A new, modular dependency solver for cabal-install

Andres Löh  Duncan Coutts
Well-Typed LLP  Well-Typed LLP

HIW 2011

23 September 2011
Why?

- Error messages are suboptimal.
- We want the solver to be more configurable.
- Solver is difficult to extend (current design is rather ad-hoc).
Why now?

We have started to work on this as part of a project funded by the Industrial Haskell Group.

Original trigger: support “private dependencies”.
Why now?

We have started to work on this as part of a project funded by the Industrial Haskell Group.

Original trigger: support “private dependencies”.

Status

► New solver is implemented.
► “At least as good” as the old solver – when used with ghc-7.*
► Ready for experimentation.
► Some new features prepared, but not yet implemented.
Talk overview

- Dependencies in Cabal
- Architecture of the new solver
- Private dependencies
- Error messages
- Remaining work
Dependencies in Cabal
When you call cabal install,
When you call `cabal install`,

1. the Hackage index is read,
2. the installed packages index is read,
When you call `cabal install`,

1. the Hackage index is read,
2. the installed packages index is read,
3. the solver is invoked to produce an install plan,
When you call `cabal install`,

1. the Hackage index is read,
2. the installed packages index is read,
3. the solver is invoked to produce an install plan,
4. the install plan is sanity checked,
When you call `cabal install`,

1. the Hackage index is read,
2. the installed packages index is read,
3. the solver is invoked to produce an install plan,
4. the install plan is sanity checked,
5. the install plan is executed (or printed).
When you call `cabal install`,

1. the Hackage index is read,
2. the installed packages index is read,
3. the solver is invoked to produce an install plan,
4. the install plan is sanity checked,
5. the install plan is executed (or printed).
When you call `cabal install`,

1. the Hackage index is read,
2. the installed packages index is read,
3. the solver is invoked to produce an install plan,
4. the install plan is sanity checked,
5. the install plan is executed (or printed).

Independent sanity check makes it relatively easy to trust the new solver.
## Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Index</strong></td>
<td>database of information about packages (can be built from many indices)</td>
</tr>
<tr>
<td><strong>Location</strong></td>
<td>an index location (such as Hackage, or your installed packages)</td>
</tr>
<tr>
<td><strong>Package Name</strong></td>
<td>a name such as <code>mtl</code> or <code>threadscope</code></td>
</tr>
<tr>
<td><strong>Version</strong></td>
<td>a version such as <code>1.1.0.2</code></td>
</tr>
<tr>
<td><strong>Instance</strong></td>
<td>a triple of a name, version and location</td>
</tr>
<tr>
<td><strong>Instance Constraint</strong></td>
<td>restrictions on version and location</td>
</tr>
<tr>
<td><strong>Dependency</strong></td>
<td>a pair of a name and a constraint</td>
</tr>
</tbody>
</table>
Cabal dependency problem

- Cabal files can specify dependencies based on boolean flags.

- The complete transitive dependency closure of a package is needed to uniquely identify it (ABI hash).

- In other words: installed instances have fixed dependencies, new instances have flexible dependencies.

- In general, one application cannot use multiple instances of the same package.

- With ghc-pkg, we can install many instances of one package, but only one instance per package version.
Cabal dependency problem

- Cabal files can specify dependencies based on boolean flags.
- The complete transitive dependency closure of a package is needed to uniquely identify it (ABI hash).
Cabal dependency problem

- Cabal files can specify dependencies based on boolean flags.
- The complete transitive dependency closure of a package is needed to uniquely identify it (ABI hash).
- In other words: installed instances have fixed dependencies, new instances have flexible dependencies.
Cabal dependency problem

- Cabal files can specify dependencies based on boolean flags.
- The complete transitive dependency closure of a package is needed to uniquely identify it (ABI hash).
- In other words: installed instances have fixed dependencies, new instances have flexible dependencies.
- In general, one application cannot use multiple instances of the same package.
Cabal dependency problem

- Cabal files can specify dependencies based on boolean flags.
- The complete transitive dependency closure of a package is needed to uniquely identify it (ABI hash).
- In other words: installed instances have fixed dependencies, new instances have flexible dependencies.
- In general, one application cannot use multiple instances of the same package.
- With ghc-pkg, we can install many instances of one package, but only one instance per package version.
Approach of the old solver

**Conservative approach**

Tries to select a unique instance per package in an install plan.
Approach of the old solver

Conservative approach

Tries to select a unique instance per package in an install plan.

Furthermore:

- Never backtracks (but looks ahead a bit).
- Exclude packages that can’t be configured (relatively new).
- Flag resolution tied to package selection.
- Maintains hardly any information about the order in which packages are resolved.
Cabal can break your system

Assume $B-1$ depends on any version of $A$. 
Cabal can break your system

You install B-1 on your system, fixing the dependency to A-1.
Cabal can break your system

Many other packages that depend on B–1 are installed later.
Now we want to install D which depends on A-2 (!) and B.
Cabal can break your system

Since B still depends on A, the install plan selects A-2.
Cabal can break your system

Upon actual installation, the old B–1 is destructively updated . . .
# About destructive updates

## Proper solution

NixOS-style package database – no destructive updates, ever.
## About destructive updates

### Proper solution

NixOS-style package database – no destructive updates, ever.

### Other options

- warn explicitly about destructive updates
- discourage or prevent destructive updates in the solver options
The new solver
The algorithm

1. Build (lazily) an approximation of the search tree. The tree contains all valid solutions (but may contain wrong ones).

Inspired by Thomas Nordin and Andrew Tolmach, Modular Lazy Search for Constraint Satisfaction Problems, JFP, 2001
The algorithm

1. Build (lazily) an approximation of the search tree. The tree contains all valid solutions (but may contain wrong ones).

2. Validate the search tree. The resulting tree contains only valid solutions.
The algorithm

1. Build (lazily) an approximation of the search tree. The tree contains all valid solutions (but may contain wrong ones).
2. Validate the search tree. The resulting tree contains only valid solutions.
3. Rearrange and optimize the search tree. Many independent traversals over the tree. This is what makes the solver modular.
The algorithm

1. Build (lazily) an approximation of the search tree. The tree contains all valid solutions (but may contain wrong ones).

2. Validate the search tree. The resulting tree contains only valid solutions.

3. Rearrange and optimize the search tree. Many independent traversals over the tree. This is what makes the solver modular.

4. Explore the search tree. Once a solution is found, we turn it into an install plan.

Inspired by . . . Thomas Nordin and Andrew Tolmach, Modular Lazy Search for Constraint Satisfaction Problems, JFP, 2001
The algorithm

1. Build (lazily) an approximation of the search tree. The tree contains all valid solutions (but may contain wrong ones).
2. Validate the search tree. The resulting tree contains only valid solutions.
3. Rearrange and optimize the search tree. Many independent traversals over the tree. This is what makes the solver **modular**.
4. Explore the search tree. Once a solution is found, we turn it into an install plan.

**Inspired by . . .**

Thomas Nordin and Andrew Tolmach, Modular Lazy Search for Constraint Satisfaction Problems, JFP, 2001
The algorithm

```
solve cfg idx userPrefs userConstraints userGoals =
    explorePhase $ 
    heuristicsPhase $ 
    preferencesPhase $ 
    validationPhase $ 
    prunePhase $ 
buildPhase 
where
        buildPhase :: Tree . . . 
        prunePhase :: Tree . . . → Tree . . . 
        . . . 
        explorePhase :: Tree . . . → Log (Assignment, RevDepMap)
```
The search tree

```haskell
data Tree a =
    PChoice QPN a (PSQ I (Tree a))
  | FChoice QFN a Bool (PSQ Bool (Tree a))
  | GoalChoice (PSQ OpenGoal (Tree a))
  | Done RevDepMap
  | Fail (ConflictSet QPN) FailReason
```
The search tree

```haskell
data Tree a =
    | PChoice QPN a (PSQ I (Tree a))
    | FChoice QFN a Bool (PSQ Bool (Tree a))
    | GoalChoice (PSQ OpenGoal (Tree a))
    | Done RevDepMap
    | Fail (ConflictSet QPN) FailReason
```

- Nodes represent goals / choices.
- Edges represent assignments.
- The search tree is **not** the dependency tree.
The search tree

Several kinds of nodes:

- **Goal Choice** to give some flexibility to the order in which goals are solved
The search tree

Several kinds of nodes:

- **Goal Choice** to give some flexibility to the order in which goals are solved
- **Package Goals** to pick an instance for a given name
- **Flag Goals** to select a boolean for a given flag for a given package
The search tree

Several kinds of nodes:

**Goal Choice** to give some flexibility to the order in which goals are solved

**Package Goals** to pick an instance for a given name

**Flag Goals** to select a boolean for a given flag for a given package

**Success** to indicate that a solution has been found

**Fail** to explicitly indicate failure
The search tree

Several kinds of nodes:

- **Goal Choice**: to give some flexibility to the order in which goals are solved
- **Package Goals**: to pick an instance for a given name
- **Flag Goals**: to select a boolean for a given flag for a given package
- **Success**: to indicate that a solution has been found
- **Fail**: to explicitly indicate failure

All nodes are equipped with additional info.

Note that goal-choice nodes have a different semantics from single-goal nodes.
Building the tree

Keep track of current goals:
- build goal choice node
- build goal-specific nodes
- add new goals depending on choice if needed
- if no goals left, end with a success node
Validating the tree

Keep track of constraints:

- use constraints to remove choices
- but keep disabled choices around (for error messages)
- add new constraints corresponding to the choices while moving down

After validation, the tree contains just correct solutions.
Reorderings

- exploration will in essence proceed depth-first, left-to-right
- the order of subtrees in the choice nodes is relevant
- we can thus express preferences by reordering
Typical preferences

- Prefer later versions over older versions.
- Prefer installed instances over non-installed ones.
Typical preferences

- Prefer later versions over older versions.
- Prefer installed instances over non-installed ones.

```haskell
packageOrderFor :: (PN → Bool) →
  (PN → I → I → Ordering) →
  Tree a → Tree a

packageOrderFor p cmp = trav go

where
  go (PChoiceF v@(Q _ pn) r cs)
    | p pn = PChoiceF v r
    | otherwise = PChoiceF v r
    (sortByKeys (flip (cmp pn)) cs)
  go x = x
```
Goal heuristics

Goal choices allow us to do some limited look-ahead:

- Prefer goals that lead to immediate failure.
- Prefer goals (package names and flags) that have only one allowed choice.
Goal heuristics

Goal choices allow us to do some limited look-ahead:

- Prefer goals that lead to immediate failure.
- Prefer goals (package names and flags) that have only one allowed choice.

Because it does never make sense to backtrack in goal-choice nodes, we actually leave only the first goal choice after applying goal heuristics.
Backjumping

We prune the tree by propagating failure information up.

**Key observation**

Not all the nodes on a path to a conflict actually contribute to that conflict.
Backjumping

We prune the tree by propagating failure information up.

**Key observation**

Not all the nodes on a path to a conflict actually contribute to that conflict.

So in many cases we can prune entire failing subtrees quickly.
We prune the tree by propagating failure information up.

**Key observation**

Not all the nodes on a path to a conflict actually contribute to that conflict.

So in many cases we can prune entire failing subtrees quickly.

We also use conflict set info for error messages.
Exploring the tree

Mainly depth-first, left-to-right being used, but with various degrees of debugging info being produced.
Exploring the tree

Mainly depth-first, left-to-right being used, but with various degrees of debugging info being produced.

Configurable backtracking: we can impose a limit on the number of backjumps performed – the old solver never backtracks.
Reflections on the implementation

- The approach is reasonably easy to work with in practice, because we can split the algorithm into multiple independent steps.
- We haven’t made any effort on clever optimizations, and the new solver has about the same speed in practice as the old one . . .
- However, while laziness is the key to modularity here, there are also very subtle laziness constraints that aren’t expressible in the type system.
Encapsulations
Encapsulations

The assumption to allow only one instance of a package per application is too conservative:

▶ The problem arises from exported types that aren’t compatible between instances.
▶ Some libraries are mostly used privately or don’t export any types.
▶ If we could declare such private dependencies, we might get better results.
Encapsulations

The assumption to allow only one instance of a package per application is too conservative:
- The problem arises from exported types that aren’t compatible between instances.
- Some libraries are mostly used privately or don’t export any types.
- If we could declare such private dependencies, we might get better results.

Idea

We allow a package A to declare that it encapsulates package B. Ideally, it should be checked that A’s interface contains no traces of B.
Encapsulations

The assumption to allow only one instance of a package per application is too conservative:

▶ The problem arises from exported types that aren’t compatible between instances.
▶ Some libraries are mostly used privately or don’t export any types.
▶ If we could declare such private dependencies, we might get better results.

Idea

We allow a package $A$ to declare that it encapsulates package $B$. Ideally, it should be checked that $A$’s interface contains no traces of $B$.

However, encapsulations are subtle . . .
Encapsulation example

Different instances can be chosen for B and C.B.
Encapsulation example

The encapsulation is invalidated by other dependencies. Both B and C.B must be the same instance.
The plan for encapsulations

- The solver is prepared for scoped goals.
- Scoped goals are introduced when packages declare encapsulations.
- Scoped goals can be invalidated by other dependencies (i.e., forced to be equal to original goals).
- Scoped goals should therefore be resolved as late as possible, to prevent unnecessary backtracking.
- Even if scoped goals can be resolved differently, the preference should still be to select a single instance per package.
- The install plan checker has to be generalized.
- The Cabal library needs to be extended to check private dependencies.
# Error messages

## Ideas

- Add as much information internally as possible, i.e., keep reasons for all decisions.
- Make traces, while being verbose, fully understandable.
- Error messages are excerpts of the full trace, removing irrelevant parts.
Ideas

▶ Add as much information internally as possible, i.e., keep reasons for all decisions.
▶ Make traces, while being verbose, fully understandable.
▶ Error messages are excerpts of the full trace, removing irrelevant parts.

Current status

▶ All the information is there.
▶ Could be presented in a better/cleaner way.
▶ There are some choices on how to produce excerpts.
Demo
Future work

- Turn cabal-install into a proper library.
- Factor out the solver into a separate package.
- Others can write their own solvers.
- Integrate cabal-dev, or make such tools use the library.
- Track external dependencies.
- Allow more configuration options.
Please try it

We appreciate early feedback:

darcs get \  
   http://darcs.haskell.org/cabal-branches/cabal-modular-solver  
  cd cabal-modular-solver  
  cabal install ./cabal ./cabal-install