Verifier Design

Jenna Wise, Cameron Wong, Jonathan Aldrich, Johannes Bader, Éric Tanter

November 27, 2018

1 Verification Design

[OVERALL VERIFICATION DESIGN - TBD]

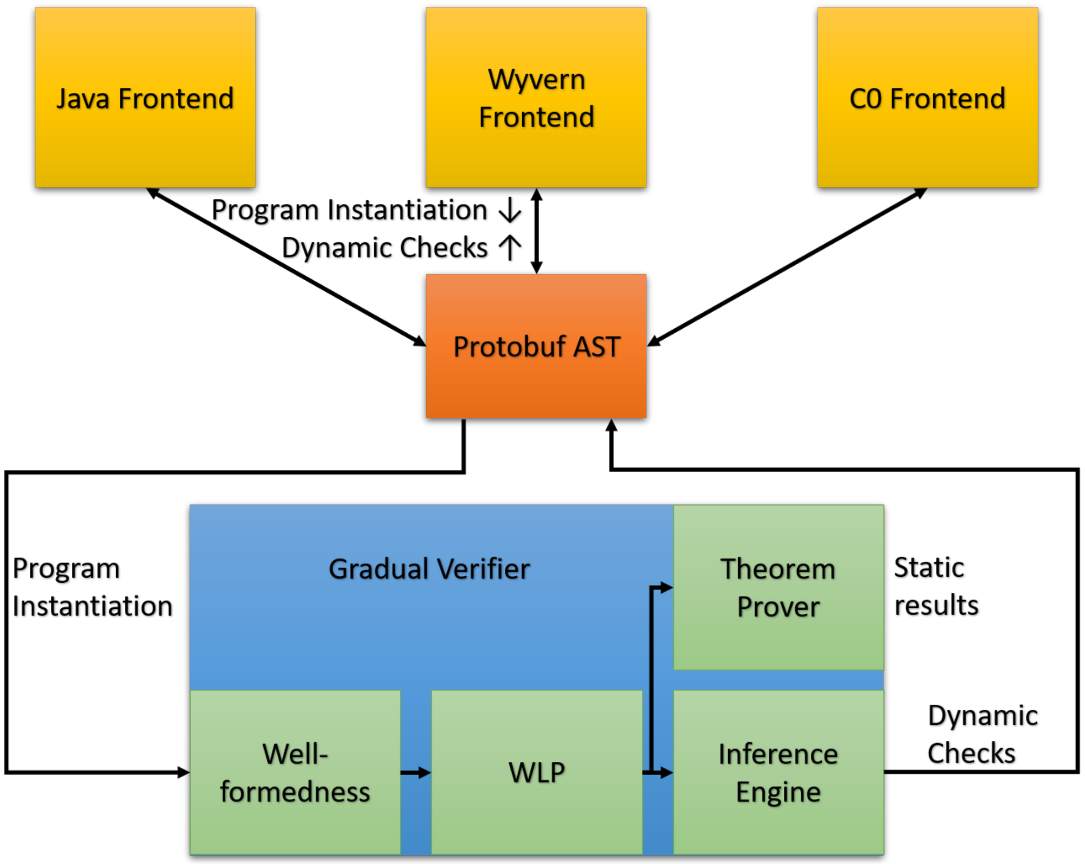

Figure 1: TBD - Not final version of diagram, changes coming, particularly to the WLP, Inference Engine, and Theorem Prover part of the diagram

1.1 Language Dependent Frontend

The gradual verifier supports multiple programming languages through protocol buffers [CITE protobuf] and language dependent frontends. Language dependent frontends must do the following:

- Check well-formedness of the program to be verified
  - This includes type-checking the program
- Use generated code from the protocol buffer message types of the gradual verifier's core language AST .proto file to
  - Translate the program to be verified into a program written in the gradual verifier's core language
  - Receive and execute dynamic checks determined necessary by the gradual verifier

Since the language dependent frontends must translate the program to be verified into a program written in the gradual verifier's core language, the programs and language features that can be verfied are restricted to the programs and language features that are supported by the verifier and its core language.

Frontends are available for the Java, Wyvern, and C0 programming languages which can be found at [LINK(S) HERE].

[DISCUSS ONE OF THEM HERE? - TBD]

1.2 Gradual Verifier

[SUMMARY OF GRADUAL VERIFIER TBD]

1.2.1 Core language

The gradual verifier is dependent on a core language seen in BNF grammar format in Figure 2. A language independent version of the core language grammar is available as protocol buffer message types in our Github repository (here).

Note that there are discrepancies between the core language grammar as seen in Figure 2 and its language independent counterpart, because we anticipate implementing additional language features for verification in the near future. The core language BNF grammar shows language features that can be verified currently, while the language independent counterpart shows language features that can be verified currently and additional features that will be verifiable in the future.

As seen in Figure 2, the core language is a simple object-oriented language with a nominal type system. Programs consist of classes and statements (recursively combined with ; into one statement acting as the "main" of a program). Classes contain publicly accessible fields and methods and have superclasses. Statements include the empty statement, statement sequences, combined variable declaration and assignment statements (including assignment of field values), if statements, field assignments, object creations, method calls, assertions, and special release and hold statements [REFERENCE APPROPRIATE SECTION FOR MORE INFORMATION ABOUT RELEASE & HOLD]. Expressions consist of variables, arithmetic expressions, field accesses, and integer, object, or null values. Methods have contracts consisting of pre- and postconditions, which are gradual formulas represented by $\widetilde{\phi}$. A gradual formula can be a concrete formula ϕ or an imprecise formula

$$
\begin{array}{rcll}
x, y, z & \in & \mathit{VAR} & \textit{(variables)} \\
v & \in & \mathit{VAL} & \textit{(values)} \\
e & \in & \mathit{EXPR} & \textit{(expressions)} \\
s & \in & \mathit{STMT} & \textit{(statements)} \\
o & \in & \mathit{LOC} & \textit{(object Ids)} \\
f & \in & \mathit{FIELDNAME} & \textit{(field names)} \\
m & \in & \mathit{METHODNAME} & \textit{(method names)} \\
C, D & \in & \mathit{CLASSNAME} & \textit{(class names)} \\
P & ::= & \overline{cls}\ s & \\
cls & ::= & class\ C\ extends\ D\ \{\overline{field}\ \overline{method}\} & \\
field & ::= & T\ f; & \\
T & ::= & int \mid C \mid \top & \\
method & ::= & T\ m(\overline{T\ x})\ contract\ \{s\} & \\
contract & ::= & requires\ \widetilde{\phi}\ ensures\ \widetilde{\phi} & \\
\oplus & ::= & + \mid - \mid * \mid \backslash & \\
\odot & ::= & \neq \mid = \mid < \mid > \mid \leq \mid \geq & \\
s & ::= & skip \mid s_1\ ;\ s_2 \mid T\ x := e \mid if\ (x \odot y)\ \{s_1\}\ else\ \{s_2\} \mid x.f := y \mid x := new\ C & \\
 & & \mid y := z.m(\overline{x}) \mid assert\ \phi \mid release\ \phi \mid hold\ \phi\ \{s\} & \\
e & ::= & v \mid x \mid e \oplus e \mid e.f & \\
x & ::= & result \mid id \mid old(id) \mid this & \\
v & ::= & n \mid o \mid null & \\
\phi & ::= & \text{true} \mid e \odot e \mid acc(e.f) \mid \phi * \phi & \\
\widetilde{\phi} & ::= & \phi \mid ? * \phi &
\end{array}
$$

Figure 2: The gradual verifier's core language in BNF grammar format

$? * \phi$ [1]. ϕ can be true, a binary relation between expressions, a heap accessibility predicate for field accesses (from implicit dynamic frames), or a separating conjunction $*$ of other ϕ (also from implicit dynamic frames) [REFERENCE IDF EXPLANATION SECTION].

Constructors are not explicitly provided in the core language, because they can be modeled with a method call to a specially defined method (this method implements the behavior of a constructor) following an allocation statement. Constructors return **void**, which can be modeled through returning an int (e.g. 0) and at call sites assigning the return value to a dummy variable [2]. Additionally, although subtyping relationships can be specified, inheritance and dynamic dispatch are

[1] $?$ is syntactic sugar for $? * \text{true}$

[2] **void** will likely be introduced as a type in the near future

not considered as language features [3]. Therefore, instance methods and their calls can be thought of as syntactic sugar on top of global functions; methods can be thought of as taking an additional parameter **this** and having an additional conjunctive form **this** $\neq$ **null** in its precondition. We leave the subtyping relationship declaration in the core language to note that language dependent frontends will need to translate their respective language to a language with a Top class that is a superclass of every other class (e.g. the Object class in Java).

1.2.2 Well-formedness

The gradual verifier will check the well-formedness properties of core language programs that are not already checked by a language dependent frontend. Therefore, the gradual verifier implements the well-formedness rules in Figure [REFERENCE FIGURE].

[EXPLANATION OF RULES]

[TBD]

1.2.3 Weakest Liberal Precondition Calculus

[TBD]

1.2.4 Static Verification

[TBD]

1.2.5 Dynamic Verification

[TBD]

[3]Inheritance and dynamic dispatch will be considered in the future