Maak
A Build System

Eelco Dolstra
`eelco@cs.uu.nl`

March 4, 2002

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$
  - ⋆ Reliable staleness checking

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$
  - ⋆ Reliable staleness checking
    - ∗ Full dependencies (all sources, tools, flags)

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$
  - ⋆ Reliable staleness checking
    - ∗ Full dependencies (all sources, tools, flags)
    - ∗ Track derivatives

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$
  - Reliable staleness checking
    - Full dependencies (all sources, tools, flags)
    - Track derivatives
    - Derive dependencies (`make depend`)

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$
  - Reliable staleness checking
    - Full dependencies (all sources, tools, flags)
    - Track derivatives
    - Derive dependencies (`make depend`)
      - Fully automatically (like Clearcase)?

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$
    - Reliable staleness checking
        - Full dependencies (all sources, tools, flags)
        - Track derivatives
        - Derive dependencies (`make depend`)
            - Fully automatically (like Clearcase)?
- Scalability

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$
  - Reliable staleness checking
    - Full dependencies (all sources, tools, flags)
    - Track derivatives
    - Derive dependencies (`make depend`)
      - Fully automatically (like Clearcase)?
- Scalability
  - Support large projects, and inter-project dependencies

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$
  - Reliable staleness checking
    - Full dependencies (all sources, tools, flags)
    - Track derivatives
    - Derive dependencies (`make depend`)
      - Fully automatically (like Clearcase)?
- Scalability
  - Support large projects, and inter-project dependencies
  - Abstraction mechanisms to support this

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$
  - ⋆ Reliable staleness checking
    - ∗ Full dependencies (all sources, tools, flags)
    - ∗ Track derivatives
    - ∗ Derive dependencies (`make depend`)
      - ○ Fully automatically (like Clearcase)?
- Scalability
  - ⋆ Support large projects, and inter-project dependencies
  - ⋆ Abstraction mechanisms to support this
- Variant builds

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$
  - ⋆ Reliable staleness checking
    - ∗ Full dependencies (all sources, tools, flags)
    - ∗ Track derivatives
    - ∗ Derive dependencies (`make depend`)
      - ○ Fully automatically (like Clearcase)?
- Scalability
  - ⋆ Support large projects, and inter-project dependencies
  - ⋆ Abstraction mechanisms to support this
- Variant builds
  - ⋆ Must be easy to specify

Requirements

- Correctness (safeness), i.e., $\mathrm{Build}(W) = \mathrm{Build}(\mathrm{Clean}(W))$
  - Reliable staleness checking
    - Full dependencies (all sources, tools, flags)
    - Track derivatives
    - Derive dependencies (`make depend`)
      - Fully automatically (like Clearcase)?
- Scalability
  - Support large projects, and inter-project dependencies
  - Abstraction mechanisms to support this
- Variant builds
  - Must be easy to specify
  - Share common derivatives

Requirements (cont’d)

- Automatic ‘meta-operations’:
  - `make clean`
  - Making (source, binary) distributions

Requirements (cont'd)

- Automatic ‘meta-operations’:
  - ⋆ `make clean`
  - ⋆ Making (source, binary) distributions

- Package management
  - ⋆ Packages are also dependent on each other

Problems with `make`

- Not enough abstraction:

```
prog: a.o b.o
    gcc -o prog ...
a.o: a.c
    gcc a.c
b.o: b.c
    gcc b.c
b.c: b.y
    yacc ...
```

while the conceptual model is:

```
prog: a.c b.y
```

Problems with `make`

- Not enough abstraction:

```
prog: a.o b.o
    gcc -o prog ...
a.o: a.c
    gcc a.c
b.o: b.c
    gcc b.c
b.c: b.y
    yacc ...
```

while the conceptual model is:

```
prog: a.c b.y
```

- Not enough expressive power
  - ⋆ Almost no genericity

Maak

- Expressive power: input is lazy functional language

Maak

- Expressive power: input is lazy functional language
- Abstraction:
  - ⋆ Tools specified separately

Maak

- Expressive power: input is lazy functional language
- Abstraction:
  - ⋆ Tools specified separately
  - ⋆ Targets specified in terms of *conceptual* dependencies; concrete dependencies and intermediates inferred automatically using tool definitions

Example: ATerm

- Multiple variants: regular, debugging, different compiler, etc.

Example: ATerm

- Multiple variants: regular, debugging, different compiler, etc.

- Currently implemented using makefile hacks:

```
%-cc.o   : %.c $(ALLINCLUDES)
CFLAGS= $(CC_COMPILE) -c $< -o $@

%-dbg.o  : %.c $(ALLINCLUDES)
$(DBG_COMPILE) -c $< -o $@

%-gcc.o  : %.c $(ALLINCLUDES)
$(GCC_COMPILE) -c $< -o $@

libATerm_dbg_a_LIBADD = $(ALLSRCS:.c=-dbg.o)
```

Example: ATerm

- Multiple variants: regular, debugging, different compiler, etc.
- Currently implemented using makefile hacks:

```
%-cc.o   : %.c $(ALLINCLUDES)
CFLAGS= $(CC_COMPILE) -c $< -o $@

%-dbg.o  : %.c $(ALLINCLUDES)
$(DBG_COMPILE) -c $< -o $@

%-gcc.o  : %.c $(ALLINCLUDES)
$(GCC_COMPILE) -c $< -o $@

libATerm_dbg_a_LIBADD = $(ALLSRCS:.c=-dbg.o)
```

- So for every variant we have to add more rules

Example: ATerm (cont'd)

- In Maak:

```
include <stdlib.mk>;

srcs = [ <aterm.c> <list.c> ... ];

<libATerm.a> { in = srcs };
<libATerm-dbg.a> { in = srcs, cflags = "-g" };
<libATerm-gcc.a> { in = srcs, cc = "gcc" };

<primes>
  { type = "exe"
  , in = [<../test/primes.c>]
  , libs = [<libATerm-dbg.a>]
  };

"default" { type = "dummy", in = [<primes>] }
```

Example: bootstrapping GCC

```
gcc = {in = [gcc.c expr.c ...]};

gcc1 = <gcc> {cc = "/usr/bin/cc"};

gcc2 = <gcc> {cc = gcc1};

gcc3 = <gcc> {cc = gcc2};
```

Tool definitions

```
suffix ".o" "loadable";
suffix ".c" "csrc";

tool "loadable" \x ->
  { force =
      x { in = force "csrc" x.in };
      fdep x x.in;
  , build = exec "cc" [x.cflags "-c" x.in "-o" x]
  };
```

Implementation

- Currently in Haskell

Implementation

- Currently in Haskell
- Very simple input language; everything is an `Expr`
  e.g., `includes` are just expressions

Implementation

- Currently in Haskell
- Very simple input language; everything is an `Expr`
  e.g., `includes` are just expressions
- Evaluation using state monad on top of I/O monad; nice: short interpreter

Problem

- Should the input language be:
  - ⋆ Impure: build the graph as a side-effect of evaluation? (Original approach)

Problem

- Should the input language be:
  - ⋆ Impure: build the graph as a side-effect of evaluation? (Original approach)
  - ⋆ Pure/declarative, i.e., yield a graph as result of evaluation?

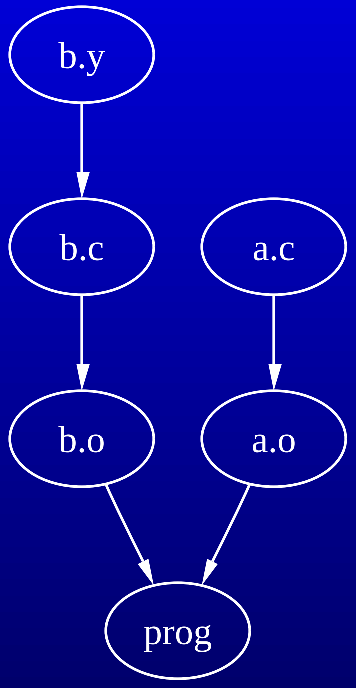
Pure variant

- Original approach was impure

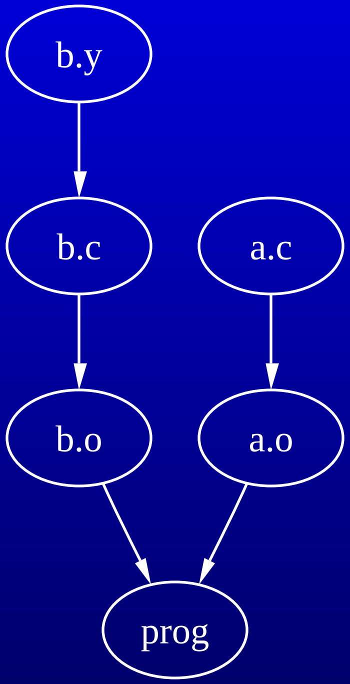
- Making it more declarative

```
default = { type = "dummy", in = [primes] };

primes =
  { type = "exe", out = <primes>
  , in = [<../test/primes.c>], libs = [libATerm]
  };

libATerm = { out = <libATerm.a>, in = srcs };

libATerm-dbg = libATerm
  | { out = <libATerm-dbg.a> }
  | propagate { cflags = "-g" };
```

Wrong model



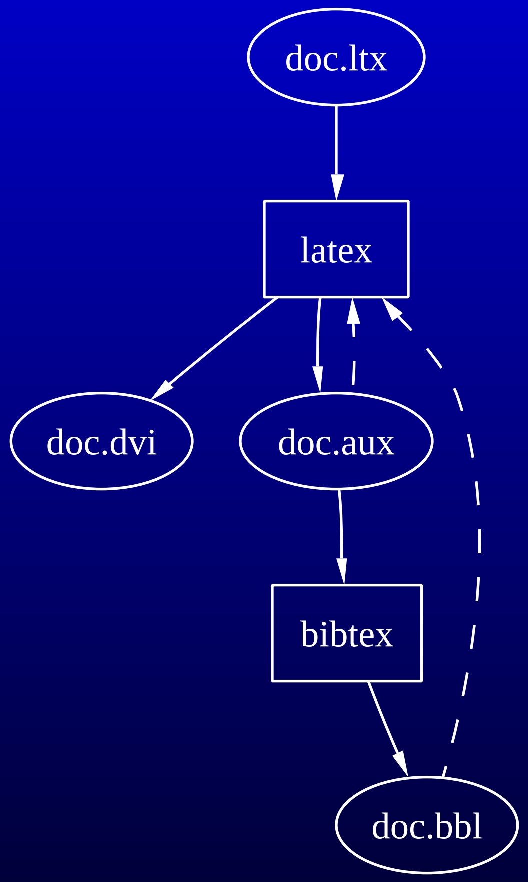
Wrong model



- Doesn’t work when tools have multiple outputs

Circular dependencies

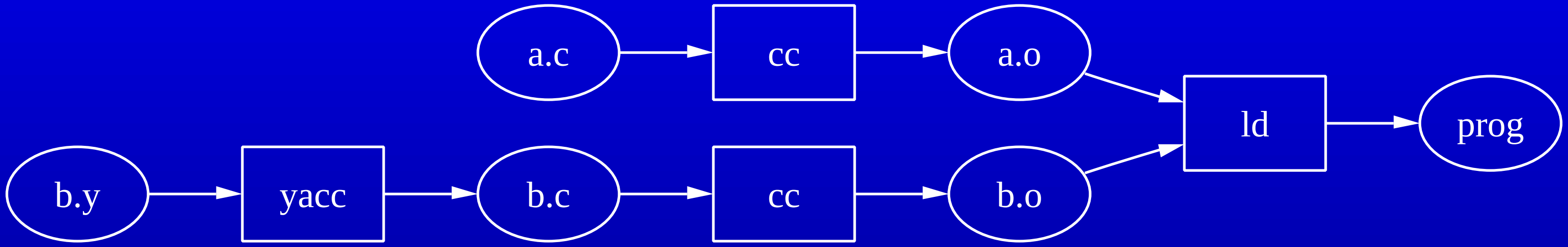
Iterate until fixpoint:



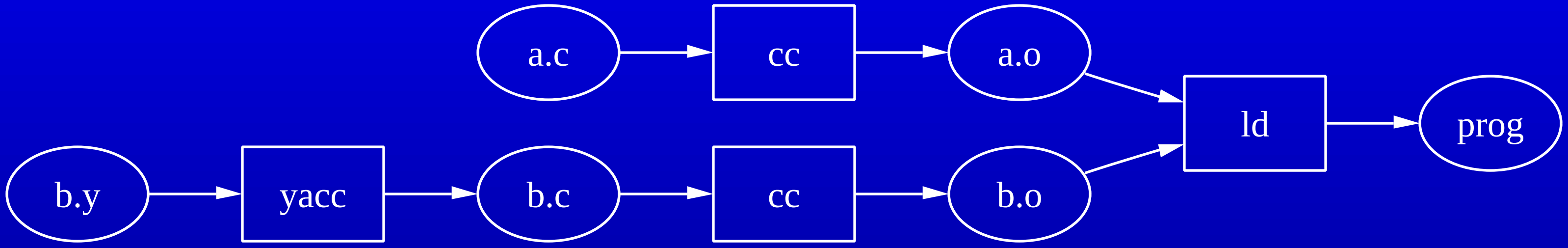
Dashed = optional dependency

Might be non-terminating!

Right model

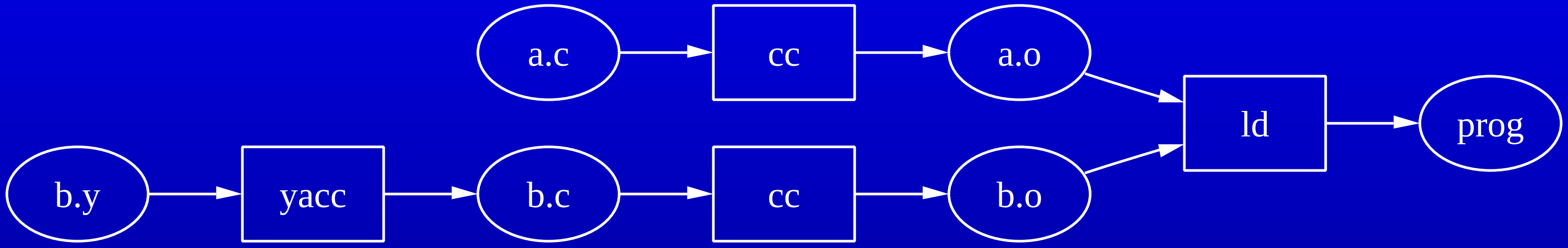


Right model



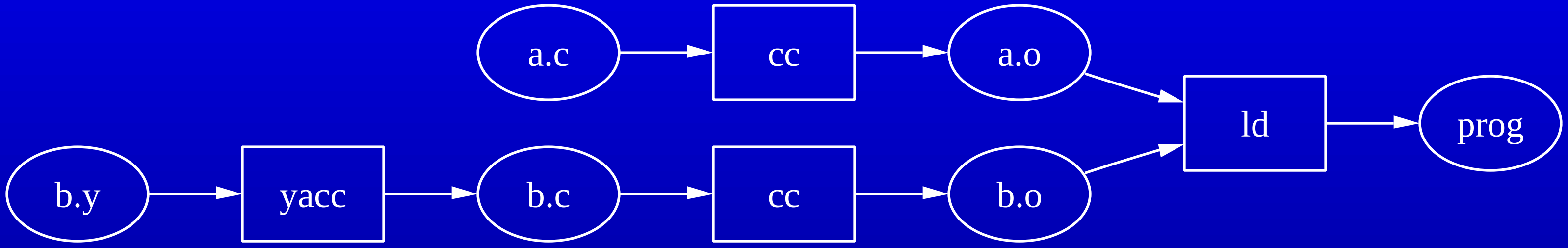
- Separate targets and tool instantiations

Right model



- Separate targets and tool instantiations
- Targets and tool instantiations both have attributes

Right model



- Separate targets and tool instantiations
- Targets and tool instantiations both have attributes
- How to specify?

Future work

- Fix model

Future work

- Fix model
- Implement statefile

Future work

- Fix model
- Implement statefile
- Dissociate filenames and target names $\Rightarrow$ packaging

Future work

- Fix model
- Implement statefile
- Dissociate filenames and target names ⇒ packaging
- Generic operations: cleaning, making distributions

Future work

- Fix model
- Implement statefile
- Dissociate filenames and target names ⇒ packaging
- Generic operations: cleaning, making distributions
- Distributed statefiles

Future work

- Fix model
- Implement statefile
- Dissociate filenames and target names ⇒ packaging
- Generic operations: cleaning, making distributions
- Distributed statefiles
- Generic inferencing of dependencies: `strace`