The Opt and Optop API
*** Draft 0.9 ***
(Opt Version 1.6.11/Optop Version 1.6.3)

Drew McDermott

November 28, 2005

This document describes the Lisp implementation of the Opt language, described in [¡¿]. The implemented system provides a syntax checker, a problem solver, and a deductive engine. It is split into two subsytems, called Opt and Optop. Optop implements the planner; Opt implements everything else.

1 Installing the System

Opt is based on several other software packages:

1. YTools, a set of macros that enhance Common Lisp.
2. Nity, a type system that provides subtyping and union types. (The name Nity stands for "Nisp types," because it was originally developed for use with Nisp, a package that adds strong typing to Lisp. However, it is not yet used for that purpose.)
3. Langutils, which provides namespace management, plus error management for a syntax checker.
4. Lisplang, which implements a core of facilities that tend to be found in Lisp-like languages (such as Opt).

Optop is based on Opt plus two packages called YNisp and Screamer. Ynisp is a version of Nisp based on YTools. (YNisp is also referred to as Nisp 2.9 elsewhere. From now on I'll just refer to it as Nisp.) Screamer is a version of Siskind and McAllester's Screamer contraint-solving program, which has been slightly revised only to make it compatible with Allegro's "modern" mode, which is the first Lisp to get case-sensitivity right.[1]

Downloading and installation:

From web page:

Follow the instructions on
`http://cs-www.cs.yale.edu/homes/dvm/index.html#software`.
You should wind up with the following directory hierarchy:

[1]It does it the obvious way, by just making every symbol keep the case it was inputted in. The price you pay for the obvious solution is that you often have to modify the code of people who refer to the same symbol as `car` in one place and `CAR` in another.

```
nity-dir
   langutils
   nity
opt-dir
     lisplang
     opt
optop
ydecl
ytools
    ytload
```

These may all be below a single directory, or they can be located wherever you want, but keep the lowest layer together as shown. The directory `ydecl` defines Nisp.
From the Zoo or the Classes-V2 server (for CS470/570 students):

If you want to run on the Zoo using CMUCL or SBCL, you should be able to start Lisp, presumably under Slime, then execute `(yt-load` *module-name*`)`, where *module-name* is `:ytools`, `:lisplang`, or `:opt`. (I'm assuming your Lisp initialization file is as described in the YTools documentation.)

If you want to install Opt on your own machine, a subset of essentially the same directory structure as found on the web page can be reconstructed from the following gzipped tar files

```
/home/classes/cs470/lib
prognity.tar.gz
      opt.tar.gz
ytools
    ytools.tar.gz
```

(You should reinstall YTools if the tar.gz file has a later date than the last one you installed.)

Set up your Lisp initialization file as described in the YTools manual. To install Opt, start Lisp, then execute

```
(yt-install :opt)
```

To install Optop, execute

```
(yt-install :optop)
```

You load the two modules by executing `yt-load` instead of `yt-install`. If you prefer to install in stages, you can install YTools and YNisp "by hand," or let them be installed automatically in the course of installing Opt and Optop. [[Currently Optop is not available on the Zoo.]]

When Opt is started, the file `opt-init.lisp` in your home directory, if it exists, is loaded.

2 The Syntax Checker

To load one or more Opt files, checking their syntactic validity enroute, execute

(opt-load *—filespecs—*)

The format of the `filespecs` is given in the YTools manual (see the entry for `fload`); a typical example would look like

(opt-load %world/ aux-dom main-dom)

Here `%world/` is a *YTools logical pathname,* defined using `def-ytools-logical-pathname`. Although the details depend on the exact details of one's operating system and file system, a typical result of the call to `opt-load` above would be to load these files:

```
/planner/domains/aux-dom.opt
  /planner/domains/main-dom.opt
```

`opt-load` creates a file *name*`.chk` containing an annotated version of the input. Any syntax errors are flagged by <<double angle brackets>>.

3 Deduction

Opt contains a deductive system used by the Optop planner, but available for other purposes as well. To understand how it works, you first have to understand how variable bindings are handled in Opt, and how "situations" work.

As in most theorem provers and logic-programming systems, the concept of *substitution* is crucial. The data structure implementing substitutions is the `Bdgenv` ("binding environment"); it consists in essence of an association list between variables (symbols) and their values. Facilities are provided for substituting variables into expressions using Bdgenvs. However, the system is unusual in that expressions are represented as pairs consisting of an S-expression and a variable identifier, or *varid*, which is just an integer. The advantage of this scheme is that we can rename variables just by picking another varid.[2] A Bdgenv maps variable-varid pairs to expression-varid pairs.

Fortunately, for many purposes the complexities of this system can be ignored. Before explaining the simple and complex ways of calling the deduction engine, let me explain situations.

For planning applications, a key concept is that time can be modeled as a series of *situations*, each a snapshot of the world.[3] Situations are generated and used by the planning system, and will mostly be ignored until section 4. One can create a situation as described in the Opt manual, or you can use the "static fact situation" associated with every domain.

Here are the main entry points to the deductive system:

`(deduce-envs` p `&key varid env sit dom):`
Return a list of Bdgenvs for all deducible instances of p. Argument `varid` defaults to 1 (actually, the value of `dummy-id*`). Argument `env` is the initial bindings for the variables in p. It defaults to the empty environment (no bindings), which happens to be `nil`. The `dom` argument defaults to the value of `default-domain *`. The `sit` defaults to the value of `default-situation*`. If either is a symbol, it is replaced by its value in the deductive universe, which is *not* the same as its value as a variable. That is, if `dom` is a symbol, it is replaced by the domain with that name; if `sit` is a symbol, it is replaced by the situation with that name in `dom`. If `sit` is unspecified and there is no default, then it is taken to be the *static fact situation* for that domain, which contains exactly the facts declared in the domain.

`(deduce-instances` p `&key varid env sit dom return-envs):`
Returns two values. The first is the set of instances of p for all the Bdgenvs found. The second is the same as the value of `deduce-envs` unless the `return-envs` argument is `nil`; in which case the second valuel is `nil`. The `varid`, `env`, `sit`, and `dom` are just as for `deduce-instancees`.

`(deducible` p `&key varid env sit dom):`
Takes the same arguments as `deduce-envs`, and returns `t` if any Bdgenvs can be found, else `nil`.

`(deduce` q i e s d`)`: This is the key subroutine called by all the routines above; q is a query, i is a varid, e is a qvar environment, s is a situation, and d is a domain. All arguments are required; the situation and domain must be an actual situation and domain, not their names; and they must have been initialized since the last time they were redefined. It returns a list of Bdgenvs, just as for `deduce-envs`.

[2]Whether the advantages outweigh the disadvantages is not clear.

[3] — almost. Because Opt allows continuous processes to be modeled, a situation includes models of the rates of change of quantities affected by active processes.

Example: `(deduce '(loves john ?who) vid* (empty-env) sit17 the-dom)`

Within an Opt file, almost all variables are declared with quantifiers or other binding constructs, and question marks are almost always optional. Obviously, that is not the case for deductive goals in Lisp code. In addition, while Opt code in files is type-checked mercilessly, no one is checking that `(loves john ?who)` is well formed (e.g., that `loves` takes two arguments, or that `?who` takes as values expressions whose Opt type is what is required for the second argument). In future, a tighter coupling between expressions in code and the type checker will be introduced, but for now just be careful.

To get a domain, use `(try-domain-with-name` sym $create$`)`, which returns the domain with name sym if it exists. If it doesn't, and $create$ is true, then it will be created. Otherwise, false will be returned.

To get a situation, use `(sit-ini` *sitname dom*`)`, where *sitname* is a named situation defined using the Opt construct

```
 (define (situation sitname)
    ...)
```

`sit-ini` this will return false if the situation doesn't exist; if it does, it will be *initialized* if it hasn't been already.

Careful: For deduction and related operations (such as planning), domains and situations must be "indexified" and "up to date." Low-level inferential operations can't afford to make sure the domain and situation are correct, so it's important for programs to check and enforce the desired properties. For instance, Optop calls `plan-problem-initialize` to make sure that a named planning problem, its situation, and its domain are all kosher. It needs to be called only once, when planning begins. To initialize a situation, call `situation-initialize`, or, `sit-ini` at the read-eval-print loop.

If the variable `deduce-auto-initialize*` is true, then the functions `deduce-envs`, `deduce-instances`, and `deducible` initialize their situation and domain arguments every time they are called. This is convenient in pedagogical contexts, but not a good idea in general. Hence the default value of this variable is false.

[[Contexts, unification]]

[[Unimplemented syntax of Opt: Lambda expressions are properly type-checked, but the unifier screws up on them. Keyword arguments ditto.]]

Debugging logical axioms can be frustrating. If an axiom is buggy, then typically queries involving it silently fail (return no answers, or fewer than you expect). One approach is to try simpler queries, hoping that inspiration will strike if you narrow down the axioms where the fault lies, and then stare at them for a while. An alternative is to trace the deducer to see exactly where its behavior departs from what you expect. The procedure `(trace-deduce [:on | :off])` turns tracing on or off. (Without an argument, tracing is toggled.)

With tracing on, one sees the following output messages:

1. When a query is first seen, the message `"Creating` *qag*`"` is printed. The *qag* is a *Query-answer-gen*, a record of a goal. Even if a goal occurs more than once, it has a unique Query-answer-gen.

2. When a rule `(<-` c `(and` a_1 a_2 `...`a_n`)` is applied to a goal g, the deducer prints the message

   ```
   Reducing g
    to [depth] a1/i
    + sketch of a2 ... an
    ans-so-far = b
    super-query so far = sketch of c
   ```

   Here "sketch" means a representation of the "current instance" of a goal. All the system needs to know is the current subgoal produced by the rule. Its "sketches" retain some variables with the "wrong" varid.

Every time an instance is found to a_1, the bindings are substituted into $a_2, \ldots, a_n$, and a `Reducing...` message is printed out for a_2 (with a "sketch" of $a_3, \ldots, a_n$), up until an instance of a_n is found.

3. When answers to a goal have been found, the following message is printed:

   ```
   [depth] [ old | new | extra ] answers to qag
           to wit:   answers
   ```

   The tag `old` is used when *qag* occurs as the "reducer" of a goal after all its answers have already been found. The tag `new` means that "all" the answers to *qag* have been found. Further answer propagation can occur later, especially given the way negation and tabling are handled. At that point, the message will be printed with tag `extra`.

   When no answers have been found on the first pass through a goal, the same message is printed, but the "answers" are of the form `:none`. Another pseudo-answer is `:many`, which means that the deductive system was unable to return a list of answers to a goal. An example is a goal such as `(> ?n 5)`, which has an infinite number of answers. However, in general, an answer of `many` means only that the number of answers is unknown; it may be 0 or infinite.

4. When a goal of the form `(not` h`)` is encountered, the system looks for immediate answers (by matching assertions in the database) to goal h. If it finds any, then there will be no answers to `(not` h`)`; which is just another way of saying the goal fails. But if there are no answers to h, it is too early to conclude that the answer to `(not` h`)` is affirmative, because further answers to h might develop later. What the Opt deducer does is put `(not` h`)` on a list of goals to be considered later, printing the message

   ```
   Postponing exploration of consequences of answerless qag for (not h)
   ```

   The list is examined after all regular deductive activity has finished. Then it is sorted in order of decreasing depth, so that goals deeper in the tree are considered first. It prints the message "`Postponed nonmons =` *qagens*." after doing the sort; the first element of the list is then removed from the list and restarted.

The Opt manual (sect. 11, "Web Mode and Namespaces") describes the machinery built into Opt for using multiple namespaces for Web applications. Unless you are using Optop for planning on the Web (e.g., finding plans for dealing with web services), you should compile Opt and Optop in "no-namespace" mode. In this mode, things that look like symbols in formulas and actions are implemented as Lisp symbols, which makes debugging and other interactions with the system much simpler.

4 Planning

To work on a problem, load in the file defining it, using `opt-load`. If no expressions are flagged, then call

```
    (search-for-plan probname)
```

The `search-for-plan` function has these optional keyword arguments:

`:metric`	A fluent expression giving the objective function to be minimized by the planner. This is any numerical fluent. Useful examples are `total-time` and `total-steps`, but any fluent defined in the domain will do. (Default: no metric, which is about the same as `total-steps`.)
`:maxtime`	Any plan whose projected time is greater than this will be rejected. (Default: no ceiling.)
`:maxlength`	Any plan whose projected length is greater than this will be rejected. (Default: 1000.)
`:maxplans`	The maximum number of plans that will be considered before the planner gives up. (Default: 10000)
`:initial`	The initial situation to use. (Default: the initial situation given by the problem definition.)

On old-favorite domains like "Ferry" and "Blocks," this should just work. On new domains, it never works, because there are always bugs in the action definitions and actions. On the first try, the result is almost always a quick failure, with no useful output.

To diagnose the problem, you have to turn on various debugging switches, and use various tools to explore the problem. `reduction-dbg*`, if true, causes a message to be printed whenever Optop is considering how to achieve (i.e., "reduce") a goal. If the goal can be achieved immediately by doing a feasible action A, Optop will tell you what A is. Most goals generate subgoals, and Optop will tell you what they are.

When it appears that Optop is not finding a reduction that you believe to be appropriate, the problem usually requires looking inside the maximal matcher. (This is the module that matches conjunctions of goals against a situation and finds variable bindings that make as many conjuncts true as possible; the remaining conjuncts become subgoals.) If you set `maxmatch-dbg*` to true, then every time the maximal matcher is called it will tell you how it arrives at the matches it arrives at, and then pause to let you inspect the resulting output. The amount of output it produces can be intimidating, but it's basically a tree. Each node is a set of conjuncts remaining to match, and the children are the ways of extending the current set of bindings to a maximal match. Typically, if the remaining conjuncts consist of n literals `(and c`$_1$ `...c`$_n$`)`, with $n > 0$there are two basic subtrees: those in which c_1 is made true in the current situation by binding its free variables, and those in which c_1 remains false as free variables are bound. In the latter, c_1 goes on the *avoid list* as the recursive matching process proceeds. A commitment to avoid c_1 can fail to work out if every way of completing the bindings makes c_1 true. (The maximal matcher will never introduce a binding with the sole purpose of keeping a conjunct from becoming true.) The sucessful leaves of the maxmatch search tree are complete bindings that do not make any formula on the avoid list true. There are always many more leaves corresponding to the various kinds of failure.

Another class of branch on the tree the maximal matcher explores are those corresponding to attempts to *reduce* c_1. If there is a backward-chaining rule `(<-` c' a`)`, and c_1 and c' unify with substitution θ, then the maximal matcher will consider replacing c_1 with $\theta(a)$. It does this only if the predicate in c_1 is *stratified*, that is, cannot lead to an infinite chain of reductions.

Yet another way to handle c_1 is to make it a *constraint*. This typically occurs when c_1 is a mathematical formula that underdetermines the values of its variables. Such formulas are postponed as long as possible, and finally handed to the Siskind-McAllester Screamer package [¡¿] to be solved. Actually, they are not dealt with by the maximal matcher at all, but are returned as constraints on the match, and become part of *effort-specs*, the internal representation of goals. (The name has to do with the "effort" numbers computed by estimated-regression planners.) When an effort-spec is reduced, the constraints get passed down to the sub-effort-specs. They finally get tested when an effort-spec appears to be satisfiable in the present situation. The bindings that make it true must satisfy all the constraints.

It may take a bit of practice to understand what the max-matcher is doing, because it must deal with all the complexities above and a few others, but if you keep the tree idea in mind, and look for leaves that fail when they should succeed, it should be clear when the max-matcher is going wrong because the formalization of the domain is wrong.

To turn on `maxmatch-dbg*` only when a certain set of goals is encountered, put the goals on the list `maxmatch-trap*`.

It occurs surprisngly often that a domain formalization can be fixed just by debugging the finding of the first action in a plan. The reason is that Optop is reasoning back from the overall goal to the initial situation on every search step. Hence it is likely to foresee a problem at step 6 when it is trying to choose step 1. This is lucky for the domain debugger because it means you can turn a lot of debugging switches on and expect to see the problems emerge right away, not after several gigabytes of output. However, if it should turn out that steps 1, 2, ..., 5 are okay, and a problem only arises after a particular plan prefix of length 5 has been found, you should probably try breaking the plan search down into two phases: first find the situation obtaining after the first 5 steps, then solve the problem starting in that situation.

To find the situation obtaining after a sequence of actions, do

`(seq-project` s a d`)`

where s is a situation (the initial situation in the application at hand), a is a list of actions, and d is a domain. It will return three values, a situation, a float, and an integer. The first value is false if the sequence of actions is infeasible in s. Otherwise, it's the situation resulting from executing that sequence starting in s. The float is the elapsed time, and the integer is the number of steps executed.

Having gotten the situation, one can now use it as the `:initial` argument to `search-for-plan` in order to skip over the search required to find that situation.

Here are all the debugging switches:

`plan-dbg-amt*`	This is a small integer, between 0 and 2. Setting it to zero disables all debugging output. Setting it to 1 gives some output that reflects the overall plan search. Setting it to 2 generates detailed information for every search state.
`maxmatch-dbg*`	See above. Prints an exhaustive account of what happens during each maximal match.
`maxmatch-trap*`	A list of goals during whose reduction `maxmatch-dbg*` is set to true.
`reduction-dbg*`	Print a message when each goal literal is reduced during construction of the regression-match graph. (See above).