

Extended Static Checking for L3

Kumar Avijit and Swapnil V. Patil
kavijit,svp @cs.cmu.edu
15-745 (Spring 2006) Project Proposal

1 Proposal

Extended static checking (ESC) is a well known technique used in different languages, like Modula-3 [1], JAVA [3] to find errors in computer programs. The goal of ESC is to prove the absence of run-time errors (like out-of-bounds arrays, incorrect locking etc.) at compile-time.

Figure 1 shows the structure of an extended static checker. The programmer inserts checker-readable statements, called *annotations*, in the source. These annotations tell the checker about *preconditions* that need to be checked. For example, a precondition can be an equality (or an inequality) of the form `var = value`. Thus, when ESC checks this annotated routine, it ensures that these preconditions are true before proceeding (or entering that module); or, issue warnings if these preconditions cannot be verified. ESC analyzes this source program to generate *verification conditions* that are valid if and only if the program adheres to these conditions. These conditions are then submitted to the *theorem prover* to be checked for satisfiability.

In this project, we propose to build a simple ESC module for the L3 language. We believe that this will be a significant contribution for the future versions of L3 used in teaching compiler design. ESC will give beginners a simple tool to help debug their programs. Also, ESC does not need the source of the whole program to run the checker. A programmer can easily annotate the program to check any module inside the program. Finally, our goal is develop mechanically-checkable annotations that are easy for the programmer to use while checking their L3 source code.

2 Challenges

In this section we identify the key components of the ESC framework for the L3 language.

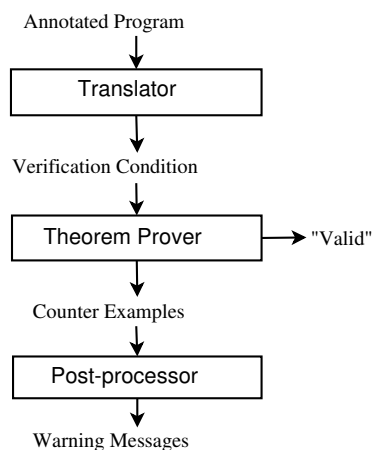


Figure 1: Components of a generic extended static checker (ESC).

2.1 Annotation Language

We propose to implement a simple annotation language with the following goals:

- **Safety:** The programmer may write annotations with his code and the typechecker decides whether the code satisfies the given properties.
- **Optimizations:** The compiler may use the annotations for optimization purposes. Here, we wish to include annotations that the compiler may follow without verifying them, and use them for further optimization.

We now briefly sketch the grammar of annotations:

$$\begin{aligned} \text{Annotations } \alpha & ::= \mathbf{assume} P \\ & \quad | \mathbf{verify} P \\ & \quad | \mathbf{invariant} P \\ \text{Predicates } P & ::= \langle var \rangle \langle binop \rangle \langle exp \rangle \\ & \quad | P_1 \wedge P_2 \\ & \quad | P_1 \vee P_2 \\ & \quad | \neg P \\ & \quad | \langle var_1 \rangle \# \langle var_2 \rangle \\ \langle stmt \rangle & ::= \dots | \mathbf{assume} P : \langle stmt \rangle | \langle stmt \rangle : \mathbf{verify} P \\ \langle control \rangle & ::= \dots | \mathbf{invariant} P : \mathbf{for}(\dots) | \mathbf{invariant} P : \mathbf{while}(\dots) \end{aligned}$$

For a Turing-complete language like L3, the problem of determining the value of a variable is undecidable. For the scope of this project, our annotation language will only support arithmetic expressions. In order to ensure that the verification of annotations remains undecidable, we do not plan to support recursion and function calls.

References present us with a potential problem due to aliasing. One way to avoid being overly conservative would be to allow the programmer to specify assumptions about aliased and non-aliased locations. The predicate $\langle var_1 \rangle \# \langle var_2 \rangle$ expresses such an assumption. Verification of such predicates would however, require pointer analysis, which is beyond the scope of this class project.

The user specified annotations can also act as refinements to types. We wish to formally describe the above framework by specifying its static and dynamic semantics, and prove type safety relative to the user-specified assumptions.

Example

Consider the following simple example to illustrate the use of annotations for optimization:

```
assume max = arr.length - 1
invariant i >= 0 ^ i < max : for (i=0; i<max; i=i+1){
    arr[i] = arr[foo(i)] + 1;
    ...
}
...
assume i = _ :
int foo (i:int){
    ...
}
verify ret = i + 1
```

The array indices for `arr` can be verified to lie within bounds using program annotations. The code for runtime bounds check can then be eliminated.

2.2 Theorem Prover

After the verification conditions have been generated, the ESC invokes a theorem prover that finds counterexamples to these conditions. The theorem prover needs to explore all possible cases to find these counterexamples.

Due to time constraints, our project will not focus on building a complicated theorem prover. Instead, we will either implement a very simple theorem prover or port an existing theorem prover to the L3 specification. The original ESC work on Modula 3 uses the *Simplify* theorem prover [2]. Simplify checks the validity of any conjecture by testing the satisfiability of the negated conjecture, using backtracking search, decision procedures and pattern-driven instantiations. For the sake of brevity, we omit details about the theorem prover; see [2] for a detailed description of the theorem prover.

References

- [1] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report Research Report 159, Compaq Systems Research center, December 1998.
- [2] D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Labs, July 2003.
- [3] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Cchecking for Java. In *ACM Programming Language Design and Implementation (PLDI '02)*, June 2002.