Casper documentation

May 26, 2013

1 Introduction

Casper is a modeling language for constraint programming with declarative search strategy specification, in the line of the Rules2CP language [5]. The Casper compiler transforms Casper models into GNU or Sicstus Prolog programs with FD constraints [3, 2] or Java programs using either the Choco constraint solver [9] or the MiniSAT solver. Search strategies are specified by, on the first hand, unordered decision trees expressed with Boolean expressions, and, on the other hand, an ordering criterion on Boolean trees. This specification is transformed into a conjunction of reified constraints and a labeling ordering on the associated Boolean variables. Section 2 specifies the user-level language, consisting in a typed functional kernel for expressing Boolean expressions on arithmetic constraints. Section 3 defines the transformation from Boolean expressions into reified constraints and labeling ordering and illustrates this transformation for some usual search strategies. Section 4 gives the most relevant parts of the compilation chain from the Casper language to the targeted solvers.

2 User-level language

2.1 Example: n-queens in Casper

The following program models the n-queens problem in Casper.

```
let n := parameter('n') in
let queens := [{row: i, column: _} | i in 0 .. n - 1] in
let not_attack(q, q') := (
  q.column != q'.column /\
  q.column + q.row != q'.column + q'.row /\
  q.column - q.row != q'.column - q'.row) in
solve({
   constraints:
      (forall q in queens, 0 <= q.column < n) /\
      (forall q in queens,
         forall q' in queens,
           q.row != q'.row => not_attack(q, q')),
   search:
      forall q in queens,
         q.column <- inf(q.column) .. sup(q.column)})
```

This example shows the two distinctive traits of Casper with respect to other constraint modelling languages such as Zinc [10] or Essence [6]. The first trait is that the parametricity of

$\mathcal{G} =$ | $\langle variable \rangle$ | $\langle integer \rangle$ | $\langle string \rangle$ | `_` | `(`$\mathcal{G}$`)`	variables and constants
| $\mathcal{G} \bowtie \mathcal{G}$ $\quad \bowtie \in \{$`+`, `-`, `*`, `/`$\}$	general arithmetic expressions and constraints
| $\mathcal{G} \bowtie \mathcal{G}$ $\quad \bowtie \in \{$`=`, `!=`, `<`, `<=`, `>`, `>=`$\}$	
| `inf(`$\mathcal{G}$`)` | `sup(`$\mathcal{G}$`)`	indexicals
| `not(`$\mathcal{G}$`)`	Boolean operators
| $\mathcal{G} \bowtie \mathcal{G}$ $\quad \bowtie \in \{$`/\`, `\/`, `=>`$\}$	
| `[`$\mathcal{G}$`, ..., `$\mathcal{G}$`]`	list constructor
| `[`$\mathcal{G}$ `|` $\langle variable \rangle$ `in` $\mathcal{G}$`]`	list comprehension
| $\mathcal{G}$`..`$\mathcal{G}$	integer range list
| $\mathcal{G}$`[`$\mathcal{G}$`]`	indexed access
| `forall` $\langle variable \rangle$ `in` $\mathcal{G}$`,` $\mathcal{G}$	quantifiers over the elements of a list
| `exists` $\langle variable \rangle$ `in` $\mathcal{G}$`,` $\mathcal{G}$	
| `sum` $\langle variable \rangle$ `in` $\mathcal{G}$`,` $\mathcal{G}$	
| `product` $\langle variable \rangle$ `in` $\mathcal{G}$`,` $\mathcal{G}$	
| `{`$\langle field \rangle$`:`$\mathcal{G}$`, ..., `$\langle field \rangle$`:`$\mathcal{G}$`}`	record constructor
| $\mathcal{G}$`.`$\langle field \rangle$	record projection
| `let` $\langle variable \rangle$ `:=` $\mathcal{G}$ `in` $\mathcal{G}$	local definition of values
| `let` $\langle variable \rangle$`(`$\langle variable \rangle$`, ..., `$\langle variable \rangle$`)` `:=` $\mathcal{G}$ `in` $\mathcal{G}$	local definition of functions
| $\langle variable \rangle$`(`$\langle variable \rangle$`, ..., `$\langle variable \rangle$`)`	function application
| $\langle variable \rangle$ `<-` $\mathcal{G}$	finite-domain labeling
| `once(`$\mathcal{G}$`)`	extra-logical cut
| `solve(`$\mathcal{G}$`)`	solve directive
| `parameter(`$\langle name \rangle$`)`	access to model parameters
| `import(`$\langle filename \rangle$`)`	basic module system

Table 1: Casper grammar

the model (the variable n taken as a parameter) is preserved in the generated intermediary code and is handled by the targetted back-end. The second trait is that the model specifies the search strategy: the example shows a simple enumerating labeling from lower-bound to upper-bound for each variable.

2.2 Casper grammar

The grammar of Casper expressions is given in table 1. A simple type-system over these expressions is defined in section 2.3. A Casper file may contain any Casper expression and a Casper program is a file that contain an expression of type `goal`. The program can refer to values defined in other files with the *import directive* `import('`$\langle filename \rangle$`')`.

An expression of type `goal` is constructed with the *solve directive* that takes a record in argument: `solve({ constraints:` c`, search:` s `})`. This record specifies the two components that define a constraint solving problem: the constraints c and the search strategy s. The expressions c and s are both of type `var(bool)`, the type of (possibly uninstantiated) Boolean expressions. In the constraints expression c, Boolean connectives are interpreted as constraints. In the search strategy s, Boolean conjunction `/\` is interpreted as a sequence and Boolean dis-

junction \/ is interpreted as a choice-point (and Boolean implication is not allowed).

The construction x `<-` *list* is used in different contexts: basically, x `<-` `[`e_1`, ..., `e_n`]` is interpreted as x `=` e_1 `\/ ... \/` x `=` e_n. Consequently, in constraint expressions, x `<-` l `..` u gives the domain $\{l, \ldots, u\}$ to the variable x. In the search strategy, indexicals for a finite-domain variable x can be accessed with `inf(`x`)` and `sup(`x`)`: we use them in conjunction with the x `<-` *list* expression in order to express basic enumeration strategies (*e.g.*, x `<-` `inf(`x`) .. sup(`x`)` for a labeling from the lower bound to the upper bound). The Casper small-step semantics is defined in section 2.4 and the expressiveness of the search strategy language is discussed in section 3.

The import directive `import('`⟨*filename*⟩`')` used in conjunction with files defining record expressions provides Casper with a simple module system. This module system is illustrated by the small standard library that is distributed with Casper: e.g., `import('optimization.csp')` returns a record containing functions for standard optimization procedures that transform expressions of type `goal` that solves CSP satisfiability into expression of type `goal` that implement optimizers (typically, `import('optimization.csp').branch_and_bound`). Similarly, `import('packing.csp')` returns a record defining functions for building constraints for packing problems such as `import('packing.csp').non_overlap`.

2.3 Type system

Casper types have the following forms.

- The basic types of Casper are `int`, `bool`, `string` and `goal`: `bool` is a subtype of `int` for the values `0` and `1`.
- The type constructor `var(`τ`)` denotes the type of finite-domain variables, where τ is either `int` or `bool`. `var(bool)` is a subtype of `var(int)`, `bool` is a subtype of `var(bool)` and `int` is a subtype of `var(int)`. `goal` is a subtype of `var(bool)`.
- The type constructor `[`τ`]` denotes the type of lists whose elements are of type τ: all lists in Casper are homogeneous. If σ is a subtype of τ, then `[`σ`]` is a subtype of `[`τ`]`.
- The type constructor `{`⟨*field*$_1$⟩`:` τ_1`, ...,` ⟨*field*$_n$⟩`:` τ_n`}` denotes the type of records that contain the pair-wise distinct fields ⟨*field*$_1$⟩, ... ⟨*field*$_n$⟩ such that the value of ⟨*field*$_i$⟩ has the type τ_i. A record type σ is a subtype of a record type of τ if all the fields in τ occur in σ and if ⟨*field*$_i$⟩ has type τ_i in τ, then ⟨*field*$_i$⟩ has type σ_i in σ and σ_i is a subtype of τ_i.
- The type constructor `(`τ_1`,...,`τ_n`)->`τ denotes the type of functions with n arguments of types $\tau_1, \ldots, \tau_n$ respectively and whose produce a value of type τ.

The typing rules are given in table 2. It is worth noticing that the type system is monomorphic: for each value (function argument or let-binding), if there is no constraint that forces another type, the type `var(int)` is chosen by default. Moreover, let-binding allows the definition of recursive values.

Lists of finite-domain variables are allowed (with the type `[var(int)]` or `[var(bool)]`). However, the length of the lists is always fixed. In particular, the construction $l..u$ is only allowed when l and u have the type `int` (and not `var(int)`).

2.4 Semantics

The semantics of Casper follows a call-by-value evaluation strategy. All the constructions of fully instantied values (*i.e.*, all the types except `var(int)` and `var(bool)`) follow the usual

$$\frac{b \in \{0,1\}}{\Gamma \vdash b : \texttt{bool}} \qquad \frac{n \geq 2}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash \langle \mathit{string} \rangle : \texttt{string}} \qquad \frac{}{\Gamma \vdash \texttt{\_} : \texttt{var(int)}} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma \text{ subtype of } \tau}{\Gamma \vdash e : \tau} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int} \qquad \bowtie \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{\Gamma \vdash e_1 \bowtie e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{var(int)} \qquad \Gamma \vdash e_2 : \texttt{var(int)} \qquad \bowtie \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{\Gamma \vdash e_1 \bowtie e_2 : \texttt{var(int)}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int} \qquad \bowtie \in \{\texttt{=}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1 \bowtie e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{var(int)} \qquad \Gamma \vdash e_2 : \texttt{var(int)} \qquad \bowtie \in \{\texttt{=}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1 \bowtie e_2 : \texttt{var(bool)}}$$

$$\frac{\Gamma \vdash e : \texttt{var(bool)}}{\Gamma \vdash \texttt{inf}(e) : \texttt{bool}} \qquad \frac{\Gamma \vdash e : \texttt{var(int)}}{\Gamma \vdash \texttt{inf}(e) : \texttt{int}} \qquad \frac{\Gamma \vdash e : \texttt{var(bool)}}{\Gamma \vdash \texttt{sup}(e) : \texttt{bool}} \qquad \frac{\Gamma \vdash e : \texttt{var(int)}}{\Gamma \vdash \texttt{sup}(e) : \texttt{int}}$$

$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{not}(e) : \texttt{bool}} \qquad \frac{\Gamma \vdash e : \texttt{var(bool)}}{\Gamma \vdash \texttt{not}(e) : \texttt{var(bool)}} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool} \qquad \bowtie \in \{\texttt{/\textbackslash}, \texttt{\textbackslash/}, \texttt{=>}\}}{\Gamma \vdash e_1 \bowtie e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{var(bool)} \qquad \Gamma \vdash e_2 : \texttt{var(bool)} \qquad \bowtie \in \{\texttt{/\textbackslash}, \texttt{\textbackslash/}, \texttt{=>}\}}{\Gamma \vdash e_1 \bowtie e_2 : \texttt{var(bool)}}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \ldots \qquad \Gamma \vdash e_n : \tau}{\Gamma \vdash \texttt{[}e_1\texttt{, ..., }e_n\texttt{]} : \texttt{[}\tau\texttt{]}} \qquad \frac{\Gamma \vdash e_2 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{[}e_1 \texttt{ | } x \texttt{ in } e_2\texttt{]} : \texttt{[}\tau\texttt{]}} \qquad \frac{\Gamma \vdash l : \texttt{int} \qquad \Gamma \vdash u : \texttt{int}}{\Gamma \vdash l \texttt{..} u : \texttt{[int]}} \qquad \frac{\Gamma \vdash l : \texttt{[}\tau\texttt{]} \qquad \Gamma \vdash i : \texttt{int}}{\Gamma \vdash l\texttt{[}i\texttt{]} : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{[}\tau\texttt{]} \qquad \Gamma, x : \tau \vdash e_2 : \texttt{bool}}{\Gamma \vdash \texttt{forall } x \texttt{ in } e_1\texttt{, } e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \texttt{[}\tau\texttt{]} \qquad \Gamma, x : \tau \vdash e_2 : \texttt{var(bool)}}{\Gamma \vdash \texttt{forall } x \texttt{ in } e_1\texttt{, } e_2 : \texttt{var(bool)}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{[}\tau\texttt{]} \qquad \Gamma, x : \tau \vdash e_2 : \texttt{bool}}{\Gamma \vdash \texttt{exists } x \texttt{ in } e_1\texttt{, } e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \texttt{[}\tau\texttt{]} \qquad \Gamma, x : \tau \vdash e_2 : \texttt{var(bool)}}{\Gamma \vdash \texttt{exists } x \texttt{ in } e_1\texttt{, } e_2 : \texttt{var(bool)}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{[}\tau\texttt{]} \qquad \Gamma, x : \tau \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{sum } x \texttt{ in } e_1\texttt{, } e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{[}\tau\texttt{]} \qquad \Gamma, x : \tau \vdash e_2 : \texttt{var(int)}}{\Gamma \vdash \texttt{sum } x \texttt{ in } e_1\texttt{, } e_2 : \texttt{var(int)}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{[}\tau\texttt{]} \qquad \Gamma, x : \tau \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{product } x \texttt{ in } e_1\texttt{, } e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{[}\tau\texttt{]} \qquad \Gamma, x : \tau \vdash e_2 : \texttt{var(int)}}{\Gamma \vdash \texttt{product } x \texttt{ in } e_1\texttt{, } e_2 : \texttt{var(int)}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \texttt{\{}l_1\texttt{:}e_1\texttt{,...,}l_n\texttt{:}e_n\texttt{\}} : \texttt{\{}l_1\texttt{:}\tau_1\texttt{,...,}l_n\texttt{:}\tau_n\texttt{\}}} \qquad \frac{\Gamma \vdash e : \texttt{\{}l\texttt{:}\tau\texttt{\}}}{\Gamma \vdash e\texttt{.}l\texttt{:}\tau} \qquad \frac{\Gamma, x : \sigma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let } x \texttt{ := } e_1 \texttt{ in } e_2 : \tau}$$

$$\frac{\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n, f : \texttt{(}\sigma_1\texttt{,...,}\sigma_n\texttt{)->}\sigma \vdash e_1 : \sigma \qquad \Gamma, f : \texttt{(}\sigma_1\texttt{,...,}\sigma_n\texttt{)->}\sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let } f\texttt{(}x_1\texttt{,...,}x_n\texttt{) := } e_1 \texttt{ in } e_2 : \tau}$$

$$\frac{\Gamma \vdash f : \texttt{(}\sigma_1\texttt{,...,}\sigma_n\texttt{)->}\sigma \qquad \Gamma \vdash e_1 : \sigma_1 \qquad \ldots \qquad e_n : \sigma_n}{\Gamma \vdash f\texttt{(}e_1\texttt{,...,}e_n\texttt{)} : \sigma} \qquad \frac{\Gamma \vdash x : \texttt{var(int)} \qquad \Gamma \vdash l : \texttt{[int]}}{\Gamma \vdash x \texttt{ <- } l : \texttt{var(bool)}}$$

$$\frac{\Gamma \vdash e : \texttt{\{constraints:var(bool),search:var(bool)\}}}{\Gamma \vdash \texttt{solve}(e) : \texttt{goal}}$$

Table 2: Typing rules

functional semantics. Partially instantiated expressions of type `var(int)` and `var(bool)` are handled as abstract terms represented as expression trees. There are two back-end specific constructions. The first construction is the handling of parameters: for the implemented back-end, the parameters are commonly read from the command-line or, if the command-line does not specify them, then they are asked interactively in the terminal. The second construction is the handling of `solve(`e`)`, which expects that e evaluates into a record with two fields: `constraints` and `search`. Both fields should be Boolean expressions of type `var(bool)`. The expression in the field `constraints` is added in the constraint store of the underlying constraint solver. The expression in the field `search` is read as a $\wedge/\vee$ decision-tree of elementary constraints. This tree is evaluated depth-first and from left to right: `/\` denotes conjunctions and `\/` denotes disjunctions. Common search strategies that are expressible in this framework are given in section 3.

3 Search strategies

Besides the basic enumeration labeling expressible by conjunctions of x `<-` l and indexicals, most search strategies are expressible in terms of $\wedge/\vee$ decision-trees.

The following function gives the definition of dichotomic (or bissection) search, where the size of the domain of a variable is reduced by two, recursively.

```
let dichotomy(x) :=
  inf(x) = sup(x) \/
  inf(x) < sup(x) /\ (
    let m := (inf(x) + sup(x)) / 2 in
    (x <= m \/ x > m) /\ dichotomy(x)) in ...
```

It is easy to notice that `sup(x) - inf(x)` strictly decreases at each recursive call: a (rudimentary) arithmetic analysis detects such specific cases and rejects recursions for which a simple terminaison argument cannot be found.

The following function gives the definition of interval splitting, where the size of the domain of a variable is reduced by a fixed factor.

```
let interval_splitting(x, k) :=
  let s := (sup(x) - inf(x)) / k in
  exists i in 0 .. k,
    (x >= i * s /\ x < i * (s + 1)) in ...
```

These definitions are provided in the module `enumeration.csp` which contains a record, the fields of which contain these functions.

Having `goal` as a subtype of constraints `var(bool)` allows decision-trees to express optimization procedures by iterating solver phases. However, there is a need to introduce mutable variables persistent over search points to remember values found in previous search phases: this is done by adding the `ref()` construction creating a new mutable variable and a binary `:=` goal for assignment.

The following functions are provided in the module `optimization.csp`. Branch-and-bound is the iteration of a goal for finding a better solution with respect to an objective variable, or proving the optimality, *i.e.*, that there is no better solution.

```
let branch_and_bound(goal, x) :=
   let best := ref(inf(x)) in
   solve(goal) /\ best := inf(x)
   \/ x < best /\ branch_and_bound(goal, x) in ...
```

The termination is guaranteed since `sup(x)` strictly decreases (since `inf(x)` $\leq$ `sup(x)`). It is worth noticing that if `solve(goal)` fails, then we have $best = inf(x)$ and `x < best` immediately fails too. Another optimization strategy is on the contrary to try to assign the objective variable to its lowest value first, and then augments the value until a solution is found.

```
let minimize_increasing(goal, x):
   let minimize :=
      let store = ref(inf(x)) in
      (
         x = inf(x) /\
         post_constraint
      )
      \/
      objective > store /\ minimize in ...
```

4 Compilation targeting constraint solvers or SAT solvers

There are three implemented back-end for Casper: Prolog with finite-domain variables (with GNU-Prolog [3] and Sicstus [2] variants), Java with the Choco solver [9] and Java with MiniSAT [4].

Besides the arithmetic constraints included in the syntax, solver-specific constraints can be used such as the `geost` global constraint [1] in Sicstus or Choco for modelling placement problems (`packing.csp` module).

Lists are implemented as lists in Prolog and as arrays in Java.

In Java, records are compiled into interfaces, with methods for functional fields and getters for other fields. For example, the interface for a `Queen`, section 2.1, is the following.

```
interface Queen {
  public IntegerVariable getColumn();
  public IntegerVariable getRow();
}
```

The MiniSAT back-end handles only models whose all variables are `var(bool)` (*i.e.*, models that do not contain `var(int)`). The following Boolean example enumerates all the siphons [8] in an SBML model [7].

```
let filename := parameter('filename') in
let graph := import('sbml.csp').load(filename) in
let siphon := [ _ | i in 0 .. length(graph.places) - 1 ] in
let enumeration := import(enumeration.csp') in
(forall i in 0 .. length(graph.places) - 1,
   0 <= siphon[i] <= 1) /\
(forall i in 0 .. length(graph.places) - 1,
   siphon[i] =>
     (forall j in graph.places[i].predecessors,
       forall k in graph.transitions[j].predecessors, siphon[k])) /\
enumeration.enumerate(
  solver({ search: (forall i in 0 .. length(graph.places) - 1, siphon[i] <- 0 .. 1)}))
```

Compilation to MiniSAT is implemented as a two-step compilation, transforming first the model in a Casper model in conjunctive normal form.

References

[1] N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In C. Bessière, editor, *Proc. CP'2007*, volume 4741 of *LNCS*, pages 180–194. Springer-Verlag, 2007. Also available as SICS Technical Report T2007:08, `http://www.sics.se/libindex.html`.

[2] M. Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, release 4 edition, 2007. ISBN 91-630-3648-7.

[3] Daniel Diaz. *GNU Prolog user's manual*, 1999–2003.

[4] Niklas Eén and Niklas Sörensson. MiniSAT web page. `http://minisat.se/`.

[5] François Fages and Julien Martin. From rules to constraint programs with the Rules2CP modelling language. In *Recent Advances in Constraints, Revised Selected Papers of the 13th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP'08*, volume 5655 of *Lecture Notes in Artificial Intelligence*, pages 66–83. Springer-Verlag, 2009.

[6] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.

[7] Michael Hucka et al. The systems biology markup language (SBML): A medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

[8] M. Kinuyama and T. Murata. Generating siphons and traps by petri net representation of logic equations. In *Proceedings of 2th Conference of the Net Theory SIG-IECE*, pages 93–100, 1986.

[9] Choco Team. CHOCO web page. `http//www.emn.fr/x-info/choco-solver/doku.php`.

[10] The Zinc team. MiniZinc web page. `http://www.minizinc.org/`.