Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 324 (2016) 3–13

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Týr: A Dependent Type System for Spatial Memory Safety in LLVM

Vítor Bujés Ubatuba De Araújo[a], Álvaro Freitas Moreira[a] and Rodrigo Machado[a]

[a] *Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)*
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

Abstract

This work proposes a dependent type system for the LLVM Intermediate Representation language for keeping track of pointer bounds information. The system employs a combination of static analysis and runtime checks to avoid spatial memory safety violations, such as buffer overflows. By working on LLVM IR, the system serves a foundation for ensuring spatial memory safety in languages which can be compiled to LLVM, such as C and C++.

Keywords: Spatial memory, dependent types, type safety, LLVM

1 Introduction

The C and C++ programming languages do not enforce spatial memory safety: they do not ensure that memory accessed through a pointer to an object in memory, such as an array, actually belongs to that object. Rather, the programmer is responsible for keeping track of allocations and bounds information and ensuring that only valid memory accesses are performed by the program. On the one hand, this provides flexibility: the programmer has full control over the layout of data in memory, and when checks are performed. On the other hand, this is a frequent source of bugs in C and C++ programs, with consequences varying from crashes and silent data corruption to security vulnerabilities, such as Heartbleed [7].

A number of techniques have been proposed to provide memory safety in C [8,11,10]. Typically such systems keep their own bounds information and instrument the program to ensure that memory safety is not violated. A different approach, named Deputy [6], employs dependent types to enable the programmer to annotate the already existing bounds information kept manually in C programs, so that the compiler can identify it and insert checks which ensure its correct usage. Because the compiler-inserted checks use the same metadata as the checks written by the

http://dx.doi.org/10.1016/j.entcs.2016.09.003

programmer, compiler optimizations can often prove inserted checks redundant, thus reducing the performance penalty of the technique. Deputy uses the CIL framework [12] for program transformation. Since CIL does not support C++, Deputy cannot be used to ensure memory safety of C++ programs.

This work proposes a new system, called Týr, which provides functionality similar to Deputy for LLVM [9], a language-agnostic compilation framework designed around a typed assembly-like language (LLVM IR) for an abstract machine. Týr consists of a dependent type system for LLVM IR, which enables associating pointers with their correspondent metadata, and a set of transformations over LLVM IR code to make it spatially memory-safe. By providing this functionality at the LLVM IR level, Týr can be used as a foundation for spatial memory safety for compilers which target LLVM, such as the Clang [5] C/C++ compiler.

This paper is organized as follows. Section 2 presents background for this work. Section 3 describes the LLVM IR language. Section 4 describes the proposed system, how it fits in the LLVM environment, and the type rules and transformations it implements. Section 5 presents conclusions and future work.

2 Background

2.1 Memory safety

Broadly speaking, a program is said to be memory-safe if it only accesses regions of memory currently allocated to some program object. A language or environment is said to be memory-safe if it ensures that its programs are memory-safe.

There are different kinds of memory safety. *Temporal memory safety* refers to the property of not attempting to access memory that has been deallocated or not allocated yet, while *spatial memory safety* refers to the property of not attempting to access memory outside of the bounds of an object. The methods for ensuring each kind are different. This work addresses only spatial memory safety. Other mechanisms, such as conservative garbage collection [4], can be used complementarily to ensure temporal memory safety in C.

Works also vary in the granularity at which memory safety is considered. Some works, such as [3], consider whole data structures as the basic unit for memory, so that access past the limits of a field in a data structure is not considered a violation if the access is still within the same data structure. This limits the damage caused by buffer overflows or overreads, but does not completely eliminate them. Other works, such as Deputy, address memory safety at the level of single variables and structure fields. The present work uses this more strict version of memory safety.

2.2 Dependent types

A dependent type system [1] is one in which types can be indexed by terms at the value level. For instance, dependent types make it possible to define a type $Vec\langle\tau, n\rangle$ for vectors of n elements of type τ. By allowing types to be parameterized by values, rather than only other types as in the case of parametric polymorphism,

dependent types enable us to assign richer and more precise types to programs, thus allowing more properties of programs to be mechanically verified, either statically or at run-time.

Dependent type systems vary in their degree of expressivity. Some dependently-typed programming languages allow any expression to appear in types, which may lead to undecidability because an expression may fail to terminate [2]. Other languages allow a subset of expressions to be used in types, thus avoiding undecidability at the expense of expressivity.

Most dependently-typed languages require type checking to happen entirely at compile-time. This is usually accomplished by either restricting type expressions to make them more amenable to static type checking, or by building some theorem proving mechanism into the language. By contrast, some systems allow type checking to be postponed to run-time when they are unable to check some property at compile-time. In this way, they provide greater flexibility, while trading off static guarantees.

2.3 Deputy

Deputy [6] is a dependent type system for the C programming language which enables programmers to add annotations relating pointers to their bounds. For instance, a function might be declared as:

```
int f(int * COUNT(len) array, int len)
```

meaning that the `array` parameter of function `f` is a pointer to a region of `len` integers. When the program is compiled, Deputy inserts assertions in the code which ensure that pointers are within the declared bounds before they are used. A second compilation pass then looks for assertions which can be proven true at compile time, and thus can be safely removed, thus reducing the performance impact of the checks. It also looks for assertions which can be proven false at compile time, which are reported as compile-time errors to the programmer, thus providing static checks when possible. Expressions which can appear in a dependent type are limited to local variables, constants, and arithmetic expressions, which is usually sufficient for describing bounds in C programs.

2.4 The LLVM Compiler Infrastructure

LLVM is a widely used language-agnostic framework for program compilation, analysis and transformation designed around a uniform Intermediate Representation (LLVM IR), a typed assembly-like language for an abstract machine. Various backends exist for translating LLVM IR to machine code of different architectures. The use of a uniform, well-defined language for code representation makes it relatively easy to extend LLVM with new analysis and transformation passes.

Clang is the C/C++ compiler provided by the LLVM Project. It takes C/C++ source code and emits LLVM IR, which is then optimized and translated to machine code by LLVM. Figure 1 shows an example code snippet in C and how it might

```
int f(int *x, int i) {
  int result;

  if (i < 0)
    result = i+i;
  else
    result = x[i];

  return result;
}
```

```
define i32 @f(i32* %x, i32 %i) {
        %result = alloca i32
        %1 = cmp slt %i, 0
        br %1, %then, %else

 then:  %2 = add %i, %i
        store %2, %result
        br %end

 else:  %3 = getelementptr %x, %i
        %4 = load %3
        store %4, %result
        br %end

 end:   %5 = load %result
        ret %5
}
```

Fig. 1. Sample C function and equivalent LLVM IR code

be represented in LLVM. [1] The LLVM IR language is explained in greater detail in Section 3.

3 LLVM Intermediate Representation language

LLVM IR is a typed assembly-like language for an abstract machine with an infinite number of registers. The LLVM IR language is in Static Single Assignment (SSA) form with respect to its registers: each register is assigned exactly once, and each definition dominates all of its uses. Memory access is done through typed pointers. The SSA restriction applies only to registers, not to memory locations.

The basic unit of compilation in LLVM is the *module*. A module contains variable, function, constant and type definitions. A function definition includes its name, the names and types of its parameters, its return type, and the function body, which is composed of one or more *basic blocks*. A basic block is a sequence of instructions ended by a *terminator*, an instruction which transfers control to another basic block (branching) or to the function caller (returning). A basic block may be preceded by a label, which can be used with the branching instruction.

For simplicity of exposition, we define a relevant subset of the LLVM IR language for use in this work (Figure 2). It is similar to the Vminus subset used by [13] in formalizing LLVM semantics, but also includes pointers, memory access and pointer arithmetic, and omits ϕ-nodes, which will be discussed later. It defines the types `i1` (1-bit integers, i.e., booleans), `i32` (32-bit integers, a prototypical integer type), and a constructor for pointer types ($*$). Values which can appear as instruction operands in this language are boolean and integer constants and registers. Registers are prefixed with `%`, and may be given either symbolic names (`%var`) or numbers (`%1`).

The arithmetic instructions (of which a `add` is a prototypical example) behave as usual. The `cmp` instruction performs a specified comparison (such as `slt`, *signed less than*) between two values and yields a boolean. `load` takes a pointer ($\tau*$) to a memory location and yields its value. `store` stores the value of its first operand into the memory location pointed by the second one. `alloca` τ allocates memory

[1] Explicit types are required before every operand in LLVM IR (e.g., `i32 0` rather than plain `0`), but they have been omitted for clarity of exposition.

Types:	τ	::=	`i1` | `i32` | τ*
Constants:	*const*	::=	`false` | `true` | `0` | `1` | `2` | ...
Registers:	*reg*	::=	`%a` | ...
Values:	*val*	::=	*reg* | *const*
Commands:	*cmd*	::=	*reg* = `add` val_1, val_2
		|	*reg* = `cmp` *op* val_1, val_2
		|	*reg* = `load` *val*
		|	*reg* = `store` val_1, val_2
		|	*reg* = `getelementptr` val_1, val_2
		|	*reg* = `alloca` τ
Terminators:	*term*	::=	`ret` *val*
		|	`br` val, lab_1, lab_2
Instructions:	*inst*	::=	*cmd* | *term*
Basic blocks:	*blk*	::=	[*name* :] $\overline{cmd}$ *term*
Function bodies:	*body*	::=	`{`$\overline{blk}$`}`

Fig. 2. Subset of LLVM IR considered in this work

for a value of type τ in the stack and yields a pointer to it.

`getelementptr` is the instruction for typed pointer arithmetic: it takes a pointer p and an offset n and returns a pointer to the nth position after p, taking in account p's type (i.e., an offset of 1 into a pointer to a 32-bit integer is actually 4 bytes). This directly corresponds to the behavior of pointer arithmetic in the C and C++ languages.

The branching instruction (`br`) takes a boolean and two labels, and jumps to either label depending on the value of the boolean. `br` *lab* is an unconditional branch to *lab*, and can be regarded as syntactic sugar for `br true`, *lab*, *lab*. `ret` returns to the function caller with the specified value.

4 Týr

We propose a new system, called Týr, which augments LLVM IR with dependent pointer types. This section describes the design decisions of the proposed system, the type rules it introduces, and the LLVM IR transformations it performs.

4.1 Design

Týr is implemented as a type system and transformation at the LLVM IR level. Although Týr is intended primarily for use with C and C++, in principle it can be applied to any language that compiles to LLVM IR. Moreover, LLVM IR is more uniform and has a smaller number of constructs than C or C++, which makes it simpler to analyze and transform.

Týr is composed of two passes. The Týr-1 pass (Figure 3) takes an LLVM IR program (typically generated by Clang from a C/C++ source file), and the dependent type information provided by the programmer in the form of annotations. The

result of Týr-1 is a new LLVM IR program augmented with run-time bounds checking. The program also carries tracing information which allows identifying the true and false branches of any inserted check. This tracing information has the form of fake function calls of the form `@.tyr.true.branch (id)` and `@.tyr.false.branch (id)`, where *id* is a unique identifier for each inserted check. Using calls to no-operation functions as a way to associate metadata with code is standard practice in LLVM, used for instance to associate debugging and liveness information with local variables. Such calls do not appear in the generated machine code.

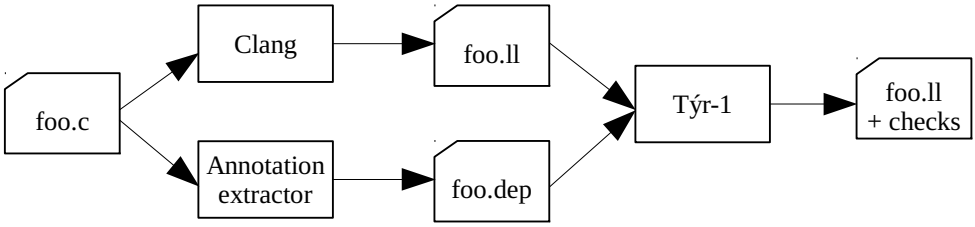

Fig. 3. Týr-1 pass: Check insertion

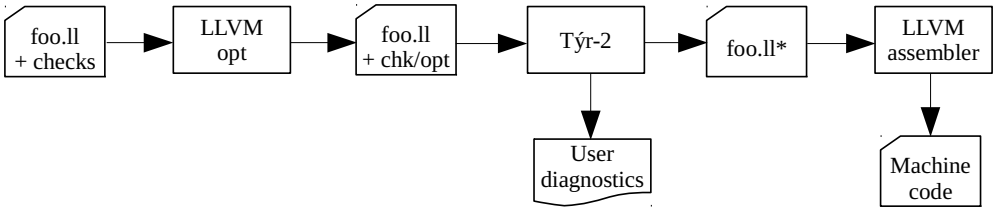

Fig. 4. Týr-2 pass: Compile-time diagnostics

This program with run-time checks and tracing information is passed on to LLVM for optimization. It is expected that the optimizations performed by LLVM will remove checks when it can prove them to be always true or always false. For instance, if LLVM can prove that a check condition is always true, it will simplify the program by eliminating the check and the false branch, and analogously when a condition is always false. Because the true and false branches of all inserted checks contain a function call to `@.tyr.true.branch` and `@.tyr.false.branch`, respectively, we can identify that a condition was proved by the compiler to be always true or always false by looking for calls to `@.tyr.true.branch` without the corresponding `@.tyr.false.branch` or vice-versa.

An inserted check that is always true is a redundant check, which can be safely removed without affecting the correctness of the program. On the other hand, an inserted check which is proved to be always false represents a memory-safety violation detected at compile time, i.e., a condition which was required to ensure spatial memory safety at run-time has been proved at compile time to never hold. The Týr-2 pass (Figure 4) looks for such occurrences in the optimized programs, and reports them as compile-time errors to the user. If no such occurrence is found, the program is accepted, the tracing function calls are removed, and the program is

Types:

$$\begin{array}{lll} \tau & ::= & \ldots \\ & | & Ptr\langle \tau, \delta_1, \delta_2 \rangle \quad \text{(bounded pointer)} \\ & | & LocalVar\langle \tau \rangle \quad \text{(variable in stack)} \end{array}$$

Dependent type expressions:

$$\begin{array}{lll} \delta & ::= & val \quad \text{(constant or register)} \\ & | & \delta_1 + \delta_2 \quad \text{(integer arithmetic)} \\ & | & \delta_1 \oplus \delta_2 \quad \text{(pointer arithmetic)} \\ & | & \delta! \quad \text{(local pointer dereference)} \end{array}$$

Fig. 5. Dependent type constructors introduced by Týr and the expressions which can appear within a dependent type

$$\textsc{type-int}\ \frac{}{\vdash i32 :: type} \qquad \textsc{type-localvar}\ \frac{\Gamma \vdash \tau :: type}{\Gamma \vdash LocalVar\langle \tau \rangle :: type} \qquad \textsc{type-ptr}\ \frac{\emptyset \vdash \tau :: type \quad \Gamma \vdash_L lo : Ptr\langle \tau, \_, \_ \rangle \quad \Gamma \vdash_L hi : Ptr\langle \tau, \_, \_ \rangle}{\Gamma \vdash Ptr\langle \tau, lo, hi \rangle :: type}$$

Fig. 6. Rules for well-formed types

passed on to the LLVM assembler to generate a machine-code executable.

4.2 Type system

We introduce dependent types in the LLVM IR language by replacing the $*$ pointer type constructor with two new type constructors (Figure 5). $Ptr\langle \tau, lo, hi \rangle$ represents a pointer to a sequence of elements of type τ between the addresses lo (inclusive) and hi (exclusive). $LocalVar\langle \tau \rangle$ represents a pointer to a local variable of type τ in the stack created by the `alloca` instruction. The kinds of expressions which can appear as bounds for a pointer type are limited to constants, registers, local variables in the stack, and arithmetic expressions involving both integers and pointers. It excludes arbitrary pointers to non-local data, because otherwise keeping track of mutations to depended values would require perfect aliasing information, i.e., being able to tell statically whether any two pointers point to the same memory region.

Figure 6 presents the rules for well-formedness of types. The first two rules are trivial: they say that $i32$ is a valid type, and that one can construct a $LocalVar\langle \tau \rangle$ type from any type τ. The third rule expresses that a pointer of type τ must be bounded by other pointers of type τ, which may be expressions using the local environment Γ, but the base type of the pointer must be valid in an empty environment. This restriction, already present in Deputy, is necessary because general (non-$LocalVar$) pointers can escape the function they are created in, and therefore the scope of local variables; moreover, we would need perfect aliasing information to ensure we can find all such escaping pointers when their type is invalidated through mutation of a local variable it depends on.

4.3 Type checking

Figure 7 shows the type rules for expressions that can appear within a dependent type. Figure 8 presents the type rules for dependent pointer types. LLVM-PTR-

$$\textsc{local-int}\ \frac{}{\vdash_L n : i32} \qquad \textsc{local-reg}\ \frac{\Gamma(\%r) = \tau}{\Gamma \vdash_L \%r : \tau} \qquad \textsc{local-add}\ \frac{\Gamma \vdash_L e_1 : i32 \qquad \Gamma \vdash_L e_2 : i32}{\Gamma \vdash_L e_1 + e_2 : i32}$$

$$\textsc{local-ptr-arith}\ \frac{\Gamma \vdash_L e_1 : Ptr\langle\tau, lo, hi\rangle \qquad \Gamma \vdash_L e_2 : i32}{\Gamma \vdash_L e_1 \oplus e_2 : Ptr\langle\tau, lo, hi\rangle} \qquad \textsc{local-localvar-deref}\ \frac{\Gamma \vdash_L e : LocalVar\langle\tau\rangle}{\Gamma \vdash_L e! : \tau}$$

Fig. 7. Type rules for dependent type expressions

$$\textsc{llvm-int}\ \frac{}{n : i32} \qquad \textsc{llvm-reg}\ \frac{\Gamma(\%r) = \tau}{\Gamma \vdash \%r : \tau} \qquad \textsc{llvm-int-arith}\ \frac{\Gamma \vdash val_1 : i32 \qquad \Gamma \vdash val_2 : i32}{\Gamma \vdash \%r = \texttt{add}\ val_1, val_2 : i32}$$

$$\textsc{llvm-cmp-int}\ \frac{\Gamma \vdash val_1 : i32 \qquad \Gamma \vdash val_2 : i32}{\Gamma \vdash \%r = \texttt{cmp}\ \text{OP}\ val_1, val_2 : i1} \qquad \textsc{llvm-cmp-ptr}\ \frac{\Gamma \vdash val_1 : Ptr\langle\tau, \_, \_\rangle \qquad \Gamma \vdash val_2 : Ptr\langle\tau, \_, \_\rangle}{\Gamma \vdash \%r = \texttt{cmp}\ \text{OP}\ val_1, val_2 : i1}$$

$$\textsc{llvm-ptr-arith}\ \frac{\Gamma \vdash val_1 : Ptr\langle\tau, lo, hi\rangle \qquad \Gamma \vdash val_2 : i32}{\Gamma \vdash \%r = \texttt{getelementptr}\ val_1, val_2 : Ptr\langle\tau, lo, hi\rangle}$$

$$\textsc{llvm-load}\ \frac{\Gamma \vdash val : Ptr\langle\tau, lo, hi\rangle}{\Gamma \vdash \%r = \texttt{load}\ val : \tau \Rightarrow val \neq 0 \wedge val \geq lo \wedge val < hi} \qquad \textsc{llvm-store}\ \frac{\Gamma \vdash val_1 : \tau \qquad \Gamma \vdash val_2 : Ptr\langle\tau, lo, hi\rangle}{\Gamma \vdash \texttt{store}\ val_1, val_2 \Rightarrow val_2 \neq 0 \wedge val_2 \geq lo \wedge val_2 < hi}$$

$$\textsc{llvm-local-load}\ \frac{\Gamma \vdash val : LocalVar\langle\tau\rangle}{\Gamma \vdash \%r = \texttt{load}\ val : \tau} \qquad \textsc{llvm-local-store}\ \frac{\Gamma \vdash val_1 : \tau \qquad \Gamma \vdash val_2 : LocalVar\langle\tau\rangle \qquad \forall(\%r : \tau_r) \in \Gamma.\ \Gamma \vdash \%r : [val_1/!val_2]\tau_r \Rightarrow \gamma_r}{\Gamma \vdash \texttt{store}\ val_1, val_2 \Rightarrow \wedge_{\%r \in Dom(\Gamma)} \gamma_r}$$

Fig. 8. Type rules for pointer usage

ARITH ensures that pointer arithmetic propagates the bounds of the original pointer. LLVM-LOAD indicates that one can load from a pointer of type $Ptr\langle\tau, lo, hi\rangle$, subject to a *run-time check*, indicated by the symbol $\Rightarrow$, that the pointer is non-null and within the declared bounds. LLVM-STORE is similar. Note that pointer arithmetic is allowed to move a pointer beyond its declared bounds; the validity of the pointer is checked only when the pointer is used to load or store values. This is because C/C++ programs often generate a pointer one past the last element of an array when iterating over its elements, and this is allowed by the language standard as

$$
\begin{array}{l}
\text{Compute}(\delta) = \\
\quad \text{case } \delta \text{ of} \\
\qquad val \quad \Rightarrow ([], val) \\
\qquad \delta_1 + \delta_2 \Rightarrow \text{let } (insts_1, \texttt{\%r}_1) = \text{Compute}(\delta_1) \\
\qquad\qquad\qquad\quad (insts_2, \texttt{\%r}_2) = \text{Compute}(\delta_2) \\
\qquad\qquad\qquad \text{in } (insts_1 \mathbin{+\!\!+} insts_2 \mathbin{+\!\!+} [\texttt{\%r}_3 \texttt{ = add \%r}_1, \texttt{\%r}_2], \texttt{\%r}_3) \\
\qquad \delta_1 \oplus \delta_2 \Rightarrow \text{let } (insts_1, \texttt{\%r}_1) = \text{Compute}(\delta_1) \\
\qquad\qquad\qquad\quad (insts_2, \texttt{\%r}_2) = \text{Compute}(\delta_2) \\
\qquad\qquad\qquad \text{in } (insts_1 \mathbin{+\!\!+} insts_2 \mathbin{+\!\!+} [\texttt{\%r}_3 \texttt{ = getelementptr \%r}_1, \texttt{\%r}_2], \texttt{\%r}_3) \\
\qquad \delta! \quad \Rightarrow \text{let } (insts_1, \texttt{\%r}_1) = \text{Compute}(\delta) \\
\qquad\qquad\qquad \text{in } (insts_1 \mathbin{+\!\!+} [\texttt{\%r}_2 \texttt{ = load \%r}_1], \texttt{\%r}_2)
\end{array}
$$

$$
\begin{array}{l}
\text{Check}(\gamma,\, oklabel) = \\
\quad \text{case } \gamma \text{ of} \\
\qquad true \quad \Rightarrow [\texttt{br true}, oklabel, oklabel] \\
\qquad \delta_1 \text{ OP } \delta_2 \Rightarrow \text{let } (insts_1, \texttt{\%r}_1) = \text{Compute}(\delta_1) \\
\qquad\qquad\qquad\quad\;\; (insts_2, \texttt{\%r}_2) = \text{Compute}(\delta_2) \\
\qquad\qquad\quad \text{in } insts_1 \mathbin{+\!\!+} insts_2 \mathbin{+\!\!+} [\ \texttt{\%}r_3\texttt{=cmp OP \%}r_1, \texttt{\%}r_2, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{br \%r}_3, oklabel, fail] \\
\qquad \gamma_1 \wedge \gamma_2 \quad \Rightarrow \text{Check}(\gamma_1, rest) \mathbin{+\!\!+} [rest :] \mathbin{+\!\!+} \text{Check}(\gamma_2, oklabel)
\end{array}
$$

Fig. 9. Algorithm for generating LLVM IR code to compute preconditions

long as the pointer is not dereferenced.

Rules for local variables are different. Loading from a local variable is guaranteed to succeed, so no check is required. On the other hand, storing to a local variable might violate an invariant expressed in dependent types involving that variable. For instance, if p is declared as a pointer to an array of n elements, changing n might invalidate the type of p. To ensure this does not happen, LLVM-LOCAL-STORE checks that all invariants that were valid before the store operation are still valid if the variable is replaced by its new value.

The algorithm for generating the instruction sequences corresponding to the run-time checks (Figure 9) can be described in terms of two functions. $Compute(\delta)$ takes a dependent type expression and returns two values: a sequence of instructions which computes the value δ, and the register where the computed value can be found. $Check(\gamma, oklabel)$ takes a precondition γ and generates a sequence of instructions which test whether γ is true and branch to *oklabel* if that is the case. Otherwise, the generated sequence of instructions jumps to a failure-handling block, which must be defined appropriately. The *oklabel* argument is used to chain sequences of tests when computing conjunctions of preconditions.

5 Conclusion and future work

This work explores an approach based on dependent types to bring spatial memory safety to C and C++ by allowing programmers to make the bounds information already latent in C and C++ programs explicit. Unlike previous works, our system aims to support both C and C++ by working at a lower level language to which both can be compiled.

Týr is a work in progress. For the system to be useful in real C programs, some extensions must be made. The most important of these is support for function calls. The rules for binding parameters to arguments when a function is called are similar to those of assignment to a local variable. Another useful extension is support for dependencies among fields of data structures, allowing for instance declaring that a field represents the length of a buffer pointed to by another field.

A LLVM-specific extension is support to ϕ-nodes, used in SSA-form to represent variables which may be assigned different values depending on control flow. C code emitted by Clang does not usually contain ϕ-nodes before optimization, and any ϕ-nodes can be removed by running LLVM's `reg2mem` pass prior to processing, but it is interesting to support the entire LLVM IR language for the sake of generality.

Finally, work still must be done on implementing the present system as a set of LLVM passes and integrating it with the Clang compilation system. Empirical experiments must also be done to check the effectiveness and performance of the system.

References

[1] Aspinall, D. and Hofmann, M. (2004). Dependent types. In Pierce, B. C., editor, *Advanced Topics in Types and Programming Languages*. MIT Press.

[2] Augustsson, L. (1998). Cayenne—a language with dependent types. In *ACM SIGPLAN Notices*, volume 34, pages 239–250. ACM.

[3] Berger, E. D. and Zorn, B. G. (2006). Diehard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Notices*, volume 41, pages 158–168. ACM.

[4] Boehm, H.-J. and Weiser, M. (1988). Garbage collection in an uncooperative environment. *Software: Practice & Experience*, 18(9):807–820.

[5] Clang (2015). Clang: a C language family frontend for LLVM. http://clang.llvm.org. Accessed in July 2015.

[6] Condit, J., Harren, M., Anderson, Z., Gay, D., and Necula, G. C. (2007). Dependent types for low-level programming. In *Programming Languages and Systems*, pages 520–535. Springer.

[7] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J. A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., and Paxson, V. (2014). The matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA. ACM.

[8] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. (2002). Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288.

[9] Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE.

[10] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. (2009). Softbound: highly compatible and complete spatial memory safety for c. In *ACM Sigplan Notices*, volume 44, pages 245–258. ACM.

[11] Necula, G. C., Condit, J., Harren, M., McPeak, S., and Weimer, W. (2005). CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526.

[12] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. (2002). CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228. Springer.

[13] Zhao, J., Nagarakatte, S., Martin, M. M., and Zdancewic, S. (2013). Formal verification of SSA-based optimizations for LLVM. In *ACM SIGPLAN Notices*, volume 48, pages 175–186. ACM.