EMini: a miniscule fragment of Essence (v1.1)

András Salamon and Christopher Stone

20230309

1 Introduction

We propose implementing a parser for a small fragment of Essence, which we call EMini. This is inspired by the fragment E_{FO} which is studied by Mitchell and Ternovska [1]. Their fragment uses enumerated types and relations, as their work is motivated by computational complexity. We also use relations but for simplicity we use integer domains instead of enumerated types. In this note we confine ourselves to defining the fragment and giving a small example of an NP-complete problem, instances of which can be expressed using EMini in a reasonably succinct way.

2 The language

An EMini specification consists of three parts.

2.1 Letting

The `letting` part consists of a sequence of domain declaration statements or definitions of constants.

Example constant definitions are of the form

```
letting a be 3
```

defining an integer constant `a`,

```
letting t be (1,2)
```

defining a constant tuple `t` of arity 2 with integer components, or

```
letting R be relation ((1,2),(1,3),(2,3))
```

defining a constant relation `R` of arity 2 containing 3 tuples.

Example domain declarations are of the form

```
letting D be domain int(a..b)
```

defining finite integer domains, where `a` and `b` are integer constants,

```
letting T be domain tuple (D,E,F)
```

where `D`, `E`, and `F` are integer domains declared as `letting`, the list of domains contains at least one element and is bracketed by parentheses, and the elements in the list are separated by `,`, or

```
letting Rtype be domain relation of (D * E * F)
```

where `D`, `E`, and `F` are integer domains declared as `letting`, the list of domains contains at least one element and is bracketed by parentheses, and the elements in the list are separated by `*` (note the syntactic disparity between tuples and relations).

At present we do not support the semantically equivalent declaration `letting Rtype be domain relation of T` which uses the notion that a relation is a set of tuples. (Doing so would diverge from the existing syntax of Essence.)

2.2 Find

The `find` part consists of a sequence of statements of the form

```
find R : relation of Rtype
```

where `Rtype` was declared as `letting`.

2.3 Such that

The `such that` part consists of a sequence of statements of the form

```
such that P
```

where `P` is an expression which is a formula of first-order logic over relations and integers, such that all quantifiers are of the form

```
forAll i : D .
```

or

```
exists i in D .
```

where `D` is an integer domain declared by a `letting` statement. The quantification is either over all elements in a domain, or all elements (tuples) in a relation.

3 Example

The following EMini spec (which is also valid Essence) defines an instance of Graph 3-Colouring, where given an input graph the task is to find a mapping of the vertices to 3 colours such that the endpoints of every edge are different. The vertices are denoted by positive integers. We use the relation `C` to map a colour to each vertex. The three constraints require `C` to be a function, which is total, and which is a proper colouring.

```
letting vertices be domain int(1..3)
letting colours be domain int(1..3)
letting G be relation((1,2),(1,3),(2,3))
letting map be domain relation of (vertices * colours)
letting T be domain tuple (vertices,colours)
find C : map
find t : T
such that
  forAll (u,c) in C .
     forAll (v,d) in C .
        ((u = v) -> (c = d))
such that
  forAll u : vertices .
     exists c : colours . C(u,c)
such that
  forAll (u,v) in G .
     forAll c,d : colours . (C(u,c) /\ C(v,d) -> (c != d))
such that
  t in C
such that
  t[1] = t[2]
```

Then `conjure solve` returns the following solution:

```
letting C be relation((1, 3), (2, 2), (3, 1))
letting t be (2, 2)
```

4 Tokens

Spaces are not significant.

Here are the tokens used in our language:

`letting be such that domain tuple int relation forAll exists in . , : * ( ) [ ] -> /\ \/ ! = != < > <= >=`

5 Expressions

Parentheses `(` and `)` are used to bracket expressions and to disambiguate. Expressions should be bracketed using parentheses to avoid problems with non-associative operators such as `->`.

The table shows the operators.

References

[1] D. G. Mitchell and E. Ternovska. Expressive power and abstraction in essence. *Constraints*, 13(3):343–384, 2008. doi:10.1007/s10601-008-9050-3.

operator	description
`->`	implication
/\	logical and
\/	logical or
`!`	logical negation
`=`	equality
`!=`	inequality
`<`	less-than
`>`	greater-than
`<=`	less-than-or-equal
`>=`	greater-than-or-equal