Project Report

Expression Grammar

Our tool supports expressions containing the following operators: ***== < <= > >= + - min max add1 sub1 && || ! = if***. You can also use ***and or not*** in place of ***&& || !***.

== is for Integer equality and ***=*** is for Boolean equality, although internally we convert these to ***=*** and ***equal?***, respectively, since ***==*** is not a Racket operator. This means that the simplification of *(== a b)* would be *(= a b)*, with *a* and *b* being inferred as Integers.

As instructed, our type inference rules default to Integer when the type of a variable cannot be established. This means that situations can arise where an ***if*** statement returns an Integer in one case, and a Boolean in the other. Since Integers can be used with the Boolean operators **&& || !** in Racket, and Rosette has no problems handling these cases, our expression grammar allows for these kinds of situations. However, please be aware that Racket's rules for using Integers with Boolean operators are rather unintuitive, and as such our tool's output may be difficult to reason about in these kinds of situations. Our tool prints a warning message when we are forced to assume the type of a variable.

We have restricted the grammar of our tool to require that operators will always have a set number of arguments. In Racket, the operators ***+ min max && || = equal?*** can all take an arbitrarily high number of arguments, but this is just a convenient shorthand. *(+ a b c)* is really *(+ (+ a b) c)*, *(= a b c)* is really *(&& (= a b) (= a c))*, etc. Therefore, our constraint does not restrict the overall expression space, and ensures that the height of given expressions is more 'honest'. From a practical point of view, this also makes working with macros quite a bit easier. We generally require operators to take two arguments, with the exception of ***! add1 sub1*** which take one, and ***if*** which takes three.

Usage

Our tool is accessed through the *simplify-exp* function in *Project.rkt*, which takes a syntax object and prints out a simplified version of the expression. A valid function call would be *(simplify-exp (syntax (|| (&& a b) (|| a b))))*. This function is exported, so if you include *Project.rkt* in an external test file, you will have access to *simplify-exp*. *ults.rkt* must also be present in the same folder.

simplify-exp will print out the given formula, followed by the result of running our manual simplification rules on it. Depending on the formula, our simplification rules will have varying effects; certain subexpressions may be simplified, the entire formula may be reduced to a single constant or variable, or there may be no change at all. If we are forced to assume that certain variables are integers, warning messages will be printed here. *simplify-exp* will then attempt to synthesize the simplified formula, starting with a depth of 0, and increasing the depth if no synthesis can be found. A line will be printed to inform you when synthesis begins at each depth.

When a successful synthesis is found, *simplify-exp* will open an external file, *data.rkt*, and write enough code to synthesize the formula and perform a *print-forms* call. The output of this call will be redirected to *solution.txt*, which *simplify-exp* will read and print out as our final simplified expression.

The simplified expression will be in the form of a definition. The result of running *(simplify-exp (syntax (|| (&& a b) (|| a b))))* will be *(define (gen a b) (|| b a))*. As you can see, the solution *(|| b a)* can be found at the end of the definition.

Note: On one of our computers, the program sometimes fails to write to solution.txt, causing a contract violation error. There has been no discernable pattern to when it fails and when it succeeds. Since this issue did not come up on Victor's computer, we are assuming that this is due to some idiosyncrasy of the machine. If this does happen to come up, you can get the output by running *data.rkt*, after *simplify-exp* has written to it (and given you the error).

<u>Simplification Rules</u>

In general, we have found that Rosette is a lot smarter than we are, and does not require help finding an optimal simplification. However, larger and deeper expressions lead to drastically increased runtime, even if Rosette will ultimately find a solution. Therefore, when developing our simplification rules, we have focused on reducing the size and depth of expressions, even if Rosette would be able to make the simplification anyways.

Our rules handle simplifications on a case-by-case basis. They can be found near the bottom of our code, in the "Manual Simplification" sections. To give some examples:

(+ x 0) simplifies to *x*

(> x x) simplifies to *#f*

(>= x x) simplifies to *#t*

(if (> (+ x y)(+ y x)) x y) simplifies to *y*

If a subexpression only contains constants, we evaluate it directly – *(+ 1 2)* simplifies to *3*, for instance.

Lastly, we have found that there are some basic simplifications with integers that Rosette does not recognize. For example, it does not simplify *(< (+ a b) (+ a c))* to *(< b c)*. Wherever we have found these situations during our tests, we have added appropriate simplification rules.