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 Constraint Satisfaction and Inference Engine, Logic Programming in Python

Due date of deliverable: December 2006

Actual Submission date: February 28, 2007

Start date of Project: 1st December 2004 Duration: 2 years

Lead Contractor of this WP: DFKI

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

Revision: Interim

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, OWL and Semantic Web
July 07	Aurélien Campéas	Logilab contributions: state of the art, logic and constraint programming
July 13	Anders Lehmann	Revisions to OWL part (DFKI)
July 27	Aurélien Campéas	Extensions and revisions (logilab part)
July 28	Stephan Busemann	First published draft, formal matters
Dec 14	Aurélien Campéas	Towards a polished version
Feb 27	Stephan Busemann	Interim version to be sent to EC

Abstract

This report describes the concurrent logic and constraint programming approaches taken in PyPy. Following some historical background, 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 one to specify and solve constraint satisfaction problems. Methods to define and extend solvers are described. A reasoning module is also described, using the semantic web language OWL.

Contents

1 Executive Summary

This report discusses logic programming and constraint programming over discrete domains in PyPy with an application to an OWL reasoner. Most of the features considered thereafter are implemented in a special object space. This demonstrates the flexibility of the PyPy infrastructure, the coroutine machinery and parts of the garbage collector framework, as described in [D7.1]. The OWL reasoner uses 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).

Several code snippets are presented, along with discussion on the state of the implementation of the various sub-components involved.

2 Introduction

2.1 Purpose of this Document

This 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. Also logic variables, along with unification and dataflow thread synchronization were 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.

There is on going work on the implementation and test of the computation space abstraction.

The OWL reasoner has been implemented by DFKI together with an interface to the query language SPARQL [SPQL]. An application to LT World is being realized that relies on SPARQL.

2.3 Related Documents

Support for Massive Parallelism and Publish about Optimization results, Practical Usages and Approaches for Translation Aspects [D7.1].

2.4 Getting Started

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
```

Or alternatively compile and run a standalone interpreter:

```
cd pypy/translator/goal
python2.4 ./translate.py targetlogicstandalone.py
```

3 Concurrent Constraint and Logic Programming

3.1 Motivation

The first step of this work was to research modern logic and constraint programming systems supporting concurrency.

The motivation was to add to Python a set of primitives enabling logic and constraint programming. Indeed, using an external library written in an efficient, compiled language is the only option currently available to the Python programmer, if she wants raw performance. And, to our knowledge, no binding for such libraries (such as gecode or choco) exists today.

On the other hand, some pure Python solutions can be found. Notably, logilab's constraint package (which implements Oz's algorithms [ECP94]) and Gustavo Niemeyer's constraint solver (implementing MAC [MAC97]) are examples. We know for sure that logilab's package is used daily in production. However, from personal experience it is known that for some moderately sized problems, performance is inadequate.

People with whom we discussed about the availability of logic or constraint tools in Python regularly mentioned other tools, such as Pylog[1], Metalog[2], Pychinko[3], CWM[4]. The first (and only one concerned with logic programming) is quite unsatisfying in that there can be only a limited set of data structures common to Python and SWI-Prolog. Metalog is biased towards semantic web applications, and also since it is written in pure Python, its engine can run in to the performance problems of the aforementioned constraint packages. The others are concerned with production rule systems, which while useful in their own right and also in conjunction with logic/constraint systems, do not provide the same services.

There was clearly some unsatisfied demand there. Thus it made sense to incorporate into PyPy one solution that would comply with many non-obvious requirements:

1. build logic and constraint programming on top of some common infrastructure,
2. have the logic/constraint programming styles be maximally integrated with the imperative object-oriented language that is Python,
3. support concurrency (because interactive programs need it), multi-processing and distribution (because exploration of vast search spaces is a processing power hungry activity),
4. be flexible and extensible,
5. be orders of magnitude faster than a pure Python implementation.

That might well be completely impossible within the current CPython implementation, which many developers deem inflexible and hard to adapt. We still doubt it could be done like we did with other Python implementations, like Jython or IronPython. Indeed, we exploited some features quite unique to PyPy in order to achieve a solution which satisfies all these constraints. This is all explained in the rest of this report.

[1]Pylog is a bridge linking CPython to SWI Prolog.

[2]Metalog "is a next-generation reasoning system for the Semantic Web. Historically, Metalog has been the first semantic web system to be designed, introducing reasoning within the Semantic Web infrastructure by adding the query/logical layer on top of RDF."

[3]Pychinko is a Python implementation of the RETE algorithm, aimed at production systems. RETE is quite different from logic/constraint programming as it is concerned with truth maintenance in the face of a changing environment

[4]CWM is "a popular Semantic Web program that can, amongst many things, perform inferences as a forward chaining FOPL inference engine". CWM uses Pychinko.

Prolog [PRLG92] and constraint solving algorithms like MAC [MAC97] were considered, but we finally decided to go with the more recent paradigm named Concurrent Constraint and Logic Programming [CCLP93], as it supports requirement #1, #2 and #3. More precisely, we decided to walk in the steps of the Oz progamming language [OPM95], which provides a CCLP framework also satisfying #4. Leveraging the PyPy architecture allowed to implement all of this while respecting #5.

We now skim very briefly over the history and state of affairs in the realm of logic and constraint programming, in order to expose the rationale behind our architectural choices.

3.2 Logic Programming in Prolog

While first touched by John MacCarthy in *Programs with common sense* [MCC59], and approached in languages like Planner, logic programming was installed firmly with Prolog. Despite Prolog having incomplete logical semantics (due to its depth-first search strategy and the existence of 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.

3.3 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* [AIJ94] interleaved.

3.4 Limits of Prolog and MAC

A typical classic Prolog implementation and MAC share several fundamental common mechanisms (backtracking and trailing); as such, modern Prolog implementations (such as GNU Prolog) have been augmented with finite domain constraint programming without much pain.

Both Prolog and MAC, however can be considered lacking [PVR02] 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 environment);

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 programming; 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++).

3.5 From Backtracking Engines to CCLP

In short, CCLP draws from research in parallelization and modularization of logic programming, integration with stateful computing and integration with constraint programming. CCLP emerged after almost two decades of research into ways to extract the anticipated inherent parallelism of a logic language.

Concurrent logic languages had a rich history with experiments like *Parlog* [PAR86], *Concurrent Prolog* [CPR87], *Guarded Horn Clauses* [GHC87].

The most significant step was the *ask* and *tell* operators brought by *Saraswat* [CCP90]. This work reconciliated parallel logic languages with constraint programming. The Andorra Kernel Language [AKL91], 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 that is riddled with low-level and error-prone locks, mutexes and semaphores. It also has clean logic semantics.

3.6 From CCLP to Oz to Python (anecdotal bits)

Expanding on AKL, the Oz language integrates CCLP into an already functional and also stateful programming framework, which is truly multi-paradigm. In this light, it was decided to meet the current Oz developer 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 shed 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 in such a language. 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.

Moreover, the encapsulated search principle deployed with AKL has been shown to pave the road to languages mixing stateful and logic/constraint programming. The ability to copy whole computations (where a computation is defined as a bag of data comprising a set of threads plus its associated heap) eliminates the difficulty of implementing custom trailing semantics for custom ad hoc data types (or objects). It does not completely alleviate the dangers inherent to the manipulation of mutable data shared accross encapsulated computations (thus obviously violating encapsulation) though. But one usual answer to such concerns is that programmer 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 (most of the time).

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 right now that the non-declaration of variables in Python is a bit of a concern) or because these styles are not well understood by most programmers.

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

3.7 Constraint Handling Rules

Constraint Handling Rules [CHR98] are a special purpose constraint definition language for user-defined constraints. They complement CCLP languages. They bring features such as automatic simplification of constraints (preserving logical equivalence) and propagation (adding redundant equivalent constraints, which may cause further simplification) over these. We don't have this in the context of the WP09 work. It is a bit too early to decide if, when and how to achieve that.

It must be noted that the need for Constraint Handling Rules can be mitigated by the availability of first class constraints, and also the advanced computation space features described in [PCS00]. Neither feature will be provided as part of the WP09 work.

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

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[5] concurrent programming.

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

4.1 Basic 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 *only 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)
```

It should be 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 Python *__eq__* operator, but with one important twist: unify mutates the state of the involved logic variables.

The following basic example shows logic variables embedded into dictionaries:

[5] lock-free but not block-free, one symptom of a badly written concurrent program using logic variables being that it hangs indefinitely.

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

Unify is thus defined as follows (it is symmetric):

Unify	**value**	**unbound var**
value	equal?	bind
unbound var	bind	alias

Given two values, it does nothing if they are equal, or raises a UnificationError exception. Binding happens whenever one value is unified with an unbound variable. Finally, two unbound variables are aliased, that is, constrained to be bound by the same future value.

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, and with complete logical semantics).

4.2 Threads and Dataflow Synchronization

When a thread tries to access a free logic variable, it is suspended until the variable is bound. This behaviour is known as "dataflow synchronization". 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).

4.3 Table of Operators

Logic variables support the following operators:

Variable Creation:

newvar/0 nothing -> logic variable

Predicates:

is_free/1 any -> bool

is_bound/1 any -> bool

alias_of/2 logic variables -> bool

Mutators:

bind/2 logic variable, any -> None | RebindingError | FutureBindingError

unify/2 any, any -> None | UnificationError

Dataflow synchronization (blocking operations):

wait/1 value -> value | AllBlockedError

wait_needed/1 logic variable -> logic variable

4.4 Implementation Aspects

As above, 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 are provided as standard PyPy builtins.

The most glaring feature, syntactically-wise, of the logic object space, is the half transparency of the logic variables. Basically all operations on application-level values are wrapped in calls to the *wait* generic function (or set of multi-methods), which then apropriately dispatches on the right action. One could reasonably argue that the run-time price is too high, and that it is not *pythonic* (the fourth, lesser known, theological virtue) to make such a feature implicit in Pyhton (the classical Python mantra being *explicit is better than implicit*).

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. Having multi-methods as an interpreter-level implementation device proved to be very convenient.

5 Programming With Search

5.1 Non-Deterministic Logic Programs

5.1.1 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
```

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 local program (we call this *don't care non-determinism*) but to another program interested in the outcomes, typically a solver exploring the space of the logic 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.

The above program contains two choice points. If given to a depth first search solver (this is the recommended 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. An illustration:

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

To allow this de-coupling between program execution and search strategy, the computation space concept comes handily. Basically, choices are exposed in speculative computations which are embedded, or encapsulated in the so-called computation space. An intuitive approximation of computation spaces is the idea of forking a program on choices: each forked process runs in an independant address space. Solutions get out as fresh values.

Let us introduce the space method behind choice:

```
choose/1
```

A call to choose blocks the space until a call to another method, commit, is made from the solver. 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.

5.1.2 Driving logic programs with computation spaces

The search engine drives the computation by selecting which branches to run when the space waits on choose calls. To do so, the solver must peruse two computation space methods, *ask* and *commit*. Thus ask/0, choose/1 and commit/1 form a three-way protocol that works exactly as follows. Ask waits for the space to be *stable*, that is when the program running inside is blocked on a choose call. It returns precisely the value provided to choose by the encapsulated program. The search engine can then, depending on its strategy, *commit* the space to one choice, thus unblocking the choose call that can return a value which represents the choice being made. 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),
- it terminates returning values, thus being a candidate solution to the problem it stands for.

Ask, 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.

5.2 Constraint Programs

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* ...).

5.3 Using the Constraint Engine

5.3.1 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.

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 zero-argument procedure defined as follows:

```
def simple_problem():
    x = domain(['spam', 'egg', 'ham'], 'x')
    y = domain([3, 4, 5], 'y')
    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. Note that the domain/1 primitive returns a constraint variable (which is a logic variable constrained by a domain).

5.3.2 Getting solutions

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

```
from constraint.solver import solve
space = newspace(simple_problem)

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

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

5.4 Table of Operators

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

Space creation:

newspace/0 nothing -> space

Finite domain creation:

domain/2 list of any, var. designator -> Constraint variable

Expressions:

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

AllDistinct/1 list of var. designators -> Expression

Space operations:

newspace/1 zero-arity function -> None

tell/1 Expression -> None

ask/0 nothing -> a positive integer i

choose/1 int -> None | AssertionError ('space is finished')

clone/1 space -> space

commit/1 integer in [1, i] -> None

merge/1 space -> list of values (solution)

5.5 Extending the Search Engine

Most of this part was inspired by Christian Schulte PhD Thesis, *Programming Constraint Services* [PCS00]. The first part of the thesis, up to chapter 9, has been a driving guide for our work. The later chapters, where computation spaces are made into composable constraint combinators, have been left out. There are two reasons for that: the implementation is quite involved, and it has been shown to perform not as well as other techniques (especially with respect to techniques yielding constraint handling rules capabilities).

Hence, the semantic of the *merge* computation space operator is merely a way to extract solutions, whereas it could have been, following Schulte's steps, a merging of two spaces's state (for instance, the binding in a space of unbound variables captured and assigned to into a subspace).

5.5.1 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 have seen, a CSP is encapsulated into a so-called first class *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:

```
1    def first_solution_dfs(space):
2        status = space.ask()
3        if status == 0:
4            return None
5        elif status == 1:
6            return space.merge()
7        else:
8            new_space = space.clone()
9            space.commit(1)
10           outcome = first_solution_dfs(space)
11           if outcome is None:
13               new_space.commit(2)
14               outcome = first_solution_dfs(new_space)
15           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 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. Well, it turns out that n-ary search as better performance caracteristics when n is 2 [PCS00].

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.

5.5.2 Using distributors

A computation space might contain one thread making calls to *choose*, and there can be only one such thread per space (not respecting this restriction leads quickly to insane, undebuggable programs). We call this thread the *distributor*.

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 distribution 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 highest 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 hard-wired) in the [MAC97] 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 two 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:

```
distribute('dichotomy')
```

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

5.6 Implementation Aspects

The non-deterministic part of this work peruses the cloning facility of PyPy's garbage collection infrastructure [D07.4]. 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 [TVC99] 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.

5.7 History of the Implementation, Credits

This work was done by Strakt, Merlinux, DFKI and Logilab.

Anders Lehmann and Aurélien Campéas wrote a first pure-Python emulation of logic variables, limited computation 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 proxying 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 support of non-deterministic computation, by Aurélien with the assistance of Anders and Alexandre Fayolle. The initial pure-python constraint solver has been partly ported to RPython, partly rewritten (with the help of Anders and Alexandre). Decisive low-level functionality in PyPy's own garbage collector framework was provided by Armin Rigo.

6 Semantic Web: 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.

Section 1.2 of [OWL] summarises the standards and tools envisaged for the Semantic Web and describes their function and interrelationships:

- 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.

7 Implementation of the OWL Reasoner

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 use the tableaux algorithm.

Our 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 that are then 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 (i.e. by using a owl:oneOf predicate) the collection of all Individuals (the extension of owl:Thing). This collection is built 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 have been processed the ontology can be checked for consistency by invoking the constraints. The constraints are used to narrow the domains down to the smallest number of items that satisfy 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 the constraint application leads to a failure.

After a consistent ontology has been converted in to a constraint problem, search can be performed by adding new constraints representing a search query.

7.1 Implementation of Builtin OWL Types

7.1.1 Implementation of owl:Class

A owl:Class is defined by the members of the class. It is implemented as a class(called ClassDomain), which is a subclass of AbstractDomain.

7.1.2 Implementation of owl:Thing

All individuals in OWL are a subclass of owl:Thing. So the domain of owl:Thing contains all individuals. Technically it is implemented as a subclass of ClassDomain.

7.1.3 Implementation of owl:Nothing

owl:Nothing is the empty set.

7.1.4 Implementation of rdf:Property

Properties are relationships between Things (or Classes). It is implemented as a subclass of Abstract-Domain, which records the subject and the object in a tuple. Internally the domain is implemented as a dictionary of a subject to a list of objects. The getValues method will deliver a list of tuples (or rather a generator, which produces the tuples) of subject, object.

There is also a method called getValuesPrKey, which produces the list of objects for a given subject.

7.1.5 Implementation of owl:FunctionalProperty

FunctionalProperty is implemented as a subclass of Property, with a FunctionalCardinalityConstraint

7.1.6 Implementation of owl:InverseFunctionalProperty

7.1.7 Implementation of owl:TransistiveProperty

7.1.8 Implementation of owl:DatatypeProperty

7.1.9 Implementation of owl:InverseProperty

7.2 Implementation of Builtin OWL Predicates

7.2.1 Implementation of owl:oneOf

There are two versions of owl:oneOf. The first defines a class by enumerating all the members of the class. The class is completely defined by this enumeration.

The second version defines a datatype by enumerating the values of the datatype.

The implementation creates a domain for the class, parses the elements of the list (which is the object of the oneOf triple), and fills the elements into the domain. The class is marked as final.

7.2.2 Implementation of rdf:Type

As OWL is an extension of RDF, the reasoner has to be able to support RDF semantics. For rdf:Type one has to distinguish between two cases. If the object of is not a builtin OWL type, the meaning of rdf:Type is implemented as a MemberConstraint that will raise an exception if the subject is not in the domain of the object. The MemberConstraint is special because it does not constrain the domain.

7.2.3 Implementation of owl:equivalentProperty

The equivalentProperty is implemented by a special constraint that ensures that the domain of both properties are identical. This constraint does not narrow the domains of either property.

7.2.4 Implementation of owl:subClassOf

The owl:subClassOf is implemented as a constraint for the subject class, that will remove all individuals that is not part of the object class. This means that the object class has to be defined, either by a oneOf statement or by having the defining constraints run for the object. To ensure that other defining constraints arerun first the cost attribute of a subClassOf constraint has a highvalue.

7.2.5 Implementation of owl:equivalentClass

The owl:equivalentClass states that the members of two classes shall be the same. This is implemented with two subClassOf statements.

7.3 Example of Using the OWL Reasoner

Here is an example of using the reasoner on an existing ontology:

```
O = Ontology() # Create a new ontology
O.load_file("ontology.rdf") # parse the ontology into triples
O.attach_fd() # Attach finitedomains and create constraints
O.consistency() # Check consistency
```

If the ontology is not consistent an exception is raised.

To search the ontology more constraints are added. For example to search for all individuals that have a specific value ('Green') for a property ('color'), you would do the following:

```
result = URIRef('result')
O.type(result, Restriction)
O.onProperty(result, 'color')
O.hasValue(result, 'Green')
O.consistency()
for res in O.variables[result]:
    print res
```

8 Using the Results for Semantic Web Applications

8.1 Searching the Language Technology World

A promising opportunity of applying the above results to real-world problems opened up when the developers of LT World, the world's largest information portal on Speech and Language Technologies, expressed interest in adding to the portal more generic search functionality. 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 means to create functionality capable of intelligent search and of answering typical user questions. The application consists of a linguistic part, which is outside the scope of this project, and a reasoning part, which is described in more detail below. The linguistic part maps natural language queries onto expressions of SPARQL [SPQL], and converts results from constraint resolution back into natural language output. There are preexisting components for this that need to be adapted from SeSame to SPARQL, which is a straightforward task.

- Describe range of SPARQL queries covered, questions covered
- Size of LT World, Performance
- Achievement: proof of concept for scalability and usability?

8.2 Assessing Benefits

- Assess performance
- General need for closed domain Question Answering at DFKI

9 Conclusion

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

Several elements proper to the PyPy architecture have made this work possible: the concept of a first class object space, which allows to wrap additional semantics around normal Python semantics (the case of logic variables), the usage of RPython, which facilitates writing interpreter-level code (by comparison with the C language), the presence of multiple dispatch methods (that facilitates the definition of builtin operators for instance), and the aspect-oriented machinery which allowed to write garbage collector-level routines for the cloning of threads (impossible with the Böhm-Weiser GC).

We can assert without doubt that this work would have been completely impossible within CPython.

References

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

[OPM95] *The Oz Programming Model*, Gert Smolka, 1995

[ECP94] *Encapsulated Search and Constraint Programming in Oz*, Christian Schulte, Gert Smolka and Jörg Würtz, 1994

[PRLG92] *The birth of Prolog*, Alain Colmerauer and Philippe Roussel, 1992

[MAC97] *Understanding and Improving the MAC Algorithm*, Daniel Sabin and Eugene Freuder, 1997

[CCLP93] *From Concurrent Logic Programming to Concurrent Constraint Programming*, Frank S. de Boer and Catuscia Palamidessi, in *Advances in Logic Programming Theory*, 1993

[MCC59] *Programs with common sense*, John McCarthy, 1959

[AIJ94] *Arc-Consistency and Arc-Consistency Again*, Christian Bessière, 1994

[PVR02] *Logic Programming in the Context of Multiparadigm Programming: The Oz Experience*, Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, Christian Schulte, in Theory and Practice of Logic Programming, 2002

[PAR86] *Parlog: Parallel Programming in Logic*, K. Clark and S. Gregory, in ACM TOPLAS Vol. 8, N. 1, 1986

[GHC87] *Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard*, Kazunori Ueda, 1987

[CPR87] *Concurrent Prolog : Collected Papers*, Ehud Shapiro, MIT Press, 1987

[CCP90] *Semantic foundations of concurrent constraint programming*, Vijay A. Saraswat, Martin Rinard and Prakash Panangaden, ACM, 1990

[AKL91] *Programming Paradigms of the Andorra Kernel Language*, Sverker Janson, Seif Haridi, in Logic Programming, Proceedings of the 1991 International Symposium, 1991

[CHR98] *Theory and Practice of Constraint Handling Rules*, Thom Frühwirth, in Journal of Logic Programming, Special Issue on Constraint Logic Programming, 1998

[TVC99] *Comparing Trailing and Copying for Constraint Programming*, Christian Schulte, 1999

[PCS00] *Programming Constraint Services*, Christian Schulte, 2000

[CPM01] *Constraint Propagation in Mozart*, Tobias Müller, 2001

[OWL] *OWL Web Ontology Language. Overview*, Deborah L. McGuinness and Frank Harmelen (eds.), W3C recommendation, http://www.w3.org/TR/2004/REC-owl-features-20040210/#s1.1, 2004

[SPQL] *SPARQL Query Language for RDF*, Eric Preud'homme and Andy Seaborne (eds.), W3C Working Draft 4 October 2006, http://www.w3.org/TR/rdf-sparql-query/, 2006