Delft University of Technology

Language Based Software Security
CS4280

Paper summary of "Dependent Types for Low-Level Programming"

Author:
Luka Janjić (4822218)

June 18, 2023

1 Introduction

This report summarizes and discusses the paper "Dependent Types for Low-Level Programming" [1]. Dependent types are a type system feature that allows types to depend on values, enabling more expressive and precise specifications. The authors present a framework for using dependent types in a low-level context, supporting mutation for variables and heap-allocated structures, accompanied by a technique for automatically deducing dependent types of local variables. They use this framework to implement Deputy, a tool for ensuring pointer bounds safety and safe tagged unions using a hybrid approach – combining static and dynamic analysis. They extend C's type system with annotations allowing programmers to express two kinds of type dependencies. Pointer types are extended to take two expressions representing their bounds in terms of other variables in scope, including the annotated variable itself. Union types are extended to take one additional "selector" expression per type argument. The type system then enforces the pointers to be within bounds, and unions' fields are used only when the appropriate selector allows. The analysis is carried out in two passes. First, the flow-insensitive type checker is run. In addition to regular type checking, it considers the bounds annotations and blindly inserts assertions before any line using or modifying pointer types, such that the program execution continues normally only if the bounds are respected. Second, a flow-sensitive optimizer is run on the output of the type checker simplifying or removing any previously inserted assertions that it can prove to always hold based on the control-flow structure of the program. The remaining assertions become compile-time instrumentation, while the rest are statically guaranteed to hold and are not emitted into the resulting code.
Here is a mapping of the assignment questions to the section that answers them:

- 1, 4 are answered in 2, up untill 2.4
- 2, 3, 5, 6, 9 are answered in 2.4
- 7, 8 are answered in 3

2 Main results

This paper discusses a general framework for implementing dependent types on top of low-level, C-like type systems, allowing expression and enforcement of dependencies among data that are not possible in the original type system. The framework is based on a hybrid approach combining static and dynamic analysis and operates in three passes which I present in the three following subsections. In the last subsection, I present and discuss Deputy, an instantiation of this framework to the problem of spatial memory safety. The results are presented in terms of a simple, generic imperative language.

2.1 Annotation inference

Only global variables are strictly required to be annotated. This is why the first pass consists of inferring any unspecified dependent types of local variables based on the annotations on global variables. This involves adding new variables to hold values of any missing annotation, whose values are automatically computed based on the context. The nice aspect of the technique for automatic dependency inference is the fact that it is independent of the actual dependent type used.

2.2 Type rules

The second pass the authors call a flow-insensitive type checker, however, this is not only an ordinary dependent type checker. Non-dependent types are checked in the usual way, but dependent types are not even attempted to be checked in the typical sense. Instead, dynamic checks, in the form of assertions, are inserted before any line where dependent typing is to be assured. The assertions enforce the condition required for the type to be satisfied in order for the program to continue execution, thus ensuring type safety in the presence of dependent types.

Inference rules for types are standard, with the exception of applying type constructors to type arguments. Due to the difficulty of perfectly resolving aliasing, the type arguments must be valid in an empty context, restricting the reference constructor ref :: type → type from being applied to dependent types:

$$\frac{\Gamma \vdash_L \tau_1 :: \text{type} \to \kappa \qquad \emptyset \vdash_L \tau_2 :: \text{type}}{\Gamma \vdash_L \tau_1\ \tau_2 :: \kappa} \quad \text{(Type-Type)}$$

In the above deduction rule, the subscript to the turnstile means that the inference is local – it only depends on local expressions. The authors define local expressions as any expressions not involving dereferencing. Typing of local expressions is standard.

Next, a bit of notation of the following form is introduced: $\Gamma \vdash e : \tau \Rightarrow \gamma$, meaning "in type environment Γ, e is a well-typed expression with type τ if γ is satisfied". This is used to define the assertions generated from the dependent types, where each expression is associated with a logical proposition representing the precondition for its well-typedness. In the case of simply typed expressions, this is simply the proposition "True". For compound expressions, such as those resulting from arithmetic operations, their conditions are merged into their conjunction. Dereferencing a pointer type propagates its condition.

Having set up the definitions for expressions, the compilation of statements is addressed. A new piece of notation is introduced, for which the authors use the same syntax as for statements. For disambiguation, I will add a subscript: $\Gamma \vdash c \Rightarrow_C c'$ stands for "In type environment Γ , command c compiles to c′, where c′ is identical to c except for added assertions". As indicated by the meaning of the notation, this defines how the input code is augmented by the assertions in order to preserve the types. Most of the rules are not particularly interesting, simply propagating the compilation and insertion of the assertions throughout the syntax tree. one rule stands out:

$$\frac{x \in \mathrm{Dom}(\Gamma) \qquad \Gamma \vdash y[e/x] : \tau_y[e/x] \Rightarrow \gamma_y \quad \forall (y : \tau_y) \in \Gamma}{\Gamma \vdash x := e \quad \Rightarrow_C \quad \mathrm{assert}(\bigwedge_{y \in \mathrm{Dom}(\Gamma)} \gamma_y);\ x := e} \quad \text{(Var-write)}$$

The remarkable aspect of this rule is the fact that it ensures the preservation of dependent types in the presence of mutable variables and is a key contribution of this type system. This rule says that when updating a variable x with the value of expression e, we must check all variables y in the current environment to see that their types still hold after substituting e for x. Essentially, it verifies that the assignment does not break any dependencies in the current scope. Moreover, the local-type restriction on base types of pointers ensures that there are no dependencies from the heap. This combination is what allows verifying mutation in the presence of dependent types.

There are two other rules which involve substitution, the rule for let-statements and the rule for heap allocation. However, since these two instructions introduce new variables they do not require checking the rest of the context, only substituting the assigned expression for the new variable in its own type. This ensures that the assignment satisfies any conditions expressed by the potentially dependent type of the new variable.

$$\frac{x \notin \mathrm{Dom}(\Gamma) \qquad \Gamma, x : \tau \vdash_L \tau :: \mathrm{type} \qquad \Gamma \vdash e : \tau[e/x] \Rightarrow \gamma \qquad \Gamma, x : \tau \vdash c \Rightarrow_C c'}{\Gamma \vdash \mathrm{let}\ x : \tau = e\ \mathrm{in}\ c \quad \Rightarrow_C \quad \mathrm{assert}(\gamma);\ \mathrm{let}\ x : \tau = e\ \mathrm{in}\ c'} \quad \text{(Let)}$$

$$\frac{x \notin \mathrm{Dom}(\Gamma) \qquad \emptyset \vdash_L \tau :: \mathrm{type} \qquad \Gamma \vdash e : \tau[e/x] \Rightarrow \gamma \qquad \Gamma, x : \mathrm{ref}\ \tau \vdash c \Rightarrow_C c'}{\Gamma \vdash \mathrm{let}\ x := \mathrm{new}\ \tau(e)\ \mathrm{in}\ c \quad \Rightarrow_C \quad \mathrm{assert}(\gamma);\ \mathrm{let}\ x := \mathrm{new}\ \tau(e)\ \mathrm{in}\ c'} \quad \text{(Alloc)}$$

Additional rules are presented for structures. The important aspect is the fact that the types of their fields may only depend on the values of their fields:

$$\frac{(f_1 : \tau_1, ...f_n : \tau_n) \vdash_L \tau_i :: \mathrm{type} \quad \forall\ 1 \leq i \leq n}{\Gamma \vdash_L \mathrm{struct}\{f_1 : \tau_1, ...f_n : \tau_n\} :: \mathrm{type}}$$

Essentially, this rule says that a type of a struct is valid under any context, as long as each of its fields is well defined in the context containing all of its fields. In other words, each field of a struct may only depend on its fields, itself included. The rest of the rules are rather messy and I will only describe them. Let ℓ be an expression of a struct type with n fields and $f_i : \tau_i$ is one of its fields. The Struct-read rule simply states that the expression of the form $\ell.f_i$, has the type $\tau_i[\ell.f_j/f_j]_{1 \leq j \leq n}$. In other words, any field names that the type depends on must be substituted for the actual value of the corresponding field. The remaining two rules are analogous to the rules for statements. The creation of a struct corresponds to the Let rule, substituting all the assigned values for their corresponding field names in all the field types. Writing into one of the struct's fields corresponds to the Var-write rule, where the substitution is carried out over all of the structs fields with the newly assigned value.

Note that no interesting conditions are inserted so far. The system is envisioned as a general and extensible framework that can handle nontrivial conditions introduced by additional type constructors. Later in the report, I will demonstrate a concrete instantiation applied to the enforcement of pointer bounds in the C language. A nontrivial constructor is introduced for the C-style arrays as pointers, which holds its bounds in addition to the type of its content.

2.3 Optimizer

At this point, the program is merely flooded with assertions with no regard to its structure, which could already be guaranteeing some of the conditions are always met. The third and final pass is the flow-sensitive optimizer, which attempts to prove the assertions obsolete by inspecting the control flow of the program. Any checks that it fails to remove, remain in the output program as compile-time instrumentation. Alternatively, the optimizer might deduce that a condition cannot be satisfied and report an error.

2.4 Deputy

Based on the presented framework, the authors implement Deputy, a dependent typing tool that allows programmers to annotate existing C code with more refined types. In particular, it allows relating a pointer with its bounds, and a union type with its tag, which is then enforced by the type system as presented above. This ensures the property that the literature refers to as spatial memory safety; the framework guarantees that no out-of-bounds accesses will occur throughout the program and that tagged unions are always accessed with respect to their tag.

The soundness of the simple type system presented in Section 2.2 is proven by the authors, guaranteeing that no dependency violation goes unnoticed by the analysis. This, however, is not the full story, since Deputy does not cover all the features of the underlying language, such as tracking dependencies of data on the heap and type casts [2]. In these cases, Deputy implicitly trusts the safety of these features, despite the fact that they can easily violate enforced properties. This means that some properties required by the dependent type annotations could be violated if the programmer carelessly uses these unsupported features: the analysis is sound with respect to the supported features, but not with respect to the language as a whole.

Now let us consider attacks to which Deputy provides resilience. Take for example a simple C function with a buffer overflow vulnerability.

```
1  void copyData(char* source) {
2      char buffer[10];
3      strcpy(buffer, source);  // Potential buffer overflow
4      printf("Copied string: %s\n", buffer);
5  }
```

The strcpy function can famously create a vulnerability to buffer overflow attacks because it does not consider any bounds of the provided input, nor the target buffer. This could be exploited to arbitrarily overwrite data surrounding the target buffer in memory, which can have a wide range of consequences from data corruption to full control exploits in case the buffer is on the stack. Similarly, a lack of bounds checks could be exploited to leak sensitive data which might be stored in an adjacent memory location to the buffer being read. Now consider a simple program using the annotated version of our function:

```
1  void copyData(char * COUNT(20) source) {
2      char buffer[10];          // type of buffer is 'char * COUNT(10)'
3      strcpy(buffer, source);   // static analysis gives an error here
4      printf("Copied string: %s\n", buffer);
5  }
6
7  int main() {
8      char input[20];           // type of input is 'char * COUNT(20)'
9      printf("Enter a string: ");
10     scanf("%s", input);       // potential malicious input
11     copyData(input);
12     return 0;
13 }
```

The only syntactic difference with the original function is in the annotation of the argument. However, now Deputy can statically deduce that this program suffers from a vulnerability due to the lack of bounds checks.

On the other hand, Deputy is not a fully precise tool either. There are cases in which Deputy will raise a false alarm, reporting a type error where in reality no violation of the type system can occur. In other words, the analysis is incomplete. To see this, consider the following snippet:

```
1  struct DynamicArray {
2      int size;
3      char * COUNT(size) data;
```

```
4 };
5 ...
6 void doubleSize(struct DynamicArray * arr) {
7     char * COUNT(arr->size * 2) new_data = malloc(2*size * sizeof(DynamicArray));
8     strncpy(new_data, arr->data, size);
9     arr->data = new_data;          // temporary violation of the dependent type
10    arr->size = 2 * arr->size;     // consistency reestablished
11 }
```

In this case, a property required by the dependent type of DynamicArray is temporarily violated in line 9, only to be reestablished in line 10. This, however, is not allowed by Deputy. The issue can be resolved by using multiple assignment, an extension proposed by the authors to remedy this class of problems.

Now consider a simple example based on the code of a Linux device driver [2]:

```
1 struct scull_qset {
2     char * COUNT(?) * data;
3     struct scull_qset * next;
4 };
5
6 struct scull_dev {
7     struct scull_qset * data;
8     int quantum, qset;
9 };
```

We run into two limitations of the analysis. On the one hand, due to the lack of support for dependent typing of data on the heap, Deputy cannot track the bounds of the array pointed to by the data field of the scrull_qset structure. On the other hand, even if this was handled, the bound of the data array could still not be expressed because dependencies between fields of different structures are unsupported.

While the lack these of features is unfortunate, the authors report that more often than not, the examples they used to test required minimal modifications of code. Moreover, the authors believe that the changes required by these limitations are anyway advisable for a higher likelihood of correctness. While this is a reasonable claim, some performance-critical code bases might not be able to afford that luxury.

3 Comparison with other techniques

As explained in Section 2 this work presents a hybrid approach, performing static analysis whenever possible and relying on dynamic analysis otherwise. In this section, I compare the presented approach with other techniques discussed in the course that could be used to tackle the same problem.

3.1 Dynamic analysis

Without the optimizer step, Deputy is effectively a dynamic analysis tool performing compile-time instrumentation. The advantage of this approach is the fact that the analysis can rely on runtime information to reduce false negatives and improve accuracy; tricky cases that could not be decided statically can be trivially decided in runtime when all the values are known. The main disadvantage here is the fact that this directly impacts performance. Checks are interspersed in the source code, adding and they execute every time an enforced property could be violated. In the case of programs that heavily rely on pointers, this could incur significant overhead. Considering the fact that the context of the application is low-level programming, where performance is often crucial and the use of pointers abides, this overhead is often not permissible. In comparison to existing solutions relying on dynamic analysis, this technique relies only on assertions and does not augment the definitions of structures containing arrays, as is the case with techniques using fat pointers [3] [4]. This gives them the advantage of preserving the original memory layout, which would require adapting the entire code base to work properly. Furthermore, with the use of the optimizer, the authors achieve a significant reduction of emitted dynamic checks in comparison to blindly instrumenting all potentially problematic locations. Alternatively, pointer bounds could be stored in a metadata table external to the program, preserving the memory layout. Such an approach is taken in the construction of SoftBound [5], which achieves complete spacial memory safety with a runtime overhead comparable to Deputy's.

An alternative purely dynamic approach is using runtime monitoring. While this is, generally speaking, a viable option, in the context of low-level programming this is not a particularly appealing option. The

overhead incurred from running a full virtual machine is out of the question, while a more lightweight runtime environment, perhaps only checking for spatial memory safety, would have to perform the same checks that Deputy does. While it would have access to runtime information to aid its optimization, it remains inconclusive whether such an approach would yield any significant results.

3.2 Abstract interpretation

The bounds-checking aspect of Deputy, along with its optimizer, bares some similarities to abstract interpretation with the abstract domain of intervals. While an abstract interpreter would keep track of the range of each value and array bounds, Deputy allows the bounds of arrays to be expressed in terms of an expression containing other variables in scope. In some cases, this has an identical effect. For example, suppose a pointer **int** $*$ ptr with a Deputy annotation COUNT(end $-$ ptr. If the abstract interpreter can infer that subtracting the ranges of end and ptr gives a strictly positive range, and from control flow the optimizer can deduce that end $-$ ptr is strictly positive, then a write to ptr will be guaranteed to be safe, with no runtime checks emitted by Deputy and no errors reported by the interpreter. In other cases, they can have different outcomes. Continuing with the previous example, if it can not be deduced from the runtime that end $-$ ptr > 0, then the abstract interpreter will report an issue since it cannot statically prove the safety of the code. On the other hand, Deputy will simply leave a dynamic check immediately before the potentially problematic instruction. This illustrates the flexibility of the solution quite well. While striving for statically enforced safety, anything that is disputed is left to be checked in runtime. Depending on the specific domain of application this could be a viable solution.

3.3 Dependent types

Several other techniques employing dependent types for imperative programming are mentioned in the paper. In [6] and [7], expressions appearing in dependent types are different from program expressions and must be decidable at compile time. In [8], arbitrary expressions from the same language are allowed, and thus the type system may be undecidable. In the presented paper, the authors attempt to find a middle ground, allowing expressive annotations in the source language while using run-time checks to keep the type checker simple and decidable, all while allowing mutable variables to appear in dependent types, which is unsupported by the mentioned techniques.

4 Conclusion

In conclusion, the hybrid approach developed by the authors seems like a well-balanced tradeoff between the advantages of static and dynamic analysis. In their experiments, the authors show that in a typical case, the output code produced by their technique has a small number of dynamic checks relative to the size of the input program: less than 11% of the lines were changed overall, with 2-4% in most cases. Moreover, this incurred a rather acceptable runtime overhead, ranging between 24% increase up to 98% in the worse case, making it competitive with other analysis tools targeting spacial memory safety.

References

[1] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C Necula. Dependent types for low-level programming. In *Programming Languages and Systems: 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007. Proceedings 16*, pages 520–535. Springer, 2007.

[2] Jeremy Paul Condit. *Dependent types for safe systems software*. University of California, Berkeley, 2007.

[3] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27 (3):477–526, 2005.

[4] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[5] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[6] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, 1999.

[7] Hongwei Xi. Imperative programming with dependent types. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 99CB36332)*, pages 375–387. IEEE, 2000.

[8] Lennart Augustsson. Cayenne—a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, 1998.