w >-> do {
    mtime <- getFileTime n;
    if (mtime==NoFileTime)
      then error ("can't make "++n++"!")
      else return mtime})
```

⇒ ⟨ case in which exception occurs is ignored (see section 9.2) ⟩

```
(w >-> make (Leaf n)) ==
  (w >-> do {
    mtime <- getFileTime n;
    return mtime})
```

= ⟨ rearranging, according to laws of do-notation ⟩

```
(w >-> make (Leaf n)) ==
  (w >-> getFileTime n)
```

⇒ ⟨ **HL.4.1.1** ⟩

```
(w >-> make (Leaf n)) == w
```

Lemma HL.4.1.3

The world that results from making a non-leaf target is either a world with all dependencies made, or a world with all dependencies made *and* the top file `n` rebuilt.

```
((w >-> make (Target n m ts)) ==
  (w >-> do {mapM make ts;exec n m}))
|| ((w >-> make (Target n m ts)) ==
     (w >-> mapM make ts))
```

Proof: Analysis of `make` body, using **HL.4.1.1.**

⟨ `make` defn. ⟩

```
(w >-> make (Target n m ts)) ==
  (w >-> do {
      mtime <- getFileTime n;
      ctimes <- update_deps ts;
      if (mtime <= (newest ctimes))
        then do {
          exec n m;
          getFileTime n}
        else return mtime})
```

= ⟨ Using **HL.4.1.1** on first command, and expanding `update_deps` defn. ⟩

```
(w >-> make (Target n m ts)) ==
  (w >-> do {
      ctimes <- mapM make ts;
      if ((w >~> getFileTime n) <= (newest ctimes))
        then do {exec n m;getFileTime n}
        else return mtime})
```

⇒ ⟨ Splitting, based on result of `if` comparison ⟩

```
((w >-> make (Target n m ts)) ==
  (w >-> do {
      ctimes <- mapM make ts;
      exec n m;
      getFileTime n}))
|| ((w >-> make (Target n m ts)) ==
     (w >-> do {
         ctimes <- mapM make ts;
         return (w >~> getFileTime n)}))
```

= ⟨ Removing unused return variable `ctimes` ⟩

```
((w >-> make (Target n m ts)) ==
  (w >-> do {
      mapM make ts;
      exec n m;
      getFileTime n}))
|| ((w >-> make (Target n m ts)) ==
     (w >-> do {
         mapM make ts;
         return (w >~> getFileTime n)}))
```
= ⟨ Rearranging, using **HL.3.2.2** ⟩
```
((w >-> make (Target n m ts)) ==
  ((w >-> do {
      mapM make ts;
      exec n m;}) >-> (getFileTime n)))
|| ((w >-> make (Target n m ts)) ==
     ((w >-> mapM make ts)
       >-> (return (w >~> getFileTime n))))
```
= ⟨ Using **HL.4.1.1** in first equality, and **HL.3.1** in second equality. ⟩
```
((w >-> make (Target n m ts)) ==
  (w >-> do {mapM make ts;exec n m}))
|| ((w >-> make (Target n m ts)) ==
     (w >-> mapM make ts))
```

Lemma HL.4.1.4

The value returned when a leaf node is built is equal to the time-stamp of the leaf node.

```
(w >~> make (Leaf n)) == (w >~> getFileTime n)
```

Proof: Very similar to **HL.4.1.2.**

Lemma HL.4.1.5

The value returned when a leaf node is built will never be `NoFileTime`.

```
(w >~> make (Leaf n)) /= NoFileTime
```

Proof: Trivial

Definition of `exec` I/O primitive

The `exec` I/O action is really just "touch", and the command is ignored altogether in the semantic model (for reasons described in section 5). The clock time is incremented, and the time-stamp associated with the file `nm` is changed to be the value of that new clock time.

```
-- logically execute command 'cmd'.
exec :: Name -> Command -> IO2 ()
exec nm cmd = IO2 (\(f,k) -> ((override nm (k+1) f, k+1),()))
```

Definition of `clock_newer`

If the following predicate is true then no file in the file-system is newer than the value of the clock. It also states that the clock-value is non-negative. The `clock_newer` predicate is half of the `pre_make` pre-condition used later.

```
-- is every file in the filesystem no newer than the clock?
-- (clock-time is forced to be non-negative)
clock_newer :: World -> Bool
clock_newer (p,k) = (k >= (foldl max 0 (map snd p)))
```

Lemma HL.4.2.1

Presuming the clock-time in a world is as new as any file before rebuilding a file, then it will be as new afterwards.

```
(clock_newer w) ==> (clock_newer (w >-> exec n m))
```

Proof: Examination of `exec` function.

⟨ assuming pre-condition ⟩
`(clock_newer w)`
= ⟨ letting `(f,k) = w`. ⟩
`(clock_newer (f,k))`
= ⟨ expanding definition: ⟩
`(k >= (foldl max 0 (map snd f)))`
⇒ ⟨ `override`, `max` properties ⟩
`((k+1) == (foldl max 0 (map snd (override n (k+1) f))))`
⇒ ⟨ weakening the equality into an inequality ⟩
`(k+1) >= (foldl max 0 (map snd (override n (k+1) f)))`
= ⟨ `clock_newer` defn. ⟩
`(clock_newer ((override n (k+1) f), k+1))`
= ⟨ `exec` defn. ⟩
`(clock_newer ((f,k) >-> exec n m))`
⟨ Reinstating `w` and adding pre-condition ⟩
`(clock_newer w) ==> (clock_newer (w >-> exec n m))`

Lemma HL.4.3.1

If the clock is as new as every file in the world before making a target, the clock will continue to be as new as every file afterwards.

```
(clock_newer w) ==> (clock_newer (w >-> make t))
```

Proof: Induction on structure of target `t`. In the inductive case, all the implications of the recursive make-calls are chained together, to show that the `clock_newer` property holds after the recursive calls. Finally, using **HL.4.2.1**, it is shown that the possible execution of an `exec` in the `make` body will not affect the `clock_newer` property.

Base case: `t=(Leaf n)`

⟨ Assume pre-condition ⟩

```
  (clock_newer w)
⇒ ⟨ adding HL.4.1.2 ⟩
  (clock_newer w) && ((w >-> make (Leaf n)) == w)
⇒ ⟨ substitution ⟩
  (clock_newer (w >-> make (Leaf n)))
⟨ add pre-condition and reinstate t ⟩
  (clock_newer w) ==> (clock_newer (w >-> make t))
```

Inductive case: `t=(Target n m ts)`

```
⟨ Inductive hypothesis ⟩
  all (\t1 -> FORALL w1:
    (clock_newer w1) ==> (clock_newer (w1 >-> make t1)))
  ts
⇒ ⟨ Instantiating w1 ⟩
  all (\(t1,w1) ->
    (clock_newer w1) ==> (clock_newer (w1 >-> make t1)))
  (zip ts (trace make ts w))
⇒ ⟨ Letting ws = trace make ts w ⟩
  all (\(t1,w1) ->
    (clock_newer w1) ==> (clock_newer (w1 >-> make t1)))
  (zip ts ws)
= ⟨ rearranging according to trace properties ⟩
  all (\(w1,w2) ->
    (clock_newer w1) ==> (clock_newer w2))
  (zip ws (tail ws))
⇒ ⟨ adding pre-condition ⟩
  (clock_newer w) &&
    (all (\(w1,w2) ->
      (clock_newer w1) ==> (clock_newer w2))
    (zip ws (tail ws)))
= ⟨ rewriting w ⟩
  (clock_newer (head ws)) &&
    (all (\(w1,w2) ->
      (clock_newer w1) ==> (clock_newer w2))
    (zip ws (tail ws)))
⇒ ⟨ chaining the implications ⟩
  (clock_newer (last ws))
= ⟨ HL.3.4.2 ⟩
  (clock_newer (w >-> mapM make ts))
⇒ ⟨ from HL.4.2.1 ⟩
  (clock_newer (w >-> mapM make ts)) &&
  (clock_newer (w >-> do {mapM make ts;exec n m}))
⇒ ⟨ adding HL.4.1.3 ⟩
  (clock_newer (w >-> mapM make ts)) &&
  (clock_newer (w >-> do {mapM make ts;exec n m}))
  &&
    (((w >-> make t) ==
      (w >-> do {mapM make ts;exec n m}))
    || ((w >-> make t)==(w >-> mapM make ts)))
⇒ ⟨ substitution ⟩
  (clock_newer (w >-> make t))
⟨ add pre-condition ⟩
  (clock_newer w) ==> (clock_newer (w >-> make t))
```

Lemma HL.4.3.2

If the `clock_newer` predicate is true before making a sequence of targets, then it will be true afterwards, and true for all intermediate worlds.

```
(clock_newer w) ==> (all clock_newer (trace make ts w))
```

Proof: Induction on the list of targets. In the inductive case, the most recent `clock_newer` fact is extracted from the list, and is used (via **HL.4.3.1**) to prove the next one.

⟨ The following stronger fact is proven: ⟩

```
((clock_newer w) && (i>0) && (i <= (length ts)+1))
  ==> (all clock_newer (take i (trace make ts w)))
```

Base Case: `i==1`

⟨ assuming one `clock_newer` pre-condition ⟩

```
  (clock_newer w)
```

⇒ ⟨ **HL.3.4.1** ⟩

```
  clock_newer (head (trace make ts w))
```

= ⟨ `take`, `all` properties ⟩

```
  all clock_newer (take 1 (trace make ts w))
```

⟨ adding pre-conditions, and introducing `i`. ⟩

```
  ((clock_newer w) && (i>0) && (i <= (length ts)+1))
    ==> (all clock_newer (take i (trace make ts w)))
```

Inductive Case: `i==j`

⟨ *inductive hypothesis* ⟩

```
  ((clock_newer w) && ((j-1) > 0) && ((j-1) <= (length ts)+1))
    ==> (all clock_newer (take (j-1) (trace make ts w)))
```

⇒ ⟨ Since local pre-conditions supersede the pre-conditions in inductive hypothesis (`j` is at least `2`, and at most `(length ts)+1`), the following is true. ⟩

```
  (all clock_newer (take (j-1) (trace make ts w)))
```

= ⟨ Letting `w = trace make ts w` ⟩

```
  (all clock_newer (take (j-1) ws))
```

⇒ ⟨ extracting a specific `clock_newer` fact ⟩

```
  (all clock_newer (take (j-1) ws)) &&
    (clock_newer (ws!!(j-2)))
```

⇒ ⟨ using **HL.4.3.1** ⟩

```
  (all clock_newer (take (j-1) ws)) &&
    (clock_newer (ws!!(j-2) >-> make (ts!!(j-1))))
```

⇒ ⟨ and, adding `trace` property **HL.3.6** ⟩

```
  (all clock_newer (take (j-1) ws)) &&
    (clock_newer (ws!!(j-2) >-> make (ts!!(j-1)))) &&
    (((ws!!(j-2)) >-> make ts!!(j-2))==ws!!(j-1))
```

⇒ ⟨ substitution ⟩

```
  (all clock_newer (take (j-1) ws)) &&
    (clock_newer (ws!!(j-1)))
```
= ⟨ rearranging ⟩
```
  (all clock_newer (take j ws))
```
⇒ ⟨ adding pre-condition and re-instating `i` ⟩
```
  ((clock_newer w) && (i>0) && (i <= length ts))
    ==> (all clock_newer (take i (trace make ts w)))
```
End of Inductive Proof

⟨ Let `i = (length ts+1)` ⟩
```
  ((clock_newer w) &&
    ((length ts)+1 > 0) && ((length ts)+1 <= (length ts)+1))
    ==> (all clock_newer (take ((length ts)+1) (trace make ts w)))
```
⇒ ⟨ rewriting ⟩
```
  ((clock_newer w) ==>
    (all clock_newer (trace make ts w)))
```

Definition of `pre_make`

For almost all lemmas associated with properties of `make` from now on, it will first be necessary for `pre_make` to be true. The predicate says two things: Firstly, in the world, no file is newer than the current value of the clock. Secondly, the target being made is a DAG-Tree.

```
-- Make's pre-condition
pre_make :: Target -> World -> Bool
pre_make t w = (clock_newer w) && (isDAGT t)
```

Lemma HL.4.4

If pre-make is true when calling make on a target, it guarantees that it will also be true for all recursive make calls.

```
let t  = (Target n m ts)
    ws = trace make ts w
in (pre_make t w) ==>
     all (\(t1,w1) -> pre_make t1 w1) (zip ts ws)
```

Proof: A combination of **HL.2.2** and **HL.4.3.2**. **HL.4.3.2** states that `clock_newer` holds for all recursive make calls. **HL.2.2** states that if `t` is a DAG-Tree, all its immediate dependencies are also DAG-Trees. The conjunction of this is `clock_newer`.

⟨ Assume pre-condition holds. ⟩
```
(pre_make t w)
```
⇒ ⟨ expanding `pre_make` defn. ⟩
```
(clock_newer w) && (isDAGT t)
```
⇒ ⟨ using **HL.4.3.2** and **HL.2.2** on the left and right-hand-side expressions respectively ⟩

```
(all clock_newer ws) && (all isDAGT ts)
```
⇒ ⟨ merge ⟩
```
all (\(t1,w1) -> (clock_newer w1) && (isDAGT t1)) (zip ts ws)
```
= ⟨ `pre_make` defn. ⟩
```
all (\(t1,w1) -> pre_make t1 w1) (zip ts ws)
```
= ⟨ add pre-condition ⟩
```
(pre_make t w) ==>
  all (\(t1,w1) -> pre_make t1 w1) (zip ts ws)
```

10.4 General File-System Proofs

Overview

Most of the proofs in this section relate to simple high-level ways `make` interacts with files in the file-system, generally touching the complexities of DAG-Trees as little as possible at this early stage.

The most important parts are as follows:

- The introduction and use of the `filesSame` predicate for reasoning about how a program alters the file-system.
- Theorem **H.1** shows that if the name of a file doesn't appear anywhere in a target, then that file won't have been touched after the target is rebuilt.
- Theorem **H.2** is a general property of `exec` and `getFileTime`, showing that after running `exec`, the file touched is newer than any other file in the file-system.
- **HL.5.5** is a small but important lemma, which states that the value returned from `make` is always equal to the resultant time-stamp of the file just built.
- Theorem **H.3** is a powerful statement about how files are affected by `make`. Either a file isn't touched at all, or it is newer than the clock-time before `make` was called.

Definition of `filesSame`

The `fileSame` predicate is quite simple. It states that for all the files in `ns` they're time-stamps are the same in `w1` as they are in `w2`.

```
-- are files 'ns' the same in two different worlds?
filesSame :: [Name] -> World -> World -> Bool
filesSame ns w1 w2 = all fileSame ns
  where
    fileSame n = (w1 >~> getFileTime n)==(w2 >~> getFileTime n)
```

Lemma HL.5.1

The predicate stating whether the modification times of a particular set of filenames differ between two worlds defines an equivalence relation on all worlds.

Symmetric

```
(filesSame ns w1 w2) == (filesSame ns w2 w1)
```

Reflexive

```
(filesSame ns w1 w1)
```

Transitive

```
((filesSame ns w1 w2) && (filesSame ns w2 w3)) ==> (filesSame ns w1 w3)
```

Proof: All trivial.

Lemma HL.5.2

If the files `ns1` don't change between two particular worlds, and the same is true of files `ns2`, then the union of `ns1` and `ns2` won't change between those worlds.

```
(filesSame ns1 w1 w2) && (filesSame ns2 w1 w2) ==>
  (filesSame (ns1++ns2) w1 w2)
```

Proof: Easy. The `filesSame` function is defined as the conjunction of statements about individual files, and `(and (b1++b2)) == ((and b1) && (and b2))`.

Lemma HL.5.3

If a file `n` is not a member of the set of all filenames in target `t`, then its modification time will not change when `t` is being made.

```
(pre_make t w) && (n `notElem` (allnames t)) ==>
  (filesSame [n] w (w >-> make t))
```

Proof: Structural induction on target `t`. The base case is easy. In the inductive case, it is necessary to show (using the inductive hypothesis, and the transitivity of the `filesSame [n]` relation (**HL.5.1**)) that the recursive make calls do not change `n`. Finally, in the case that an `exec` occurs, we can show using the pre-condition that the top file of `t`, `(name t)`, is not equal to `n`.

Base Case: `t = (Leaf n1)`

⟨ **HL.4.1.2** ⟩

```
  (w >-> make (Leaf n1)) == w
```

⇒ ⟨ **HL.5.1** (reflexivity of `filesSame [n]`) ⟩

```
  (filesSame [n] w (w >-> make (Leaf n1)))
```

⇒ ⟨ adding pre-condition ⟩

```
  (pre_make t w) && (n `notElem` (allnames t)) ==>
    (filesSame [n] w (w >-> make t))
```

Inductive Case: `t = (Target n1 m ts)`

⟨ Inductive hypothesis ⟩

```
  (all (\t1 -> FORALL w1: (pre_make t1 w1) && (n `notElem` (allnames t1))
    ==> (filesSame [n] w1 (w1 >-> make t1))) ts)
```

⇒ ⟨ Firstly, let `ws = trace make ts w`. Then, instantiating all `w1` in the inductive hypothesis ⟩

```
  all (\(t1,w1) -> (pre_make t1 w1) &&
    (n `notElem` (allnames t1))
      ==> (filesSame [n] w1 (w1 >-> make t1))) (zip ts ws)
```

⇒ ⟨ Assuming local `pre_make` pre-condition, and using **HL.4.4** ⟩

```
  all (\(t1,w1) -> (n `notElem` (allnames t1))
      ==> (filesSame [n] w1 (w1 >-> make t1))) (zip ts ws)
```

⇒ ⟨ Also, since `((allnames t1) `subset` (allnames t))`, assuming the second local pre-condition yields: ⟩

```
  all (\(t1,w1) -> (filesSame [n] w1 (w1 >-> make t1)))
    (zip ts ws)
```

= ⟨ rewriting ⟩

```
  all (\(w1,w2) -> filesSame [n] w1 w2)) (zip ws (tail ws))
```

⇒ ⟨ Since `filesSame` is an equivalence relation (**HL.5.1**), and `fst (ws!!i) == snd (ws!!(i-1))`: ⟩

```
  (filesSame [n] w (last ws))
```

= ⟨ **HL.3.4.2** ⟩

```
  (filesSame [n] w (w >-> update_deps ts))
```

⇒ ⟨ Adding `make` defn., from **HL.4.1.3** ⟩

```
  ((w >-> make (Target n m ts)) ==
    (w >-> do {mapM make ts;exec n m})
  || ((w >-> make (Target n m ts)) ==
       (w >-> mapM make ts)))
  && (filesSame [n] w (w >-> mapM make ts))
```

If `exec` occurs:

⟨ removing other expression ⟩

```
    (w >-> make t) ==
      (w >-> do {mapM make ts;exec n1 m})
    && (filesSame [n] w (w >-> mapM make ts))
```

⇒ ⟨ and, since from initial `filesSame` pre-condition, `n/=n1` ⟩

```
    (w >-> make t) ==
      (w >-> do {mapM make ts;exec n1 m})
    && (filesSame [n] w (w >-> mapM make ts))
    && (filesSame [n] (w >-> mapM make ts)
      (w >-> do {mapM make ts;exec n1 m}))
```

⇒ ⟨ substitution ⟩

```
    (filesSame [n] w (w >-> make t))
```

If no `exec` occurs:

⟨ removing other expression ⟩

```
    (w >-> make t) == (w >-> mapM make ts)
    && (filesSame [n] w (w >-> mapM make ts))
```

⇒ ⟨ substitution ⟩

```
    (filesSame [n] (w >-> make t))
```

End-If

⟨ Adding assumed pre-conditions ⟩

```
  (pre_make t w) && (n `notElem` (allnames t)) ==>
    (filesSame [n] w (w >-> make t))
```

Theorem H.1

If no file in a set of files `ns` is a member of the set of all filenames in target `t`, then the modification times of the files `ns` will not have changed after making `t`.

```
(pre_make t w) && (ns `disjoint` (allnames t))
  ==> (filesSame ns w (w >-> make t))
```

Proof: A direct consequence of **HL.5.3**. Lemma **HL.5.2** is applied directly – any list of files can be represented as the concatenation of a number of one-element lists.

Lemma HL.5.4

A non-constructive definition of the `clock_newer` pre-condition.

```
(clock_newer (f,k)) ==> ((f,k) >~> getFileTime n) <= (FileTime k)
```

Proof: Relatively straight-forward. The `clock_newer` pre-condition states a fact about all files – this lemma takes an arbitrary file `n` and removes any information stated about other files.

Case: `(lookup n f) == Nothing`

⟨ `getFileTime` defn. ⟩

```
  ((f,k) >~> getFileTime n) ==
    (case (lookup n f) of
           Nothing  -> NoFileTime
           (Just t) -> FileTime t)
```

= ⟨ rewriting ⟩

```
  ((f,k) >~> getFileTime n) == NoFileTime
```

⇒ ⟨ defn. of `<` on `FileTime`s. ⟩

```
  ((f,k) >~> getFileTime n) < (FileTime _)
```

⇒ ⟨ instantiating ⟩

```
  ((f,k) >~> getFileTime n) < (FileTime k)
```

⇒ ⟨ adding pre-condition ⟩

```
  (clock_newer (f,k)) ==>
    ((f,k) >~> getFileTime n) <= (FileTime k)
```

Case: `(lookup n f) == (Just k1)`

⟨ assume `(clock_newer (f,k))`. ⟩

```
  (clock_newer (f,k)) && (lookup n f)==(Just k1)
```

= ⟨ `clock_newer` defn. ⟩

```
  (k >= (foldl max 0 (map snd f))) && (lookup n f)==(Just k1)
```

⇒ ⟨ Using `max` and `lookup` defns. ⟩

```
  (k >= k1) && (lookup n f)==(Just k1)
```

⇒ ⟨ `getFileTime` defn. ⟩

```
  (k >= k1) && ((f,k) >~> getFileTime n)==(FileTime k1)
```

⇒ ⟨ `FileTime` comparison ⟩

```
  ((f,k) >~> getFileTime n) <= (FileTime k)
```

⟨ adding pre-condition. ⟩

```
  (clock_newer (f,k)) ==>
    ((f,k) >~> getFileTime n) <= (FileTime k)
```

Theorem H.2

Directly after "executing" (or touching) a file, that file will be newer than any other file in the filesystem.

```
(clock_newer w) && (n/=n1) ==>
  ((w >~> do {exec n m; getFileTime n}) >
   (w >~> do {exec n m; getFileTime n1}))
```

Proof: Using **HL.5.4**, and properties of `exec`, it can be shown that the time-stamp of `n1` afterwards is less than or equal to the value of the clock beforehand. Also, again from properties of `exec`, the timestamp of `n` is greater than the value of the clock beforehand. Therefore the `n`'s time-stamp is greater than that of `n1`.

```
let (f,k) = w
```

⟨ assuming pre-condition ⟩

```
  (clock_newer w) && (n/=n1)
```

⇒ ⟨ using **HL.5.4** ⟩

```
  (w >~> getFileTime n1)<=(FileTime k) && (n/=n1)
```

⇒ ⟨ `exec` properties, since `n/=n1` ⟩

```
  (w >~> getFileTime n1)<=(FileTime k) &&
    (filesSame [n1] w (w >-> exec n m))
```

= ⟨ `filesSame` defn. ⟩

```
  (w >~> getFileTime n1)<=(FileTime k) &&
  (w >~> getFileTime n1) == (w >-> (exec n m)) >~> getFileTime n1
```

$\Rightarrow$ ⟨ equality ⟩

```
  (((w >-> exec n m) >~> getFileTime n1)
    <= (FileTime (snd w))
```

$=$ ⟨ Let `w1 = (w >-> exec n m)` ⟩

```
  (w1 >~> getFileTime n1) <= (FileTime k)
```

$\Rightarrow$ ⟨ `exec` property ⟩

```
  ((w1 >~> getFileTime n1) <= (FileTime k))
    && (FileTime ((snd w1)-1))==(FileTime k)
    && (w1 >~> getFileTime n)==(FileTime (snd w1))
```

$\Rightarrow$ ⟨ equality ⟩

```
  ((w1 >~> getFileTime n1) <= (FileTime ((snd w1)-1)))
    && (w1 >~> getFileTime n)==(FileTime (snd w1))
```

$\Rightarrow$ ⟨ algebra ⟩

```
  ((w1 >~> getFileTime n1) < (FileTime (snd w1)))
    && (w1 >~> getFileTime n)==(FileTime (snd w1))
```

$\Rightarrow$ ⟨ equality ⟩

```
  ((w1 >~> getFileTime n1) < (w1 >~> getFileTime n))
```

$=$ ⟨ rewriting `w1` ⟩

```
  (((w >-> exec n m) >~> getFileTime n1) <
  ((w >-> exec n m) >~> getFileTime n))
```

$=$ ⟨ using **HL.3.2.3** ⟩

```
  ((w >~> do {exec n m; getFileTime n1}) <
   (w >~> do {exec n m; getFileTime n}))
```

⟨ adding pre-condition ⟩

```
(clock_newer w) (n/=n1) ==>
  ((w >~> do {exec n m; getFileTime n}) >
   (w >~> do {exec n m; getFileTime n1}))
```

Lemma HL.5.5

The time-stamp returned after making `t` is always equal to the resultant modification time of `t`'s root filename.

```
(pre_make t w) ==>
  (w >~> do {make t;getFileTime (name t)}) == (w >~> make t)
```

Proof: Case analysis on target. The leaf case is easy. In the non-leaf case, case-analysis is performed on the `if` expression. If the `exec` takes place, then it is obvious from `exec` and `return` semantics that the two values are the same. If not, it must be shown that the value `mtime` in the `make` body is still the correct time associated with top file *after* all the sub-targets are made. Since this is the value eventually returned, it will represent the actual time of the file `n`.

Leaf case: `t = (Leaf n)`

⟨ **HL.4.1.4** ⟩

```
  (w >~> make (Leaf n)) == (w >~> getFileTime n)
```

⇒ ⟨ therefore, because `getFileTime` does not change the world (**HL.4.1.1**): ⟩

```
(w >~> do {make t;getFileTime (name t)})==(w >~> make t)
```

⟨ and, adding pre-condition: ⟩

```
(pre_make t w) ==>
  (w >~> do {make t;getFileTime (name t)}) == (w >~> make t)
```

Non-Leaf case: `t = (Target n m ts)`

⟨ `make` defn. ⟩

```
(w >=> make t) ==
  (w >=> do {
    mtime <- getFileTime n;
    ctimes <- update_deps ts;
    if (mtime <= newest ctimes)
      then do {exec n m; getFileTime n}
      else return mtime})
```

= ⟨ since `getFileTime` is idempotent (**HL.4.1.1**): ⟩

```
(w >=> make t) ==
  (w >=> do {
    ctimes <- update_deps ts;
    if ((w >~> getFileTime n) <= newest ctimes)
      then do {exec n m; getFileTime n}
      else return (w >~> getFileTime n)})
```

= ⟨ monad laws, and **HL.3.2.1** ⟩

```
(w >=> make t) ==
  ((w >-> update_deps ts) >=> do {
    if ((w >~> getFileTime n) <=
        newest (w >~> update_deps ts))
      then do {exec n m; getFileTime n}
      else return (w >~> getFileTime n)})
```

If `(mtime <= newest ctimes):`

```
  (w >=> make t) ==
    ((w >-> update_deps ts) >=>
      do {exec n m; getFileTime n})
```

= ⟨ **HL.3.2.1** ⟩

```
  (w >=> make t) ==
    (w >=> do {
      update_deps ts;
      exec n m; getFileTime n})
```

⇒ ⟨ because `getFileTime` is an idempotent action (**HL.4.1.1**) ⟩

```
  (w >~> make t) == ((w >=> make t) >~> getFileTime n)
```

= ⟨ **HL.3.2.3** ⟩

```
  (w >~> make t) == (w >~> do {make t; getFileTime (name t)})
```

If `(mtime > newest ctimes):`

```
  (w >=> make t) ==
    ((w >-> update_deps ts) >=>
      return (w >~> getFileTime n))
```

⇒ ⟨ because `t` is a DAG-Tree, using **HL.2.8.1** ⟩

```
(w >=> make t) ==
  ((w >-> mapM make ts) >=>
    return (w >~> getFileTime n))
&& all (\t1 -> n `notElem` (allnames t1)) ts
```

$\Rightarrow$ ⟨ **HL.5.3**, applied to all recursive makes: ⟩

```
(w >=> make t) ==
  ((w >-> mapM make ts) >=>
    return (w >~> getFileTime n))
&& all (\(w1,w2) -> filesSame [n] w1 w2)
  (zip (trace make ts w)
    (tail (trace make ts w)))
```

$\Rightarrow$ ⟨ **HL.5.1**, **HL.3.4.2** ⟩

```
(w >=> make t) ==
  ((w >-> mapM make ts) >=>
    return (w >~> getFileTime n))
&& (filesSame [n] w (w >-> mapM make ts))
```

$\Rightarrow$ ⟨ `filesSame` defn. ⟩

```
(w >=> make t) ==
  ((w >-> mapM make ts) >~> getFileTime n)
```

$\Rightarrow$ ⟨ since `getFileTime` is idempotent (**HL.4.1.1**) ⟩

```
(w >~> make t) == (w >~> do {make t; getFileTime (name t)})
```

End-If

⟨ .. and adding pre-condition: ⟩

```
(pre_make t w) ==>
  (w >~> do {make t; getFileTime (name t);}) == (w >~> make t)
```

Lemma HL.5.6

After calling make, files only get newer or don't change - they don't get older.

```
(pre_make t w) ==>
  (w >~> getFileTime n) <=
    (w >~> do {make t w; getFileTime n})
```

Proof: Induction on target.

Lemma HL.5.7

After running make, the clock only gets newer or stays the same.

```
(pre_make t w) ==> (snd w)<=(snd (w >-> make t))
```

Proof: Induction on target.

Theorem H.3

After running make, either the modification time of a file hasn't changed, or it is greater than the clock-time before running make.

```
(pre_make t w) ==>
  let k = w >~> do {make t; getFileTime n}
  in (k == w >~> getFileTime n) || (k > (FileTime (snd w)))
```

Proof:

Induction on the target structure. The base case is straight-forward.

The inductive case first deals with the inductive hypothesis, then case analysis is performed on whether the arbitrary file `n` is or is not touched by recursive make calls. If it is, it can be shown with **HL.5.6** and **HL.5.7** that will be newer than the clock before running make. If it isn't, then it hasn't changed. The disjuntion of these two possibilities yields the first part of the proof.

After making the sub-targets, it's possible an `exec` will have taken place. If it has, then once again, either `n` will stay the same or it's value will be greater than that of the clock before running `make`.

Base Case: `t = (Leaf n1)`

⟨ **HL.4.1.2** ⟩

```
  (w >-> make (Leaf n1)) == w
```

⇒ ⟨ function application ⟩

```
  ((w >-> make t) >~> getFileTime n) == (w >~> getFileTime n)
```

= ⟨ **HL.3.2.1** ⟩

```
  (w >~> do {make t; getFileTime n}) == (w >~> getFileTime n)
```

⇒ ⟨ using conjunction, and adding pre-condition: ⟩

```
(pre_make t w) ==>
  let k = w >~> do {make t; getFileTime n}
  in (k == w >~> getFileTime n) || (k > (FileTime (snd w)))
```

Inductive Case: `t = (Target n1 m1 ts)`

⟨ *Inductive hypothesis:* ⟩

```
  all (\t1 -> FORALL w1: (pre_make t1 w1) ==>
    let k1 = w1 >~> do {make t1; getFileTime n}
    in (k1 == w1 >~> getFileTime n) || (k1 > (FileTime (snd w1)))
  ) ts
```

⇒ ⟨ Letting `ws = trace make ts w`, ⟩

```
  all (\(t1,w1) -> (pre_make t1 w1) ==>
    let k1 = w1 >~> do {make t1; getFileTime n}
    in (k1 == w1 >~> getFileTime n) || (k1 > (FileTime (snd w1)))
  ) (zip ts ws)
```

⇒ ⟨ Assuming `(pre_make t w)` pre-condition, and using **HL.4.4**: ⟩

```
  all (\(t1,w1) ->
    let k1 = w1 >~> do {make t1; getFileTime n}
    in (k1 == w1 >~> getFileTime n) || (k1 > (FileTime (snd w1)))
  ) (zip ts ws)
```

= ⟨ Or, alternatively, using **HL.3.6** ⟩

```
all (\(t1,w1,w2) -> (w2 >~> getFileTime n)==(w1 >~> getFileTime n)
  || (w2 >~> getFileTime n)>(FileTime (snd w1))
) (zip3 ts ws (tail ws))
```

If file `n` is touched:

⟨ Let `i` be a number such that making target `ts!!i` causes the modification time of file `n` to be changed. Using the inductive hypothesis, this means, firstly, that: ⟩

```
(ws!!(i+1) >~> getFileTime n)>(FileTime (snd (ws!!i)))
```

⟨ Secondly, using **HL.5.7**, we can show: ⟩

```
all (\(w1,w2) ->
  (FileTime (snd w1)) <= (FileTime (snd w2))
) (take (i-1) (zip ws (tail ws)))
```

⇒ ⟨ algebra ⟩

```
(FileTime (snd (ws!!i))) >= (FileTime (snd w))
```

⟨ Thirdly, applying **HL.5.6** (and **HL.4.4**): ⟩

```
all (\(w1,w2) ->
  (w1 >~> getFileTime n) <= (w2 >~> getFileTime n))
) (drop (i+1) (zip ws (tail ws)))
```

⇒ ⟨ using algebra ⟩

```
(ws!!(i+1) >~> getFileTime n) <= ((last ws) >~> getFileTime n)
```

⟨ So, using the following three proven facts: ⟩

```
(FileTime (snd (ws!!i))) >= (FileTime (snd w))
(ws!!(i+1) >~> getFileTime n)>(FileTime (snd (ws!!i)))
(ws!!(i+1) >~> getFileTime n) <= ((last ws) >~> getFileTime n)
```

⟨ infer ⟩

```
((last ws) >~> getFileTime n) > (FileTime (snd w))
```

= ⟨ **HL.3.4.2**, `update_deps` defn. ⟩

```
((w >-> update_deps ts) >~> getFileTime n) > (FileTime (snd w))
```

If file `n` is not touched:

⟨ No number `i` exists, as in the previous section, so the modification time never changes. ⟩

```
all (\(w1,w2) ->
  (w1 >~> getFileTime n) == (w2 >~> getFileTime n)
) (zip ws (tail ws))
```

⇒ ⟨ equality ⟩

```
((head ws) >~> getFileTime n) ==
  ((last ws) >~> getFileTime n)
```

= ⟨ **HL.3.4.1**, **HL.3.4.2** ⟩

```
(w >~> getFileTime n)==
  ((w >-> update_deps ts) >~> getFileTime n)
```

If-lifting, this yields

```
let w' = (w >-> update_deps ts)
in  (w' >~> getFileTime n) > (FileTime (snd w))
      || (w >~> getFileTime n)==(w' >~> getFileTime n)
```

⟨ After running `update_deps`, the file `n1` may or may not be touched: ⟩

If `n1` is not touched:

```
    w' == (w >-> make t)
```

⇒ ⟨ rearranging ⟩

```
    let k = w >~> do {make t; getFileTime n}
    in (k == w >~> getFileTime n) || (k > (FileTime (snd w)))
```

If `n1` is touched, but `n1/=n`:

```
    filesSame [n] w' (w >-> make t)
```

= ⟨ `filesSame` defn. ⟩

```
    (w' >~> getFileTime n) == ((w >-> make t) >~> getFileTime n)
```

⇒ ⟨ rearranging: ⟩

```
    let k = w >~> do {make t; getFileTime n}
    in (k == w >~> getFileTime n) || (k > (FileTime (snd w)))
```

If `n1` is touched, and `n1==n`:

```
    (w >-> make t) == (w' >-> exec n1 m1)
```

⇒ ⟨ `exec` defn. ⟩

```
    ((w >-> make t) >~> getFileTime n)==
      (FileTime (snd (w >-> make t)))
    && (FileTime (snd (w >-> make t))) > (FileTime (snd w'))
```

⟨ And, using the following previously proven fact: ⟩

```
    (FileTime (snd w')) <= (FileTime (snd w))
```

⟨ infer ⟩

```
    ((w >-> make t) >~> getFileTime n) > (FileTime (snd w))
```

⇒ ⟨ adding disjunction and rearranging ⟩

```
    let k = w >~> do {make t; getFileTime n}
    in (k == w >~> getFileTime n) || (k > (FileTime (snd w)))
```

If-Lifting, and adding pre-condition:

```
  (pre_make t w) ==>
    let k = w >~> do {make t; getFileTime n}
    in (k == w >~> getFileTime n) || (k > (FileTime (snd w)))
```

10.5 Make "`newest_dep`" Proofs

Overview

This section introduces the `newest_dep` function, and then proceeds to prove that after `make` is called, the time-stamp associated with the top file in a target is always newer than all the files anywhere else in the tree (Theorem **H.4**).

The proof of Theorem **H.4** is huge – lemmas **HL.6.3.1**, **HL.6.3.2** and **HL.6.4**, all large and cumbersome, are required to bootstrap the inductive proof of Theorem **H.4**.

Apart from that main proof (which is the longest and most difficult in the document), there is little else of interest in this section.

Definition of `newest_dep`

The `newest_dep` function returns the newest time associated with every dependency of a target (excluding the target itself).

```
-- get the newest time of the *dependencies* of the target.
newest_dep :: World -> Target -> FileTime
newest_dep w t =
  newest (map (\n -> w >~> getFileTime n) (allnames2 t))
```

Lemma HL.6.1

A useful fact about `newest_dep`.

```
(newest_dep t w) ==
  (newest (map (\t1 -> newest
    [w >~> getFileTime (name t1), newest_dep t1 w]) (deps t)))
```

Proof: Function expansion and substitution.

Lemma HL.6.2

If, after making a target, the head file is newer than all sub-targets, and that same head-file was never touched at all, then none of the files associated with the sub-targets will have been touched either.

```
let w1 = w >-> make t
in
    (pre_make t w) && (filesSame [name t] w w1)
    && (w1 >~> getFileTime (name t)) > (newest_dep t w1)
    ==> (filesSame (allnames t) w w1)
```

Proof: The top file after running `make` must not be greater than the value of the clock before running make, from the `clock_newer` pre-condition in `pre_make`. **H.3** states that all the files in the target after running make either haven't changed, or are greater than the clock before running make. However, the latter is impossible, since the top file is newer than all the other files in the target. Therefore, no files in the target can have changed.

⟨ First, the `filesSame` pre-condition states the following. **(1)** ⟩

`(w >~> getFileTime (name t)) == (w1 >~> getFileTime (name t))`

⟨ Secondly, the `clock_newer` pre-condition implies ⟩

`(clock_newer w)`

⇒ ⟨ **HL.5.4**. **(2)** ⟩

`(w >~> getFileTime (name t)) <= (FileTime (snd w))`

⟨ Thirdly, the third pre-condition is.. ⟩

`(w1 >~> getFileTime (name t)) > (newest_dep t w1)`

= ⟨ `newest_dep` defn. ⟩

```
w1 >~> getFileTime (name t)) >
  (newest [w1 >~> getFileTime n | n <- allnames2 t])
```

= ⟨ rearranging according to `newest` properties. **(3)** ⟩

```
all (\n -> (w1 >~> getFileTime (name t)) >
  (w1 >~> getFileTime n)) (allnames2 t)
```

⟨ Merging **(1)**, **(2)** and **(3)**. ⟩

```
all (\n -> (FileTime (snd w)) > (w1 >~> getFileTime n))
  (allnames2 t)
```

⇒ ⟨ **H.3** ⟩

```
all (\n -> (w >~> getFileTime n)) > (w1 >~> getFileTime n))
  (allnames2 t)
```

= ⟨ `filesSame` defn. ⟩

```
(filesSame (allnames2 t) w w1)
```

⇒ ⟨ and, since `(name t)` doesn't change either, ⟩

```
(filesSame (allnames t) w w1)
```

Lemma HL.6.3.1

At any stage during the making of a sequence of sub-targets, the value of the newest time-stamp returned from the making of all preceding targets is equal to the newest of the time-stamps of the top files in each of the preceding targets. (Assuming that after running `make`, the time returned is newer than the time-stamp of all the target's dependencies.)

```
let t  = (Target n m ts)
    ks = w >~> mapM make ts
    ws = trace make ts w
in
    (pre_make t w) && (i >= 0) && (i <= length ts)
      && (all (\(t1,w1,k1) -> k1 > newest_dep t1 w1)
           (take i (zip3 ts (tail ws) ks)))
     ==> (newest (take i ks))==(newest [(ws!!i) >~>
           getFileTime (name t1) | t1 <- (take i ts)])
```

Proof: Induction on list of sub-targets. The base case is quite simple. The inductive case requires the use of algebraic properties of `max` to split a single `newest` expression into a "nested" `newest` expression. Each individual inner expression is then converted to a simpler form, and then the nested expression is folded down to a single non-nested one. The conversion that takes place internally is somewhat technical, but it is logically sound.

Base case:`i==0`

⟨ `newest` defn. ⟩

```
(newest []) == NoFileTime
```

⇒ ⟨ trivial equality ⟩

```
(newest []) == (newest [])
```

⇒ ⟨ introducing values for two `[]`s. ⟩

```
(newest (take 0 ks))==(newest [(ws!!0) >~>
  getFileTime (name t1) | t1 <- (take 0 ts)])
```

⇒ ⟨ adding pre-conditions ⟩

```
(pre_make t w) && (i >= 0) && (i <= length ts)
  && (all (\(t1,w1,k1) -> k1 > newest_dep t1 w1)
      (take i (zip3 ts (tail ws) ks)))
==>
  (newest (take i ks))==(newest [(ws!!i) >~>
    getFileTime (name t1) | t1 <- take i ts ])
```

Inductive case: `i==j`

⟨ *Inductive hypothesis:* ⟩

```
  (pre_make t w) && (j-1 >= 0) && (j-1 <= (length ts))
    && (all (\(t1,w1,k1) -> k1 > newest_dep t1 w1)
        (take (j-1) (zip3 ts (tail ws) ks)))
   ==>
    (newest (take (j-1) ks))==(newest [(ws!!(j-1)) >~>
      getFileTime (name t1) | t1 <- (take (j-1) ts)]))
```

⟨ Assume pre-conditions, which supersede inductive hypothesis pre-conditions, yielding ⟩

```
  (newest (take (j-1) ks))==(newest [(ws!!(j-1)) >~>
    getFileTime (name t1) | t1 <- (take (j-1) ts)]))
  && ((ks!!(j-1)) > newest_dep (ts!!(j-1)) (ws!!j))
```

⇒ ⟨ Adding fact, from **HL.5.5** ⟩

```
  (newest (take (j-1) ks))==(newest [(ws!!(j-1)) >~>
    getFileTime (name t1) | t1 <- (take (j-1) ts)]))
  && (ks!!(j-1) == (ws!!j) >~> getFileTime (name (ts!!(j-1))))
  && ((ks!!(j-1)) > newest_dep (ts!!(j-1)) (ws!!j))
```

⇒ ⟨ combining two `max` expressions ⟩

```
  (newest (take j ks))==(newest [(ws!!(j-1)) >~>
    getFileTime (name t1) | t1 <- (take (j-1) ts)]++
    [(ws!!j) >~> getFileTime (name (ts!!(j-1))]))
  && ((ks!!(j-1)) > newest_dep (ts!!(j-1)) (ws!!j))
```

⟨ Now, assuming implicitly that the inequality at the bottom of the above statement holds, proceed to show that the left-to-right equality which we desire to prove is indeed correct. ⟩

```
(newest (take j ks))
```

= ⟨ from equality above ⟩

```
(newest [(ws!!(j-1)) >~>
  getFileTime (name t1) | t1 <- (take (j-1) ts)]++
  [(ws!!j) >~> getFileTime (name (ts!!(j-1))]))
```

= ⟨ using commutative/idempotent properties of `max` ⟩

```
newest [
  newest [(ws!!(j-1)) >~> getFileTime (name t1),
          (ws!!j) >~> getFileTime (name (ts!!(j-1)))]
  | t1 <- (take (j-1) ts)
]
```

⟨ To perform the next (crucial) step, it is necessary to perform case-analysis on the different ways the two different files can interact. Firstly, define four new variables: ⟩

```
  let t2 = (ts!!(j-1))
      k2 = (ws!!j) >~> getFileTime (name t2)
      k1old = (w1!!(j-1)) >~> getFileTime (name t1)
      k1new = (w1!!j) >~> getFileTime (name t1)
```

⟨ The following is clearly true from **HL.5.6**: ⟩

```
  k1old <= k1new
```

Case: `(name t1)==(name t2)`

⟨ since it's the exact same file: ⟩

```
    (k1old <= k1new) && (k1new==k2) && (k1old <= k2)
```

⇒ ⟨ `k1old` is no greater than any other value ⟩

```
    (newest [k1old,k2]) == (newest [k1new,k2])
```

Case: `(name t1) ‘elem‘ (allnames2 t2)`

⟨ since, from assumption, `k2` is newer than all its dependencies: ⟩

```
    (k1old <= k1new) && (k2 > k1new)
```

⇒ ⟨ `k2` is definitely greater than all values ⟩

```
    (newest [k1old,k2]) == (newest [k1new,k2])
```

Case: `(name t1) ‘notElem‘ (allnames t2)`

⟨ `(name t1)` is not touched at all, so, from **HL.5.3**: ⟩

```
    (k1old==k1new)
```

⇒ ⟨ therefore, trivially: ⟩

```
    newest [k1old,k2]) == (newest [k1new,k2])
```

End of case-analysis.

⟨ The following has been shown: ⟩

```
  newest [(ws!!(j-1)) >~> getFileTime (name t1),
          (ws!!j) >~> getFileTime (name (ts!!(j-1)))]
  ==
  newest [(ws!!j) >~> getFileTime (name t1),
          (ws!!j) >~> getFileTime (name (ts!!(j-1)))]
```

⟨ So, the previous expression... ⟩

```
newest [
  newest [(ws!!(j-1)) >~> getFileTime (name t1),
          (ws!!j) >~> getFileTime (name (ts!!(j-1)))]
  | t1 <- (take (j-1) ts)
]
```

= ⟨ .. by simple substitution ⟩

```
newest [
  newest [(ws!!j) >~> getFileTime (name t1),
          (ws!!j) >~> getFileTime (name (ts!!(j-1)))]
  | t1 <- (take (j-1) ts)
]
```

= ⟨ rearranging according to `max` properties ⟩

```
newest [
  (ws!!j) >~> getFileTime (name t1) | t1 <- (take j ts)
]
```

⟨ Now, the following equality has been fully proven: ⟩

```
newest (take j ks)) ==
  newest [(ws!!j) >~> getFileTime (name t1)
           | t1 <- (take j ts)]
```

⟨ Finally, to conclude, substitute `i` for `j`, and reinstate all assumed pre-conditions: ⟩

```
(pre_make t w) && (i >= 0) && (i <= length ts)
  && (all (\(t1,w1,k1) -> k1 > newest_dep t1 w1)
      (take i (zip3 ts (tail ws) ks)))
==>
newest (take i ks)) == newest [(ws!!i) >~>
  getFileTime (name t1) | t1 <- (take i ts)]
```

Lemma HL.6.3.2

After running make on a sequence of targets, the time-stamp associated with the newest top file of all the targets will be equal to the time-stamp of the newest file anywhere in the targets (including all dependencies). (This is under the assumption that directly after running `make` on a target, the top file of the target is newer than all dependencies of that target. This is proven later in Theorem **H.4**.)

```
let t  = (Target n m ts)
    ws = trace make ts w
in
    (pre_make t w) && (i >= 0) && (i <= length ts)
      && (all (\(t1,w1) -> (w1 >~> getFileTime (name t1))
        > newest_dep t1 w1) (take i (zip ts (tail ws))))
     ==> ((newest [ws!!i >~> getFileTime (name t1)
                   | t1 <- (take i ts)])
         >= (newest [newest_dep t1 ws!!i
                     | t1 <- (take i ts)]))
```

Proof: Induction on the list of sub-targets. The base case is pretty simple. The inductive case, however, requires case analysis. If the top file of the new sub-target to have been made has not been changed, then, since it is newer than all its dependencies, then all its dependencies haven't changed either. If, however, the top file of the new sub-target has been changed, then using **H.3** it can be shown that the time associated with the top-file of the new sub-target will be newer than *everything*. The specific aspects of these two facts can be generalised by an inequality – that inequality is the goal of this proof.

Base case: `i==0`

⟨ `newest` defn. ⟩

```
(newest []) == NoFileTime
```

⇒ ⟨ trivial equality ⟩

```
(newest []) == (newest [])
```

⇒ ⟨ introducing values for two []s. ⟩

```
((newest [ws!!i >~> getFileTime (name t1)
               | t1 <- (take 0 ts)])
    >= (newest [newest_dep t1 ws!!i
                | t1 <- (take 0 ts)]))
```

⇒ ⟨ and, adding pre-conditions: ⟩

```
(pre_make t w) && (i >= 1) && (i <= length ts)
  && (all (\(t1,w1) -> (w1 >~> getFileTime (name t1))
    > newest_dep t1 w1) (take i (zip ts (tail ws))))
 ==> ((newest [ws!!i >~> getFileTime (name t1)
               | t1 <- (take i ts)])
    >= (newest [newest_dep t1 ws!!i
                | t1 <- (take i ts)]))
```

Inductive Case: `i==j`

⟨ *Inductive hypothesis* ⟩

```
(pre_make t w) && (j-1 >= 1) && (j-1 <= length ts)
  && (all (\(t1,w1) -> (w1 >~> getFileTime (name t1))
    > newest_dep t1 w1) (take (j-1) (zip ts (tail ws))))
 ==> ((newest [ws!!(j-1) >~> getFileTime (name t1)
               | t1 <- (take (j-1) ts)])
   >= (newest [newest_dep t1 ws!!(j-1)
               | t1 <- (take (j-1) ts)]))
```

⇒ ⟨ Assuming pre-conditions, and merging with inductive hypothesis ⟩

```
(pre_make t w) &&
  (newest [ws!!(j-1) >~> getFileTime (name t1)
           | t1 <- (take (j-1) ts)])
     >= (newest [newest_dep t1 ws!!(j-1)
                 | t1 <- (take (j-1) ts)])
  && (w!!j >~> getFileTime (name ts!!(j-1))) >
       (newest_dep ts!!(j-1) w!!j)
```

⇒ ⟨ Changing `pre_make` slightly using **HL.4.4**. ⟩

```
(pre_make ts!!(j-1) ws!!(j-1)) &&
  (newest [ws!!(j-1) >~> getFileTime (name t1)
           | t1 <- (take (j-1) ts)])
     >= (newest [newest_dep t1 ws!!(j-1)
                 | t1 <- (take (j-1) ts)])
  && (w!!j >~> getFileTime (name ts!!(j-1))) >
       (newest_dep ts!!(j-1) w!!j)
```

⟨ For simplicity, we define some new variables as follows: ⟩

```
let ts1 = (take (j-1) ts)
    t2  = (ts!!(j-1))
    w0  = ws!!(j-1)
    w1  = ws!!j
```

⟨ The statement above then becomes: ⟩

```
(pre_make t2 w0)
  && (newest [w0 >~> getFileTime (name t1) | t1 <- ts1])
      >= (newest [newest_dep t1 w0 | t1 <- ts1])
  && (w1 >~> getFileTime (name t2)) > (newest_dep t2 w1)
```

If `(filesSame [name t2] w0 w1)`**:**

⟨ Since it is known that: ⟩

```
  (pre_make t2 w0) && (filesSame [name t2] w0 w1) &&
    (w1 >~> getFileTime (name t2)) > (newest_dep t2 w1)
```

⇒ ⟨ **HL.6.2** ⟩

```
  (filesSame (allnames t2) w0 w1)
```

⟨ Adding fact based on **H.1** ⟩

```
  (filesSame (allnames t2) w0 w1) &&
    (filesSame ((allnames (Target n m ts1))
        \\\ (allnames t2)) w0 w1)
```

= ⟨ merging, using **HL.5.2** ⟩

```
  (filesSame ((allnames t2)++(allnames (Target n m ts1)))
    w0 w1)
```

⟨ Clearly, no files change at all between `w0` and `w1`. Therefore, the previously proven fact ⟩

```
  (newest [w0 >~> getFileTime (name t1) | t1 <- ts1])
     >= (newest [newest_dep t1 w0 | t1 <- ts1])
  && (w1 >~> getFileTime (name t2)) > (newest_dep t2 w1)
```

⇒ ⟨ from `filesSame` property ⟩

```
  (newest [w1 >~> getFileTime (name t1) | t1 <- ts1])
     >= (newest [newest_dep t1 w1 | t1 <- ts1])
  && (w1 >~> getFileTime (name t2)) > (newest_dep t2 w1)
```

⇒ ⟨ merge both inequalities ⟩

```
  (newest [w1 >~> getFileTime (name t1) | t1 <- ts1++[t2]])
     >= (newest [newest_dep t1 w1 | t1 <- ts1++[t2]])
```

If `(not (filesSame [name t2] w0 w1))`**:**

⟨ Define three more variables as follows: ⟩

```
  ns1 = (allnames (Target n m ts1))
  ns2 = (allnames t2)
  ns3 = ns1 \\\ ns2
```

⟨ From **H.1**. **(1)** ⟩

```
  (filesSame ns3 w0 w1)
```

⟨ And, from **H.3**. **(2)** ⟩

```
  (w1 >~> getFileTime (name t2)) > (FileTime (snd w0))
```

⟨ and, since `(clock_newer w0)` **(3)** ⟩

```
  (FileTime (snd w0)) >=
     (newest [w0 >~> getFileTime (name t1) | t1 <- ts1])
  && (FileTime (snd w0)) >=
      newest [newest_dep t1 w0 | t1 <- ts1])
```

⟨ putting **(1)**, **(2)** and **(3)** together: ⟩

```
  (w1 >~> getFileTime (name t2)) >
    (newest [w1 >~> getFileTime n1 | n1 <- ns3)
```

⟨ and, adding the already proven fact: ⟩

```
  (w1 >~> getFileTime (name t2)) >
    (newest [w1 >~> getFileTime n1 | n1 <- ns3)
  && (w1 >~> getFileTime (name t2)) > (newest_dep t2 w1)
```

= ⟨ And, using `newest_dep` defn., the second expression changes ⟩

```
  (w1 >~> getFileTime (name t2)) >
    (newest [w1 >~> getFileTime n1 | n1 <- ns3)
  && (w1 >~> getFileTime (name t2)) >
  (newest (map (\n1 -> w1 >~> getFileTime n1)
    (allnames2 t2)))
```

⇒ ⟨ since `(allnames t2)` is just `(name t2)` and `(allnames2 t2)` together. (**HL.2.5**) ⟩

```
  (w1 >~> getFileTime (name t2)) >
    (newest [w1 >~> getFileTime n1 | n1 <- ns3)
  && (w1 >~> getFileTime (name t2)) >
  (newest [w1 >~> getFileTime n1 | n1 <- ns2)
```

= ⟨ merging ⟩

```
  (w1 >~> getFileTime (name t2)) >=
    (newest [w1 >~> getFileTime n1 | n1 <- ns2++ns3)
```

⟨ The time associated with `(name t2)` is at least as new as every other file. Therefore, the following holds trivially: ⟩

```
  (newest [w1 >~> getFileTime (name t1) | t1 <- ts1++[t2]])
      >= (newest [newest_dep t1 w1 | t1 <- ts1++[t2]])
```

End-If

⟨ Removing all new variables, the following fact has been proven: ⟩

```
(newest [(ws!!j) >~> getFileTime (name t1)
       | t1 <- (take (j-1) ts)++[ts!!(j-1)]])
    >= (newest [newest_dep t1 ws!!j
       | t1 <- (take (j-1) ts)++[ts!!(j-1)]])
```

= ⟨ `take` defn. ⟩

```
(newest [(ws!!j) >~> getFileTime (name t1)
       | t1 <- take j ts]])
    >= (newest [newest_dep t1 ws!!j
       | t1 <- take j ts]])
```

⇒ ⟨ Adding pre-conditions, and replacing `i` ⟩

```
(pre_make t w) && (i >= 1) && (i <= length ts)
  && (all (\(t1,w1) -> (w1 >~> getFileTime (name t1))
    > newest_dep t1 w1) (take i (zip ts (tail ws))))
 ==> ((newest [ws!!i >~> getFileTime (name t1)
               | t1 <- (take i ts)])
    >= (newest [newest_dep t1 ws!!i
                | t1 <- (take i ts)]))
```

Lemma HL.6.4

After making a sequence of sub-targets, the newest value of all time-stamps returned by `make` will equal the time-stamp of the newest file of all of `t`'s dependencies. (Assuming that after each recursive make call, the time-stamp returned is newer than the time-stamp associated with any of that sub-target's dependencies – this is proven later in Theorem **H.4**)

```
let t  = (Target n m ts)
    ks = w >~> mapM make ts
    ws = trace make ts w
in
    (pre_make t w)
      && (all (\(t1,w1,k1) -> k1 > newest_dep t1 w1)
            (zip3 ts (tail ws) ks))
     ==> (newest ks)==(newest_dep t (last ws))
```

Proof: A merging of **HL.6.3.1** and **HL.6.3.2**. The former compares the returned time-stamps to the actual times associated with the top files of each sub-target. The latter compares those actual times with the times of *all* dependencies of the sub-targets in the form of an inequality – one which can easily be changed into an equality. This lemma, through simple equality substitution, removes the intermediate step.

⟨ Firstly, taking **HL.6.3.1**: ⟩

```
(pre_make t w) && (i >= 1) && (i <= length ts)
  && (all (\(t1,w1,k1) -> k1 > newest_dep t1 w1)
       (take i (zip3 ts (tail ws) ks)))
 ==> (newest (take i ks))==(newest [(ws!!i) >~>
       getFileTime (name t1) | t1 <- (take i ts)])
```

⇒ ⟨ Letting `i = length ts`, and using known facts about lengths of `ks`, `ts` and `ws`. ⟩

```
(pre_make t w) &&
  && (all (\(t1,w1,k1) -> k1 > newest_dep t1 w1)
       (zip3 ts (tail ws) ks))
 ==> (newest ks)==(newest
       [(last ws) >~> getFileTime (name t1) | t1 <- ts])
```

⇒ ⟨ and, assuming the (same) pre-conditions in the local proof: **(1)** ⟩

```
(newest ks) ==
  (newest[(last ws) >~> getFileTime (name t1) | t1 <- ts])
```

⟨ Next, take **HL.6.3.2**: ⟩

```
(pre_make t w) && (i >= 1) && (i <= length ts)
  && (all (\(t1,w1) -> (w1 >~> getFileTime (name t1))
    > newest_dep t1 w1) (take i (zip ts (tail ws))))
 ==> ((newest [ws!!i >~> getFileTime (name t1)
               | t1 <- (take i ts)])
     >= (newest [newest_dep t1 ws!!i
                 | t1 <- (take i ts)]))
```

⇒ ⟨ Instanciating `i` as `(length ts)` ⟩

```
(pre_make t w)
  && (all (\(t1,w1) -> (w1 >~> getFileTime (name t1))
    > newest_dep t1 w1) (zip ts (tail ws)))
 ==> ((newest [(last ws) >~> getFileTime (name t1)
               | t1 <- ts])
     >= (newest [newest_dep t1 (last ws)
                 | t1 <- ts]))
```

⇒ ⟨ Assuming (local) pre-conditions, and applying **HL.5.5** so them so that the pre-conditions to the above fact are met ⟩

```
(newest [(last ws) >~> getFileTime (name t1)
   | t1 <- ts])
 >= (newest [newest_dep t1 (last ws) | t1 <- ts])
```

⇒ ⟨ changing the RHS of the inequality to make it a simple equality. ⟩

```
(newest [(last ws) >~> getFileTime (name t1)
      | t1 <- ts])
== (newest [
     (newest [newest_dep t1 (last ws) | t1 <- ts]),
     (newest [(last ws) >~> getFileTime (name t1)
              | t1 <- ts])
  ])
```

= ⟨ applying **HL.6.1**. **(2)** ⟩

```
(newest [(last ws) >~> getFileTime (name t1) | t1 <- ts]) ==
  (newest_dep t (last ws))
```

⟨ Combining **(1)** and **(2)**: ⟩

```
(newest ks) == (newest_dep t (last ws))
```

⇒ ⟨ Adding pre-condition. ⟩

```
(pre_make t w)
  && (all (\(t1,w1,k1) -> k1 > newest_dep t1 w1)
       (zip3 ts (tail ws) ks))
 ==> (newest ks)==(newest_dep t (last ws))
```

Theorem H.4

After calling make on a target `t`, the returned time-stamp will be newer than any of `t`'s dependencies.

```
(pre_make t w) ==>
  let (w2,k) = w >=> make t
  in  k > (newest_dep t w2)
```

Proof: Induction on target structure. It has been shown (in **HL.6.4**) that assuming certain pre-conditions, through analysing the time-stamps returned from all recursive make calls, we can determine the age of the most recently changed file anywhere in all the dependencies. Using this we can prove, in the inductive case, that if the top file in `t` isn't newer than its dependencies, then an `exec` will take place to make it so.

Base case: `t=(Leaf n)`

⟨ **HL.4.1.5** ⟩

```
((w >~> make t)/=NoFileTime)
```
⟨ And, for all worlds `w1`, ⟩
```
(newest_dep t w1) == NoFileTime
```
⟨ merging the two facts together ⟩
```
((w >~> make t) > (newest_dep t w1))
```
⇒ ⟨ instantiating `w1` ⟩
```
(w >~> make t) > (newest_dep t (w >-> make t))
```
= ⟨ rearranging, and adding pre-condition: ⟩
```
(pre_make t w) ==>
  let (w2,k) = w >=> make t
  in  k > (newest_dep t w2)
```

Inductive case: `t=(Target n m ts)`

⟨ *Inductive hypothesis* ⟩
```
all (\t1 -> FORALL w1: (pre_make t1 w1) ==>
  (w1 >~> make t1) > (newest_dep t1 (w1 >-> make t1))) ts
```
⟨ Now, define two new variables: ⟩
```
  ws = trace make ts w
  ks = w >~> mapM make
```
⟨ instantiate `w1` in inductive hypothesis ⟩
```
all (\(t1,w1) -> (pre_make t1 w1) ==>
  (w1 >~> make t1) > (newest_dep t1 (w1 >-> make t1)))
(zip ts ws)
```
⇒ ⟨ Using **HL.4.4**, and assuming local `pre_make` pre-condition ⟩
```
all (\(t1,w1) -> (w1 >~> make t1) >
  (newest_dep t1 (w1 >-> make t1))) (zip ts ws)
```
= ⟨ Introducing `ks` ⟩
```
all (\(t1,w1,k1) ->
  k1 > (newest_dep t1 (w1 >-> make t1))) (zip ts ws ks)
```
= ⟨ rewriting, using **HL.3.6** ⟩
```
all (\(t1,w1,k1) -> k1 > (newest_dep t1 w1))
  (zip ts (tail ws) ks)
```
⇒ ⟨ **HL.6.4** ⟩
```
(newest ks)==(newest_dep t (last ws))
```
⇒ ⟨ adding in the final `make` definition ⟩
```
(w >=> make t) ==
  (w >=> do {mtime <- getFileTime n;
             ctimes <- update_deps ts;
             if (mtime <= newest ctimes)
               then do {exec n m; getFileTime n}
               else return mtime})
&& (newest ks)==(newest_dep t (last ws))
```
= ⟨ since `getFileTime` doesn't change the world (**HL.4.1.1**). ⟩

```
(w >=> make t) ==
  (w >=> do {ctimes <- update_deps ts;
             if ((w >~> getFileTime n) <= newest ctimes)
               then do {exec n m; getFileTime n}
               else return (w >~> getFileTime n)})
&& (newest ks)==(newest_dep t (last ws))
```

= ⟨ `update_deps`, `ks` defn. ⟩

```
(w >=> make t) ==
  ((last ws) >=> do {
           if ((w >~> getFileTime n) <= newest ks)
             then do {exec n m; getFileTime n}
             else return (w >~> getFileTime n)})
&& (newest ks)==(newest_dep t (last ws))
```

= ⟨ replacing `newest ks`. ⟩

```
(w >=> make t) ==
  ((last ws) >=> do {
    if ((w >~> getFileTime n) <= (newest_dep t (last ws)))
      then do {exec n m; getFileTime n}
      else return (w >~> getFileTime n)})
```

= ⟨ using **HL.5.1**, **HL.2.8.1** ⟩

```
(w >=> make t) ==
  ((last ws) >=> do {
    if (((last ws) >~> getFileTime n)
        <= (newest_dep t (last ws)))
      then do {exec n m; getFileTime n}
      else return ((last ws) >~> getFileTime n)})
```

If some dependency is as new as `n`:

```
  (w >=> make t) ==
    ((last ws) >=> do {exec n m; getFileTime n})
```

⟨ The following two facts are known (**HL.2.8.1**, **HL.4.4**): ⟩

```
  (all (\n1 -> n1/=n) (allnames2 t)) && (clock_newer (last ws))
```

⇒ ⟨ Therefore, from **H.2** ⟩

```
  all (\n1 -> (last ws) >~> do {exec n m; getFileTime n} >
              (last ws) >~> do {exec n m; getFileTime n1})
     (allnames2 t)
```

⇒ ⟨ rearranging ⟩

```
  (last ws) >~> do {exec n m; getFileTime n} >
    (newest [(last ws) >~> {exec n m; getFileTime n1}
      | n1 <- allnames2 t])
```

= ⟨ `newest_dep` defn. ⟩

```
  (last ws) >~> do {exec n m; getFileTime n} >
    (newest_dep t ((last ws) >-> exec n m))
```

= ⟨ monad and `getFileTime` properties ⟩

```
  ((last ws) >~> do {exec n m;getFileTime n}) >
    (newest_dep t ((last ws) >-> do {exec n m; getFileTime n}))
```

= ⟨ rewriting in terms of `w`, `make` and `t` ⟩

```
  (w >~> make t) > (newest_dep t (w >-> make t))
```

If `n` is newer than all dependencies:

```
  (w >=> make t)==
    ((last ws) >=> return ((last ws) >~> getFileTime n))
  && ((last ws) >~> getFileTime n) > (newest_dep t (last ws))
```

= ⟨ splitting first equality in two ⟩

```
  (w >-> make t)==(last ws)
  && (w >~> make t)==((last ws) >~> getFileTime n)
  && ((last ws) >~> getFileTime n) > (newest_dep t (last ws))
```

⇒ ⟨ rearranging ⟩

```
  (w >~> make t) > (newest_dep t (w >-> make t))
```

End-If

⟨ Adding pre-condition ⟩

```
(pre_make t w) ==>
  let (w2,k) = w >=> make t
  in (k > newest_dep t w2)
```

10.6 Make "`deps_older`" Proofs

Overview

This section takes Theorem **H.4** and uses it to prove the final more fundamental facts about `make`.

The important lemmas and theorems are as follows:

- Theorem **H.5** takes **HL.6.2**, and, using Theorem **H.4**, shows a standard fact about `make`.
- Lemma **HL.7.3**, proven using DAG-Tree Induction (**HL.2.13**), is extremely important. It states that if two targets are compatible, and both are rebuilt in succession, then the files shared by the two targets will not be touched during the second make.
- Theorem **H.6** applies **HL.7.3** recursively, showing eventually that after running make *every* file in a target is newer than all its dependencies.

Theorem **H.6** is the last theorem to be proven, and says the most about the expected behaviour of `make`. However, with a little thought, it can be seen that Lemma **HL.7.3** has proven another essential `make` property:

Since every sub-target in a DAG-Tree is compatible with one another (**HL.2.6**), the building of a target is really just the sequential building of the sub-targets. Therefore, it can be seen that no single file will ever be built more than once.

Definition of `deps_older`

The `deps_older` predicate states that every file in a target is newer than all its dependencies. It is defined as a recursive statement about `newest_dep`, and that recursive definition works perfectly with the inductive proofs in **HL.7.3**, and **H.6**.

```
-- are all targets newer than their dependencies?
deps_older :: World -> Target -> Bool
deps_older w (Leaf n) = True
deps_older w t@(Target n m ts) =
  ((w >~> getFileTime n) > (newest_dep w t)) &&
  (all (deps_older w) ts)
```

Theorem H.5

After making a target, if the head-file was never touched at all, then none of the files associated with the sub-targets will have been touched either.

```
let w1 = w >-> make t
in
  (pre_make t w) && (filesSame [name t] w w1)
  ==> (filesSame (allnames t) w w1)
```

Proof: A merging of **H.4** and **HL.6.2**. All the hard work here is effectively done in **HL.6.2**, but this time around, a fact which had to be assumed as a pre-condition, has been proven (in **H.4**) to actually always be true.

⟨ Assuming `pre_make` pre-condition, from **H.4** ⟩
```
((w >~> make t) > newest_dep t (w >-> make t))
```
= ⟨ From **HL.5.5** ⟩
```
((w >~> do {make t; getFileTime (name t)})
  > newest_dep t (w >-> make t))
```
= ⟨ introducing variable `w1` ⟩
```
let w1 = w >-> make t
in ((w1 >~> getFileTime (name t)) > newest_dep t w1)
```
⇒ ⟨ Applying **HL.6.2** ⟩
```
let w1 = w >-> make t
in
  (pre_make t w) && (filesSame [name t] w w1)
  ==> (filesSame (allnames t) w w1)
```

Lemma HL.7.1

If every file in target `t` is newer than all its dependencies, then all the files in `t` will be unchanged after `t` is made.

```
(pre_make t w) && (deps_older w t) ==>
  (filesSame (allnames t) w (w >-> make t))
```

Proof: Induction on the target, using **HL.6.4**.

Lemma HL.7.1.2

If every file in target `t` is newer than all its dependencies, then no file *anywhere* will be changed after the making `t`.

```
(pre_make t w) && (deps_older w t) ==>
  (filesSame ns w (w >-> make t))
```

Proof: Combining **HL.7.1**, and **H.1** (all the filenames in `ns` are either in `(allnames t)` or they aren't.)

Lemma HL.7.1.3

If every file in target `t0` is newer than all its dependencies, `t0` and `t1` are compatible DAG-Trees, and `t1` is a sub-target of `t0`, then every file in `t1` will also be newer than all its dependencies.

```
(couldBeDAGT t0 t1) && (deps_older w t0) && (t1 `elem` alldeps t0)
  ==> (deps_older w t1)
```

Proof: DAG-Tree induction. In the base case, if `t0` and `t1` are "Apart" or "B-Within", then `t1` is not a sub-target of `t0`, so the pre-condition fails. If they are "Equal", then it is trivially true. In the inductive case, from the inductive hypothesis (using the recursive definition of `deps_older`), the sub-targets of `t0` will also have the `deps_older` property true of them. Since `(name t0)` is not in `t1`, the property will remain true.

Lemma HL.7.2

If target `t1` is compatible with target `t`, and each file in `t1` is newer than all its dependencies, then after making `t`, all the files in `t1` will remain unchanged.

```
(pre_make t w) && (couldBeDAGT t t1) && (deps_older w t1)
  ==> (filesSame (allnames t1) w (w >-> make t))
```

Proof: DAG-Tree-Induction (**HL.2.13**). The base case shows that (1) if `t1` is inside `t` ("equal", "b-within") then files will not have changed (using **HL.7.1.3**), and (2) if `t1` is totally outside `t` ("apart") then files will not have changed, using **H.1**. The inductive case, using the inductive hypothesis, shows that the files won't have changed during recursive calls. Then, since the top file `(name t)` cannot be inside `t1`, even if `(name t)` is rebuilt using an `exec` then no files in `t1` will have changed.

⟨ First, define the predicate we wish to prove: ⟩

```
let p =
  (\t0 t1 -> FORALL w: (pre_make t0 w) &&
    (couldBeDAGT t0 t1) && (deps_older w t1)
    ==> (filesSame (allnames t1) w (w >-> make t0)))
```

Base case:

```
  (apartT t0 t1) || (equalT t0 t1) || (bwithinT t0 t1)
```
⟨ Firstly, from **H.1**: ⟩
```
  (apartT t0 t1) ==> (filesSame (allnames t1) w (w >-> make t0))
```
⟨ Also, since ⟩
```
  ((equalT t0 t1) || (bwithinT t0 t1)) ==> (t1 `elem` alldeps t0)
```
⇒ ⟨ because of `deps_older` pre-condition, using Lemma **HL.7.1.3** ⟩
```
  ((equalT t0 t1) || (bwithinT t0 t1)) ==> (deps_older w t0)
```
⇒ ⟨ **HL.7.1** ⟩
```
  ((equalT t0 t1) || (bwithinT t0 t1)) ==>
    (filesSame (allnames t1) w (w >-> make t0))
```
⟨ Put together, adding pre-conditions: ⟩
```
  (couldBeDAGT t0 t1) &&
  ((apartT t0 t1) || (equalT t0 t1) || (bwithinT t0 t1))
    ==> (p t0 t1)
```

Inductive case:
```
  (crushedT t0 t1) || (dwithinT t0 t1)
```
⇒ ⟨ From **HL.2.12.2** and **HL.2.12.3**, this implies the following fact: ⟩
```
  (name t0) `notElem` (allnames t1)
```
⟨ The above fact will be used later. For the moment, concentrate on assumed inductive hypothesis: ⟩
```
  (all (\t -> p t t1) (deps t0))
```
= ⟨ expanding `p`. ⟩
```
  (all (\t -> FORALL w1: (pre_make t w1) &&
    (couldBeDAGT t t1) && (deps_older w1 t)
    ==> (filesSame (allnames t1) w1 (w1 >-> make t)))
  (deps t0))
```
⟨ Explicitly adding assumed (local) pre-conditions: ⟩
```
  (pre_make t0 w) && (couldBeDAGT t0 t1) && (deps_older w t1)
  && (all (\t -> FORALL w1: (pre_make t w1) &&
    (couldBeDAGT t t1) && (deps_older w1 t)
    ==> (filesSame (allnames t1) w1 (w1 >-> make t)))
  (deps t0))
```
⇒ ⟨ Let `ws` = `(trace make (deps t0) w)`. ⟩
```
  (pre_make t0 w) && (couldBeDAGT t0 t1) && (deps_older w t1)
  && (all (\(t,w1) -> (pre_make t w1) &&
    (couldBeDAGT t t1) && (deps_older w1 t)
    ==> (filesSame (allnames t1) w1 (w1 >-> make t)))
  (zip (deps t0) ws))
```
⇒ ⟨ applying **HL.4.4**, all recursive `pre_make`s vanish ⟩
```
  (pre_make t0 w) && (couldBeDAGT t0 t1) && (deps_older w t1)
  && (all (\(t,w1) ->
    (couldBeDAGT t t1) && (deps_older w1 t)
    ==> (filesSame (allnames t1) w1 (w1 >-> make t)))
  (zip (deps t0) ws))
```

⇒ ⟨ using **HL.2.6**, and through recursive nature of `deps_older` defn., all recursive pre-conditions vanish ⟩

```
(pre_make t0 w) && (couldBeDAGT t0 t1) && (deps_older w t1)
&& (all (\(t,w1) ->
  (filesSame (allnames t1) w1 (w1 >-> make t))
(zip (deps t0) ws)))
```

= ⟨ rewriting, using **HL.3.6** ⟩

```
(pre_make t0 w) && (couldBeDAGT t0 t1) && (deps_older w t1)
&& (all (\(w1,w2) -> (filesSame (allnames t1) w1 w2))
  (zip ws (tail ws)))
```

⇒ ⟨ **HL.5.1**, and **HL.3.4.2** ⟩

```
(filesSame (allnames t1) w (w >-> mapM make (deps t0)))
```

⇒ ⟨ adding **HL.4.1.3** ⟩

```
((w >-> make t0) == (w >-> mapM make (deps t0)) ||
  (w >-> make t0) ==
    (w >-> do {mapM make (deps t0);
               exec (name t0 ) (cmd t0)}))
&& (filesSame (allnames t1) w (w >-> mapM make (deps t0)))
```

If `exec` doesn't take place:

⟨ removing `exec` part ⟩

```
  (w >-> make t0) == (w >-> mapM make (deps t0))
    && (filesSame (allnames t1) w (w >-> mapM make (deps t0)))
```

⇒ ⟨ substitution ⟩

```
  (filesSame (allnames t1) w (w >-> make t0))
```

Else-If `exec` does take place:

⟨ removing non-`exec` section, and adding initial assumed fact (from outer case-analysis) ⟩

```
  ((w >-> make t0) ==
      (w >-> do {mapM make (deps t0);
                 exec (name t0 ) (cmd t0);}))
  && (filesSame (allnames t1) w (w >-> mapM make (deps t0)))
  && ((name t0) `notElem` (allnames t1))
```

⇒ ⟨ `exec` property, guaranteed since `exec`ing `t0` won't touch `t1`'s files. ⟩

```
  ((w >-> make t0) ==
      (w >-> do {mapM make (deps t0);
                 exec (name t0 ) (cmd t0);}))
  && (filesSame (allnames t1) w (w >-> mapM make (deps t0)))
  && (filesSame (allnames t1)
    (w >-> mapM make (deps t0))
    (w >-> do {mapM make (deps t0); exec (name t0) (cmd t0)}))
```

⇒ ⟨ merging both `filesSame` clauses using **HL.5.1** ⟩

```
  ((w >-> make t0) ==
      (w >-> do {mapM make (deps t0);
                 exec (name t0 ) (cmd t0)}))
  && (filesSame (allnames t1) w
    (w >-> do {mapM make (deps t0); exec (name t0) (cmd t0)}))
```

⇒ ⟨ substitution ⟩
```
    (filesSame (allnames t1) w (w >-> make t0))
```
End-If

⟨ adding all assumed pre-conditions ⟩
```
  (couldBeDAGT t0 t1) && (all (\t -> p t t1) (deps t0)) &&
  ((crushedT t0 t1) || (dwithinT t0 t1))
    ==> (p t0 t1)
```
End of Inductive Proof

⟨ Merging these two statements using **HL.2.13**: ⟩
```
  (couldBeDAGT t t1) ==> (p t t1)
```
⇒ ⟨ expanding `p` ⟩
```
  (couldBeDAGT t t1) ==> ((pre_make t w) &&
    (couldBeDAGT t t1) && (deps_older w t1)
    ==> (filesSame (allnames t1) w (w >-> make t)))
```
= ⟨ rearranging ⟩
```
  (pre_make t w) && (couldBeDAGT t t1) && (deps_older w t1)
    ==> (filesSame (allnames t1) w (w >-> make t)))
```

Lemma HL.7.3

If the files in target `t` are the same in two different worlds, then the 'deps_olderness' of the target will not change between the two worlds.

```
(filesSame (allnames t) w1 w2) ==>
  (deps_older w1 t) == (deps_older w2 t)
```

Proof: Simple induction on target.

Lemma HL.7.4

Assuming pre-conditions hold, and sub-target `t0` is fully ordered, then after calling make on `t1`, the target `t0` will still have its dependencies ordered.

```
LET t  = (Target n m [t0,t1])
    w1 = w >-> make t1
IN  (pre_make t w) && (deps_older w t0) && (deps_older w1 t1)
      ==> (deps_older w1 t0)
```

Proof: Relatively trivial – **HL.7.2** does all the hard work. The reason `t0`'s dependencies are still ordered is because no files anywhere in `t0` have been changed.

⟨ First, assume always that the three pre-conditions hold ⟩
```
  (pre_make t w) && (deps_older w t0) && (deps_older w1 t1)
```
⇒ ⟨ expanding the `pre_make` properties ⟩
```
  (pre_make t1 w) && (couldBeDAGT t0 t1) &&
  (deps_older w t0) && (deps_older w1 t1)
```
⇒ ⟨ apply **HL.7.2** ⟩
```
  (filesSame (allnames t0) w w1) && (deps_older w t0)
```
⇒ ⟨ apply **HL.7.3** ⟩
```
  (deps_older w1 t0)
```

Theorem H.6

After making any target, each file in that target will be newer than its dependencies.

```
(pre_make t w) ==> (deps_older (w >-> make t) t)
```

Proof: Induction on target. In the inductive case, the "uninteresting" parts of the `make` body are taken out, and we are left with a statement about the execution of the recursive make calls. By proving another stronger fact (by induction), using **HL.7.4** in the (inner) inductive step, the proof of the theorem is completed.

Base case: `t = (Leaf n)`

⟨ From `deps_older` defn. ⟩

```
  (deps_older _ (Leaf n)) == True
```

⇒ ⟨ trivially ⟩

```
  (pre_make t w) ==> (deps_older (w >-> make t) t)
```

Inductive Case: `t = (Target n m ts)`

⟨ *Inductive hypothesis* ⟩

```
  all (\t1 -> FORALL w1:
    (pre_make t1 w1) ==> deps_older (w1 >-> make t1) t1) ts
```

⟨ Instantiate all the intermediate worlds with a new variable `ws`. ⟩

```
  ws = trace make ts w
```

⟨ Rewriting the inductive hypothesis: ⟩

```
  all (\(t1,w1) -> (pre_make t1 w1) ==>
    deps_older (w1 >-> make t1) t1) (zip ts ws)
```

⇒ ⟨ Assuming `pre_make` pre-condition and using **HL.4.4** ⟩

```
  all (\(t1,w1) -> deps_older (w1 >-> make t1) t1)
    (zip ts ws)
```

= ⟨ From `trace` defn. ⟩

```
  all (\(t1,w2) -> deps_older w2 t1) (zip ts (tail ws))
```

⟨ The above expression becomes the refined inductive hypothesis used later in the proof. For now, define `wf` to be the final world which make returns. ⟩

```
  wf = (w >-> make t)
```

⟨ Since, from **HL.5.5**, we know ⟩

```
  (wf >~> getFileTime n) == (w >~> make t)
```

⟨ And, from **H.5**, we know ⟩

```
  (w >~> make t) > (newest_dep wf t)
```

⇒ ⟨ Merging these together: ⟩

```
  (wf >~> getFileTime n) > (newest_dep wf t)
```

⇒ ⟨ Adding the `deps_older` definition. ⟩

```
  ((wf >~> getFileTime n) > (newest_dep wf t)) &&
  (deps_older wf t) ==
    ((wf >~> getFileTime n) > (newest_dep wf t) &&
      (all (deps_older wf) ts))
```

⇒ ⟨ rearranging ⟩

```
(all (deps_older wf) ts) ==> (deps_older wf t)
```

⟨ The main proof obligation now becomes `(all (deps_older wf) ts)`. Now, examine the possible worlds that result from running `make`. (**HL.4.1.3**) ⟩

```
(wf == (w >-> update_deps ts)) ||
(wf == (w >-> do {update_deps ts; exec n m;}))
```

= ⟨ rewriting using `ws`. ⟩

```
(wf == (last ws)) || (wf == ((last ws) >-> exec n m))
```

⇒ ⟨ Since the only difference between `wf` and `(last ws)` is that an `exec` may have taken place, and (from **HL.2.8.1**), it is known that `n` is not in any sub-targets, ⟩

```
all (\t1 -> n `notElem` (allnames t1)) ts
```

⇒ ⟨ therefore, from `exec` properties ⟩

```
all (\t1 -> filesSame (allnames t1) wf (last ws)) ts
```

⇒ ⟨ Using **HL.7.3** ⟩

```
(all (deps_older (last ws)) ts) ==> (all (deps_older wf) ts)
```

⟨ Therefore, the proof obligation becomes `(all (deps_older (last ws)) ts)`. To prove this, it is necessary to prove a stronger fact in the process, namely: ⟩

```
((i >=0) && (i <= length ts)) ==>
   all (deps_older (ws!!i)) (take i ts))
```

Base case: `i==0`

```
(take i ts) == []
```

⇒ ⟨ `all` defn. ⟩

```
all f (take i ts) == True
```

⇒ ⟨ instantiating `f` ⟩

```
all (deps_older (ws!!i)) (take i ts)
```

Inductive Case: `i==k`

⟨ *Inductive hypothesis* **(1)** ⟩

```
((k-1 >= 0) && (k-1 <= length ts)) ==>
   all (deps_older (ws!!(k-1))) (take (k-1) ts))
```

⟨ First, examine the pre-condition properties. From **HL.2.6**: ⟩

```
(and [couldBeDAGT t0 t1 | t0 <- ts, t1 <- ts])
```

⇒ ⟨ merging with `clock_newer` property ⟩

```
and [pre_make (Target n m [t0,t1]) (ws!!(k-1))
          | t0 <- ts, t1 <- ts]
```

⇒ ⟨ filtering unwanted targets. **(2)** ⟩

```
all (\t0 -> pre_make (Target n m [t0,(ts!!k)]) ws!!(k-1))
  (take (k-1) ts)
```

⟨ Also, from the (outer) inductive assumption: ⟩

```
(deps_older (ws!!(k-1) >-> make (ts!!(k-1)))) ts!!(k-1))
```

⟨ `trace` property of `ws` ⟩

```
(deps_older (ws!!k) (ts!!(k-1)))
```

⇒ ⟨ Trivially, this becomes. **(3)** ⟩

```
all (\t0 -> deps_older (ws!!k) (ts!!(k-1)))
  (take (k-1) ts)
```

⟨ Using **(1)**, **(2)** and **(3)**, and **HL.7.4**, this yields ⟩

```
(k-1 >= 0) && (k-1 <= length ts) ==>
  all (deps_older (ws!!k)) (take (k-1) ts)
```

⇒ ⟨ since it has been proven that `(deps_older (ws!!k) (ts!!(k-1)))`, (and previous pre-condition is superseded by new one.) ⟩

```
(k >= 0) && (k <= length ts) ==>
all (deps_older (ws!!k)) (take k ts)
```

⇒ ⟨ substituting `i==k`. ⟩

```
(i >= 0) && (i <= length ts) ==>
  all (deps_older (ws!!i)) (take i ts)
```

End of Inductive Proof.

⟨ Finally, letting `i=(length ts)` ⟩

```
(length ts >= 0) && (length ts <= length ts) ==>
  all (deps_older (ws!!(length ts)))
    (take (length ts) ts)
```

⟨ removing true pre-condition, and adding `trace` properties: ⟩

```
(all (deps_older (ws!!(length ts)))
  (take i (length ts))) &&
ws!!(length ts)==(last ws) &&
(take (length ts) ts)==ts
```

⇒ ⟨ substitution ⟩

```
(all (deps_older (last ws)) ts)
```

⇒ ⟨ and, as shown earlier, this yields ⟩

```
(deps_older (w >-> make t) t)
```

⟨ and, adding pre-condition: ⟩

```
(pre_make t w) ==> (deps_older (w >-> make t) t)
```

11 Clean Proofs

We present the Clean proofs in the same style as the Haskell proofs already seen. In many cases we will find that proofs for the Clean programs are either identical or only trivially different to the corresponding Haskell proofs. In those cases we have chosen not to present the Clean proofs in detail, but to refer to the corresponding Haskell proofs instead. To make it easier to locate the corresponding proofs we have chosen to use the same numbering scheme for these proofs – this has sometimes occasioned a gap in the numbering of the Clean proofs, where a lemma related to the Haskell I/O system was not required.

Lemma CL.2.1

```
((prodall r s) && (subset s2 s) ==> (prodall r s2)
```

Proof: trivial

Lemma CL.2.2

```
(isDAGT t) ==> all isDAGT (deps t)
```

Proof: see Lemma **HL.2.2**

Proofs Lemma **CL.2.3** to Lemma **CL.2.13** are identical to the corresponding Haskell proofs, and are omitted here in the interests of brevity.

There are no proofs numbered Lemma **CL.3.1** to Lemma **CL.3.2**. In order to obtain consistent numbering across the Clean and Haskell proofs we resume the numbering at proof Lemma **CL.3.3**

`trace` definition

```
trace :: (a World -> (a,World)) [a] World -> [World]
trace f [] w = [w]
trace f [p:ps] w = [w : trace a ps (w >-> a p))
```

Lemma CL.3.3

```
length (trace a ps w) == (length ps)+1
```

Proof: induction on a list of parameters. **Base case:** `ps = []`
⟨ LHS: ⟩
`length (trace a ps w)`
=⟨ replace `ps` ⟩
`length (trace a [] w)`
=⟨ `trace defn.` ⟩
`length [w]`
=⟨ `length` defn ⟩
`length [] + 1`
=⟨ reintroducing `ps` ⟩
`length ps + 1`
Inductive case: `ps = [p1:ps1]`
⟨ *Inductive hypothesis* ⟩
`(length (trace a ps1 w1)) = (length ps1)+1`
⟨ LHS ⟩
`length (trace a ps w)`
=⟨ `trace` definition ⟩
`length [w: trace a ps1 (w >-> a ps)]`
=⟨ Letting `w1 = w >-> a p1` ⟩
`length [w:trace a ps1 w1]`
=⟨ Using inductive hypothesis, algebra ⟩
`length ps1 + 2`
=⟨ relationship of `ps` to `ps1` ⟩
`length ps + 1`

Lemma CL.3.4.1

```
head (trace a ps w) == w
```

Proof: Trivial (Expension of `trace` function)

Lemma CL.4.1.1

```
(w >-> getFileTime n)==w
```

Proof: Trivial

Lemma CL.4.1.2

```
(w >-> make (Leaf n)) == w
```

Proof:
⟨ Definition of `make` on leaves, defn. `>->` ⟩

```
w >-> make (Leaf n) == fst
        letb (w,time) <- getFileTime n w in
        if (time==NoFileTime) (error "Can't make "+++n+++"!")
                              (w,mtime)
```

⇒⟨ Leaf nodes exist (by precondition - see section 9.2) ⟩

```
w >-> make (Leaf n) == fst
                           letb (w,time) = getFileTime n w in
                                           (w,time)
```

=⟨ eliminating `letb` ⟩

```
 w >-> make (Leaf n) == fst (getFileTime n w)
```

⇒⟨ Lemma **CL.4.1.1** ⟩

```
w >-> make (Leaf n) == w
```

Lemma CL.4.1.3

```
((w >-> make (Target n m ts)) ==
  w >-> \w -> letb (w,_) = update_deps ts w in exec n w)
  ||  ((w >-> make (Target n m ts)) == w >-> \w -> update_deps ts w)
```

Proof: analysis of make body, and Lemma **CL.4.1.1**
⟨ `make` definition, `>->` definition ⟩

```
w >-> make (Target n m ts) ==
   fst (letb (w,times) = update_deps ts w in
        letb (w,time)  = getFileTime n w in
        if (time <= maxList times)
           (getFileTime n (fst (exec n m w)))
           (w,time))
```

⟨ Case analysis: time ≤ maxList times ⟩

```
w >-> make (Target n m ts) ==
   fst (letb (w,times) = update_deps ts w in
        letb (w,time)  = getFileTime n w in
        ((fst (exec exec n m w)),
         (snd (getFileTime n (fst (exec n m w))))))
```

=⟨ Lemma **CL.3.1.1** ⟩

```
w >-> make (Target n m ts) ==
   (letb (w,times) = update_deps ts w in
      (exec n m w))
```

=⟨ definition of `>->`; introduce lambda expression ⟩

```
w >-> make (Target n m ts) ==
   w >-> \w -> (letb (w,times) = update_deps ts w in
                (exec n m w))
```

⟨ Case analysis: time > maxList times ⟩

```
w >-> make (Target n m ts) ==
   fst (letb (w,times) = update_deps ts w in
        letb (w,time)  = getFileTime n w in
        (w,time))
```

=⟨ expanding `letb` ⟩

```
w >-> make (Target n m ts) ==
   fst (letb (w,times) = update_deps ts w in
        getFileTime n w)
```

⇒⟨ Lemma **CL.4.1.1**, introducing lambda expression and `>->` ⟩

```
w >-> make (Target n m ts) == w >-> \w -> (update_deps ts w)
```

⇒⟨ Uniting cases using `||` ⟩

```
((w >-> make (Target n m ts)) ==
     w >-> \w -> letb (w,_) = update_deps ts w in
                 exec n m w)
  ||  ((w >-> make (Target n m ts)) ==
           w >-> \w -> update_deps ts w)
```

The proofs for lemmas Lemma **CL.4.1.4** to Lemma **CL.5.1** are all identical to the correspondingly numbered Haskell proofs, and are omitted here in the interests of brevity.

Lemma CL.5.1

`filesSame` defines an equivalence relation. Proof: see Lemma **HL.5.1**

Lemma CL.5.1

```
  (filesSame ns1 w1 w2) && (filesSame ns2 w1 w2) ==>
    (filesSame (ns1++ns2) w1 w2
```

Proof: see Lemma **HL.5.1**

Lemma CL.5.3

```
(pre_make t w) && (notElem n (allnames t)) ==>
  (filesSame [n] w (w >-> make t))
```

Proof: Structural induction on `t`. **Base case:** `t = (Leaf n1)`
⟨ Lemma **CL.3.1.2** ⟩

```
(w >-> make (Leaf n1)) == w
```

⇒⟨ Lemma **HL.5.1** (reflexivity of `filesSame`) ⟩

```
(filesSame [n] w (w >-> make (Leaf n1)))
```

⇒⟨ Adding pre-condition ⟩

```
(pre_make t w) && (notElem n (allnames t)) ==>
  (filesSame [n] w (w >-> make t))
```

Inductive case: `t = (Target n1 m ts)`
⟨ Inductive hypothesis ⟩

```
(all (\t1 -> FORALL w1: (pre_make t1 w1) && (notElem n (allnames t1))
  ==> (filesSame [n] w1 (w1 >-> make t1))) ts)
```

⇒⟨ Firstly, let `ws = trace make ts w`, then instantiate all `w1` in inductive hypothesis ⟩

```
all (\(t1,w1) -> (pre_make t1 w1) && (notElem n (allnames t1))
  ==> (filesSame [n] w1 (w1 >-> make t))) (zip ts ws)
```

⇒⟨ assuming local `pre_make` and Lemma **CL.3.4** ⟩

```
all (\(t1,w1) -> (notElem n (allnames t1))
  ==> (filesSame [n] w1 (w1 >-> make t1))) (zip ts ws)
```

⇒⟨ Since `subset (allnames t1) (allnames t)`, assuming precondition ⟩

```
all (\(t1,w1) -> filesSame [n] w1 (w1 >-> make t1))) (zip ts ws)
```

=⟨ rewriting ⟩

```
all (\(w1,w2) -> filesSame [n] w1 w2)) (zip ws (tail ws))
```

⇒⟨ since `filesSame` is an equivalence relation (Lemma **CL.5.1**) and `fst (w1!!i) == snd (ws!!(i-1))` ⟩

```
filesSame [n] w (last ws)
```

=⟨ by definition of `make` ⟩

```
(filesSame [n] w (w >-> update_deps ts))
```

⇒⟨ adding `make` definition from Lemma **CL.3.1.3** ⟩

```
((w >-> make (Target n m ts)) ==
     w >-> \w -> letb (w,_) = update_deps ts w in
                 getFileTime n (fst (exec n m w)))
||  ((w >-> make (Target n m ts)) ==
         w >-> \w -> update_deps ts w)
&& (filesSame [n] w (w >-> \w -> update_deps ts w))
```

If `exec` occurs:
⟨ Removing other expression ⟩

```
((w >-> make (Target n m ts)) ==
     w >-> \w -> letb (w,_) = update_deps ts w in
                 getFileTime n (fst (exec n m w)))
&& (filesSame [n] w (w >-> \w -> update_deps ts w))
```

⇒⟨ and, since from initial `filesSame` pre-condition, n≠n1 ⟩

```
((w >-> make (Target n m ts)) ==
     w >-> \w -> letb (w,_) = update_deps ts w in
                 exec n m w)
&& (filesSame [n] w (w >-> \w -> update_deps ts w))
&& (filesSame [n] (w >-> \w -> update_deps ts w)
         (w >-> \w -> letb (w,_) = update_deps ts w in exec n m w))
```

⇒⟨ Lemma **HL.5.1** ⟩

```
  (filesSame [n] w (w >-> make t))
```

If no `exec` occurs:
⟨ removing other expression ⟩

```
  ((w >-> make (Target n m ts)) == w >-> \w -> update_deps ts w)
    && (filesSame [n] w (w >-> \w -> update_deps ts w))
```

⇒⟨ Substitution ⟩

```
  (filesSame [n] (w >-> make t))
```

End-If
⟨ Adding assumed pre-conditions ⟩

```
  (pre_make t w) && (notElem n (allnames t)) ==>
    (filesSame [n] w (w >-> make t))
```

Theorem C.1

```
  (pre_make t w) && (disjoint ns (allnames t)) ==>
    (filesSame [n] w (w >-> make t))
```

A consequence of Lemma **CL.5.3**; see Theorem **H.1**

Lemma CL.5.4

```
  (clock_newer (f,k)) ==> ((f,k) >~> getFileTime n) <= (FileTime k)
```

Proof: see Lemma **HL.5.4** for approach

Theorem C.2

```
  (clock_newer w) && (n <> n1) ==>
    ((w >~> (w,time) = exec n m w in getFileTime n w) >
     (w >~> (w,time) = exec n m w in getFileTime n1 w))
```

Proof: See Theorem **H.2** for explanation of approach.

```
  let (f,k) = w
```

⟨ assuming pre-condition ⟩

```
(clock_newer w) && (n <> n1)
```

⇒⟨ Using Lemma **HL.5.4** ⟩

```
(w >~> getFileTime n1) <= (FileTime k) && (n <> n1)
```

⇒⟨ `exec` properties, since `n<>n1` ⟩

```
(w >~> getFileTime n1) <= (FileTime k) &&
  (filesSame [n1] w (w >-> exec n m))
```

⇒⟨ `filesSame` defn. ⟩

```
(w >~> getFileTime n1) <= (FileTime k) &&
(w >~> getFileTime n1) == (w >-> (exec n m)) >~> getFileTime n1
```

⇒⟨ equality ⟩

```
(((w >-> exec n m) >~> getFileTime n1)
  <= (FileTime (snd w)))
```

=⟨ Let `w1 = (w >-> exec n m)` ⟩

```
(w1 >~> getFileTime n1) <= (FileTime k)
```

⇒⟨ `exec` property ⟩

```
((w1 >~> getFileTime n1) <= (FileTime k))
 && (FileTime ((snd w1)-1))==(FileTime k)
 && (w1 >~> getFileTime n) ==(FileTime (snd w1))
```

⇒⟨ equality ⟩

```
((w1 >~> getFileTime n1) <= (FileTime ((snd w1)-1)))
 && (w1 >~> getFileTime n)==(FileTime (snd w1))
```

⇒⟨ algebra ⟩

```
((w1 >~> getFileTime n1) < (FileTime (snd w1)))
 && (w1 >~> getFileTime n)==(FileTime (snd w1))
```

⇒⟨ equality ⟩

```
((w1 >~> getFileTime n1) < (w1 >~> getFileTime n))
```

⇒⟨ Rewriting `w1` ⟩

```
(((w >-> exec n m) >~> getFileTime n1) <
 ((w >-> exec n m) >~> getFileTime n))
```

⇒⟨ Introducing local bindings for `time` and `w` ⟩

```
((w >~> (w,time) = exec n m w in getFileTime n w) >
 (w >~> (w,time) = exec n m w in getFileTime n1 w))
```

⇒⟨ Adding pre-condition ⟩

```
(clock_newer w) && (n <> n1) ==>
  ((w >~> (w,time) = exec n m w in getFileTime n w) >
   (w >~> (w,time) = exec n m w in getFileTime n1 w))
```

Lemma CL.5.5

```
(pre_make t w) ==>
  ((w >-> make t) >~> getFileTime n) == (w >~> make t)
```

Proof: Case analysis on target

Leaf case: `t = Leaf n`
⟨ CL.3.1.4 ⟩

```
(w >~> make (Leaf n)) == (w >~> getFileTime n)
```

⇒⟨ therefore, because of Lemma **CL.4.1.1** ⟩

```
((w >-> make t) >~> getFileTime n) == (w >~> make t)
```

=⟨ adding pre-condition ⟩

```
(pre_make t w) ==>
  ((w >-> make t) >~> getFileTime n) == (w >~> make t)
```

Non-Leaf case: `t = (Target n m ts)`
⟨ `make` defn. ⟩

```
(w >~> make t) == (w >~> \w->
   letb (w,times) = update_deps ts w in
   letb (w,time)  = getFileTime n w in
   if (time <= maxList times)
        (getFileTime n (fst (exec n c w)))
        (w,time))
```

⇒⟨ Lemma **CL.4.1.1** ⟩

```
(w >~> make t) == (w >~> \w->
   letb (w,times) = update_deps ts w1 in
   if ((w >~> getFileTime n) <= maxList times)
        (getFileTime n (fst (exec n c w)))
        (getFileTime n w))
```

Case analysis: `time <= maxList times`
⟨ Dropping other branch ⟩

```
(w >~> make t) == (w >~> \w->
   letb (w,times) = update_deps ts w1 in
   getFileTime n (fst (exec n c w)))
```

⇒⟨ Lemma **CL.4.1.1** ⟩

```
(w >-> make t) == (w >-> \w->
   letb (w,times) = update_deps ts w1 in
   getFileTime n (fst (exec n c w))) &&
((w >-> make t) >~> getFileTime n) == ((w >-> (\w->
   letb (w,times) = update_deps ts w1 in
   getFileTime n (fst (exec n c w))))
     >~> getFileTime n)
```

⇒⟨ substitution ⟩

```
((w >-> make t) >~> getFileTime n) == (w >~> make t)
```

Case: `time > maxList times`
⟨ Dropping other branch ⟩

```
(w >~> make t) == (w >~> \w->
   letb (w,times) = update_deps ts w1 in
   getFileTime n w)
```

⇒⟨ Lemma **CL.4.1.1** ⟩

```
(w >-> make t) == (w >-> \w->
   letb (w,times) = update_deps ts w1 in
   getFileTime n w) &&
((w >-> make t) >~> getFileTime n) == (w >-> (\w->
   letb (w,times) = update_deps ts w1 in
   getFileTime n w))
     >~> getFileTime n)
```

⇒⟨ substitution ⟩

```
((w >-> make t) >~> getFileTime n) == (w >~> make t)
```

```
End-If
```

⟨ Adding pre-condition ⟩

```
(pre_make t w) ==>
  ((w >-> make t) >~> getFileTime n) == (w >~> make t)
```

Lemmas Lemma **CL.5.6** and Lemma **CL.5.7** are identical to the correspondingly numbered Haskell proofs and are omitted here.

Theorem C.3

```
(pre_make t w) ==>
  let (k = w >~> \w -> letb (w,time) = make t w in getFileTime n w) in
      ( k == w >~> getFileTime n) || (k > (FileTime (snd w)))
```

Proof: Induction on the target structure. See Theorem **H.3** for discussion.

Base case: `t = Leaf n1`

⟨ CL.4.1.2 ⟩

```
(w >-> make (Leaf n1)) == w
```

⇒⟨ Application ⟩

```
((w >-> make t) >~> getFileTime n) == (w >~> getFileTime n)
```

⇒⟨ Rewriting ⟩

```
( w >~> \w -> letb (w,time) = make t w in getFileTime n w)
== (w >~> getFileTime n)
```

⇒⟨ Conjunction and adding pre-condition ⟩

```
(pre_make t w) ==>
  let (k = w >~> \w -> letb (w,time) = make t w in getFileTime n w) in
      ( k == w >~> getFileTime n) || (k > (FileTime (snd w)))
```

Inductive case: `t = (Target n1 m1 ts)`

⟨ *Inductive Hypothesis*: ⟩

```
all (\t1 -> FORALL w1: (pre_make t1 w1) ==>
  let k1 = w1 >~> (\w -> letb (w,time) = make t1 w in getFileTime n w) in
      (k1 == w1 >~> getFileTime n) || (k1 > (FileTime (snd w1)))) ts
```

⇒⟨ Letting `ws = trace make ts w` ⟩

```
all (\(t1,w1) -> (pre_make t1 w1) ==>
  let k1 = w1 >~> \w -> (letb (w,time) = make t1 w in getFileTime n w) in
    (k1 == w1) >~> getFileTime n) || (k1 > (FileTime (snd w1))) (zip ts ws)
```

⇒⟨ Assuming `pre_make t w` pre-condition, and using Lemma **CL.3.4** ⟩

```
all (\(t1,w1) ->
  let k1 = w1 >~> \w -> (letb (w,time) = make t1 w in getFileTime n w) in
    (k1 == w1) >~> getFileTime n) || (k1 > (FileTime (snd w1)))
```

If file `n` is touched

```
  (ws!!(i+1) >~> getFileTime n)>(FileTime (snd (ws!!i)))
```

⟨ Secondly, using Lemma **CL.5.7** we can show: ⟩

```
all (\(w1,w2)-> (FileTime (snd w2)) <= (FileTime (snd w2)))
    (take i-1) (zip ws (tail ws)))
```

⇒⟨ algebra ⟩

```
(FileTime (snd (ws!!i))) >= (FileTime (snd w))
```

⟨ Thirdly, applying Lemma **CL.5.6** and Lemma **CL.3.4** ⟩

```
all (\(w1,w2) -> (w1 >~> getFileTime n) <= (w2 >~> getFileTime n))
    (drop (i+1) (zip ws (tail ws)))
```

⇒⟨ Algebra: ⟩

```
(ws!!(i+1)>~>getFileTime n)<=((last ws)>~>getFileTime n)
```

⟨ So, using these three facts: ⟩

```
(FileTime (snd (ws!!i))) >= (FileTime (snd w))
(ws!!(i+1) >~> getFileTime n) > (FileTime (snd (ws!!i)))
(ws!!(i+1) >~> getFileTime n) <= ((last ws) >~> getFileTime n)
```

⟨ we infer ⟩

```
((last ws) >~> getFileTime n) > (FileTime (snd w))
```

=⟨ By definition of `update_deps` ⟩

```
((w >-> update_deps ts) >~> getFileTime n) > (FileTime (snd w))
```

If file `n` is not touched

```
all (\(w1,w2)-> (w1 >~> getFileTime n)
== (w2 >~> getFileTime n)) (zip ws (tail ws))
```

⇒⟨ equality ⟩

```
((head ws)>~>getFileTime n)==((last ws) >~> getFileTime n)
```

=⟨ rewriting ⟩

```
(w >~> getFileTime n)==((w >-> update_deps ts) >~> getFileTime n)
```

Unifying steps gives:

```
let w' = (w >-> update_deps ts) in
   (w' >~> getFileTime n) > (FileTime (snd w))
 ||(w  >~> getFileTime n)==(w' >~> getFileTime n)
```

If `n1` is not touched:

```
  w' == (w>->make t)
```
⇒⟨ rewriting ⟩
```
let k = w >~>
   (\w -> letb (w,time) = make t w in getFileTime n) in
   (k == w >~> getFileTime n) || (k > (FileTime (snd w)))
```
If `n1` is touched, but `n1`≠`n`:
```
filesSame [n] w' (w >-> make t)
```
=⟨ `filesSame` defn. ⟩
```
(w' >~> getFileTime n) == ((w >-> make t) >~> getFileTime n)
```
⇒⟨ rearranging ⟩
```
let k = w >~>
    (\w -> letb (w,time) = make t w in getFileTime n w) in
    (k== w >~> getFileTime n)||(k > (FileTime (snd w)))
```
If `n1` is touched, and `n1==n2`:
```
(w >-> make t) == (w' >-> exec n1 m1)
```
⇒⟨ `exec` definition ⟩
```
((w >-> make t) >~> getFileTime n) ==
  (FileTime (snd (w >-> make t))) &&
  (FileTime (snd (w>->make t)))>(FileTime(snd w'))
```
⟨ And, using the following previously proven fact: ⟩
```
(FileTime (snd w')) <= (FileTime (snd w))
```
⟨ we infer: ⟩
```
((w >-> make t) >~> getFileTime n) > (FileTime (snd w))
```
⇒⟨ Adding disjunction and rearranging: ⟩
```
let k = w >~>
  (\w -> letb (w,time) = make t w in getFileTime n w) in
  (k == w >~> getFileTime n) || (k > (FileTime (snd w)))
```
Adding pre-condition:
```
(pre_make t w)==>
  let k = w >~> (\w -> letb (w,time) = make t w in getFileTime n w) in
    (k == w >~> getFileTime n) || (k > (FileTime (snd w)))
```

Lemma CL.6.1

```
  (newest_dep t w) == (maxList (map (\t1 -> newest
     [w >~> getFileDate (name t1), newest_dep t1 w]) (deps t)))
```

Proof: Function expansion and substitution

Lemma CL.6.2

```
  let w1 = w >-> make t in
      (pre_make t w) && (filesSame [name t] w w1)
   && (w1 >~> getFileTime (name t)) -> (newest_dep t w1)
   ==> (filesSame (allnames t) w w1)
```

Proof: See Lemma **HL.6.2**

Lemma CL.6.3.1

```
let t  = (Target n m ts)
    ks = w >~> update_deps ts
    ws = trace make ts w
in
  (pre_make t w) && (i >= 0) && (i <= length ts)
   && (all (\(t1,w1,k1) -> k1 > newest_dep t1 w1)
      (take i (zip3 ts (tail ws) ks))
   ==> (maxList (take i ks)) ==
       (newest [(ws!!i) >~> getFileTime (name t1) \\ t1 <- (take i ts)])
```

Proof: See Lemma **HL.6.3.1** for approach, noting that `update_deps ts = mapM make ts` in that proof.

Lemma CL.6.3.2

```
let t  = (Target n m ts)
    ws = trace make ts w
in
 (pre_make t w) && (i >= 0) && (i <= length ts)
    && (all (\(t1,w1) -> (w1 >~> getFileTime (name t1))
      > newest_dep t1 w1) (take i (zip ts (tail ws))))
   ==> ((maxList [ws!!i >~> getFileTime (name t1) \\ t1 <- (take i ts)])
     >= (maxList [newest_dep t1 ws!!i \\ t1<-(take i ts)])
```

Proof: See Lemma **HL.6.3.2**

Lemma CL.6.4

```
let t  = (Target n m ts)
    ks = w >~> update_deps ts
    ws = trace make ts w
in
 (pre_make t w)
    && (all (\(t1,w1,k1) -> k1 > newest_dep t1 w1)
    (zip3 ts (tail ws) ks)) ==> (newest ks)==(newest_dep t (last ws))
```

Proof: See Lemma **HL.6.4** for approach, noting that `update_deps ts = mapM make ts` in that proof.

Theorem C.4

```
  (pre_make t w)==> let (w2,k) = w >=> make t in k>(newest_dep t w2)
```

Proof: Induction on target structure.
Base case: `t = (Leaf n)`
Identical to Theorem **H.4**.
Inductive case: `t = (Target n m ts)`
⟨ *Inductive hypothesis:* ⟩

```
all (\t1 -> FORALL w1: (pre_make t1 w1) ==>
  (w1 >~> make t1) > (newest_dep t1 (w1 >-> make t1))) ts
```

⟨ Define two new variables: ⟩

```
ws = trace make ts w
ks = w >~> update_deps ts
```

⟨ instantiate `w1` in ind. hyp. ⟩

```
all (\(t1,w1) -> (pre_make t1 w2) ==>
 (w1 >~> make t1) > (newest_dep t1 (w1 >-> make t1))) (zip ts ws)
```

⇒⟨ Using Lemma **CL.4.4** and assuming local `pre_make`: ⟩

```
all (\(t1,w1) -> (w1 >~> make t1) >
      (newest_dep t1 (w1 >-> make t1))) (zip ts ws)
```

=⟨ Introducing `ks` ⟩

```
all (\(t1,w1,k1) -> k1 > (newest_dep t1 (w1 >-> make t1)))
    (zip ts ws ks)
```

=⟨ Rewriting ⟩

```
all (\(t1,w1,k1) -> k1 > (newest_dep t1 w1)) (zip ts (tail ws) ks)
```

⇒⟨ Lemma **CL.6.4** ⟩

```
(maxList ks) == (newest_dep t (last ws))
```

⇒⟨ Definition of `make` ⟩

```
(w >=> make t) ==
  (w >=> \w -> letb (w,times) = update_deps ts w in
               letb (w,time)  = getFileTime n w in
               if (time <= maxList times)
                    (getFileTime n (fst (exec n c w)))
                    (w,time))
  && (maxList ks)==(newest_dep t (last ws))
```

=⟨ `update_deps`, `ks` defn ⟩

```
(w >=> make t) ==
  (w >=> \w -> letb (times,w) = update_deps ts w in
                   if ((w >~>getFileTime n w) <= maxList times)
                      (exec n c w) (w>~>getFileTime n,w))
  && (maxList ks)==(newest_dep t (last ws))
```

=⟨ defns. of `update_deps` and `ks` ⟩

```
(w >=> make t) ==
  ((last ws) >=> \w -> if ((w >~> getFileTime n)<=maxList ks)
                          (letb (w,_) = exec n m w in getFileTime n w)
                          (getFileTime n w))
  && (maxList ks)==(newest_dep t (last ws))
```

=⟨ Replacing `maxList ks` ⟩

```
(w >=> make t) ==
  ((last ws) >=> \w -> if ((w >~> getFileTime n)<=(newest_dep t (last ws))
                          (letb (w,_) = exec n m w in getFileTime n w)
                          (getFileTime n w))
  && (maxList ks)==(newest_dep t (last ws))
```

⇒⟨ Using Lemma **CL.5.1** and Lemma **CL.2.8.1**, rewriting lambda expr. ⟩

```
(w >=> make t) ==
  ((last ws) >=> if (((last ws) >~> getFileTime n)<=(newest_dep t (last ws))
                     (letb (w,_) = exec n m w in getFileTime n w)
                       (getFileTime n w))
  && (maxList ks)==(newest_dep t (last ws))
```

Cases: if some dependency is as new as n:

```
(w >=> make t) == ((last ws)>=> \w -> letb (w,_) = exec n m
                                       in getFileTime n w)
```

⟨ The following two facts are known from Lemma **CL.4.8.1** and Lemma **CL.3.4** ⟩

```
(all (\n1 -> n1<>n) (allnames2 t)) && (clock_newer (last ws))
```

⇒⟨ Therefore, from Theorem **C.2** ⟩

```
all (n1 -> (last ws) >~>
            (\w -> letb (w,_) = exec n m w in getFileTime n w) >
           (last ws) >~>
            (\w -> letb (w,_) = exec n m w in getFileTime n1 w))
    (allnames2 t)
```

⇒⟨ Rearranging ⟩

```
(last ws) >~> (\w -> letb (w,_) = exec n m w in getFileTime n w) >
      (maxList [(last ws) >~>
         (\w -> letb (w,_) = exec n m w
                in getFileTime n1 w) \\ n1 <- allnames2 t])
```

=⟨ `newest_dep` definition ⟩

```
(last ws) >~> (\w -> letb (w,_) = exec n m w in getFileTime n w) >
  (newest_dep t ((last ws) >-> exec n m))
```

=⟨ `getFileTime` properties ⟩

```
(last ws) >~> (\w -> letb (w,_) = exec n m w in getFileTime n w) >
  (newest_dep t ((last ws) >->
                   (\w -> letb (w,_) = exec n m w in getFileTime n w)))
```

⇒⟨ rewriting using defns of `w`, `make` and `t` ⟩

```
  (w >~> make t) > (newest_dep t (w >-> make t))
```

If n is newer than all dependencies:

```
  (w >=> make t) ==
    ((last ws) >=> \w -> (last ws) >~> getFileTime n)
  && ((last ws) >~> getFileTime n) > (newest_dep t (last ws))
```

=⟨ Splitting the first equality in two ⟩

```
  (w >-> make t) == (last ws)
  && (w >~> make t) == (last ws) >~> getFileTime n)
  && ((last ws) >~> getFileTime n) > (newest_dep t (last ws))
```

⇒⟨ Rearranging: ⟩

```
 (w >~> make t) > (newest_dep t (w >-> make t))
```

End-if ⟨ Adding pre-condition ⟩

```
  (pre_make t w) ==> let (w2,k) = w >=> make t in
                     (k > newest_dep t w2)
```

Theorem C.5

```
let w1 = w >-> make t in
  (pre_make t w) && (filesSame [name t] w w1)
  ==> (filesSame (allnames t) w w1)
```

Proof: See Theorem **H.5**.

The final lemmas, numbered Lemma **CL.7.1.1** to Lemma **CL.7.4** are all essentially identical to the correspondingly numbered Haskell proofs, and are omitted here. It is worth including the statement of the property of Lemma **CL.7.2** as it differs slightly from the Haskell formulation:

Lemma CL.7.2

```
  (pre_make t w) && (couldBeDAGT t t1) && (deps_older w t1)
  ==> (filesSame (allnames t1) w (w >-> make t))
```

Proof: See Lemma **HL.7.2** for approach, noting that `update_deps ts = mapM make ts`.

A Set Properties

Overview

Sometimes is is useful to think of standard functional lists with an equality relation defined on them as sets instead. The Haskell `List` library has some support for this but it was necessary to build some operations that treat lists specifically as if they were sets. This is the same with Clean.

One specific example is that of the list-difference operator `\\`. The standard implementation of this in Haskell doesn't work for lists with duplicate elements, so a new `\\\` operator had to be defined.

No facts were actually *proven* about these new operations, but we are convinced that the lemmas shown are indeed true.

Haskell Set functions

```
disjoint, notdisjoint :: Eq a => [a] -> [a] -> Bool
disjoint l1 l2 = (l1 `intersect` l2) == []
notdisjoint l1 l2 = not (l1 `disjoint` l2)

-- subset and proper-subset operators
subset, psubset :: Eq a => [a] -> [a] -> Bool
subset [] s2 = True
subset (i1:s1) s2 = (i1 `elem` s2) && (s1 `subset` s2)
psubset s1 s2 = (s1 `subset` s2) && (not (s2 `subset` s1))

-- Set equality
```

```
setEq :: Eq a => [a] -> [a] -> Bool
setEq [] [] = True
setEq [] _  = False
setEq _  [] = False
setEq (x:xs) xs1 =
  if (x `elem` xs1)
    then setEq (filter (/= x) xs) (filter (/= x) xs1)
    else False

infix 5 \\\
-- set difference operator
(\\\) :: Eq a => [a] -> [a] -> [a]
(\\\) s1 [] = s1
(\\\) s1 (i2:s2) = (prune s1) \\\ (prune s2)
  where prune = filter (/= i2)
```

Clean Set functions

```
disjoint l1 l2 = intersect l1 l2 == []

intersect [] ys = []
intersect [x:xs] ys | isMember x ys = [x: intersect xs ys]
                    | otherwise     = intersect xs ys

notdisjoint l1 l2 = not (disjoint l1 l2)

subset [] s2 = True
subset [i1:s1] s2 = (isMember i1 s2) && (subset s1 s2)

psubset s1 s2 = subset s1 s2 && (not (subset s2 s1))

setEq [] [] = True
setEq [] _  = False
setEq _  [] = False
setEq [x:xs] xs1 =
  if (isMember x xs1)
    (setEq (filter (\x1 = x1 <> x) xs)
       (filter (\x1 = x1 <> x) xs1))
    (False)

(\\\) infix 5 :: [a] [a] -> [a] | == a
(\\\) s1 [] = s1
(\\\) s1 [i2:s2] = (prune s1) \\\ (prune s2)
  where prune = filter (\i1 = i1 <> i2)
```

Lemma HL.1.1, CL.1.1

Set equality, `setEq`, is an equivalence relation.

Reflexivity

```
(setEq s1 s2)
```

Symmetry

```
(setEq s1 s2) == (setEq s2 s1)
```

Transitivity

```
(setEq s1 s2) && (setEq s2 s3) ==> (setEq s1 s3)
```

Lemma HL.1.2, CL.1.2

```
(setEq s1 (reverse s1))
```

Lemma HL.1.3

```
(setEq s1 (nub s1))
```

Lemma CL.1.3

```
(setEq s1 (removeDup s1))
```

Lemma HL.1.4

```
(setEq s1 s2) ==> ((elem e s1) == (elem e s2))
```

Lemma CL.1.4

```
(setEq s1 s2) ==> ((isMember e s1) == (isMember e s2))
```

Lemma HL.1.5, CL.1.6

```
(subset s1 s2) && (subset s1 s2) ==> (setEq s1 s2)
```

Lemma HL.1.6, CL.1.6

If binary operation `op` forms a commutative, idempotent monoid with identity `i`, then if two sets are equal (under `setEq`) then folding them with `op` will yield the same result.

```
(FORALL e1,e2: ((op e1 e2)==(op e2 e1)) && ((op e1 e1)==e1)
  && ((op i e1)==e1)) && (setEq s1 s2)
    ==> (foldl op i s1) == (foldl op i s2)
```

Lemma HL.1.7, CL.1.7

```
(setEq s1 s2) && (setEq s3 s4) ==> (setEq (s1++s3) (s2++s4))
```

B Working Haskell Make Code

```
module HaskMake where
import System
import Posix

data FileTime = FileTime EpochTime | NoFileTime
type Name = FilePath
type Command = String
data Target = Target Name Command [Target] | Leaf Name

instance Eq FileTime where
  NoFileTime   == NoFileTime    = True
  NoFileTime   == _             = False
  _            == NoFileTime    = False
  (FileTime t) == (FileTime s)  = (t==s)

instance Ord FileTime where
  compare NoFileTime   NoFileTime   = EQ
  compare NoFileTime   _            = LT
  compare _            NoFileTime   = GT
  compare (FileTime t) (FileTime s) = (compare t s)

testTarg = Target "a.out" "gcc prog.c intpair.o" [
  Leaf "prog.c",
  Target "intpair.o" "gcc -c intpair.c" [
    Leaf "intpair.h",
    Leaf "intpair.c"]]

-- given a file name, return its modification time
getFileTime :: Name -> IO FileTime
getFileTime fn =
  catch
    (do
      flst <- Posix.getFileStatus fn
      return (FileTime (modificationTime flst)))
    (\ex -> return NoFileTime)

-- execute command 'cmd'. 'nm' is unused here.
exec :: Name -> Command -> IO ()
exec nm cmd = do
  putStrLn ("Running \""++cmd++"\"..")
  exitc <- system cmd
  return ()

-- make the target, and return the modification time of the
-- newest file of all recursive dependencies.
make :: Target -> IO FileTime
make (Leaf nm) = do
```

```
  mtime <- getFileTime nm
  if (mtime==NoFileTime)
    then error ("can't make "++nm++"!")
    else return mtime
make (Target nm cmd depends) = do
  -- get modification times of this file
  mtime <- getFileTime nm
  -- build and get the times of all children
  ctimes <- update_deps depends
  -- if its older than the newest child, rebuild, and return time
  if (mtime <= (newest ctimes))
    then do
      exec nm cmd     -- execute the command
      getFileTime nm  -- return the update time of the file
    else
      return mtime  -- just return mtime

-- given a list of times, find the newest
newest :: [FileTime] -> FileTime
newest times = foldl max NoFileTime times

-- update all the targets, returning all their times
update_deps :: [Target] -> IO [FileTime]
update_deps = mapM make
```

C Working Clean Make Code

```
module cm

import StdEnv, StdLibMisc, Directory, StdTime

// Types for Make
:: Name :== String
:: Target = Target Name Command [Target]
          | Leaf Name
:: Makefile :== [Target]

// Type to assist in implementation
:: *WorldState :== (*File,*World)

// Make the default target, which is the head of the list
make_main :: Makefile *WorldState -> *WorldState
make_main [t:_] w # (_, _, (f,w)) = make t w
                  = (f, w)

// Recursively build a specific target
make :: Target *WorldState -> (Bool, FileTime, *WorldState)
make (Leaf n) (f,w)
```

```
  # (t, w) = filedate n w
  | t==NoFileTime =
    (False, NoFileTime,
      (f<<<("***No rule to make target "+++n),w))
  | otherwise     = (True, t, (f,w))

make (Target n c deps) (f,w)
  # (ok, times,(f,w))          = update_deps deps (f,w)
  | not ok                     = (False, NoFileTime, (f,w))
  # (this_time,w)              = filedate n w
  | up_to_date this_time times = (True, this_time,(f,w))
  # (err, newTime, f, w)       = execute_command c f w
  | isErr err                  = (False, NoFileTime, (f<<<errmsg n err,w))
  | otherwise                  = (True,newTime,(f,w))
    where
      up_to_date _ [] = False
      up_to_date t l  = t > (maxList l)
      isErr ExecErrorNone = False
      isErr _             = True
      errmsg nam err      =
          "*** ["+++n+++"] Error "+++(toString err)+++"\n"

// Make all the dependencies listed and return their updated times
update_deps :: [Target] *WorldState -> (Bool, [FileTime], *WorldState)
update_deps [] w = (True, [],w)
update_deps [x:xs] w # (ok, t,w)  = make x w
                     | not ok     = (False, [], w)
                     # (ok, ts,w) = update_deps xs w
                     | not ok     = (False, [],w)
                     | otherwise  = (True, [t:ts], w)


//
// Sample data:
//
TestData = [ Target "a.out" "gcc f1.o f2.o -o a.out" [
               Target "f1.o" "gcc -c f1.c" [
                 Leaf "f1.c", Leaf "f2.h"
               ],
               Target "f2.o" "gcc -c f2.c" [
                 Leaf "f2.c", Leaf "f2.h", Target "fred" "fail" []
               ]
           ]]

// Implementation

Start w # (f,w) = (make_main TestData (stdio w))
        = snd (fclose f w)

// Useful utilities.
```

```
:: ExecError = ExecErrorNone
             | ExecErrorCode Int

:: FileTime = NoFileTime
            | FileTime (Date,Time)
:: Command :== String

// Execute command, with command echoing.
execute_command :: Command *File *World ->
                      (ExecError, FileTime, *File, *World)
execute_command cmd f w
      # (ok_code, w) = exec cmd w
      # (ts,w) = now w
      = (mkErr ok_code, ts, f <<< cmd <<< '\n', w)
        where mkErr 0 = ExecErrorNone
              mkErr n = ExecErrorCode n

// Integer is the return code of the command that was run
exec :: !Command *World -> (Int,*World)
exec c w = c_exec_cmd c w

// Current system data and time (as a FileTime)
now :: *World -> (FileTime, *World)
now w # (t,w) = getCurrentTime w
      # (d,w) = getCurrentDate w
      = (FileTime (d,t), w)

// Utilities
// Current date and time of a named file, as a FileTime. Aborts
// on errors except for missing file type errors, which generate
// NoFileTime instead. Not sure what errors they could be, but we
// catch them anyway.
filedate :: !String !*World -> (FileTime,*World)
filedate n w # ((ok,p),w)    = pd_StringToPath n w
             | not ok        = (NoFileTime,w)
             # ((err,inf),w) = getFileInfo p w
             | exists err    = (FileTime inf.pi_fileInfo.lastModified, w)
             | errorfree err = (NoFileTime,w)
             | otherwise     = abort ("Error getting time of "+++n)
             where
               errorfree NoDirError  = True
               errorfree DoesntExist = True
               errorfree _           = False
               exists    DoesntExist = False
               exists    _           = True

// Ordering on dates and times. NoFileTime is older than
// everything except another NoFileTime.
instance < FileTime
```

```
where
  (<) _          NoFileTime = False
  (<) NoFileTime _          = True
  (<) (FileTime (d1,t1)) (FileTime (d2,t2))
              | d1.year < d2.year       = True
              | d1.year > d2.year       = False
              | d1.month < d2.month     = True
              | d1.month > d2.month     = False
              | d1.day < d2.day         = True
              | d1.day > d2.day         = False
              | t1.hours < t2.hours     = True
              | t1.hours > t2.hours     = False
              | t1.minutes < t2.minutes = True
              | t1.minutes > t2.minutes = False
              | t1.seconds < t2.seconds = True
              | otherwise               = False
instance == FileTime
where
  (==) NoFileTime NoFileTime = True
  (==) (FileTime (d1,t1)) (FileTime (d2,t2)) = d1==d2 && t1==t2
  (==) _ _ = False

instance == Time
where
  (==) t1 t2 =
        t1.hours == t2.hours &&
        t1.minutes == t2.minutes &&
        t1.seconds == t2.seconds

instance == Date
where
  (==) d1 d2 =
        d1.year == d2.year &&
        d1.month == d2.month &&
        d1.day == d2.day

instance toString ExecError
where
  toString ExecErrorNone = "0"
  toString (ExecErrorCode n) = toString n
```

We provide a definition for `exec` in terms of a C function, referred to as `c_exec_cmd` in the source. This function can be provided through Clean's foreign language call facility in terms of the appropriate system library function (`exec` or `system`, for instance).

D Haskell Semantic Model

```
module HaskMakeMProof where
```

```
import System
import List
import Maybe

type EpochTime = Integer

data FileTime = FileTime EpochTime | NoFileTime
  deriving Show
type Name = FilePath
type Command = String
data Target = Target Name Command [Target] | Leaf Name
  deriving Eq

type FS = [(Name,EpochTime)]
type World = (FS,EpochTime)

instance Eq FileTime where
  NoFileTime   == NoFileTime    = True
  NoFileTime   == _             = False
  _            == NoFileTime    = False
  (FileTime t) == (FileTime s)  = (t==s)

instance Ord FileTime where
  compare NoFileTime   NoFileTime   = EQ
  compare NoFileTime   _            = LT
  compare _            NoFileTime   = GT
  compare (FileTime t) (FileTime s) = (compare t s)

-----------------------------------------------
----- IO2 monad definition and operators -----
-----------------------------------------------

newtype IO2 a = IO2 (World -> (World, a))

instance Monad IO2 where
  return v                = retf
    where retf = IO2 (\w -> (w,v))
  (IO2 f1) >>= ac2    = IO2 bindf
    where
      bindf w =
        let (w1,v) = (f1 w)
            (IO2 f2) = (ac2 v)
        in  (f2 w1)

infix 1 >=>, >~>
infixl 1 >->

(>=>) :: World -> (IO2 a) -> (World, a)
w >=> (IO2 f) = (f w)
```

```
(>->) :: World -> (IO2 a) -> World
w >-> act = fst (w >=> act)

(>~>) :: World -> (IO2 a) -> a
w >~> act = snd (w >=> act)


-----------------------------------------
----- The two primitive actions on IO2 -----
-----------------------------------------

-- given a file name, return its modification time
getFileTime :: Name -> IO2 FileTime
getFileTime fn = IO2 (\(f,k) -> ((f,k), (td f)))
  where
    td f = case (lookup fn f) of
           Nothing  -> NoFileTime
           (Just t) -> FileTime t

-- logically execute command 'cmd'.
exec :: Name -> Command -> IO2 ()
exec nm cmd = IO2 (\(f,k) -> ((override nm (k+1) f, k+1),()))

---------------------------------------------------------------
----- The rest of "make", identical to the working version ------
---------------------------------------------------------------

-- make the target, and return the modification time of the
-- newest file of all recursive dependencies.
make :: Target -> IO2 FileTime
make (Leaf nm) = do
  mtime <- getFileTime nm
  if (mtime==NoFileTime)
    then error ("can't make "++nm++"!")
    else return mtime
make (Target nm cmd depends) = do
  -- get modification times of this file
  mtime <- getFileTime nm
  -- build and get the times of all children
  ctimes <- update_deps depends
  -- if its older than the newest child, rebuild, and return time
  if (mtime <= (newest ctimes))
    then do
      exec nm cmd     -- execute the command
      getFileTime nm  -- return the update time of the file
    else
      return mtime  -- just return mtime

-- given a list of times, find the newest
newest :: [FileTime] -> FileTime
```

```
newest times = foldl max NoFileTime times

-- update all the targets, returning all their times
update_deps :: [Target] -> IO2 [FileTime]
update_deps = mapM make


------------------------------------------------------
----- Additional functions for getting target info -----
------------------------------------------------------

-- get the name of a target
name :: Target -> Name
name (Leaf n) = n
name (Target n _ _) = n

-- get the immediate descendants of a target
deps :: Target -> [Target]
deps (Leaf _) = []
deps (Target _ _ ts) = ts

-- get the command of a target
cmd :: Target -> Command
cmd (Leaf _) = ""
cmd (Target _ m _) = m

-- get all the descendant nodes of a node, including the node itself
alldeps :: Target -> [Target]
alldeps t = [t] ++ (concatMap alldeps (deps t))

-- get all names of descendant nodes of a target, including
-- that node itself
allnames :: Target -> [Name]
allnames t = [name t]++(concatMap allnames (deps t))

-- get all names of descendant nodes in a target, excluding
-- that node itself
allnames2 :: Target -> [Name]
allnames2 t = (map name (deps t))++(concatMap allnames2 (deps t))

isSubTreeOf :: Target -> Target -> Bool
isSubTreeOf t t1 = t `elem` alldeps t1

----------------------------
----- Proof Expressions -----
----------------------------

-- Make's pre-condition
pre_make :: Target -> World -> Bool
pre_make t w = (clock_newer w) && (isDAGT t)
```

```
-- is every file in the filesystem no newer than the clock?
-- (clock-time is forced to be non-negative)
clock_newer :: World -> Bool
clock_newer (p,k) = (k >= (foldl max 0 (map snd p)))

-- are all targets newer than their dependencies?
deps_older :: World -> Target -> Bool
deps_older w (Leaf n) = True
deps_older w t@(Target n m ts) =
  ((w >~> getFileTime n) > (newest_dep w t)) &&
  (all (deps_older w) ts)

-- get the newest time of the *dependencies* of the target.
newest_dep :: World -> Target -> FileTime
newest_dep w t =
  newest (map (\n -> w >~> getFileTime n) (allnames2 t))

override :: Eq a => a -> b -> [(a,b)] -> [(a,b)]
override a b []          = [(a,b)]
override a b ((a',b'):m1)
  | a'==a                = (a,b):m1
  | otherwise            = (a',b'):(override a b m1)

disjoint, notdisjoint :: Eq a => [a] -> [a] -> Bool
disjoint l1 l2 = (l1 `intersect` l2) == []
notdisjoint l1 l2 = not (l1 `disjoint` l2)

-- subset and proper-subset operators
subset, psubset :: Eq a => [a] -> [a] -> Bool
subset [] s2 = True
subset (i1:s1) s2 = (i1 `elem` s2) && (s1 `subset` s2)
psubset s1 s2 = (s1 `subset` s2) && (not (s2 `subset` s1))

-- Set equality
setEq :: Eq a => [a] -> [a] -> Bool
setEq [] [] = True
setEq [] _  = False
setEq _  [] = False
setEq (x:xs) xs1 =
  if (x `elem` xs1)
    then setEq (filter (/= x) xs) (filter (/= x) xs1)
    else False

infix 5 \\\
-- set difference operator
(\\\) :: Eq a => [a] -> [a] -> [a]
(\\\) s1 [] = s1
(\\\) s1 (i2:s2) = (prune s1) \\\ (prune s2)
  where prune = filter (/= i2)
```

```
infixr 1 ==>
(==>) :: Bool -> Bool -> Bool
a ==> b = (not a) || b

isDAGT :: Target -> Bool
isDAGT t = prodall safe (alldeps t)

-- is (r a1 a2) true for all a's in the given list.
prodall :: (a -> a -> Bool) -> [a] -> Bool
prodall r lst = and [r a1 a2 | a1 <- lst, a2 <- lst]

-- if the two names are the same, are the commands and
-- dependencies also the same?
safe :: Target -> Target -> Bool
safe t1 t2 =
  ((name t1)==(name t2)) ==>
    ((cmd t1)==(cmd t2) && (deps t1)==(deps t2))

-- if (couldBeDAGT t0 t1), then they're exactly one of the following.
apartT, bwithinT, crushedT, dwithinT, equalT :: Target -> Target -> Bool
apartT t0 t1   = (allnames t0) `disjoint` (allnames t1)
bwithinT t0 t1 = (allnames t0) `psubset` (allnames t1)
crushedT t0 t1 =
  (allnames t0) `notdisjoint` (allnames t1) &&
  (not ((allnames t0) `subset` (allnames t1))) &&
  (not ((allnames t1) `subset` (allnames t0)))
dwithinT t0 t1 = (allnames t1) `psubset` (allnames t0)
equalT t0 t1   = (allnames t0) `setEq` (allnames t1)

-- if target t1 and t2 were "put together" they would make a valid DAGT
couldBeDAGT :: Target -> Target -> Bool
couldBeDAGT t1 t2 = prodall safe ((alldeps t1) ++ (alldeps t2))

-- are files 'ns' the same in two different worlds?
filesSame :: [Name] -> World -> World -> Bool
filesSame ns w1 w2 = all fileSame ns
  where
    fileSame n = (w1 >~> getFileTime n)==(w2 >~> getFileTime n)


-- run mapM on a world, and return all intermediate worlds.
trace :: (a -> IO2 b) -> [a] -> World -> [World]
trace a [] w = [w]
trace a (p:ps) w = (w : (trace a ps (w >-> a p)))

-----------------------
----- Test values -----
-----------------------
```

```
testTarg :: Target
testTarg = Target "a.out" "gcc prog.c intpair.o" [
  Leaf "prog.c",
  Target "intpair.o" "gcc -c intpair.c" [
    Leaf "intpair.h",
    Leaf "intpair.c"]]

testWorld :: World
testWorld = ([("prog.c",2),("intpair.h",3),("intpair.c",4)], 6)
```

E Clean Semantic Model

```
module cm_abstr
import StdEnv

:: EpochTime :== Int

:: FileTime = FileTime EpochTime
            | NoFileTime

:: Name :== String
:: Command :== String
:: Target = Target Name Command [Target]
          | Leaf Name

:: FS :== [(Name,EpochTime)]
:: World2 :== (FS,EpochTime)

:: Nil = Nil

instance == FileTime where
  (==) NoFileTime   NoFileTime   = True
  (==) NoFileTime   _            = False
  (==) _            NoFileTime   = False
  (==) (FileTime t) (FileTime s) = t==s

instance < FileTime where
  (<) NoFileTime   NoFileTime   = True
  (<) NoFileTime   _            = False
  (<) _            NoFileTime   = False
  (<) (FileTime t) (FileTime s) = t<s


(>=>) infixl 9 :: World2 (World2 -> (World2,a)) -> (World2,a)
(>=>) w f = f w
```

```
(>->) infixl 9 :: World2 (World2 -> (World2,a)) -> World2
(>->) w f = fst (w >=> f)

(>~>) infixl 9 :: World2 (World2 -> (World2,a)) -> a
(>~>) w f = snd (w >=> f)

testTarg = Target "a.out" "gcc prog.c intpair.o" [
  Leaf "prog.c",
  Target "intpair.o" "gcc -c intpair.c" [
    Leaf "intpair.h",
    Leaf "intpair.c"]]

testWorld = ([("prog.c",2),("intpair.h",3),("intpair.c",4)], 6)

//
// IO Primitives
//
getFileTime :: Name World2 -> (World2,FileTime)
getFileTime n (fs,k) = ((fs,k),lookup n fs)
    where lookup n []         = NoFileTime
          lookup n [(fn,t):xs] | n==fn     = FileTime t
                               | otherwise = lookup n xs

exec :: Name Command World2 -> (World2,Nil)
exec nm cmd (fs,k) = ((override nm (k+1) fs, k+1), Nil)
    where override a b [] = [(a,b)]
          override a b [(c,d):xs] | a==c = [(a,b):xs]
                                  | otherwise = [(c,d):override a b xs]

make :: Target World2 -> (World2, FileTime)
make (Leaf n) w = make' (getFileTime n w)
    where make' (w,NoFileTime)  = abort ("No rule to make file "+++n)
          make' (w,FileTime t) = (w,FileTime t)
make (Target n c depends) w
     # (w,times)                    = update_deps depends w
     # (w,this_time)                = getFileTime n w
     | this_time <= (maxList times) = getFileTime n (fst (exec n c w))
     | otherwise                    = (w,this_time)

update_deps :: [Target] World2 -> (World2,[FileTime])
update_deps [] w = (w,[])
update_deps [x:xs] w # (w,n)  = make x w
                     # (w,ts) = update_deps xs w
                     = (w,[t:ts])

//
// Functions for proofs, etc.
//

name :: Target -> Name
```

```
name (Leaf n) = n
name (Target n _ _) = n

deps :: Target -> [Target]
deps (Leaf _)       = []
deps (Target _ _ d) = d

cmd :: Target -> Command
cmd (Leaf _)        = ""
cmd (Target _ m _)  = m

// Needed for removeDup
instance == Target where
  (==) (Leaf n1) (Leaf n2) = n1==n2
  (==) (Target n1 c1 ts1) (Target n2 c2 ts2) =
            (n1==n2) && (c1==c2) && (ts1==ts2)
  (==) _ _ = False

alldeps :: Target -> [Target]
alldeps t = [t]++(concat (map alldeps (deps t)))
            where concat = foldr (++) []

allnames :: Target -> [Name]
allnames t = [name t]++(concat (map allnames (deps t)))
              where concat = foldr (++) []

allnames2 :: Target -> [Name]
allnames2 t =
    ((map name (deps t)) ++ (concat (map allnames2 (deps t))))
    where concat = foldr (++) []

isSubTreeOf :: Target Target -> Bool
isSubTreeOf t1 t2 = isMember t1 (alldeps t2)

// "Proof" expressions

pre_make :: Target World2 -> Bool
pre_make t w = (clock_newer w) && (isDAGT t)

clock_newer :: World2 -> Bool
clock_newer (fs,k) = k > (foldl max 0 (map snd fs))

deps_older :: World2 Target -> Bool
deps_older w (Leaf n) = True
deps_older w (Target n m ts) =
  ((w >~> getFileTime n) > (newest_dep w (Target n m ts))
    && (all (deps_older w) ts))

newest_dep :: World2 Target -> FileTime
newest_dep w t =
```

```
  maxList (map (\n -> w >~> getFileTime n) (allnames2 t))

//disjoint :: [a]  [a] -> Bool
disjoint l1 l2 = intersect l1 l2 == []

//intersect :: [a] [a] -> [a]
intersect [] ys = []
intersect [x:xs] ys | isMember x ys = [x: intersect xs ys]
                    | otherwise     = intersect xs ys

notdisjoint l1 l2 = not (disjoint l1 l2)

subset [] s2 = True
subset [i1:s1] s2 = (isMember i1 s2) && (subset s1 s2)

psubset s1 s2 = subset s1 s2 && (not (subset s2 s1))

//setEq :: [a] [a] -> Bool | == a
setEq [] [] = True
setEq [] _  = False
setEq _  [] = False
setEq [x:xs] xs1 =
  if (isMember x xs1)
    (setEq (filter (\x1 = x1 <> x) xs)
       (filter (\x1 = x1 <> x) xs1))
    (False)

(\\\) infix 5 :: [a] [a] -> [a] | == a
(\\\) s1 [] = s1
(\\\) s1 [i2:s2] = (prune s1) \\\ (prune s2)
  where prune = filter (\i1 = i1 <> i2)

(==>) infixr 1 :: Bool Bool -> Bool
(==>) a b = (not a)||b

isDAGT :: Target -> Bool
isDAGT t = prodall safe (alldeps t)

prodall :: (a a -> Bool) [a] -> Bool
prodall r lst = and [r a1 a2 \\ a1 <- lst, a2 <- lst]

safe :: Target Target -> Bool
safe t1 t2 =
    ((name t1) == (name t2)) ==>
        (((deps t1)==(deps t2)) && (cmd t1)==(cmd t2))

apartT t0 t1 = disjoint (allnames t0) (allnames t1)

bwithinT t0 t1 = psubset (allnames t0) (allnames t1)
```

```
crushedT t0 t1 = (notdisjoint (allnames t0) (allnames t1))
    && (not (subset (allnames t0) (allnames t1)))
    && (not (subset (allnames t1) (allnames t0)))

dwithinT t0 t1 = psubset (allnames t1) (allnames t0)

equalT t0 t1 = (allnames t0)==(allnames t1)

couldBeDAGT :: Target Target -> Bool
couldBeDAGT t1 t2 =
    prodall safe ((alldeps t1)++(alldeps t2))

filesSame :: [Name] World2 World2 -> Bool
filesSame ns w1 w2 = all fileSame ns
    where fileSame n = (w1 >~> getFileTime n)==(w2 >~> getFileTime n)

trace :: (a World2 -> (World2,b)) [a] World2 -> [World2]
trace a [] w     = [w]
trace a [p:ps] w = [w : (trace a ps (w >-> a p))]

//Start = trace make [testTarg,testTarg,testTarg] testWorld
Start = testWorld >-> make testTarg
```

References

[aA90] Mícheál Mac an Airchinnigh. *Conceptual Models and Computing*. PhD thesis, University of Dublin, Trinity College, Department of Computer Science, 1990.

[BS01] Andrew Butterfield and Glenn Strong. Proving correctness of programs with i/o — a paradigm comparison. In Markus Mohnen Thomas Arts, editor, *Proceedings of the 13th International Workshop, IFL2001*, number LNCS2312, pages 72–87, 2001.

[BW82] Friedrich L. Bauer and Hans Wossner. *Algorithmic Language and Program Development*, pages 272–273. Springer Verlag, 1982.

[Fel79] Stuart I. Feldman. Make-a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–65, 1979.

[HPJWe92] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler (editors). Report on the programming language haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, Mar, 1992.

[Hug00] Arthur Hughes. *Elements of an Operator Calculus*. PhD thesis, University of Dublin, Trinity College, Department of Computer Science, 2000.

[PvE01] Rinus Plasmeijer and Marko van Eekelen. Concurrent clean version 2.0 language report. `http://www.cs.kun.nl/~clean/`, December 2001.

[SM00] Richard M. Stallman and Roland McGrath. *GNU Make: A Program for Directing Recompilation, for Version 3.79*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, 2000.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.