Rewrite Systems for Symbolic Evaluation of C-like Preprocessing

Mario Latendresse

Northrop Grumman Information Technology

Science and Technology Advancement Team – FNMOC

U.S. Navy

Outline

- Introduction to C-like preprocessing
- Statement of the objectives
- Some of the technical issues
- The conditional values
- The symbolic algorithm
- The problem of mixing macro substitution, evaluation and parsing
- Three rewrite systems
- Future work

Introduction

Text preprocessing *à la* `cpp` is widely used: From C to Haskell.

Text preprocessing is composed of

- conditional compilation `#if` ... `#endif`
- macro definition `#define ...`
- substitution of macro identifier about *anywhere* in the source code
- substitution and evaluation on `#if` conditionals
- inclusion of files with `#include`

Objectives: An Example

```
#if defined(X)
#define Y  5
#elif Y > 20
#define X  10
#else
#define X  30
#endif
....
#if Y > 4
v = X;
#endif
```

Questions we want to answer:

- Under which condition is ‘`v = X;`’ compiled? *If X is defined or the initial $Y > 4$.*
- What are the possible values of `X` for that statement? And under which conditions are they obtained? *X may have values ‘10’ ($Y > 20$), ’30’ ($4 < Y \leq 20$), ’X’ if not given any value, or any initial value given at compile time.*

General Objectives

The general objectives are:

- Find for every line of source code the condition (Boolean expression) under which it is compiled (or reached).
- Find for every line of source code the values of each preprocessing variable (macro) and under which conditions they are obtained.

The result is a static representation of all possibilities independent of any configuration (initial values of variables).

Symbolic Evaluation of C-like Preprocessing

We apply symbolic evaluation to the preprocessing code to answer the objectives.

In general, for symbolic evaluation, the values of preprocessing variables are unknown.

The variable names are even unknown!!!

To be complete, we have to assume the most general case:

> *Any unbound identifier V on if-conditions is assumed to be a variable name and is assigned the symbolic value V_I.*

Inferring Preprocessing variables: An Example

```
int main(char **argv[], int argc){
#if defined(E)
#define S "HELLO"
#endif

#if defined(S)
  printf(S);
#endif
}
```

```
gcc -DE f.c
gcc -DS='"ALLO"' f.c
gcc -DS -D'printf(X)=printf("!")'
```

Any identifier may be a free preprocessing variable.
But we can only infer what the source code tells us.

In this case we can infer that `S` and `E` are variables, no more.

Substitution and Evaluation

The concrete evaluation of an if-directive condition is done in three phases:

1. The `defined` operators are evaluated resulting in ‘0’ or ‘1’.
2. The macro substitution is done with string operations if any. A list of lexems is obtained.
3. Arithmetic and logical operations are done. At this point the conditional expression should be a valid arithmetic/logic expression.

The special operator `defined`

```
#define R(x) 2##x
#define H(x) R(x)
#if H(defined(W)) == 20
 /* W is not defined */
#else
 /* W is defined */
#endif
```

The concrete expansion algorithm has a special case:

All `defined` operators are expanded to ‘0’ or ‘1’; not the values 0 or 1. The *evaluation* changes a ‘0’ to 0 and a ‘1’ to a 1.

This code shows that `defined` is evaluated, then `H` is expanded; not the other way around.

Handling Dubious Conditionals

```
#if defined(X)
# define M   3 <
# define Y   4
#else
# define M   3 ==
# define Y   0
#endif
#if M Y
  x += 2;
#else
  ++x;
#endif
```

The condition `M Y` is not a syntacticaly valid conditional. But expansion will always make it valid.

We should be able to handle this, since we can infer that, in all cases, the condition is syntactically valid.

Symbolic Evaluation on the CFG of Preprocessing

- The source code directives are parsed and transformed into a Control Flow Graph (CFG) where the nodes are blocks of lines.
- The edges are the unevaluated conditions found on if-directives.
- So no evaluation of the if-conditions is done at that point. It is not feasible to do so in general.
- All included files are processed once independently of the control flow.
- Loops may occur in the CFG as some files may recursively include themselves.

Symbolic Evaluation of C-like Preprocessing

7. **Procedure** $V(n, c_c)$ {
8. add c_c to condition list of node n;
9. test node n for possible infinite iteration;
10. **Case** node n
11. block of code: nothing to do;
12. define: add definition to top table of S;
13. if: Let c be its expanded/simplified condition

...

28. **Procedure** Merge(T, E, S, c) {
29. **For-each** variable x in T or E
30. Bind x with $c\,?\,v_t \diamond v_f$ into the top table of S
31. **where** v_t is $v(x, T : S)$,
32. v_f is $v(x, E : S)$
33. }

Line 13 is the main focus of the paper.

Conditional Values to the Rescue

A conditional value is denoted $c\,?\,e_1 \diamond e_2$, and means that if c is true, e_1 is the resulting value, otherwise it is e_2.

There are two classes of conditional values (c-values):

1. One class to bind preprocessing variables (macro) to a set of basic values under some conditions. These are unparsed and unevaluated c-values. $\mathcal{T}_C$
2. The other for parsing, substitution and evaluation of if-conditionals. These are parsed and evaluated c-values. $\mathcal{T}_{C\uparrow}$

Unparsed and Unevaluated Conditional Values $\mathcal{T}_C$

An unparsed and unevaluated conditional value (c-value) is in general a tree where the leaves are sequences of lexems and interior nodes are Boolean expressions.

In general, a preprocessing variable is bound to a c-value during symbolic evaluation.

Examples:

1. $(Y = 2)\,?\,$‘3’$\,\diamond\,$‘4’
2. $\widehat{\texttt{def}}(OFILE)\,?\,$‘`fprintf`’$\,\diamond\,$‘`printf`’
3. $(Y = 2)\,?\,(X = 3\,?\,$‘4’$\,\diamond\,$‘5’$)\,\diamond\,$‘6’

The Unparsed c-value Terms $\mathcal{T}_C$ and the Concrete (preprocessing) Terms $\mathcal{T}_P$

The unparsed c-value terms:

$$
\begin{array}{lcll}
\mathcal{T}_C & := & e & e \in \mathcal{T}_P \\
 & | & x_I & x_I \in \text{FPVar} \\
 & | & c\,?\,e_1 \diamond e_2 & c \in \mathcal{T}_B, e_1, e_2 \in \mathcal{T}_C \\
\text{FPVar} & := & \text{Free Preprocessing Variables} &
\end{array}
$$

The Unparsed c-value Terms $\mathcal{T}_C$ and the Concrete (preprocessing) Terms $\mathcal{T}_P$

The unparsed c-value terms:

$$
\begin{array}{lcll}
\mathcal{T}_C & := & e & e \in \mathcal{T}_P \\
 & | & x_I & x_I \in \text{FPVar} \\
 & | & c\,?\,e_1 \diamond e_2 & c \in \mathcal{T}_B, e_1, e_2 \in \mathcal{T}_C
\end{array}
$$

FPVar := Free Preprocessing Variables

The concrete terms:

$$
\begin{array}{lcll}
\mathcal{T}_P & := & \top & \\
 & | & \bot & \\
 & | & t_1 \ldots t_n & n > 0, t_i \in \text{VTok}
\end{array}
$$

VTok := Valid Tokens

The Parsed and Evaluated c-value Terms $\mathcal{T}_{C\uparrow}$

$$
\begin{array}{lcll}
\mathcal{T}_{C\uparrow} & := & e & e \in \mathcal{T}_B \\
 & | & \bar{\mathrm{Z}}(e) & e \in \mathcal{T}_{C\uparrow} \\
 & | & c\,?\,e_1 \diamond e_2 & c \in \mathcal{T}_B, e_1, e_2 \in \mathcal{T}_{C\uparrow}
\end{array}
$$

These are never bound to preprocessing variables, but are the result of expansion, parsing and evaluation of if-conditionals.

$\bar{\mathrm{Z}}(0) \rightarrow$ `false`

$\bar{\mathrm{Z}}(n) \rightarrow$ `true`, $n \neq 0$

$\bar{\mathrm{Z}}(r) \rightarrow \mathrm{error}, r$ not a number

The terms $\mathcal{T}_X$ after macro expansion

$$\begin{array}{lcll} \mathcal{T}_X & := & e & e \in \mathcal{T}_C \\ & | & e_1\ e_2 & e_1, e_2 \in \mathcal{T}_X \end{array}$$

A $\mathcal{T}_X$ term is essentially a sequence of $\mathcal{T}_C$

Example, Symbolic Evaluation

```
#if L == "french" || L == "spanish"
#define X iso_accents
#elif L = "english"
#define X ascii
#endif
....
set_input_function(X)
```

The general value of X is composed of several basic values, and these depend on L.

This can be expressed as

$(L = \texttt{"french"} \vee L = \texttt{"spanish"})\,?\,\texttt{`iso\_accents'} \quad \diamond$

$(L = \texttt{"english"}\,?\,\texttt{`ascii'} \diamond X_I)$

The final Boolean terms $\mathcal{T}_B$

$\mathcal{T}_B$	:=	b	$b \in$ BVal
	$\mid$	$\texttt{def}(x_I)$	$x_I \in$ FPVar
	$\mid$	$\neg e$	$e \in \mathcal{T}_B$
	$\mid$	$e_1\, o\, e_2$	$e_1, e_2 \in \mathcal{T}_B$, $o \in \{\wedge, \vee\}$
	$\mid$	$\bar{\mathrm{Z}}(e)$	$e \in \mathcal{T}_E$
	$\mid$	r	$r \in$ Err
def	:	CVal $\rightarrow$ BVal	
$\bar{\mathrm{Z}}$	:	ECst $\rightarrow$ BVal $\cup$ Err	
CVAL	:=	$\{\top, \bot, t_1 \ldots t_n\}$	
ECst	:=	Constants	
Err	:=	Errors	

Why Rewrite Systems?

- This formalism works well to express the type of transformations to apply.
- They are language independent and represent a simple formulation of algorithms.
- They are easier to use than a sequential algorithm to prove termination and confluence (if any!).

The General Transformation of an if-condition

$$\mathcal{T}_X \xrightarrow[\exp]{R_d} \mathcal{T}_X \xrightarrow{R_e} \mathcal{T}_{C\uparrow} \xrightarrow{PE} \mathcal{T}_{C\uparrow} \xrightarrow{\bar{\mathrm{Z}}} \mathcal{T}_{C\uparrow} \xrightarrow{R_f} \mathcal{T}_B$$

R_d: applied to the `defined` operators.

$\exp$: the expansion of macros (aka substitution).

R_e: transform into one c-value with no variables, or a list of tokens and free vars with no c-values.

$\bar{\mathrm{Z}}$: wrap everything into an evaluation function to `true` or `false`.

PE: parsing, evaluation of arithmetic operators.

R_f: simplification and transformation into a Boolean Expression.

Rewrite System R_d for Operator `defined`

During symbolic evaluation, the `defined` operators, used on if-conditions, are translated to $\widehat{\texttt{def}}$s and variables to their c-values.

Then they are rewritten according to the following system.

Rewrite System R_d for Operator `defined`

$$R_d = \left\{ \begin{array}{lcl} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \end{array} \right.$$

This rule traverses the tree to its leaves.

Rewrite System R_d for Operator `defined`

$$R_d = \begin{cases} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) & \rightarrow & \text{`}\texttt{0}\text{'} \\ \widehat{\texttt{def}}(\top) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) & \rightarrow & \text{`}\texttt{1}\text{'} \end{cases}$$

The next three rules process the concrete values of preprocessing.

Rewrite System R_d for Operator `defined`

$$R_d = \left\{ \begin{array}{rcl} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) & \rightarrow & \text{`}\texttt{0}\text{'} \\ \widehat{\texttt{def}}(\top) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(x_I) & \rightarrow & \texttt{def}(x_I)\,?\,\text{`}\texttt{1}\text{'} \diamond \text{`}\texttt{0}\text{'} \end{array} \right.$$

Rewrite System R_d for Operator `defined`

$$R_d = \left\{ \begin{array}{rcl} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) & \rightarrow & \text{`}\texttt{0}\text{'} \\ \widehat{\texttt{def}}(\top) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(x_I) & \rightarrow & \texttt{def}(x_I)\,?\,\text{`}\texttt{1}\text{'} \diamond \text{`}\texttt{0}\text{'} \end{array} \right.$$

The last rule is necessary to handle text processing of ‘1’ and ‘0’.

Rewrite System R_d for Operator `defined`

$$R_d = \begin{cases} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) & \rightarrow & \text{`}\texttt{0}\text{'} \\ \widehat{\texttt{def}}(\top) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(x_I) & \rightarrow & \texttt{def}(x_I)\,?\,\text{`}\texttt{1}\text{'} \diamond \text{`}\texttt{0}\text{'} \end{cases}$$

Example:

```
#define R(x) 2##x
#define H(x) R(x)
#if H(defined(W)) == 20
```

Rewrite System R_d for Operator `defined`

$$R_d = \begin{cases} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) & \rightarrow & \text{`}\texttt{0}\text{'} \\ \widehat{\texttt{def}}(\top) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(x_I) & \rightarrow & \texttt{def}(x_I)\,?\,\text{`}\texttt{1}\text{'} \diamond \text{`}\texttt{0}\text{'} \end{cases}$$

```
#define R(x) 2##x
#define H(x) R(x)
```

#if H($\widehat{\texttt{def}}(W_I)$) == 20

Rewrite System R_d for Operator `defined`

$$R_d = \begin{cases} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) & \rightarrow & \text{`0'} \\ \widehat{\texttt{def}}(\top) & \rightarrow & \text{`1'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) & \rightarrow & \text{`1'} \\ \widehat{\texttt{def}}(x_I) & \rightarrow & \texttt{def}(x_I)\,?\,\text{`1'} \diamond \text{`0'} \end{cases}$$

```
#define R(x) 2##x
#define H(x) R(x)

#if H(def(W_I) ? '1' ◇ '0') == 20
```

Rewrite System R_d for Operator `defined`

$$R_d = \begin{cases} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) & \rightarrow & \text{`}\texttt{0}\text{'} \\ \widehat{\texttt{def}}(\top) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(x_I) & \rightarrow & \texttt{def}(x_I)\,?\,\text{`}\texttt{1}\text{'} \diamond \text{`}\texttt{0}\text{'} \end{cases}$$

`#if` $2(\texttt{def}(W_I)\,?\,\text{`}\texttt{1}\text{'} \diamond \text{`}\texttt{0}\text{'})$ `== 20`

Rewrite System R_d for Operator `defined`

$$R_d = \left\{ \begin{array}{rcl} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) & \rightarrow & \texttt{`0'} \\ \widehat{\texttt{def}}(\top) & \rightarrow & \texttt{`1'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) & \rightarrow & \texttt{`1'} \\ \widehat{\texttt{def}}(x_I) & \rightarrow & \texttt{def}(x_I)\,?\,\texttt{`1'} \diamond \texttt{`0'} \end{array} \right.$$

`#if` $\texttt{def}(W_I)\,?\,\texttt{`21'} \diamond \texttt{`20'}$ `== 20`

Rewrite System R_d for Operator `defined`

$$R_d = \left\{ \begin{array}{rcl} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) & \rightarrow & \texttt{`0'} \\ \widehat{\texttt{def}}(\top) & \rightarrow & \texttt{`1'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) & \rightarrow & \texttt{`1'} \\ \widehat{\texttt{def}}(x_I) & \rightarrow & \texttt{def}(x_I)\,?\,\texttt{`1'} \diamond \texttt{`0'} \end{array} \right.$$

`#if` $\texttt{def}(W_I)\,?\,\texttt{`21==20'} \diamond \texttt{`20==20'}$

Rewrite System R_d for Operator `defined`

$$R_d = \begin{cases} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) & \rightarrow & \text{`}\texttt{0}\text{'} \\ \widehat{\texttt{def}}(\top) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(x_I) & \rightarrow & \texttt{def}(x_I)\,?\,\text{`}\texttt{1}\text{'} \diamond \text{`}\texttt{0}\text{'} \end{cases}$$

`#if` $\texttt{def}(W_I)\,?\,\texttt{false} \diamond \texttt{true}$

Rewrite System R_d for Operator `defined`

$$R_d = \begin{cases} \widehat{\texttt{def}}(c\,?\,e_1 \diamond e_2) & \rightarrow & c\,?\,\widehat{\texttt{def}}(e_1) \diamond \widehat{\texttt{def}}(e_2) \\ \widehat{\texttt{def}}(\bot) & \rightarrow & \text{`}\texttt{0}\text{'} \\ \widehat{\texttt{def}}(\top) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(t_1 \ldots t_n) & \rightarrow & \text{`}\texttt{1}\text{'} \\ \widehat{\texttt{def}}(x_I) & \rightarrow & \texttt{def}(x_I)\,?\,\text{`}\texttt{1}\text{'} \diamond \text{`}\texttt{0}\text{'} \end{cases}$$

`#if` $\neg\texttt{def}(W_I)$

The System R_e is applied over $\mathcal{T}_X$ before parsing and evaluation

$$R_e = \begin{cases} (c\,?\,e_1 \diamond e_2)\ t & \rightarrow \quad c\,?\,(e_1\ t) \diamond (e_2\ t) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ t\ (c\,?\,e_1 \diamond e_2) & \rightarrow \quad c\,?\,(t\ e_1) \diamond (t\ e_2) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ (c_1\,?\,e_1 \diamond e_2)(c_2\,?\,e_3 \diamond e_4) & \rightarrow \quad c_1\,?\,(c_2\,?\,(e_1\ e_3) \diamond (e_1\ e_4)) \\ & \qquad\quad (c_2\,?\,(e_2\ e_3) \diamond (e_2\ e_4)) \end{cases}$$

Note: two adjacent VTOK or FPVar terms remains the same.

The System R_e is applied over $\mathcal{T}_X$ before parsing and evaluation

$$R_e = \begin{cases} (c\,?\,e_1 \diamond e_2)\ t & \rightarrow \quad c\,?\,(e_1\ t) \diamond (e_2\ t) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ t\ (c\,?\,e_1 \diamond e_2) & \rightarrow \quad c\,?\,(t\ e_1) \diamond (t\ e_2) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ (c_1\,?\,e_1 \diamond e_2)(c_2\,?\,e_3 \diamond e_4) & \rightarrow \quad c_1\,?\,(c_2\,?\,(e_1\ e_3) \diamond (e_1\ e_4)) \\ & \qquad (c_2\,?\,(e_2\ e_3) \diamond (e_2\ e_4)) \end{cases}$$

Note: two adjacent VTOK or FPVar terms remains the same.
Example:

The System R_e is applied over $\mathcal{T}_X$ before parsing and evaluation

$$R_e = \begin{cases} (c\,?\,e_1 \diamond e_2)\ t & \rightarrow \quad c\,?\,(e_1\ t) \diamond (e_2\ t) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ t\ (c\,?\,e_1 \diamond e_2) & \rightarrow \quad c\,?\,(t\ e_1) \diamond (t\ e_2) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ (c_1\,?\,e_1 \diamond e_2)(c_2\,?\,e_3 \diamond e_4) & \rightarrow \quad c_1\,?\,(c_2\,?\,(e_1\ e_3) \diamond (e_1\ e_4)) \\ & \qquad\quad (c_2\,?\,(e_2\ e_3) \diamond (e_2\ e_4)) \end{cases}$$

Note: two adjacent VTOK or FPVar terms remains the same.
Example:

$$((Y = 1)\,?\,\text{‘2’} \diamond \text{‘3’})(== 1+)((X = 4)\,?\,\text{‘5’} \diamond \text{‘6’})$$

The System R_e is applied over $\mathcal{T}_X$ before parsing and evaluation

$$R_e = \begin{cases} (c\,?\,e_1 \diamond e_2)\ t & \rightarrow \quad c\,?\,(e_1\ t) \diamond (e_2\ t) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ t\ (c\,?\,e_1 \diamond e_2) & \rightarrow \quad c\,?\,(t\ e_1) \diamond (t\ e_2) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ (c_1\,?\,e_1 \diamond e_2)(c_2\,?\,e_3 \diamond e_4) & \rightarrow \quad c_1\,?\,(c_2\,?\,(e_1\ e_3) \diamond (e_1\ e_4)) \\ & \qquad\quad (c_2\,?\,(e_2\ e_3) \diamond (e_2\ e_4)) \end{cases}$$

Note: two adjacent VTOK or FPVar terms remains the same.
Example:

$$((Y = 1)\,?\,\text{`2 == 1 +'} \diamond \text{`3 == 1 +'})((X = 4)\,?\,\text{`5'} \diamond \text{`6'})$$

The System R_e is applied over $\mathcal{T}_X$ before parsing and evaluation

$$R_e = \begin{cases} (c\,?\,e_1 \diamond e_2)\ t & \rightarrow \quad c\,?\,(e_1\ t) \diamond (e_2\ t) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ t\ (c\,?\,e_1 \diamond e_2) & \rightarrow \quad c\,?\,(t\ e_1) \diamond (t\ e_2) \\ \textbf{where } t \in \text{VTok} \cup \text{FPVar} & \\ (c_1\,?\,e_1 \diamond e_2)(c_2\,?\,e_3 \diamond e_4) & \rightarrow \quad c_1\,?\,(c_2\,?\,(e_1\ e_3) \diamond (e_1\ e_4)) \\ & \qquad\quad (c_2\,?\,(e_2\ e_3) \diamond (e_2\ e_4)) \end{cases}$$

Note: two adjacent VTOK or FPVar terms remains the same.
Example:

$$(Y = 1)\,?\,(X = 4\,?\,\text{`2 == 1 + 5'} \diamond \text{`2 == 1 + 6'}) \diamond$$
$$(X = 4\,?\,\text{`3 == 1 + 5'} \diamond \text{`3 == 1 + 6'})$$

The System R_f is applied over $\mathcal{T}_{C\uparrow}$ after evaluation

$$
R_f = \left\{
\begin{array}{lcl}
\texttt{true} \,?\, e_1 \diamond e_2 & \longrightarrow & e_1 \\
\texttt{false} \,?\, e_1 \diamond e_2 & \longrightarrow & e_2 \\
c \,?\, \texttt{true} \diamond \texttt{false} & \longrightarrow & c \\
c \,?\, \texttt{false} \diamond \texttt{true} & \longrightarrow & \neg c \\
\ldots & & \\
\bar{\mathrm{Z}}(r) & \longrightarrow & r \\
\textbf{where } r \in \mathsf{Err} & & \\
\bar{\mathrm{Z}}(e_1 \, o \, e_2) & \longrightarrow & \bar{\mathrm{Z}}(e_1) \, o \, \bar{\mathrm{Z}}(e_2) \\
\textbf{where } o \in \{\wedge, \vee\} & & \\
\bar{\mathrm{Z}}(c \,?\, e_1 \diamond e_2) & \longrightarrow & c \,?\, \bar{\mathrm{Z}}(e_1) \diamond \bar{\mathrm{Z}}(e_2) \\
c \,?\, e_1 \diamond e_2 & \longrightarrow & c \wedge e_1 \vee \neg c \wedge e_2 \\
\textbf{where } e_1, e_2 \in \mathcal{T}_B & &
\end{array}
\right.
$$

Future Work

- Implement the approach in a open IDE, for example `emacs`, or `xemacs`.
- Design a language to let the user specify the possible variable values. It could be simple: list of values. Or complex: conditional values, regular expression, etc.

The End

Thank You

Questions, comments?