Mypy incremental checking

*What follows is the original design doc for incremental checking. Much of this has been implemented (especially see the list of **UPDATES** below). Much of it has been left to do as an intern project, especially the section on Technical debt.*

Mypy is fairly slow (~3K LOC/sec). One strategy for improving the perceived speed during the typical development work cycle (edit-check-edit-check-etc.) is to save the results of type-checked files on disk and only re-check files that have actually changed. This document explores that idea, and variants, e.g. for finer granularity at the class or function level. The architecture described here also lays the groundwork for parallelization and highly interactive type checking.

A useful discussion of this topic already existed in a GitHub issue for mypy: https://github.com/python/mypy/issues/932. The topic of parallel checking is discussed in https://github.com/python/mypy/issues/933.

Problem description

Let's consider two modules, A and B, where A imports B. In the current, non-incremental scheme, whenever the developer edits A and runs "mypy A", mypy must also check B, plus everything imported by B (all the way to the builtins module). In a large program (such as the Dropbox server) this usually means that whenever one module is edited, hundreds of modules must be checked.

Approach

The basic idea for incremental checking is to dump the symbol table for B to disk after B has been checked. Then on a subsequent run to check A, when B isn't changed, we can load its symbol table instead of parsing and checking B from scratch. The assumption here is that loading the symbol table can be done much faster than parsing and type-checking the module. (This hypothesis appears to be borne out by tests: mypy can check itself, nearly 30K LOC, in 9 seconds, and in incremental mode it takes under 1.5 seconds when no files are changed.)

Complications happen due to dependencies between modules — if B in turn imports C, B's symbol table likely contains cross-references to C, so we must load C before B, and after loading B we must fix up those cross-references. (Similar dependencies exist for parsing and checking.)

That's all doable, but the real killer happens when there's an import cycle, e.g. if C in turn imports B. (The non-incremental checker doesn't solve this perfectly either.) In that case it looks like we must load the symbol tables for B and C and then fix up both their cross-references, before we can start checking A. Note that in the checker we can't handle import cycles the way we deal with them at run time — to the checker all dependencies are equal, and the delayed import trick doesn't work. Also note that import cycles are a major issue at Dropbox.

Another form of complication is due to the structure of the symbol table. The top level of a module's symbol table simply maps names (e.g. class and function names, and of course global variables including imports) to symbol table nodes which can contain a variety of information. For a function, the symbol table node contains information about the function's signature. For a class, the symbol table node contains a nested symbol table that maps the class's methods, subclasses, and instance variables (as well as class variables — mypy currently doesn't distinguish between the two). Special kinds of symbol table nodes also exist to hold cross-references to other modules (these represent imported modules) and type variables.

Whenever a symbol table node references a *type* (e.g. an argument or return type for a function, or the type of a global variable, or a base class) it contains a different kind of node, a type node. There are many kinds of type nodes, representing e.g. Any, Void/None, Callable, Tuple, and so on. Many type nodes reference other type nodes. There's at least one special case: Instance, named thus because it represents an instance of a generic type, i.e. a concrete type. It references a TypeInfo object which is what's contained in the symbol table node for a class definition. These are cross-references, and care should be taken to serialize Instance objects as *references* to TypeInfo objects (as opposed to simply serializing the contents of the TypeInfo).

Further optimizations

If we edit some module U that is imported (directly or indirectly) by hundreds of other modules, then re-checking the program using the above approach still requires re-parsing and re-checking those hundreds of other modules. There are probably ways to avoid most of that work most of the time. For example if, after the new version of U has been checked, the (relevant) information in its symbol table is the same as before, we can conclude that we don't need to re-check any module depending on U after all (assuming nothing else changed). Another, more complex, approach might be to keep track of dependencies between modules at the class or function level and only re-check code that depends directly on a changed class or function.

Project status

I've come up with (and implemented) a design for solving the complications due to symbol table cross-references, module dependencies, and (to a lesser extent) import cycles. There are still issues with cross-references and I haven't been able to prove the speed of the scheme. My

code is in a private branch of the mypy repo; I've been reluctant to publish my code mostly because it doesn't yet work and is full of debugging code.

UPDATE: I have most of the basic logic working, although it still crashes when pointed at mypy itself. So far I can measure a modest speed-up, approximately 2x.
UPDATE 3/3/16: It no longer crashes on mypy itself (though there are probably still crashes possible). Fixing various corner cases slowed it down a bit, but I'm optimistic that there are many good opportunities for speeding things up (I have intentionally not worried about speed at all so far).
UPDATE 3/8/16: Bigger speedup (~5x) due to reduced volume of cache files; fixed several corner cases.
UPDATE 3/12/16: With David F's help, wrote a new module dependency manager. It works and solves most problems, and is a much better starting point for further improvements. Read the description and the code.
UPDATE 4/7/16: Some rounds of code review later, ready for the final stretch. Everything works, speed is adequate (5x), cache file sizes are reasonable.
UPDATE <later>: It's been committed, but not widely used; it's been broken frequently due to lack of tests.

Code walk-through

I've tried to limit my changes to build.py, nodes.py, types.py, and a new file fixup.py. (The pull request can be seen here.) The logic is distributed over these file roughly as follows:

- build.py: keep track of module dependencies and cache file management
- nodes.py: code to serialize/deserialize nodes (symbol table and certain parse tree nodes)
- types.py: code to serialize/deserialize types
- fixup.py: specialized visitors (mostly) used to fix up cross-references after deserialization

Other changes:

- semanal.py: some changes related to NamedTuple
- tests
- files changed due to merges from master (at some point I decided to merge rather than rebase)

Cache file format

- The cache is a directory named ".mypy_cache" in the current directory. There's a subdirectory for each Python version, e.g. "2.7" or "3.5". (We may allow user control over the cache root in the future.)
- For each module foo.py analyzed, two JSON files are written to the cache: foo.meta.json contains the "metadata" (including dependencies) while foo.data.json contains the serialized symbol table.

- For a submodule of a package, e.g. x/y/z.py, the cached files are x/y/z.{meta,data}.json.
- Cache files are also written for stub files (even for builtins.pyi).
- The path in the cache is derived from the module name only: module x.y.z translates to x/y/z.{meta,data}.json no matter where x/y/z.py or x/y/z.pyi was found.

Cache metadata format (foo.meta.json)

Metadata files contain a single JSON object with the following keys:

- id (str): the full module name; e.g. “x.y.z”
- path (str): the path where the file was found; e.g. “x/y/z.py”
- mtime (float): the mtime from stat(path), as a POSIX timestamp
- size (int): the size of path, in bytes
- dependencies (List[str]): directly imported module ids; e.g. ["typing”, “abc”]
- data_mtime (float): the mtime of the cache *data* file

Cache data format (foo.data.json)

Data files contain a single JSON object representing the serialized symbol table for a module. This is a highly recursive object.

- Each object has a “.class” key giving the class name from which that object was serialized; this is a class name in the nodes or types module (without the package prefix).
- The top level object represents the MypyFile object.
- Most objects have keys corresponding to the instance variables/attributes/properties of that object, where the corresponding values are the serialized attribute values. For example, the type Union[int, str] is represented as follows (modulo whitespace):

```
{".class": "UnionType",
 "items": [
  {".class": "Instance", "type_ref": "builtins.int"},
  {".class": "Instance", "type_ref": "builtins.str"}
 ]
}
```

- Instance object is an exception: in memory it contains a pointer to a TypeInfo object (which represents a class), but in serialized form it has a special key “type_ref” that gives the full name of the class. This must be fixed up after deserialization; the fixup.py module handles this.
- For SymbolTableNode objects, the “kind” field is encoded as a string rather than an int; the strings make the JSON a little more readable. (We might reconsider this for speed.)
- Some fields present in memory are not serialized because they are redundant and can be recomputed from other serialized data. For example, we don’t serialize CallableType.min_args, since it is recomputed by the constructor from arg_kinds.
- We also don’t serialize TypeInfo.mro; there’s a second fixup pass that calculates the mro for all deserialized TypeInfo objects by calling their calculate_mro() method. (Jukka suggested that we can skip this pass by putting the mro in the serialized data; the second fixup pass is not used for anything else.)
- We also don’t serialize “back pointers”, in particular the info attribute of class methods and other class attributes. These are restored by the first fixup pass. It might be possible

to do this in the deserialization code instead by slightly reworking it, though it's not clear whether this would represent a speed-up (the first fixup pass is still needed to fix module cross-references).

Cache validation and dependency management

Cache validation happens in several stages. All these happen in build.py. It is closely related to dependency management, because the cache should not be used for a module if any of its dependencies weren't loaded from the cache (even if the dependent module's own cache is valid).

- Whenever we find the cache data or metadata for a given module is invalid we fall back on parsing and analyzing its source code.
- Whenever mypy encounters a module (either given on the command line or by following an import), first of all we locate the source file using the module search path (lib_path) and stat() it. We then compute the cache metadata filename (e.g. foo.py → foo.meta.json).
- If the cache metadata file exists, can be read, is well-formed JSON, and matches our schema (see cache metadata format above), we check that the fields match the source file and its stat() data (i.e. id, path, mtime and size must all match). If we don't get this far we fall back on the source.
- Next we stat() the corresponding data file (e.g. foo.data.json). Its mtime must match the data_mtime field in the metadata; if not, we fall back on the source. But even if it's up to date, we don't read the data file yet! (Since we may still decide to fall back on the source anyway.)
- The rest of the algorithm is best summarized as follows:
    - (a) collect the full graph of dependencies;
    - (b) find strongly connected components (SCCs);
    - (c) do a topological sort of the SCCs and process each SCC in order;
    - (d) run each analysis pass on all nodes of an SCC before starting the next pass.
- There are quite a few wrinkles, perhaps the most important one being the order in which to run the analysis passes on the nodes of an SCC. I ended up doing this in the *reverse* order in which the modules were encountered during a breadth-first traversal of the dependencies (which AFAICT is how the original code handled cycles).
- For more information read the [description of the new dependency manager].

Serialization/deserialization API

- Every serializable class has an instance method serialize() returning a JsonDict object (this is an alias for Dict[str, Any]).
- Base classes may define a serialize() method that raises NotImplementedError.
- Corresponding to every serialize() instance method is a class method deserialize(data) that takes a JsonDict object and returns an instance of that class.

- Base classes define a deserialize() method that looks at the data and dispatches to the appropriate subclass's deserialize() method; if the subclass doesn't define deserialize() this raises NotImplementedError.
- The dispatch by the base class deserialize() method is typically done based on the ".class" key in the data.
- However for optimization some special cases may apply. (For example we should probably serialize common cases of Instance as a string instead of a dict.)

Technical debt

Main TO DO list

- A few TODOs left in the code.
- A bunch of asserts that I hope will never trigger.
- If there are any errors for a module, we're not writing it to the cache. This may be slow if errors are not dealt with. (Possible solution: write the errors to the cache too.)
- Should multiple references to the same Var object be combined?
- Support other flags (--silent-imports, --implicit-any) that affect the contents of symbol tables (I think --implicit-any may not have any effect; but --silent-imports is tricky).
- More command-line flags to control caching: cache root dir, clear cache, debugging.

Possible speed improvements

- There are too many UnboundType objects; Argument shouldn't need to serialize these.
- Some classes have a lot of boolean attributes to indicate various properties (e.g. FuncDef has is_classmethod, is_staticmethod, and quite a few others). We could reduce the size of the serialized data by only serializing such attributes when they differ from the default.
- Don't encode the kind of a SymbolTableNode as a string — use the native int.
- Replace basic Instance {".class": "Instance", "type_ref": "blah.blah"} with the string "blah.blah".
- Serialize the mro of a class instead of recomputing it; this saves the 2nd fixup pass (it's slightly irregular since it's a list of TypeInfos — we can serialize it as a list of strings).
- Experimentally, using json.dumps() and then writing the string is faster than using json.dump() directly to the file. (Why? Maybe TextWrapper is slow?) Also, passing indent=2, sort_keys=2 makes it a little slower (though much more readable, so we may need this as an option for debugging).

Advanced ideas:

- When done parsing and type-checking a module, compare what *would* be written to the cache with what's already in the cache. If these match, cached data for modules

depending on this one need not be invalidated. (This might be important to handle the huge cycle in the server, or in general to handle changes to utility modules that are imported everywhere.)
- Parallelize processing (on a typical laptop this only offers a 2x speedup — that's something)
- (Very advanced:) keeping track of dependencies per class or function.

Abandoned ideas:
- Currently the serialize() and deserialize() methods have no state. We could perhaps improve performance by giving them an extra argument to pass state around. Such state could be used to hold a "memo" data structure used to reduce the size of the serialization and improve speed.
- It's possible that the serialization could be structured as a NodeVisitor and a TypeVisitor (the way the fixup code is structured). For the deserialization this won't work literally (because there's no instance to visit yet!), but a similar structure could work.
- Some classes have attributes that cannot be set through their constructors (these are always set by some later pass, e.g. semantic analysis). This slightly complicates the deserialization code. We might add additional arguments (with suitable default values) for these. But honestly the current solution isn't that bad.

Questions for Jukka

- Does the type checking pass actually change the contents of the symbol table, or is that settled at the end of semantic analysis? (Oddly I don't know and it hasn't mattered so far.)
- Why are we buffering errors again?
- Why is there no visit/accept protocol for TypeInfo?
- What's the meaning/use of mod_id in SymbolTableNode?
- Is there any logic to when to define __repr__ vs. __str__ ?(E.g. Type.__repr__ , but Node.__str__ .)

Potential cleanups (more technical debt)

- Clean up things based on Jukka's answers in the previous section.
- Add __repr__ or__str__ to some classes e.g. SymbolTableNode, maybe also SymbolTable (it's too large sometimes to print).
- Property vs. accessor method. E.g. filename, filename(). Also line, get_line().
- Some objects have a .name that's in quotes, e.g. '"foo of C"' -- maybe use a different attribute name for this? E.g. display_name.
- assert False vs. raise RuntimeError()?

Race condition

UPDATE: This is no longer relevant.

Background: Long ago, Google discovered a race condition in writing .pyc files: when multiple processes are all writing the .pyc file simultaneously a process that is reading it can see a valid header followed by an invalid data block (as it is overtaken by the second writer).

We might end up in a similar situation where two mypy processes both write x.data.json and x.meta.json and a third reads the data in the middle. Hopefully I've prevented the race by writing to x.{data,meta}.json.<nonce> and using atomic rename. But I've heard that on Windows (or ntfs mounted elsewhere) rename isn't atomic?

There are other possible mitigations; e.g. keep the none in the data file and give the nonce (better, the full data file) in the meta file. The downside of that is that you'd need to vacuum the cache directory occasionally to remove all the expired data files.

Anyway, it would be good if someone carefully went over the various failure scenarios and proved my theorem (that it's safe). Note that it would be safe if it occasionally causes us to re-parse files that haven't changed. But it wouldn't be safe if occasionally we'd read corrupt data (either malformed JSON or well-formed JSON that can't be deserialized).

NOTE: We could use os.replace() to avoid the Windows issue, except it doesn't exist in 3.2. See https://github.com/python/mypy/pull/1651 — we can merge that once we leave 3.2 behind.