Unclean! Unclean! or Purity Issues in Declarative Constraint Logic Programming

Ralph Becket

28 March 2006

Outline

- What is Constraint Logic Programming?
- Mercury before CLP
- Adding CLP to Mercury
- Rogues gallery
- Conclusion

What is Constraint Logic Programming?

Vanilla logic programming a la Mercury

- Conjunction, disjunction, negation
- Variables are free or ground
- Backtracking generate-and-test search
- Simple semantics; pure

CLP

- Variables can be constrained (neither free nor ground)
- Constrain-and-generate search
- More expressive
- Complex semantics; some impurity seems necessary

Mercury in a nutshell

- Mercury is a syntax for FOL augmented with types and modes.
- (If you think Prolog is a declarative language, then you're wrong.)
- Types specify the domains of variables: ints, floats, strings, algebraic types, etc.
- Logical connectives:

conjunction	`P, Q, ...`
disjunction	`( P ; Q ; ... )`
negation	`not P`
conditionals	`( if P then Q else R )`
unification	`X = Y` (...in four yummy flavours)

Mercury in a nutshell

- Modes constrain the direction of data flow (i.e., specify which predicate arguments are inputs and which are outputs).
- Modes are used to constrain the operational semantics of a Mercury program.
- The denotational semantics of a Mercury program are given directly by the predicate definitions in the program (it's just a syntax for FOL). The denotation of a predicate is the set of ground terms for which it is true.

Mercury in a nutshell

```
:- pred append(list(T), list(T), list(T)).
:- mode member(in, in, out) is det.
:- mode member(out, out, in) is multi.

append([], Ys, Ys).

append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

Mercury in a nutshell

```
:- pred append(list(T), list(T), list(T)).
:- mode member(in, in, out) is det.
:- mode member(out, out, in) is multi.
```

```
append(H1, Ys, H3) :-
    H1 = [],                % Test
    H3 = Ys.                % Assign

append(H1, Ys, H3) :-
    H1 = [X | Xs],          % Deconstruct
    append(Xs, Ys, Zs),
    H3 = [X | Zs].          % Construct
```

Mercury in a nutshell

```
:- pred append(list(T), list(T), list(T)).
:- mode member(in, in, out) is det.
:- mode member(out, out, in) is multi.


append(H1, H2, Ys) :-
    H1 = [],                % Test
    H2 = Ys.                % Assign

append(H1, H2, H3) :-
    H3 = [X | Zs],          % Deconstruct
    append(Xs, Ys, Zs),
    H1 = [X | Xs].          % Construct
```

The life of an ordinary Mercury variable

$$\text{free} \longrightarrow \text{ground}$$

- free = contains junk
- ground = completely defined value
- Nothing in between (i.e., no partial instantiation, no variable aliasing).
- Compiler reorders code to ensure that each variable has a value before it is used.
- Very efficient: modes checked statically, no untrailing, no dereferencing, can specialise representation for each type.

Negation: the root of all pain

- Consider the goal `not(X < 3)`
- If `X` is ground, then this is easy.
- If `X` isn't ground, then we have a problem.
- We don't want to suspend this goal waiting for `X` to become ground because that requires exorbitantly expensive bookkeeping (and if the negation is a compound goal then... yikes!)
- So we adopt the Closed World Assumption, allowing us to implement Negation as Failure, hence we require that 'input' variables in negated goals be ground.

If-then-else goals: slightly painful

- `( if P then Q else R )` $\Longleftrightarrow$ `( P, Q ; not(P), R )`
- The condition `P` is actually in a negated context.
- So the negated goal rules have to apply.

If-then-else goals: slightly painful

Assuming `Dictionary` and `Key` are ground...

- Bad:

```
( if   lookup(Dictionary, Key, Value)
  then true
  else false
),
NewValue = Value + 1
```

- Good:

```
( if   lookup(Dictionary, Key, Value0)
  then Value = Value0
  else false
),
NewValue = Value + 1
```

The negation issue will come back to haunt us again...

Constraint Logic Programming

- In CLP you put constraints on variables *before* giving them values.
- Improves search performance by backtracking as early as possible.
- More expressive: much less bookkeeping code required in the application program.

Constraint Logic Programming Example

The cryptarithm

```
  S E N D
+ M O R E
---------
M O N E Y
```

can be simply expressed and efficiently solved by a CLP program:

```
subset([S, E, N, D, M, O, R, Y], 0 .. 9),
all_different([S, E, N, D, M, O, R, Y]),
S > 0,
M > 0,
1000*(S + M) + 100*(E + O) + 10*(N + R) + (D + E) =
    10000*M + 1000*O + 100*N + 10*E + Y,
label([S, E, N, D, M, O, R, Y])
```

Constraint Logic Programming Example

An equivalent plain LP solution is far more work

```
member(D, 0..9),
member(E, 0..9), E \= D,
Y = (D + E) mod 10, C1 = (D + E) / 10,
Y \= D, Y \= E,
member(N, 0..9), N \= D, N \= E, N \= Y,
member(R, 0..9), R \= N, R \= D, R \= E, R \= Y,
E = (N + R) mod 10, C2 = (N + R) / 10,
...
```

It's also probably much less efficient.

Adding CLP support to Mercury

Big questions:

- How can we add constrained variables to Mercury (previously we assumed a variable was either free — junk — or ground — completely defined)?
- How can we do this without compromising the efficiency of the whole language (we don't want to pay for supporting CLP when we aren't using it)?
- How can we do this without compromising the clean semantics of the language?

Here be dragyns...

‘any’, a new variable state

An ordinary Mercury variable’s instantiation state (‘inst’) is either free or ground:

- If `X` is free then it can unify with any ground value.
- If `X` is ground then

```
all [Y, Z] (
    unifiable(X, Y) /\ unifiable(X, Z)
=>
    unifiable(Y, Z)
)
```

‘any’, a new variable state

Generally a CLP variable is neither free nor ground:
consider `X = f(_)`, `Y = f(a)`, and `Z = f(b)`.

- `X` unifies with `Y` and `Z`
- but `Y` does not unify with `Z`

We introduce a new inst, ‘`any`’ to describe `X`, meaning `X` is not free (i.e., it is being managed by some *constraint solver*), but is not known to be ground either.

‘any’, a new variable state

Whether a variables inst is free, ground, or any is important for compilation. Consider the disjunction

```
(
    X = f(a), ...
;
    X = f(b), ...
)
```

- If X is free, then we have a choice point (i.e., the disjunction is non-deterministic) and either unification with X will succeed.
- If X is ground, then we have a switch (i.e., at most one of the disjuncts will be executed).
- If X is any, then we have a choice point, but there is no guarantee that either of the unifications with X will succeed.

Solver types, a new kind of type

- There are all sorts of CLP domains we want to be able to handle: LP, FD, SAT (clausal, non-clausal, BDD-based, ...), sets, Herbrand terms, etc.
- Each CLP domain will have its own representation and implementation.
- Also, while CLP variables have to support aliasing (all variables have to support equality), we do not want to lose the efficiency of non-CLP variables by requiring that all variables support aliasing.

Solver types, a new kind of type

A solver type connects variables in a particular domain to a particular constraint solver. Management of solver variables is the responsibility of their constraint solvers.

```
:- solver type cfloat
    where representation is cfloat_rep,
          initialisation is cfloat_init,
          equality is cfloat_eq.
```

- This introduces a new constrained type called `cfloat`.
- Values of the new type are represented using type `cfloat_rep`.
- These values are initialised automatically by the `cfloat_init` predicate.
- Equality between cfloats is handled by a predicate called `cfloat_eq`.

Representation types

- Q. Why do we need a separate representation type?
- A. Because the representation type is (usually) a ground value indexing into the constraint store for the type (e.g., an int):

  ```
  :- type cfloat_rep == int.
  ```

- The semantics of the solver type are not those of the representation type! In particular, different `cfloat_reps` (that obviously won't unify) may represent two cfloats that *have* been unified.

Representation types

The solver type declaration causes the compiler to introduce some casting functions:

```
:- impure func 'representation of cfloat'(cfloat::in(any)) =
    (cfloat_rep::out(ground)) is det.
:- impure func 'representation to cfloat'(cfloat_rep::in(ground)) =
    (cfloat::out(any)) is det.
```

And this is where things start to get sordid.

- Since we can only understand these functions through their operational semantics, we insist that they be marked as impure.
- And all callers of them have to be marked as impure.
- And so on up the call graph.
- Until we get a promise from the programmer that everything is all right.

More on this later...

Constraint stores

- The constraint store is part of the constraint solver implementation.
- It is not explicitly passed around in "user code" (that would defeat the purpose: we want to reason about constrained variables in terms of the constraints placed upon them, not in terms of states of the corresponding constraint stores).
- So where do we keep the constraint store? Time for more unpleasantness:

```
:- mutable(constraint_store, cfloat_constraints, initial_value, ...).
```

- This constructs a mutable value called `constraint_store` with type `cfloat_constraints`, whose starting value is `initial_value`.
- The compiler provides access predicates:

```
:- impure pred get_constraint_store(cfloat_constraints::out) is det.
:- impure pred set_constraint_store(cfloat_constraints::in) is det.
```

Hiding the nastiness

We can conceal impurity with promises:

```
:- pred cfloat_init(cfloat::out(any)) is det.

cfloat_init(X) :-
    promise_pure (
        impure get_cfloat_counter(N),
        impure X = 'representation to cfloat'(N),
        impure set_cfloat_counter(N + 1)
    ).
```

- The `promise_pure` scope is a declaration by the constraint solver implementer that `cfloat_init` is pure, even though its implementation is not.
- In other words, you can't use `cfloat_init` in any way to subvert the declarative semantics of a program.

The story so far...

- Solver types add support for CLP to Mercury.
- A solver type is managed by a constraint solver.
- Solver variables have inst any.
- Solver types are abstract. Only the constraint solver implementation gets to see the actual representation.
- The constraint solver keeps track of current constraints in a constraint store.
- Private mutables are used to hold the constraint store.
- All impure code has to be marked as such.
- Pure predicates can be implemented using impure code. It is up to the implementer to explicitly promise purity at the interface. (The compiler will get you if you lie.)

Pear-shaped places

Solver types have some problem areas:

- getting the value of ground solver variables
- labelling solver variables
- existential quantification of solver variables
- solver variables in negated contexts
- solver variables in closures
- sometimes you need impurity

Research is still in progress.

Obtaining values for solver variables

- Once one has set up a system of constraints and found a solution, the next step is to do something with it (e.g., print it out).
- This is an awkward spot: the domain of a solver variable (i.e., the set of values it can take on) depends upon the constraints on that variable.
- There are two ways to do this, each with problems
- Method 1:

```
solve_for_all_variables,
impure ask_ground_value(X, Y)
```

Method 2:

```
label_variable(X, Y)
```

Method 1

```
solve_for_all_variables,
impure ask_ground_value(X, Y)
```

- `solve_for_all_variables` searches for a complete solution
- But because it is pure, the compiler is free to reorder it, possibly before some of the goals that posted constraints!
- `solve_for_all_variables` also has no outputs, so the compiler will complain if we declare it can succeed more than once.
- Maybe we should just make it impure (although we have a glimmer of a better solution that I'll come to in a moment...)

Method 1

```
solve_for_all_variables,
impure ask_ground_value(X, Y)
```

- `ask_ground_value(X, Y)` succeeds binding `Y` to the ground value of `X`, if it is ground, and fails otherwise.
- this clearly depends upon the current state of the constraint store, hence it must be impure. Hmm...

Method 2

```
label_variable(X, Y)
```

- Non-deterministically binds `X` to a value `Y` in its domain
- `label_variable` may well not check that this solution for `X` can be extended to a solution for all other variables (it may well be a performance disaster to do so...)
- In fact, that may well be what we want. E.g., for an hybrid propagation based solver.
- Plus, we'd rather the compiler didn't reorder these goals either, even though they are pure.

Existential quantification

- Consider the following where X, Y, and Z are solver variables and p and q constrain their arguments:

```
some [Z] (
    p(X, Z),
    q(Y, Z)
)
```

- It is possible that there is no solution for Z.
- The problem here is that p and q will probably post their constraints lazily (i.e., will not do complete consistency checking, instead leaving that task to the variable labelling predicate). This is quite often necessary for performance reasons because we don't have any other consistency checking mechanism other than searching for a solution.
- This isn't such a problem if we do *eventually* check that Z has a solution.

Existential quantification

- But how can we realistically enforce completeness checking?

```
main(!IO) :-
    ( if   some [X] ( post(X > 3), post(X < 3) )
      then print("Hello, World!", !IO)
      else print("Goodbye, cruel world", !IO)
    ).
```

- Here we never ask for a solution for X. But what does this program mean? Operationally, we'd probably expect it to output "Hello, World!". Denotationally we'd expect a program that prints "Goodbye, cruel world".
- So what do we do? Requiring (somehow) that existentially quantified solver variables have a solution before they go out of scope (or at least deciding the question one way or another) could easily be a performance killer for reasons already stated.

Negated contexts, again

```
( if p(X) then Q else R )
```

- If X is a solver variable (e.g., a cfloat) with inst any, then seems to have the same problems that free (non-solver) variables have.
- That is, because we don't know whether X is semantically ground, we can't make the closed-world assumption, hence negation as failure isn't sound.
- Ouch.

Negated contexts, again

```
( if p(X) then Q else R )
```

- Currently has to be marked impure
- Or be in a `promise_pure` scope.
- The latter arises quite often. For example,

```
promise_pure (
  if length(Xs) < 10 then Q else R
)
```

- A better candidate solution would be to introduce a special inst describing predicates/closures can only be called in non-negated contexts...

Closures containing solver variables

- if `X` and `Y` are solver variables then it is fine to say

```
X < Y
```

- Surely, then, it is also reasonable to say

```
P = ((pred) is semidet :- X < Y),
P
```

- Hmm, but what about

```
P = ((pred) is semidet :- X < Y),
( if P then Q else R )
```

- Oh no!
- But if we had a special inst that recognised that `P` was not ground, then this problem would go away!
- As would most issues with `length(Xs)` and `solve_for_all_variables`

Summary

- Adding CLP is non-trivial
- It does raise the expressive power of the language
- But the price is it complicates the semantics in many places:
  - impurity
  - promises
  - negation
  - closures
  - ordering issues
  - quantification
  - completeness/correctness vs. performance
- We have some ideas on how to fix some things
- The correctness issue is crucial