MANUAL FOR REFLECTION

KLAUS THIEL

Abstract. Reflection is a simplification procedure for quantifier free formulae over a semi-ring with 0 and 1 and decidable equality. Given the goal G reflection produces a new formula G' and a proof of $G' \to G$ which is then applied to G thus one is left with the new goal G' which should be easier to prove.

Contents

1. Introduction

Let us consider the proof of $\forall x, y, z \in \mathbb{Z}(x + y + z = z + y + x)$. In order to show this statement by hand from the semi-ring-axioms one needs six steps:

```
(sg "int1+int2+int3=int3+int2+int1")
(strip)
(simp "IntPlusCom")
(simp-with "IntPlusComm" (pt "int1") (pt "int2"))
(simp "IntPlusAssoc")
; ?_5: T
(use "Truth-Axiom")
; Proof finished.
```

The amount of steps increases quickly if one deals with more complex goals. The proof search is not always able to find a proof:

```
(sg "int1+int2+int3=int3+int2+int1")
(search)    ; no proof found
(strip)
(prop)    ; Not provable ...
```

With the reflection procedure one needs only three steps whatever the goal might be — provided it is provable from the semi-ring axioms.

```
(sg "int1+int2+int3=int3+int2+int1")
(strip)
```

```
(reflection "int")
; ?_3: int1+int2+int3=int1+int2+int3
(use "Truth-Axiom")
; Proof finished.
```

2. How To Use It

`(reflection)` is defined in the file `reflection.scm`. This file depends on `nat.scm`, `list.scm` and `reflection_thms.scm`.

`(reflection)` can only be applied to a quantifier free goal over some semi-ring. But before doing so, we need to prepare the reflection procedure for which we use the function

```
(prepare-reflection
    Ring
    Null Unum
    RingAdd RingTimes
    RingAddAssoc RingAddNeutral RingAddComm
    RingTimesAssoc RingTimesNeutral RingTimesComm
    Distr)
```

This will generate the proofs and theorems used for the simplification and its proof. The twelve expected arguments have to be strings with the following meaning:

`Ring`	name of semi-ring
`Null`	name of neutral element for `RingAdd`
`Unum`	name of neutral element for `RingTimes`
`RingAdd`	name of addition
`RingTimes`	name of multiplication
`RingAddAssoc, RingAddNeutral, RingAddComm`	
`RingTimesAssoc, RingTimesNeutral, RingTimesComm, Distr`	name of theorem or global assumption

In order to invoke the reflection procedure one uses the command `(reflection Ring)` where `Ring` is a string naming the semi-ring to which reflection is applied to.

Example.

```
(exload "ordinals/reflection.scm")
(libload "numbers.scm")

(aga "IntPlusAssoc"    (pf "all i,j,j2.i+(j+j2) = (i+j)+j2"))
(aga "IntPlusNeutral"  (pf "all i.IntZero+i = i"))
(aga "IntPlusComm"     (pf "all i,j.i+j = j+i"))
(aga "IntTimesAssoc"   (pf "all i,j,j2.i*(j*j2) = (i*j)*j2"))
(aga "IntTimesNeutral" (pf "all i. (IntPos One)*i = i"))
(aga "IntTimesComm"    (pf "all i,j.i*j = j*i"))
(aga "IntDistr"        (pf "all i,j,j2.i*(j+j2) = (i*j)+(i*j2)"))

(prepare-reflection "int" "IntZero" "IntPos One"
                    "IntPlus"  "IntTimes"
                    "IntPlusAssoc"  "IntPlusNeutral"  "IntPlusComm"
                    "IntTimesAssoc" "IntTimesNeutral" "IntTimesComm"
                    "IntDistr")

(sg "i+2+j+2+i*j2+j+j = i*(1+j2)+j*2+4+j")
(strip)
```

```
(reflection "int")

; ?_3: 4+i+3*j+i*j2=4+i+3*j+i*j2
(use-with "Truth-Axiom")
; Proof finished
```

3. How Does It Work ?

Let us assume that we have a goal G over a semi-ring R. The reflection method simplifies the goal to G', builds a proof of $G' \to G$ and applies that proof to G thus we are left with the new goal G'.

Which Goals Can Reflection Be Applied To ? The goal must be an atomic formula consisting of a binary boolean valued function applied to two arguments. The general pattern is $f(t_1, t_2)$ where f is of type $R => R => boole$ and t_1 and t_2 are terms of type of R. Moreover R must be discrete, e.g. a finitary algebra with boolean valued equality $=$.

How Does Reflection Simplify The Goal ? The simplification from G to G' is not done within R but on the meta-level. That is why this method is called reflection. `(prepare-reflection)` introduces an expression algebra named "ring-expr", where ring is replaced by the semi-ring's name. The algebra has the following constructors:

```
ringVar  :nat=>ringexpr
ringConst:ring=>ringexpr
ringAdd  :ringexpr=>ringexpr=>ringexpr
ringMult :ringexpr=>ringexpr=>ringexpr
```

If one applies `(reflection)` to some goal $f(t_1, t_2)$, then the terms t_1 and t_2 are reflected into that algebra. The mapping of semi-ring-terms into the expression algebra is done by the functions `(term-to-linarith-expr-and-env )` and `(terms-to-list-term )`. For example if t_1 is a variable (term in var-form), then it will be assigned the expression `ringVar Zero`.

In order to reobtain the corresponding semi-ring-term from an expression, one needs to know which number has been assigned to which variable. Therefore while reflecting t_1 and t_2 into the expression algebra a list of semi-ring-variables called "environment" is produced. The i^{th} variable in the environment has been assigned the expression `ringVar i` where i is a numeric term of the algebra `nat`.

Syntacticly different terms will be assigned to two different expressions which are semantically different in the expression algebra, For example in the case of the semi-ring `int`:

```
Goal  IntPos 2+IntPos 2 = IntPos 2*IntPos 2
lhs   intAdd(intConst 2)(intConst 2)
rhs   intMult(intConst 2)(intConst 2)
env   (Nil int)
```

```
Goal  i+2+j+2+i*j2+j+j=i*(1+j2)+j*2+4+j

lhs   intAdd
      (intAdd
       (intAdd
        (intAdd(intAdd(intAdd(intVar Zero)(intConst 2))(intVar(Succ Zero)))
         (intConst 2))
        (intMult(intVar Zero)(intVar(Succ(Succ Zero)))))
       (intVar(Succ Zero)))
      (intVar(Succ Zero))

rhs   intAdd
      (intAdd
       (intAdd(intMult(intVar Zero)(intAdd(intConst 1)(intVar(Succ(Succ Zero)))))
        (intMult(intVar(Succ Zero))(intConst 2)))
       (intConst 4))
      (intVar(Succ Zero))

env   i::j::j2:
```

What Reflection Cannot Do. `(reflection)` is not a complete decision procedure. It cannot be one because it is applied to quantifier free formulae with free variables in the full language of Arithmetic, i.e. including addition and multiplication. For example `(reflection)` is not able to solve Fermat's Last Theorem, not even the instance $n = 3$. The goal $k * k * k + m * m * m \neq n * n * n$ is not changed by `(reflection)` does not normalise to `false`.