IST FP6-004779

PYPY

Researching a Highly Flexible and Modular Language Platform and Implementing it by Leveraging the Open Source Python Language and Community

STREP

IST Priority 2

D9.1 (Draft): Initial and Improved Constraint Satisfaction and Inference Engine, Logic Programming in Python

Due date of deliverable: August 2006

Actual Submission date: July 31, 2006

Start date of Project: 1st December 2005 Duration: 2 years

Lead Contractor of this WP: DFKI

Authors: Aurélien Campéas (Logilab), Anders Lehmann (DFKI)

Revision: Draft July 28, 2006

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level: PU (Public)

Revision History

Date	Name	Reason of Change
June 01	Anders Lehmann	created document
July 07	Aurélien Campéas	Logilab contributions: state of the art, logic and constraint programming
July 13	Anders Lehmann	Revisions to OWL part
July 27	Aurélien Campéas	Extensions and revisions (logilab part)
July 28	Stephan Busemann	First published draft, formal matters

Abstract

This report describes the concurrent logic and constraint programming approaches taken in PyPy. After reviewing the state of the art, logic programming in PyPy is presented. Python is extended by the concept of logic variable that is implemented in a special object space, along with a non-deterministic choice construct. Constraint Programming allows to specify and solve constraint satisfaction problems. Methods to define and extend solvers are described. A reasoning module is described, using the semantic web language OWL.

Contents

1 Executive Summary

This report discusses logic programming and constraint programming in PyPy with an application to an OWL reasoner. Most of the features considered thereafter are implemented in a special object space leveraging the flexibility of the PyPy infrastructure and reusing the coroutining machinery described in (D7.1). The OWL reasoner is using the constraint solver for making inferences on OWL ontologies.

The distinctive features added to PyPy in this work are subsumed under the concept of Concurrent Constraint and Logic Programming (abbreviated as CCLP in the remainder of this document).

This report reflects in part the current state of the development effort. Several important pieces are being constructed now and some of the existing components are not integrated together as they will be in the final version. Because of this, some heterogeneity lies in the way we account for our work.

We give several code snippets as well as statements on the state of the implementation of the various involved subcomponents.

2 Introduction

2.1 Purpose of this Document

This interim report drafts the concurrent logic and constraint programming approaches taken in PyPy with an application to an OWL reasoner.

2.2 Scope of this Document

This document describes the concept and implementation of a framework for concurrent constraint and logic programming in Python.

DFKI and Logilab started with an exploratory implementation of a constraint solver using the Computation Space abstraction in order to have a reference for the further work. Also logic variables, along with unification and dataflow thread synchronization was experimented with (using Python threads and condition variables).

The Logic Object Space was implemented by Strakt, Merlinux, Logilab and DFKI. This work has been completed for all deterministic logic operators, a specific interpreter-level scheduler, exception propagation semantics between threads bound by logic variables, and preliminary support of non-deterministic computation. The initial pure-python constraint solver has been partly ported to RPython, partly rewritten.

The current work is on the implementation and test of the computation space abstraction.

The OWL reasoner has been implemented by DFKI, as the application to LT World. What remains to be done there is an implementation of the query language SPARQL.

2.3 Related Documents

(Annex I) PyPy Description of Work

(D7.1) Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects

To try the examples in this document, one can invoke the PyPy interpreter with the Logic object space as:

```
python2.4 pypy/bin/py.py -o logic --usemodule=_stackless
```

Or alternatively compile and run a standalone interpreter:

```
cd pypy/translator/goal
python2.4 ./translate.py --gc=framework --no-backendopt targetlogicstandalone.py
```

This one however needs at least a couple gigabytes of RAM and a fast processor to yield a working interpreter after around three hours of compilation.

3 Concurrent Constraint and Logic Programming

This report discusses logic programming and constraint programming with an application to an OWL reasoner.

The first step of this work was to research state-of-the-art logic and constraint programming systems with an eye on concurrency, so as to benefit from the stackless features described in (D7.1). While the well-known language Prolog and industrial strength constraint solving algorithms like MAC were considered, we finally decided to go with the more recent paradigm named Concurrent Constraint and Logic Programing (CCLP), as it adheres best to our objectives, that is embedding logic and constraint programming features into an existing imperative, object-oriented and now concurrent language like Python.

3.1 Logic Programming in Prolog

While first touched by John MacCarthy in Programs with common sense, and approached in languages like Planner, logic programming was installed soundly with Prolog. While Prolog has incomplete logical semantics (due to its depth-first search strategy, the cut operator), it has found a vast range of successful applications -- from (natural) language processing to sophisticated user interface development, passing by knowledge management and expert systems.

The algorithmic basis of Prolog is first-order resolution, which combines backtracking and unification of variables.

3.2 Constraint Programming and MAC

The MAC (maintain arc consistency) algorithm for solving constraint satisfaction problems is currently considered the best in its category. One important vendor, ILOG, based its commercial solver on it with great success.

The algorithmic bases of MAC are backtracking and arc-consistency checking interleaved.

3.3 Limits of Prolog and MAC

As one can see, even if too briefly, a typical classic Prolog implementation and MAC have several fundamental common mechanisms; as such, modern Prolog implementations (such as GNU Prolog) have been augmented with finite domain constraint programming.

While Prolog and Mac have seen many uses both in the academic world and in the industry, they can be considered lacking with respect to the following aspects:

- the search strategy is hard-wired into the engine; a different search strategy would have to be implemented, not easily, at the program level;
- it is hard to make them efficiently concurrent, which is due to the use of trailing (stack disciplined, and data-type dependant memorization of previous bindings) as technique to remember the previous values taken before engaging in a choice point (they can be distributed, but without exploiting the inherent parallelism or concurrency of a distributed setting);

Specific to Prolog, as a programming language:

- it tries to satisfy both the need for search and algorithmic problems (those that have known efficient algorithms); the resulting compromise is insufficient on both sides;
- it does not support higher-order programing; in other words, it is not possible to mix functional style, logic style and even imperative style;

Constraints in MAC solver are not restricted to Horn Clause style; their expression is bound to whatever language the implementor uses (in the case of ILOG solver, for instance, it is C++).

3.4 From backtracking engines to CCLP

In short, CCLP draws from research on parallelisation and modularisation of logic programming, integration with stateful computing, integration with constraint programming. It really emerged after almost two decades of research on ways to extract the anticipated inherent parallelism of a logic language.

Concurrent logic languages had a rich history with experiments like Parlog, Concurrent Prolog, Guarded Horn Clauses.

The most significant step was the *ask* and *tell* operators brought by Saraswat. This work reconciliated parallel logic languages with constraint programming. The Andorra Kernel Language (AKL), that embodied these ideas, added state (with ports), and encapsulated search.

It appeared that in CCLP, the logic or constraint based mechanism could be seen as a concurrent process calculus. Ask is indeed a synchronization primitive, and shared variables serve as communication devices between processes. Concurrent programming in this style alleviates the imperative style riddled with low-level and error-prone locks, mutexes and semaphores. It also has clean logic semantics.

3.5 From CCLP to Oz to Python

Expanding on AKL, the Oz language immerges CCLP into an already functional and also stateful programing framework, which is truly multi-paradigm. In this light, it was decided to meet the current Oz developper group in Louvain-la-Neuve to get insights on the feasibility of extending Python with the powerful abstractions present in Oz. We had to understand how one single language can bring so many seemingly different techniques into a coherent whole.

The meeting was fruitful and quite enthusiastic (some people working on Oz happen to be happy Python users). Some skepticism was sched with respect to Python's informal, and in fact very complex semantics, and how difficult it could be to add something radically different like logic programming as first class citizen. It was stated that the clean, formalized and well-understood, Onion-structured semantic layers of Oz was one key to the harmony of the whole.

The most important lesson learned however is that, at the end, the programmer that uses such a rich language is responsible for the proper encapsulation of different styles in a modular way. With power comes responsibility. This bodes well with the Python philosophy of considering the programmer a responsible grown-up.

We do not know yet, of course, if our extension to Python will prove practical in the long run; it could fail either because of language issues (we can see now that the non-declaration of variables in Python is a concern) or because these styles are not well understood by most programmers, or because we provide something too slow ...

The most important (and novel) concept imported from Oz is that of Computation Space. We will not, in the interim report, define it precisely; it is nonetheless exposed in the sections that deal with logic programming and constraint programming in PyPy.

3.6 Constraint Handling Rules

Constraint Handling Rules are a special purpose constraint definition language for user-defined constraints. They complements CCLP languages. They bring automatic simplification of constraints (preserving logical equivalence) and propagation (add redundant equivalent constraints, may cause further simplification) over these. We won't have this in WP09. But in some sense, it is not specially needed; Python will provide the necessary support to write constraints, maybe with the help of a more powerful lambda construct that is under discussion.

Operations on constraints can always be retrofitted later into the framework.

3.7 Status of the implementation

This work was done by Strakt, Merlinux and Logilab.

Anders Lehmann and Aurélien Campéas wrote a first pure-Python emulation of logic variables, limited computaiton space for the constraint solver, so as to discover the shape of the problems.

At the end of the meeting with the Oz team in Belgium, Samuele Pedroni and Carl Friedrich Bolz wrote the logic variable filtering code exploiting PyPy object spaces flexibility, along with some threading capabilities to illustrate the dataflow behaviour.

This work has been completed by all deterministic Logic operators, a specific interpreter-level scheduler, exception propagation semantics between threads bound by logic variables, and preliminary support of non-deterministic computation, by Aurélien. The initial pure-python constraint solver has been partly ported to RPython, partly rewritten (with the help of Anders and Alexandre).

In some sense, the hardest part is yet to come. The exact semantics of Oz computation spaces have some weird corners (with respect to newspace/0 and merge/0) and we don't know yet up to which point we will respect these. Something likely slightly simpler will probably emerge. But the main difficulty now lies in the development cycle, which involves compiling a whole PyPy interpreter, for tests that involve cloning -- and since the search and non-deterministic features rely on it, it concerns a large area -- , and doing so takes something like three hours, without backend optimizations, on a modern fast computer. But, after all, in the old days programmers used to write their programs down, have them transformed into punch cards, be scheduled, run, and the results sent back, possibly by mail, after even more wait...

4 Logic Programming in PyPy

The first addition to standard Python is the concept of logic variable. Logic variables serve two main, distinct purposes:

- they are a building block for a declarative style of programming, in which a program has obvious logical semantics (by contrast with a random Python program, whose logical semantics can become obscure extremely quickly);
- they dynamically drive the scheduling of threads in a multi-threaded system, allowing lock-free concurrent programming (lock-free but not necessarily block-free, one symptom of a badly written concurrent program using logic variables being that it hangs indefinitely ...).

In this section we describe the design, the usage and the implementation of logic variables in PyPy.

4.1 Basics, elements serving deterministic Logic programming

Logic variables are similar to Python variables in the following sense: they map names to values in a defined scope. But unlike normal Python variables, they have two states: free and bound. A bound logic variable is indistinguishable from a normal Python value, which it wraps. A free variable can only be bound once (it is also said to be a single-assignment variable). Due to the extended PyPy still using Python variables, it is good practice to denote logic variables with a beginning capital letter, so as to avoid confusion.

The following snippet creates a new logic variable and asserts its state:

```
X = newvar()
assert is_free(X)
```

Logic variables are bound thusly:

```
bind(X, 42)
assert is_bound(X)
assert X / 2 == 21
```

The single-assignment property is easily checked:

```
bind(X, 'hello') # would raise a FailureException
bind(X, 42)      # is tolerated (it does not break monotonicity)
```

In the current state of the logic object space, a generic Exception will be raised. It is quite obvious from this that logic variables are really objects acting as boxes for Python values. No syntactic extension to Python is provided yet to lessen this inconvenience.

The bind operator is low-level. The more general operation that binds a logic variable is known as unification. Unify is an operator that takes two arbitrary data structures and tries to assert their equalness, much in the sense of the == operator, but with one important twist: unify mutates the state of the involved logic variables.

Unifying structures devoid of logic variables is equivalent to an assertion about their equalness:

```
unify([1, 2], [1, 2])
```

The following basic example shows logic variables embedded into dictionaries:

```
Z, W = newvar(), newvar()
unify({'a': 42, 'b': Z},
      {'a':  Z, 'b': W})
assert Z == W == 42
```

Unifying one unbound variable with some value (a) means assigning the value to the variable (which then satisfies equalness), unifying two unbound variables (b) aliases them (they are constrained to reference the same - future - value).

Assignment or aliasing of variables is provided underneath by the 'bind' operator.

We now turn to an example involving custom data types:

```
class Foo(object):
    def __init__(self, a):
        self.a = a
        self.b = newvar()

f1 = Foo(newvar())
f2 = Foo(42)
unify(f1, f2)
assert f1.a == f2.a == 42    # assert (a)
assert alias_of(f1.b, f2.b)  # assert (b)
unify(f2.b, 'foo')
assert f1.b == f2.b == 'foo' # (b) is entailed indeed
```

Finally, note that by stating:

```
X = newvar()
X = 42
```

the logic variable referred to by X is actually replaced by the value 42. This is not equivalent to bind(X, 42).

What is provided here is sufficient to achieve deterministic logic programming: we need an additional operator to declare non deterministic choice points in order to get the expressive power of pure Prolog (understood as Prolog without cut).

4.2 Logic and Search in non-deterministic programs

4.2.1 Choice points

A logic program in the extended PyPy will be any Python program making use of the 'choice' operator. The program's entry point must be a zero arity procedure, and must be given to a solver.

The Python grammar needs only be extended like this to support the choice operator:

```
choice_stmt: 'choice:' suite ('or:' suite)+
```

For instance:

```
def foo():
    choice:
        return bar()
    or:
        from math import sqrt
        return sqrt(5)

def bar():
   choice: return -1 or: return 2

def entry_point():
    a = foo()
    if a < 2:
       fail()
    return a
```

What should happen when we encounter a choice ? One of the choice points ought to be chosen 'non-deterministically'. That means that the decision to choose does not belong to the program itself (we call this *don't care non-determinism*) but to some external entity having interest in the program outcomes.

Such an entity is typically a 'solver' exploring the space of the program outcomes. The solver can use a variety of search strategies, for instance depth-first, breadth-first, discrepancy search, best-search, A* ... It can provide just one solution or some or all of them. It can be completely automatic or end-user driven (like the Oz explorer).

Thus the program and the methods to extract information from it are truly independant, contrarily to Prolog programs which are hard-wired for depth-first exploration.

4.2.2 Choice, continued

The above program contains two choice points. If given to a depth first search solver (this is the recomended option if one wants to execute a program with logic semantics, the more fancy solvers are interesting in the case of constraint solving), it will produce three spaces: the two first spaces will be failed and thus discarded, the third will be a solution space and ready to be later merged. We leave the semantics of space merging for another report. An illustration:

```
entry_point -> foo : choice
                      /\
                     /  \
                    /    sqrt(5)  (solution)
           bar : choice
                  /\
                 /  \
                /    \
              -1      2
          (failure)(solution)
```

4.2.3 Encapsulated search and Computation Spaces

To allow this de-coupling between program execution and search strategy, the computation space concept comes handily. Basically, choices are made, and program branches are taken, in speculative computations which are embedded, or encapsulated in the so-called computation space, and thus do not affect the whole program, nor each other, until some outcome (failure, values, updated speculative world) is considered and eventually merged back into the parent world (which could be the 'top-level' space, aka normal Python world, or another speculative space).

For this we need another method of spaces:

```
choose/1
```

The choice operator can be written straightforwardly in terms of choose:

```
def foo():
    choice = choose(3)
    if choice == 1:
        return 1
    elif choice == 2:
        from math import sqrt
        return sqrt(5)
    else: # choice == 3
        return 3
```

Choose is more general than choice since the number of branches can be determined at runtime. Conversely, choice is a special case of choose where the number of branches is statically determined. It is thus possible to provide syntactic sugar for it.

It is important to see the relationship between ask(), choose() and commit(). Ask and commit are used by the search engine running in the top-level space, in its own thread, as follows :

Ask() waits for the space to be 'stable', which means that the program running inside is blocked on a choose() call. It returns precisely the value provided by choose, thus notifying the search engine about the number of available choice points. The search engine can then, depending on its strategy, commit() the space to one of its branches, thus unblocking the choose() call and giving a return value which represents the branch to be taken. The program encapsulated in the space can thus run until:

- it encounters another choice point,
- it fails (by way of explicitly calling the fail() operator, or letting an unhandled exception bubble up to its entry point),
- properly terminating, thus being a candidate 'satisfying' solution to the problem it stands for.

Commit and choose allow a disciplined communication between the world of the search engine (the normal Python world) and the speculative computation embedded in the space. Of course the search and the embedded computation run in different threads. We need at least one thread for the search engine and one per space for this scheme to work.

4.3 Threads and dataflow synchronization

When a piece of code tries to access a free logic variable, the thread in which it runs is blocked (suspended) until the variable becomes bound. This behaviour is known as "dataflow synchronization" and mimics exactly the dataflow variables from Oz. With respect to behaviour under concurrency conditions, logic variables come with two operators:

- wait/1 (defined on a variable) suspends the current thread until the variable is bound, it returns the value otherwise,
- wait_needed/1 (on a variable also) suspends the current thread until the variable has received a wait message. It has to be used explicitly, typically by a producer thread that wants to lazily produce data.

In this context, binding a variable to a value will make runnable all threads previously blocked on this variable.

Using the "future" builtin, which spawns a coroutine and applies the second to nth arguments to its first argument, we show how to implement a producer/consumer scheme:

```
def generate(n, limit):
    if n < limit:
        return (n, generate(n + 1, limit))
    return None

def sum(L, a):
    Head, Tail = newvar(), newvar()
    unify(L, (Head, Tail))
    if Tail != None:
        return sum(Tail, Head + a)
    return a + Head

X = newvar()
S = newvar()

unify(S, future(sum, X, 0))
unify(X, future(generate, 0, 10))

assert S == 45
```

Note that this eagerly generates all elements before the first of them is consumed. Wait_needed helps us write a lazy version of the generator. But the consumer will be responsible of the termination and must thus be adapted, too:

```
def lgenerate(n, L):
    """lazy version of generate"""
    wait_needed(L)
    Tail = newvar()
    bind(L, (n, Tail))
    lgenerate(n+1, Tail)

def lsum(L, a, limit):
    """this summer controls the generator"""
```

```
        if limit > 0:
            Head, Tail = newvar(), newvar()
            wait(L)
            unify(L, (Head, Tail))
            return lsum(Tail, a+Head, limit-1)
        else:
            return a

Y = newvar()
T = newvar()

future(lgenerate, 0, Y)
unify(T, future(lsum, Y, 0, 10))

wait(T)
assert T == 45
```

The 'future' statement immediately returns a restricted logic variable, which is bound only when the thunk it got as first argument has returned, yielding an actual return value. It is restricted in the sense that the thread which asks for a future can not assign it. Futures are implemented as a specialization of logic variables.

Note that in the current state of PyPy, we deal with coroutines, not threads (thus we can't rely on preemptive scheduling to lessen the problem with the eager consummer/producer program).

Finally, we observe that the bind, wait pair of operations is quite similar to the asynchronous send, blocking receive primitives commonly used in message-passing concurrency (like in Erlang or just plain Stackless).

4.4 Table of Operators

Logic variables support the following operators (listed with their arity):

Predicates

is_free/1 any -> bool
is_bound/1 any -> bool
alias_of/2 logic variables -> bool

Variable Creation

newvar/0 nothing -> logic variable

Mutators

bind/2 logic variable, any -> None
unify/2 any, any -> None

Dataflow synchronization (blocking operations)

wait/1 value -> value
wait_needed/1 logic variable -> logic variable
choose/1 int -> None

4.5 Syntactic and semantic aspects

The most glaring feature, syntactically-wise, of the logic object space, is the relative transparency of the logic variables once they have been instantiated. In some sense, we fill that they are too transparent. Especially since no mechanism currently permits to lexically distinguish normal Python value from unbound logic variables, we feel the need to adopt the convention that the later be capitalized. Some ideas have been floating around for some syntactic shortcuts, like the following (there are actually more):

```
def foo_let(x, y):
    """requires adding a new node in the AST
    the scoping rules need to be clarified, but not necessarily bent
    in the direction of Python (leaking behaviour)
    """
    let X, Y:
        unify(X, x)
        unify(Y, y)
        xx = X
        return X+Y
    print X #-> raises name error
    print xx # no NameError (?)

def foo_logic_kw(x,y):
    """needs a new keyword
    the logic keyword does not introduce a new scope
    (compare with let above)"""
    logic X,Y
    unify(X, x)
    unify(Y, y)
    return X+Y

@logic('X','Y')
def foo_decorator(x,y):
    """can this work at all ? The decorator would add local variables
    to the function with the provided names, initialized as logic
    variable"""
    unify(X, x)
    unify(Y, y)
    return X+Y

def foo_visual_basic(x,y):
    """style from an industrial-strength programming
    language used by millions"""
    Dim X, Y as LogicalVariables
    unify(X, x)
    unify(Y, y)
    return X+Y
```

One expects from a new syntactic construct that it be *pythonic* (the fourth, lesser known, theological virtue), that it allows some lexical tracing by human programmers as well as compilers or code checkers.

The problem with = being clearly dinstinguished from bind or unify is still open. Writing X = 42 instead of bind(X, 42) just throws away the logic variable, and some thread blocked on it would wait forever.

4.6 Implementation aspects

As said supra, the distinctive features of WP09 are grouped in a special object space, the Logic object space. This was motivated by the implementation of Logic variables. All other functionality could be provided as standard PyPy builtins embedded in some modules.

4.6.1 Logic variables

Logic variables pose indeed a problem with respect to normal Python semantics. While they can be emulated in pure Python (this was done during the first exploration steps), they syntactically obscure the code and would impose a painful discipline to the programmer. The idea then, is to make them as transparent as possible, in short: avoid at all cost the pain to riddle a program with explicit calls to wait/1.

This is achieved by wrapping all function calls into something that effectively puts wait/1 statements around every passed argument. While the runtime price for such filtering is expected to be important, the implementation effort is quite minimal. An effort to efficiently integrate Logic variables could be made later, but we expect some hurdles; Python was just not designed with Logic variables in mind.

A specific type W_Var is introduced at the interpreter level. This type is never accessible as such from the application level. It is only provided as an anchor for the new builtins. Thus, bind and unify, for instance, dispatch on their arguments being normal Python values or logic variables, and can hence provide adequate behaviour. We used the interpreter-level implementation of multi-methods already used in other parts of PyPy to implement many of the builtins.

4.6.2 Non determinism

The non-deterministic part of this work, which is still under early development at the time of this writing, peruses the (recently implemented) cloning facility of PyPy's coroutine infrastructure (WP07.4). It was shown by the Oz group that cloning was not much less efficient than trailing (as used classically in Logic languages implementations); it is even competitive with trailing in the case of constraint solving if cloning is mixed with recomputation steps, so as to balance the CPU/Memory consumption trade-off. Of course, cloning inherently opens the ability to distribute the load amongst several CPUs (yielding in principle linear performance increases) whereas trailing does not support parallelism at all.

The coroutines cloning facility is used thus: at each choice point, the solver can decide to take a snapshot of the current state of the computation so as to explore the disjoint computation spaces opened by the choices. The computation space concept wraps a constraint store and a set of running coroutines. The cloning of a space is then nothing more complicated than cloning the store (which present no difficulty) and the set of coroutines.

There is a downside to this currently: the coroutine cloning facility is only available when using the GC framework. As in the standard implementation of the Oz language (the Mozart platform), the actual copying routines are part of the GC mechanism. In PyPy, a mark-and-sweep and a semi-space copying collector, implemented as translation aspects, are available (while in Mozart, the standard built-in copying collector is 'reused' to effectively copy computation spaces state). So it is impossible to run a program perusing cloning on interpreted PyPY or on an interpreter compiled with the Böhm-Weiser collector (for C backends) or builtin collectors of higher-level target languages (Lisp, CLI). It also means that the development cycle (especially the testing parts) is quite slowed down since one of the steps must be a compilation of a whole PyPy. Since we typically want to test application level code (tests for cloning in pure RPython already exists and do not entail a whole recompilation), there is no other solution than recompiling the whole thing. This takes around three hours on average on a fast machine.

5 Constraint Programming with PyPy

The logic object space comes with a modular, extensible constraint solving engine. While regular search strategies such as depth-first or breadth-first search are provided, you can write better, specialized strategies (an example would be best-first search). This section describes how to use the solver to specify and get the solutions of a constraint satisfaction problem, and then highlights how to write a new solver.

Let us mention some possible kinds of solvers:

- basic : takes no argument, enumerates all solutions;
- lazy search (this one is actually the default one since in Python writing a generator is dead easy);
- general purpose (variable recomputation distance -- to tune tradoff of time and memory use), asynchronous kill to stop infinite search;
- parallel search: will spread the search on a set of machines, yielding linear speedups;
- explorer search, using a concurrent GUI engine to help a user drive the search;
- orthogonally to all the previous, it is of course possible to devise any suitable strategy (depth-first search, best-search, A* ...).

5.1 Using the constraint engine

5.1.1 Specification of a problem

A constraint satisfaction problem (CSP) is defined by a triple (X, D, C) where X is a set of finite domain variables, D the set of domains associated with the variables in X, and C the set of constraints, or relations, that bind together the variables of X.

Note that the constraint variables are NOT logic variables. Not yet anyway.

So we basically need a way to declare variables, their domains and relations; and something to hold these together. The later is what we call a "computation space". The notion of computation space is broad enough to encompass constraint and logic programming, but we use it there only as a box that holds the elements of our constraint satisfaction problem.

A problem is a one-argument procedure defined as follows:

```
def simple_problem(cs):
    cs.var('x', FiniteDomain(['spam', 'egg', 'ham']))
    cs.var('y', FiniteDomain([3, 4, 5]))
```

As one can immediately see, the space is made explicit there. It will be entirely implicit in the final version (like in the examples on non-deterministic logic programming). Basically all 'inside' space operations take an implicit space as argument.

This snippet defines a couple of variables and their domains, on the 'cs' argument which is indeed a computation space. Note that we didn't take a reference of the created variables. We can query the space to get these back if needed, and then complete the definition of our problem. Our problem, continued:

```
... x = cs.find_var('x')
    y = cs.find_var('y')
    cs.tell(make_expression([x,y], 'len(x) == y'))

    return x, y
```

We must be careful to return the set of variables whose candidate values we are interested in. The rest should be sufficiently self-describing...

5.1.2 Getting solutions

Now to get and print solutions out of this, we must:

```
import solver
cs = newspace()
cs.define_problem(simple_problem)

for sol in solver.solve(cs):
    print sol
```

The builtin solve function is a generator, producing the solutions as soon as they are requested.

5.2 Table of Operators

Note that below, "variable/expression designators" really are strings.

Space creation

newspace/0

Finite domain creation

FiniteDomain/1 list of any -> FiniteDomain

Expressions

make_expression/2 list of var. designators, expression designator -> Expression
AllDistinct/1 list of var. designators -> Expression

Space methods (really constraint store methods)

var/2 var. designator, FiniteDomain -> constraint variable instance
find_var/1 var. designator -> constraint variable instance
tell/1 Expression -> None
define_problem/1 procedure (space -> tuple of constraint variables) -> None

5.3 Extending the search engine

5.3.1 Writing a solver

Here we show how the additional builtin primitives allow you to write, in pure Python, a very basic solver that will search depth-first and return the first found solution.

As we've seen, a CSP is encapsulated into a so-called "computation space". The space object has additional methods that allow the solver implementor to drive the search. First, let us see some code driving a binary depth-first search:

```
def first_solution_dfs(space):
    status = space.ask()
    if status == 0:
        return None
    elif status == 1:
        return space.merge()
    else:
        new_space = space.clone()
        space.commit(1)
        outcome = first_solution_dfs(space)
        if outcome is None:
            new_space.commit(2)
            outcome = first_solution_dfs(new_space)
        return outcome
```

This recursive solver takes a space as its argument, and returns the first solution or None. In such code, the space will remain explicit as it is a first class entity manipulated from 'the outside'. Let us examine it piece by piece and discover the basics of the solver protocol.

The first thing to do is "asking" the space about its status. This may force the "inside" of the space to check that the values of the domains are compatible with the constraints. Every inconsistent value is removed from the variable domains. This phase is called "constraint propagation". It is crucial because it prunes as much as possible of the search space. Then, the call to ask returns a non-negative integer value which we call the space status; at this point, all (possibly concurrent) computations happening inside the space are terminated.

Depending on the status value, either:

- the space is failed (status == 0), which means that there is no combination of values of the finite domains that can satisfy the constraints,
- one solution has been found (status == 1): there is exactly one valuation of the variables that satisfy the constraints,
- several branches of the search space can be taken (status represents the exact number of available alternatives, or branches).

Now, we have written this toy solver as if there could be a maximum of two alternatives. This assumption holds for the simple_problem we defined above, where a binary "distributor" (see below for an explanation of this term) has been chosen automatically for us, but not in the general case. See the sources for a more general-purpose solver and more involved sample problems (please not that at the time of this writing, only conference_scheduling is up to date with the current API).

In line 8, we take a clone of the space; nothing is shared between space and newspace (the clone). We now have two identical versions of the space that we got as parameter. This will allow us to explore the two alternatives. This step is done, line 9 and 13, with the call to commit, each time with a different integer value representing the branch to be taken. The rest should be sufficiently self-describing.

This shows the two important space methods used by a search engine: ask, which waits for the stability of the space and informs the solver of its status, and commit, which tells a space which road to take in case of a choice point.

5.3.2 Using distributors

A distributor is a non-deterministic program running in a computation space. In the final version of this work, it will be any Python program augmented with calls to non-deterministic choice points (choose, choice).

In the case of a CSP, the distributor is a simple piece of code, which works only after the propagation phase has reached a fixpoint. Its policy will determine the fanout, or branching factor, of the current computation space (or node in the abstract search space).

Here are two examples of distribution strategies:

- take the variable with the smallest domain, and remove exactly one value from its domain; thus we always get two branches: one with the value removed, the other with only this value remaining,
- take a variable with a small domain, and keep only one value in the domain for each branch (in other words, we "instantiate" the variable); this makes for a branching factor equal to the size of the domain of the variable.

There are a great many ways to distribute... Some of them perform better, depending on the characteristics of the problem to be solved. But there is no single preferred distribution strategy. Note that the second strategy given as example there is what is used (and hardwired) in the MAC algorithm.

Currently we have two builtin distributors in the logic object space (note that they are different from the two first above):

- NaiveDistributor, which distributes domains by splitting the smallest domain in 2 new domains; the first new domain has a size of one, and the second has all the other values,
- SplitDistributor, which distributes domains by splitting the smallest domain in N equal parts (or as equal as possible). If N is 0, then the smallest domain is split in domains of size 1; a special case of this, DichotomyDistributor, for which N == 2, is also provided and is the default one.

To explicitly specify a distributor for a constraint problem, you need to state, in the procedure that defines the problem:

```
cs.set_distributor(NaiveDistributor())
```

It is currently not possible to write distributors in pure Python; this is scheduled for PyPy version 1.

5.4 Remaining Space Operators

For solver writers

ask/0	nothing -> a positive integer i
commit/1	integer in (1, i) -> None
merge/0	nothing -> list of values (solution)
clone/0	space -> space

6 Reasoning in OWL

OWL -- the Web ontology Language -- is a knowledge representation language with well-defined formal properties. It is designed for applications that need to process meaning rather than media. This way, OWL can form one of the corner stones of the Semantic Web, which is expected to allow application-independent, simultaneous access to a multitude of data, and to relationships between them.

The Semantic Web is a W3C initiative to provide means to make standards for providing data descriptions including machine readable semantics. This would allow computers to reason about the data.

The following information about the contents of the Semantic Web technologies is drawn from Wikipedia.

7 Semantic Web

The Semantic Web is a W3C initiative to provide means to make standards for providing data descriptions including machine readable semantics. This would allow computers to reason about the data.

The Wikipedia states the following about the contents of the Semantic Web technologies:

"" " The Semantic Web comprises the standards and tools of XML, XML Schema, RDF, RDF Schema and OWL. The OWL Web Ontology Language Overview describes the function and relationship of each of these components of the Semantic Web:

- XML provides a surface syntax for structured documents, but imposes no semantic constraints on the meaning of these documents.
- XML Schema is a language for restricting the structure of XML documents.
- RDF is a simple data model for referring to objects ("resources") and how they are related. An RDF-based model can be represented in XML syntax.
- RDF Schema is a vocabulary for describing properties and classes of RDF resources, with a semantics for generalization-hierarchies of such properties and classes.
- OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.

In PyPy, an OWL reasoner was developed using the constraint solving package from Logilab and the Python RDF parser from rdflib.

This approach is different from other approaches such as Pellet and razerpro which uses the tableaux algorithm.

The idea is to convert the OWL ontology into a constraint problem, solvable by a constraint solver. To that end the OWL ontology is parsed into RDF triples, and the triples are transformed into variables, domains and constraints utilising the semantics of OWL. The variables of the constraint problem are then the OWL classes, OWL properties and OWL individuals.

For each of the variables a domain of possible values are given. If the ontology does not explicitly supply values for a variable (ie. using a owl:oneOf predicate) the collection of all Individuals (the extension of owl:Thing). This collection is build during the parsing of the rdf triples. For each triple the semantics of the predicate is used to add constraints on the variables (the subject and object). When all the triples in the Ontology ahs been passed the Ontology can be checked for consistency, by invoking the constraints. The constaints are used to narrow the domains to the smallest number of items that satisfies the constraints. If all the domains contain at least one element the Ontology is consistent. If on the other hand a domain becomes empty or a fixed size domain (created by a oneOf predicate) is being reduced the Ontology is not consistent and leads to a failure.

When a consistent Ontology has been converted in to a constraint problem, search can be performed by adding new constraints.

Note: By June 2006 the OWL reasoner is mostly done. The missing parts are bugfixes to support reasoning on schemas and support for SPARQL queries.

8 Using the Results for Semantic Web Applications

Note: the following is a plan that is subject to approval by the EU.

A promising opportunity of applying the above results opened up when the developers of LT World, the world's largest information portal on Speech and Language Technologies. LT World has been developed by, and is maintained at, DFKI. The platform informs about scientists, companies, institutions, projects, products, patents etc., and news related to speech and language technologies. The conceptual basis is entirely realized in OWL.

Applying PyPy results to LT World will create a user interface capable of intelligent search and of answering typical user questions.

This application is both challenging and rewarding. If successful, a proof of concept for scalability and usability is delivered.

9 Conclusion

PyPy includes a logic and constraint programming framework, which is still a work-in-progress. It is sufficiently advanced however to allow one OWL reasonner to be written on top of it.

10 Glossary of Abbreviations, References

References

(D07.4) *Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects*, PyPy EU-Report, 2006