UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

VÍTOR BUJÉS UBATUBA DE ARAÚJO

Týr: a dependent type based code transformation for spatial memory safety in LLVM

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Álvaro Freitas Moreira
Coadvisor: Prof. Dr. Rodrigo Machado

Porto Alegre
January 2018

CIP — CATALOGING-IN-PUBLICATION

Araújo, Vítor Bujés Ubatuba De

Týr: a dependent type based code transformation for spatial memory safety in LLVM / Vítor Bujés Ubatuba De Araújo. – Porto Alegre: PPGC da UFRGS, 2018.

124 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor: Álvaro Freitas Moreira; Coadvisor: Rodrigo Machado.

1. Dependent types. 2. Memory safety. 3. Program transformation. 4. Systems programming. I. Moreira, Álvaro Freitas. II. Machado, Rodrigo. III. Título.

Safety-runes you must know
If safety you wish to have
And carve them within your types
Some in the variables
Some in the structures
And twice call the name of Týr.

— SIGRDRÍFUMÁL (paraphrased)

ACKNOWLEDGEMENTS

First of all, I must thank my advisors, Álvaro Moreira and Rodrigo Machado, without whose guidance, support and patience this work would not have been possible.

I'd like to thank my father, Jorge, who, by giving me a computer with GNU/Linux when I was 9, has put me in the path that led me where I am today. I'd also like to thank my mother, Andréia, a person whose wisdom I've come to appreciate more and more as I grew older, for her support during this time. Thanks also to all family members who have supported me in one way or another during this time.

I'd also like to thank all friends who have provided support and joyful times during this period. Special thanks go to Vítor Rey, who embarked with me in this journey, and Carolina Nogueira, "for outstanding work in enabling theses, making heads work again, spreading fun, hugs and good vibrations, maintaining world peace, and being generally ^awesome^."

I would like to thank Julie Fowlis, Clannad, Mari Boine, and other artists whose music has helped me keep some peace of mind during the making of this work. I'd also like to thank the people at *Speculative Grammarian*, and especially the *Language Made Difficult* podcast, who have helped me keep some sanity during the rough paths in this journey through academia.

This work has been partly funded by CAPES and CNPq scholarships. This work has also been partly funded by my grandmother, Eronilda, to whom I am grateful.

Last, and definitely not least, I would like to thank my cousin Hélio, who has put up with me (and vice-versa) since we were kids, and who has been my colleague during both the Bachelor's and the Master's, for the lifelong friendship. Thank you very much.

ABSTRACT

The C programming language does not enforce spatial memory safety: it does not ensure that memory accessed through a pointer to an object, 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 and security vulnerabilities in C programs.

A number of techniques have been proposed to provide memory safety in C. Typically such systems keep their own bounds information and instrument the program to ensure that memory safety is not violated. This has a number of drawbacks, such as changing the memory layout of data structures and thus breaking binary compatibility with external libraries and/or increased memory usage. A different approach is to use dependent types to describe the bounds information already latent in C programs and thus allow the compiler to use that information to enforce spatial memory safety. Although such systems have been proposed before, they are tied specifically to the C programming language. Other languages such as C++ suffer from similar memory safety problems, and thus could benefit from a more language-agnostic approach.

This work proposes Týr, a program transformation based on dependent types for ensuring spatial memory safety of C programs at the LLVM IR level. It allows programmers to describe at the type level the relationships between pointers and bounds information already present in C programs. In this way, Týr ensures spatial memory safety by checking the consistent usage of this pre-existing metadata, through run-time checks inserted in the program guided by the dependent type information. By targeting the lower LLVM IR level, Týr aims to be usable as a foundation for spatial memory which could be easily extended in the future to other languages that can be compiled to LLVM IR, such as C++ and Objective C. We show that Týr is effective at protecting against spatial memory safety violations, with a reasonably low execution time overhead and nearly zero memory consumption overhead, thus achieving performance competitive with other systems for spatial memory safety, in a more language-agnostic way.

Keywords: Dependent types. memory safety. program transformation. systems programming.

Týr: uma transformação de código baseada em tipos dependentes para segurança espacial de memória em LLVM

RESUMO

A linguagem C não provê segurança espacial de memória: não garante que a memória acessada através de um ponteiro para um objeto, tal como um vetor, de fato pertence ao objeto em questão. Em vez disso, o programador é responsável por gerenciar informações de alocações e limites, e garantir que apenas acessos válidos à memória são realizados pelo programa. Por um lado, isso provê flexibilidade: o programador tem controle total sobre o layout dos dados em memória, e sobre o momento em que verificações são realizadas. Por outro lado, essa é uma fonte frequente de erros e vulnerabilidades de segurança em programas C.

Diversas técnicas já foram propostas para prover segurança de memória em C. Tipicamente tais sistemas mantêm suas próprias informações de limites e instrumentam o programa para garantir que a segurança de memória não seja violada. Isso causa uma série de inconvenientes, tais como mudanças no layout de memória de estruturas de dados, quebrando assim a compatibilidade binária com bibliotecas externas, e/ou um aumento no consumo de memória. Uma abordagem diferente consiste em usar tipos dependentes para descrever a informação de limites já latente em programas C e assim permitir que o compilador use essa informação para garantir a segurança espacial de memória. Embora tais sistemas tenham sido propostos no passado, eles estão atrelados especificamente à linguagem C. Outras linguagens, como C++, sofrem de problemas similares de segurança de memória, e portanto poderiam se beneficiar de uma abordagem mais independente de linguagem.

Este trabalho propõe Týr, uma transformação de código baseada em tipos dependentes para garantir a segurança espacial de memória de programas C ao nível LLVM IR. O sistema permite que o programador descreva no nível dos tipos as relações entre pointeiros e informação de limites já presente em programas C. Dessa maneira, Týr provê segurança espacial de memória verificando o uso consistente desses metadados pré-existentes, através de verificações em tempo de execução inseridas no programa guiadas pela informação de tipos dependentes. Ao trabalhar no nível mais baixo do LLVM IR, Týr tem por objetivo ser usável como uma fundação para segurança espacial de memória que possa ser facilmente estendida no futuro para outras linguagens compiláveis para LLVM IR, tais

como C++ e Objective C. Demonstramos que Týr é eficaz na proteção contra violações de segurança espacial de memória, com um overhead de tempo de execução relativamente baixo e de consumo de memória próximo de zero, atingindo assim um desempenho competitivo com outros sistemas para segurança espacial de memória de uma maneira mais independente de linguagem.

Palavras-chave: segurança de memória, tipos dependentes, programação de sistemas, transformação de código.

LIST OF ABBREVIATIONS AND ACRONYMS

ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
GPU	Graphics Processing Unit
IR	Intermediate Representation
NUL	Null character
POSIX	Portable Operating System Interface
SSA	Static Single Assignment
SSL	Secure Sockets Layer

LIST OF FIGURES

Figure 1.1 Example C code and resulting instrumented program....30
Figure 1.2 Flow of the compilation process in Týr....31
Figure 1.3 Example C program and dependent type annotation file....33
Figure 1.4 Execution of the example program with no run-time memory safety errors.33
Figure 1.5 Execution of the program with an error inside the function **sum**....34
Figure 1.6 Execution of the program with an error in the call to **sum**....34

Figure 2.1 A 2-dimensional array (left) and an array of pointers to two 1-dimensional arrays (right). C accesses both kinds of structures using the same syntax. 45
Figure 2.2 Sample C function and equivalent LLVM IR code....46
Figure 2.3 Subset of LLVM IR considered in this work. An overline indicates a sequence of zero or more of the overlined element. 47
Figure 2.4 Example of ϕ-node in SSA-form....54

Figure 3.1 Grammar of Týr types 59
Figure 3.2 Default mapping $\lceil\cdot\rceil$ from LLVM IR types to Týr types 62
Figure 3.3 Mapping $\lfloor\cdot\rfloor$ from Týr types to LLVM IR types....62
Figure 3.4 Global environment generation pass....65
Figure 3.5 Local instrumentation pass 66
Figure 3.6 Grammar of boolean check conditions 96

LIST OF TABLES

Table 4.1 Non-blank lines of code in source file and in type annotations file 108
Table 4.2 Benchmark results 109
Table 4.3 Benchmark results for modified versions of revcomp 111

CONTENTS

RESUMO ESTENDIDO 13
1 INTRODUCTION 25
1.1 Background 25
1.2 Proposal 30
1.3 Scope and limitations 31
1.4 Usage 32
1.5 Results 34
1.6 Outline 35
2 BACKGROUND 36
2.1 Defining memory safety 36
2.2 Mechanisms for ensuring memory safety 37
2.2.1 Temporal memory safety 38
2.2.2 Spatial memory safety 39
2.3 Dependent types 40
2.4 Memory in C: pointers and arrays 43
2.5 The LLVM Intermediate Representation language 45
2.5.1 Top-level structure 48
2.5.2 Types 48
2.5.3 Values 49
2.5.4 Instructions 50
2.5.5 Omissions from the full language 53
2.5.6 Phi instructions 54
2.6 Correspondence between C and LLVM IR 55
2.6.1 Types 55
2.6.2 Variables and functions 56
2.7 Summary 56
3 THE TÝR CODE TRANSFORMATION 57
3.1 Design considerations 57
3.2 Týr types 58
3.2.1 Overview of Týr types 59
3.2.2 Correspondence between LLVM IR and Týr types 61
3.2.3 Type compatibility 62
3.3 Instrumentation overview 64
3.3.1 Global environment construction pass 65
3.3.2 Local pass 65
3.3.3 Instrumentation rules 66
3.4 The Týr typing and instrumentation rules 67
3.4.1 Typing values 68
3.4.2 Common pointers 68
3.4.3 String pointers and arrays 72
3.4.4 Local pointers 77
3.4.5 Invariance of pointer types 82
3.4.6 Structures 83
3.4.7 Functions 86
3.4.8 Bitcast 87
3.4.9 Other instructions 89
3.5 Global environment, revisited 90

3.6 Emitting instrumentation code 91
3.6.1 Computing expressions 92
3.6.2 Checks 96
3.7 Soundness 99
3.8 Efficiency considerations 101
3.9 Summary 103
4 EXPERIMENTAL RESULTS 104
4.1 Usage 104
4.2 Benchmarks 106
4.3 Results 108
5 RELATED WORK 113
5.1 Memory safety in C 113
5.2 Memory debugging tools 116
5.3 Safe systems programming languages 117
5.4 Summary 118
6 CONCLUSION 119
REFERENCES 122

RESUMO ESTENDIDO

Introdução

A maioria das linguagens de alto nível provê *segurança espacial de memória*: essas linguagens garantem que um programa nunca acessa regiões de memória fora dos limites de objetos previamente alocados pelo programa. Isso normalmente é obtido através de uma combinação de mecanismos, tais como um sistema de tipos forte, gerenciamento automático de memória através de *garbage collection*, e verificação de limites de vetor. Esses mecanismos possuem componentes estáticos (de tempo de compilação), tais como sistemas de tipos que não permitem construções que poderiam levar a violações de segurança de memória (por exemplo, converter um inteiro para ponteiro e então usar o ponteiro resultante para acessar a memória), e componentes dinâmicos, tais como verificações em tempo de execução para garantir que um objeto está sendo acessado dentro de seus limites. Verificações dinâmicas requerem que o programa mantenha *metadados* suficientes em tempo de execução a fim de permitir verificar a validade de um acesso à memória. Por exemplo, no ambiente de execução de linguagens como Java, vetores são tipicamente armazenados em memória como um inteiro representando seu comprimento, seguido dos elementos de fato do vetor; dessa maneira, quando se realiza o acesso a uma posição do vetor, o ambiente de execução possui informação suficiente para comparar o índice da posição desejada com o comprimento armazenado do vetor para garantir que o índice é válido, e sinalizar um erro caso não o seja.

Em contraste, a linguagem de programação C não garante segurança de memória: cabe ao programador gerenciar os limites das regiões alocadas e garantir que nenhum acesso inválido à memória seja realizado em tempo de execução. Estruturas de dados não incluem metadados implícitos em C: um vetor de 3 inteiros é constituído simplesmente de três inteiros contíguos em memória, sem nenhum metadado extra indicando quantos elementos estão presentes ou em que ponto o vetor acaba. Manter essa informação extra é deixado a cargo do programador.

Programadores C empregam um número de construções idiomáticas para manter essa informação. Uma construção idiomática comum é passar um ponteiro para um vetor juntamente com seu comprimento para funções que manipulam vetores. Outra construção comum é armazenar o ponteiro para um vetor juntamente com seu tamanho como campos em uma única estrutura de dados. Ainda outra técnica é armazenar um valor nulo distinto

ao final do vetor para indicar seu final; isso é tipicamente usado para strings, onde o caractere ASCII NUL (todos os bits zero) marca o final da string. Os programadores têm então que incluir manualmente verificações para garantir que esses limites são respeitados.

Considere por exemplo a função **main** em programas C. Essa função é chamada com dois argumentos, representando os argumentos de linha de comando passados para o programa. O primeiro argumento da função, convencionalmente chamado **argc**, é um inteiro representando o número de argumentos de linha de comando passados para o programa. O segundo argumento da função, **argv**, é um ponteiro para um vetor contendo **argc** elementos, cada um dos quais é um ponteiro para uma string delimitada por nulo. O limite do vetor acessível através do segundo argumento está representado pelo inteiro passado como primeiro argumento, e os limites de cada string está representado pelo byte nulo ao seu final. Porém, é encargo do programador usar essa informação corretamente para garantir que os limites não serão violados; a linguagem não impedirá o programa de acessar o vetor além do limite dado pelo argumento **argc**, ou além do terminador nulo de uma string. Da mesma forma, também não impedirá uma função como a **main** de ser chamada com um valor de **argc** que não condiz com o tamanho real do vetor apontado por **argv**.

Como outro exemplo, considere a função POSIX **writev**, que realiza a escrita de dados coletados de múltiplos buffers em um descritor de arquivo aberto. Essa função recebe três argumentos. O primeiro argumento é o descritor de arquivo para o qual os dados serão enviados. O segundo argumento é um ponteiro para um vetor de elementos do tipo **struct iovec**, cada um dos quais é uma estrutura de dados contendo um ponteiro para uma região de memória e um inteiro representando seu tamanho. O terceiro argumento é um inteiro indicando quantos elementos **struct iovec** estão presentes no vetor apontado pelo segundo argumento. Novamente, cabe ao programador garantir que os ponteiros e comprimentos dentro de cada **struct iovec** são consistentes entre si, e que o número de **struct iovec**s apontados pelo segundo argumento é consistente com o inteiro passado como terceiro argumento; a linguagem não impedirá que comprimentos incorretos sejam passados como argumentos.

Por um lado, essa falta de metadados e verificações em tempo de execução implícitos dá maior controle ao programador. A ausência de metadados implícitos associados com vetores e outros dados permite um controle mais fino do layout em memória das estruturas de dados. Isso é especialmente importante em *programação de sistemas*, isto é, programação de componentes de baixo nível de sistemas, tais como kernels de sis-

temas operacionais e ambientes de execução de linguagens de programação, onde um controle preciso do layout de estruturas de dados pode ser necessário. A ausência de verificações em tempo de execução implícitas também permite que o programador decida onde e quando realizar verificações, o que pode trazer benefícios em termos de desempenho.

Por outro lado, isso é uma fonte frequente de bugs em programas C e C++, uma vez que é muito fácil omitir ou realizar incorretamente tais verificações, ou gerenciar incorretamente as informações de limite controladas manualmente. Nesses casos, o resultado é um programa inseguro com relação a memória, com consequências variando desde *crashes* até corrupção silenciosa de dados e vulnerabilidades de segurança. Um grande número de vulnerabilidades de segurança encontradas em software no mundo real é causado por *buffer overflows* e *overreads*, isto é, a exploração da ausência ou incorretude de verificação de limites de algum buffer do programa para ganhar acesso a uma região de memória que não deveria ser acessível, obtendo assim informações que não deveriam ser reveladas, ou alterando o comportamento subsequente do programa, potencialmente possibilitando a execução de código arbitrário da escolha do atacante. Um exemplo disso é o bug Heartbleed (DURUMERIC et al., 2014), descoberto em 2014 na biblioteca OpenSSL, amplamente empregada para a comunicação segura entre clientes e servidores na Internet, na qual a ausência de uma verificação de limites permitia que atacantes obtivessem o conteúdo de regiões arbitrárias da memória do servidor, potencialmente revelando informações sensíveis tais como usuários, senhas, e as chaves privadas de certificados de comunicação SSL.

Diversas técnicas já foram propostas para prover segurança de memória em C (JIM et al., 2002; NECULA et al., 2005; NAGARAKATTE et al., 2009). Tipicamente, tais sistemas mantêm seus próprios metadados representando as informações de limite do programa, e instrumentam o programa com verificações em tempo de execução para garantir que a segurança de memória não seja violada, simulando assim os mecanismos empregados por linguagens de nível mais alto como Java. Isso tem um número de inconvenientes:

- Se os metadados são mantidos juntamente com os dados aos quais se referem, eles alteram a representação das estruturas de dados. Isso é indesejado em programação de sistemas onde, como mencionado anteriormente, frequentemente é necessário o controle do layout em memória dos dados. Isso também introduz problemas de interoperabilidade com código externo, tais como bibliotecas e chamadas de

sistema operacional, que não esperam que esses metadados estejam presentes nas estruturas de dados que lhes são passadas.

- Se os metadados são mantidos em uma estrutura de dados separada para evitar esses problemas, há um aumento no custo em tempo de acesso aos metadados associados a um dado buffer.
- Diferentemente de linguagens de programação de nível mais alto, C possui ponteiros, que podem apontar para um endereço de memória no meio de um objeto. Enquanto em uma linguagem como Java o ambiente de execução pode obter eficientemente os metadados associados com um vetor buscando-os em uma posição fixa a partir do início do vetor, ponteiros em C não necessariamente apontam para o início do vetor, e não é possível no caso geral encontrar o início de um vetor dado um ponteiro para um ponto arbitrário do mesmo. Assim, em vez de os metadados ficarem associados aos vetores, é necessário que cada ponteiro carregue consigo informações de limites inferior e superior (chamada de representação *fat pointer*), o que causa um overhead significativo em uso de memória.

Uma abordagem diferente baseia-se na observação de que, em um programa C sem violações de segurança de memória, todos os metadados necessários *já estão* presentes, na forma das construções idiomáticas que os programadores C usam para gerenciar a informação de limites. Porém, como essa informação é mantida de uma maneira *ad hoc* pelo programador sem suporte da linguagem, o compilador não tem como verificar o uso correto dessa informação. Se o programador tivesse uma maneira de informar ao compilador como essa informação está sendo usada, o compilador poderia verificar mecanicamente que os acessos à memória estão dentro dos limites empregando os próprios metadados providos pelo programador para realizar as verificações, e que os metadados providos são consistentes com os dados aos quais estão associados. Isso pode ser realizado por meio de tipos dependentes.

Tipos dependentes (ASPINALL; HOFMANN, 2004) são tipos indexados por expressões. Da mesma maneira que linguagens com polimorfismo paramétrico, também conhecido como *generics* no mundo orientado a objetos, permite parametrizar um tipo com outro tipo (e.g., **Array<Int>**, um vetor de inteiros), uma linguagem com tipos dependentes permite parametrizar tipos com expressões (e.g., **Array<Int, n>**, um vetor de **n** inteiros, onde **n** é uma variável do programa). Agora, se **p** é um ponteiro em C para um vetor de inteiros cujo comprimento está armazenado em uma variável **len**, poderíamos dar a **p** um tipo como **Ptr<int, 0, len>**, isto é, um ponteiro para uma região de inteiros cujo

limite inferior é 0 e cujo limite superior é **len**. Se o programador anota a variável **p** com um tal tipo, o compilador possui informação suficiente para saber onde buscar os limites da região associada com **p**. Quando o programa tentar acessar a memória através desse ponteiro usando uma expressão como **p[i]**, o compilador pode então inserir uma verificação de tempo de execução como **assert(i≥0 && i<len)** para garantir que o índice **i** é válido antes de realizar o acesso. Isso provê uma solução para os problemas mencionados anteriormente:

- Como o sistema usa os metadados já presentes no programa tais como providos pelo programador, não há necessidade de modificar o layout dos dados para carregar consigo informação adicional. Dessa maneira, mantêm-se o controle do programador sobre o layout de memória e a compatibilidade com bibliotecas externas.
- Como não são introduzidos metadados adicionais além dos já presentes no programa, o impacto em consumo de memória é mínimo.
- Finalmente, como o compilador insere verificações em termos das mesmas variáveis usadas pelo programa em vez de usar metadados mantidos separadamente, técnicas padrão de otimização utilizadas por compiladores conseguem provar redundantes e eliminar muitas das verificações inseridas.

O último ponto requer elaboração. Considere uma função C que computa a soma dos elementos de um vetor de inteiros, usando a típica construção idiomática de passar o comprimento do vetor juntamente com um ponteiro para o mesmo como argumentos separados (Figura 1a). A função usa uma variável de índice **i** inicializada com **0**, e ao fim de cada iteração o índice é incrementado. Finalmente, a iteração para quando a condição **i < len** se torna falsa. Essa é uma típica construção idiomática de iteração sobre os elementos de um vetor em C.

Agora considere que demos a essa função um tipo dependente como:

sum: Fn int (array: Ptr(int, 0, len), len: int)

indicando que **sum** é uma função que recebe dois argumentos: **array**, um ponteiro para inteiros cujos limites são **0** e **len**; e **len**, um inteiro. Dada essa informação, o compilador pode instrumentar o código de tal maneira que, antes do ponto em que **array[i]** é acessado, é inserido um teste de tempo de execução para verificar que **i** está dentro dos limites do ponteiro. Isso resultará em um código como o da Figura 1b. Porém, como a verificação inserida é escrita em termos de **i** e **len**, as mesmas variáveis usadas no fluxo de controle do laço, otimizações padrão de compiladores são capazes de determinar que a verificação

Figura 1: Exemplo de código C e programa instrumentado resultante

(a) Programa original

```
int sum(int *array, int len) {
    int result = 0;
    for (int i=0; i<len; i++) {
        result += array[i];
    }
    return result;
}
```

(b) Programa instrumentado

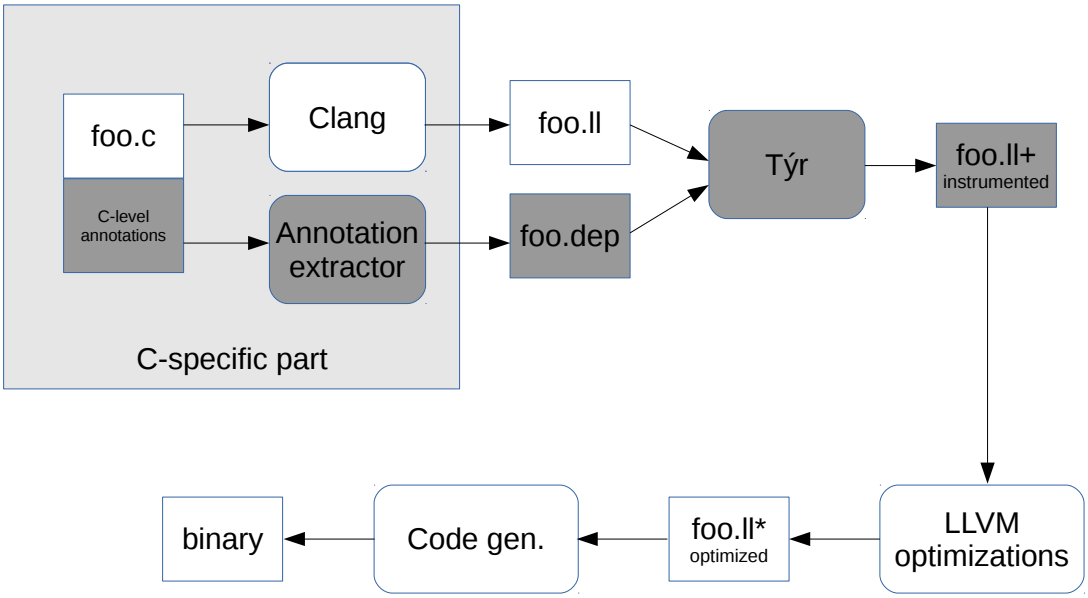
```
int sum(int *array, int len) {
    int result = 0;
    for (int i=0; i<len; i++) {
        assert(i>=0 && i<len);
        result += array[i];
    }
    return result;
}
```

Fonte: O autor

é na verdade redundante neste caso: sempre é o caso que **i>=0 && i<len** dentro do corpo do laço, já que o laço itera para **i** de **0** a **len** (exclusive). Portanto, o compilador é capaz de eliminar esse teste. Essa eliminação de verificações redundantes seria mais difícil de realizar se os testes inseridos pelo compilador fossem escritos em termos de metadados implícitos mantidos separadamente pela instrumentação, já que não seria evidente a partir do fluxo de controle que o teste é sempre verdadeiro dentro do laço. Assim, testes que podem ser provados verdadeiros em tempo de compilação podem ser removidos, reduzindo o impacto em desempenho da instrumentação, e testes que não podem ser provados verdadeiros em tempo de compilação são deixados como verificações a serem realizadas em tempo de execução, garantindo assim que os acessos à memória são seguros. Por exemplo, suponha que, na Figura 1a, o programador tivesse equivocadamente escrito **for (int i=0; i<=len; i++)**, substituindo **<** por **<=**, e assim levando a um erro de *off-by-one* em que a primeira posição após o último índice válido do vetor é acessada. Nesse caso, a verificação inserida não teria sido removida pelas otimizações do compilador, e garantiria que a execução seria interrompida antes que ocorresse o acesso inválido.

Deputy (CONDIT et al., 2007) introduz um tal sistema de tipos dependentes para C, que permite aos programadores expressar tipos de ponteiros dependentes por meio de anotações no código fonte C. Porém, Deputy foi implementado usando CIL (NECULA et al., 2002), um framework para análise e transformação de programas específico para a linguagem C. Outras linguagens, como C++, apresentam os mesmos problemas de segurança de memória de C, e portanto seria vantajoso poder aplicar as mesmas técnicas de uma maneira mais independente de linguagem.

Figura 2: Fluxo do processo de compilação em Týr



Fonte: O autor

Proposta

O presente trabalho propõe Týr, uma transformação de código baseada em tipos dependentes para a linguagem de representação intermediária (IR) do LLVM. LLVM (LATTNER; ADVE, 2004) é um framework independente de linguagem para análise, transformação e compilação de código, projetado em torno de uma linguagem intermediária bem definida. Clang (CLANG, 2015) é um compilador C/C++ que emite código LLVM IR, e emprega a infraestrutura do LLVM para realizar otimizações e emitir código de máquina. Por trabalhar em termos da linguagem LLVM IR, em um nível mais baixo, Týr pode ser aplicado para garantir a segurança espacial de memória de diversas linguagens que possam ser compiladas para LLVM IR, tais como C e C++. Týr introduz uma linguagem de tipos rica para descrever informação de limites em programas LLVM IR, e implementa uma transformação de código que instrumenta o programa com verificações de tempo de execução para garantir a conformidade com os limites descritos.

A Figura 2 mostra o fluxo do processo de compilação de um programa no framework Týr. Os componentes em cinza claro são parte do fluxo normal do processo de compilação no Clang e LLVM: um programa C é dado como entrada para o compilador Clang, o qual emite código de nível mais baixo correspondente em LLVM IR. Esse código passa então pelo pipeline de otimizações do LLVM, gerando um programa LLVM IR

otimizado, o qual é finalmente passado para o passo de geração de código do LLVM, que emite código de máquina para uma arquitetura específica.

Os componentes em cinza escuro são os adicionados pelo framework Týr. Primeiramente, o código fonte C deve ser enriquecido com anotações de tipos dependentes providas pelo programador, tais como aquela provida para a função **sum** no exemplo acima. O programa C é então passado ao Clang como de costume para a geração de código LLVM IR; adicionalmente, as anotações de tipos dependentes são extraídas do programa C e mapeadas para anotações de tipo equivalentes expressas em termos do program em nível LLVM IR. Em seguida, o programa LLVM IR gerado pelo Clang e as anotações extraídas são passadas para o passo Týr propriamente dito, que varre o programa LLVM IR verificando que o programa está em confirmidade com as anotações providas. O passo também instrumenta o programa com verificações necessárias para garantir que as restrições expressas nas anotações sejam asseguradas em tempo de execução, gerando assim um programa LLVM IR instrumentado. Esse programa é então passado ao pipeline usual de otimizações, com a possível eliminação de código de verificação redundante, e então passado para o passo de geração de código para produzir um binário instrumentado.

Vale a pena observar que o extrator de anotações, que mapeia os tipos dependentes expressos em termos de C em tipos equivalentes expressos em termos de LLVM IR, é o único componente da arquitetura de Týr que é específico à linguagem C. Em princípio, Týr pode ser adaptado para qualquer outra linguagem que compile para LLVM IR, desde que um mapeamento possa ser definido dos tipos da linguagem de alto nível para os tipos do LLVM IR, substituindo-se o extrator de anotações por um adequado à linguagem que se deseja suportar.

Escopo e limitações

O objetivo do presente trabalho é focar no componente no nível de LLVM IR da arquitetura. Definimos um conjunto de anotações baseadas em tipos dependentes que podem ser usadas para descrever relações entre ponteiros e seus metadados associados em LLVM IR, um conjunto de regras que governam o uso de valores e a inserção de verificações de tempo de execução guiada por esses tipos, e um algoritmo para a transformação de um programa LLVM IR de entrada mais anotações de tipo providas pelo programador em um programa LLVM IR de saída instrumentado com verificações de tempo de execução. Implementamos um protótipo do sistema, e o aplicamos a um número de progra-

mas de benchmark para avaliar o impacto em desempenho da instrumentação e a eficácia da mesma em detectar erros relacionados à segurança de memória.

O sistema proposto visa a segurança *espacial* de memória (isto é, verificar que os objetos em memória são acessados dentro de limites válidos), não segurança *temporal* de memória (verificar que os objetos em memória não sejam acessados depois de terem sido desalocados). Esses conceitos são descritos em maior detalhe no Capítulo 2. No geral, os mecanismos que visam a segurança espacial e temporal de memória são distintos e ortogonais.

O extrator de anotações em nível de C foi deixado fora do escopo do trabalho presente. Em nosso protótipo atual, é necessário prover anotações no nível do programa LLVM IR, em vez do nível mais alto do programa fonte C (isto é, anotamos o programa LLVM IR emitido pelo Clang, em vez do programa C em si). Esta é uma limitação do protótipo atual que pode ser suprida em um trabalho futuro.

Outra limitação do presente sistema é que, em certas circunstâncias, o mesmo assume que as anotações providas pelo programador estejam corretas. Isso se dá em parte devido a limitações da implementação atual, e em parte porque Týr permite que um programa instrumentado interaja com código externo não instrumentado, tal como bibiliotecas do sistema. Se o programador prover anotações corretas para as funções e dados externos, Týr é capaz de verificar o uso correto dessas funções e dados a partir da parte instrumentada do código. Porém, como Týr não tem controle sobre o código não instrumentado, Týr precisa assumir que o código externo de fato está em conformidade com as anotações providas pelo programador.

Uso

O binário instrumentado contém código que garante que certas condições são satisfeitas quando ponteiros ou os limites associados aos mesmos são manipulados. Se uma condição falha em ser satisfeita em tempo de execução, a execução é abortada e uma mensagem de erro é apresentada. Se o programa é compilado com informação de depuração ativada, as mensagens emitidas também conterão o nome do arquivo de código fonte, o nome da função, a linha e a coluna do programa onde a condição foi violada.

Por exemplo, considere um arquivo de código fonte C **example.c** (Figura 3a), que consiste da função **sum** apresentada anteiormente, mais uma função **main** que define um vetor de 3 números, chama **sum** com o mesmo, e imprime o resultado. Considere

Figura 3: Programa C de exemplo e arquivo de anotações de tipos dependentes

(a) example.c

```
1. #include <stdio.h>
2.
3. int sum(int *array, int len) {
4.     int result = 0;
5.     for (int i=0; i<len; i++) {
6.         result += array[i];
7.     }
8.     return result;
9. }
10.
11. int main() {
12.     int a[] = { 10, 20, 30 };
13.     int result = sum(a, 3);
14.     printf("%d\n", result);
15.     return 0;
16. }
```

(b) example.dep

```
sum: Fn i32 (array: Ptr(i32, 0, len), len: i32)
```

Fonte: O autor

Figura 4: Execução do programa de exemplo sem erros de segurança de memória em tempo de execução

```
1. $ ./tyrcc.sh -g example.c
2. $ ./example-tyr
3. 60
```

Fonte: O autor

também um arquivo de anotação de tipos dependentes **example.dep** (Figura 3b), contendo a anotação a nível de LLVM IR para a função **sum**.[1]

Se compilarmos este program com Týr e executarmos o binário resultante, obteremos a soma resultante impressa como esperado (Figura 4). Porém, considere que modifiquemos o teste na linha 5 de **i<len** para **i<=len**, criando assim um programa que acessaria uma posição de memória além do limite do vetor. Se recompilarmos e re-executarmos o programa, Týr detectará a tentativa de acesso à posição de memória inválida e abortará a execução com um erro (Figura 5). A mensagem de erro inclui diversas informações do nível LLVM IR que não são diretamente relevantes para o programador, mas são úteis para entender o funcionamento do protótipo, tais como a linha do código LLVM IR que causou a violação (linhas 3 e 4 da saída) e uma descrição do teste realizado em termos do funcionamento interno do protótipo Týr (linha 5). Porém, como compilamos o programa com informação de depuração (opção **-g** do Clang), Týr também é capaz de informar (linha 6) que o acesso problemático jaz na função **sum()**, linha 6, coluna 19, isto é, o acesso **array[i]** fora dos limites.

Considere agora que, em vez de mudar o teste na linha 5 do código de exemplo (Figura 3a), modifiquemos a chamada de função na linha 13 de **sum(a, 3)** para **sum(a,**

[1]Como observado na seção anterior, na implementação atual é necessário prover as anotações de tipos dependentes em termos do nível LLVM IR em vez do nível C. Neste exemplo, o mapeamento entre os dois níveis é direto, sendo a única diferença perceptível entre os níveis C e LLVM IR o uso de **i32** em vez de **int**.

Figura 5: Execução do programa de exemplo com um erro dentro da função **sum**

```
1. $ ./tyrcc.sh -g example.c
2. $ ./example-tyr
3. ERROR: LLVM line 36:
4.     %12 = load i32, i32* %11, !dbg !35
5. Check (sub %.tyr.deref80 %9) sgt 0# violated (%.tyr.fail94, type-deref/3)
6. example.c: sum() line 6, column 19
```

Fonte: O autor

Figura 6: Execução do programa de exemplo com um erro na chamada a **sum**

```
1. $ ./tyrcc.sh -g example.c
2. $ ./example-tyr
3. ERROR: LLVM line 68:
4.     %4 = call i32 @sum(i32* %3, i32 4), !dbg !50
5. Check 3# sge 4# violated (%.tyr.fail214, fits/3)
6. example.c: main() line 13, column 18
```

Fonte: O autor

4), passando um tamanho incorreto junto com o vetor. Após recompilar e executar o programa novamente, obtemos um erro diferente (Figura 6). Desta vez, o erro jaz na instrução LLVM IR que chama **sum** (reportada na linha 3 da saída). O teste (linha 5) indica um conflito entre o valor inteiro provido (4) e o limite superior do vetor (3), que se esperava que fosse maior ou igual (**sge**) do que o inteiro provido. A localização no código fonte da violação (linha 6) é na função **main()**, linha 13, coluna 18, isto é, a chamada incorreta a **sum**. Týr aborta a execução antes que a função **sum** seja chamada, uma vez que o valor provido para o segundo argumento não condiz com o ponteiro passado como primeiro argumento.

Resultados

Realizamos uma série de benchmarks para medir o impacto em desempenho da instrumentação inserida por Týr em termos de tempo de execução e consumo de memória relativos ao programa não instrumentado. Em nossos benchmarks, observamos um overhead médio de 25.6% em tempo de uso de CPU, com uma mediana de 12.1%. Com exceção de um dos benchmarks, o overhead observado ficou sempre abaixo de 27%. No benchmark que demonstrou um overhead maior, encontramos dois bugs de segurança de memória, ambos os quais foram detectados pela instrumentação de Týr. Em todos os casos, o overhead em consumo de memória ficou perto de zero. Esses números são similares aos reportados para Deputy (CONDIT et al., 2007), e no geral melhores que outras abor-

dagens baseadas em software que baseiam-se na manutenção de metadados separados.

As contribuições deste trabalho podem ser summarizadas em:

- O desenvolvimento de uma linguagem de tipos dependentes para LLVM IR capaz de expressar dependências entre ponteiros e seus limites, incluindo tipos dependentes para ponteiros, funções, vetores e estruturas. Ao visar à linguagem LLVM IR, em um nível mais baixo, essa abordagem é mais geralmente aplicável a qualquer linguagem que possa ser compilada para LLVM IR, desde que os tipos na linguagem fonte sejam mapeados para os tipos dependentes correspondentes no nível LLVM IR.
- Uma transformação de código LLVM IR para LLVM IR que garante a segurança espacial de memória através da inserção de verificações de tempo de execução dirigidas pela informação de tipos provida pelo programador.
- Um protótipo de implementação do sistema proposto, com avaliação experimental demonstrando que o sistema pode ser usado para garantir a segurança espacial de programas C com um overhead de desempenho geralmente razoável.

Organização

O restante deste trabalho está organizado como segue. O Capítulo 2 apresenta o background conceitual e técnico do trabalho. O Capítulo 3 descreve os tipos dependentes e a transformação de código Týr proposta neste trabalho. O Capítulo 4 apresenta resultados experimentais demonstrando a eficácia do sistema. O Capítulo 5 apresenta trabalhos relacionados. O Capítulo 6 apresenta uma conclusão e direções para trabalhos futuros.

1 INTRODUCTION

1.1 Background

Most high-level programming languages enforce *spatial memory safety*: they ensure that a program never accesses regions of memory outside the limits of objects previously allocated by the program. This is usually achieved through a combination of mechanisms, such as a strong type system, automatic memory management though garbage collection, and array bounds checking. These mechanisms have static (compile-time) componentes, such as a type system which disallows constructions that might lead to memory safety violations (for example, casting integers to pointers and then using the resulting pointer to access memory), and dynamic (run-time) components, such as run-time checks to ensure that an object is accessed within its bounds. Dynamic checks require that the program keep enough *metadata* at run-time to allow checking the validity of a memory access. For instance, in the runtime of languages like Java, arrays are typically stored in memory as an integer representing its length followed by the actual elements of the array; in this way, when an access to a position of the array is performed, the runtime has enough information to compare the index of the desired position against the stored length of the array to ensure that the index is valid, and signal an error in case it is not valid.

By contrast, the C programming language does not enforce memory safety: it is up to the programmer to keep track of the bounds of allocated regions and ensure that no invalid memory access is performed at run-time. Data structures include no implicit metadata in C: an array of 3 integers is just three contiguous integers in memory, with no extra metadata indicating how many elements are present or where the array ends. Keeping track of this extra information is left to the programmer.

C programmers employ a number of idioms to keep track of such information. A common idiom is to pass a pointer to an array along with its length as arguments to array-handling functions. Another common idiom is to store the pointer to an array and its length as fields in a single data structure. Yet another idiom is to store a distinguished null value at the end of an array to indicate its end; this is typically used for strings, where the ASCII NUL (all bits zero) character marks the end of the string. Programmers then have to manually include checks to ensure those bounds are respected.

Consider for example the **main** function in C programs. This function is called

with two arguments, representing the command-line arguments passed to the program. The first function argument, conventionally called **argc**, is an integer representing the number of command-line arguments passed in. The second function argument, **argv**, is a pointer to an array of **argc** elements, each of which is a pointer to a null-terminated string. The bounds of the array accessible through the second argument are represented by the integer passed in as the first argument, and the bounds of each string is represented by the null byte stored at its end. However, it is up to the programmer to use this information correctly to ensure that bounds will not be violated; the language will not prevent a program from accessing the array beyond the limit given by **argc**, or past the null-terminator of a string. Neither will it prevent a function like **main** from being called with an **argc** value which does not match the actual length of the array pointed to by **argv**.

As another example, consider the POSIX function **writev**, which writes data collected from various buffers to an open file descriptor. This function takes three arguments. The first one is the file descriptor the data will be sent to. The second argument is a pointer to an array of **struct iovec** elements, each of which is a data structure containing a pointer to a region of memory and an integer representing its length. The third argument is an integer indicating how many **struct iovec** elements are present in the array pointed to by the second argument. Again, it is up to the programmer to ensure that the pointers and lengths within each **struct iovec** are consistent with each other, and that the number of **struct iovec**s pointed to by the second argument matches the integer passed in as the third argument; the language will not prevent incorrect lengths from being passed in as arguments.

On the one hand, this lack of implicit metadata and implicit run-time checks gives the programmer greater control. The lack of implicit metadata associated with arrays and other data allows finer control of the memory layout of data structures. This is especially important in *systems programming*, i.e., programming of low-level system components such as operating system kernels and programming language runtimes, where precise control over data structure layout may be required. The lack of implicit run-time checks also allows the programmer to decide when and where to perform checks, which can lead to performance benefits.

On the other hand, this is a frequent source of bugs in C and C++ programs, since it is very easy to leave out or incorrectly perform such checks, or to mismanage the manually tracked bounds information. The result is a memory-unsafe program, with consequences varying from crashes to silent data corruption and security vulnerabilities. A large number

of security vulnerabilities found in real-world software is caused by *buffer overflows and overreads*, i.e., by exploiting the absence or incorrectness of bounds checking of some program buffer to gain access to a region of memory that should not be accessible, thus either obtaining information that should not be revealed, or altering the subsequent behavior of the program, potentially enabling execution of arbitrary code of the attacker's choice. An example of this is the Heartbleed bug (DURUMERIC et al., 2014) discovered in 2014 in the widely-deployed OpenSSL library for secure communication between clients and servers on the Internet, in which the absence of a bounds check allowed an attacker to obtain the contents of arbitrary regions of the server's memory, potentially revealing sensitive data such as user names, passwords, and the private keys of SSL security certificates.

A number of techniques have been proposed to provide memory safety in C (JIM et al., 2002; NECULA et al., 2005; NAGARAKATTE et al., 2009). Typically, such systems keep their own metadata tracking the program's bounds information and instrument the program with run-time checks to ensure that memory safety is not violated, thus simulating the mechanisms employed by higher-level languages such as Java. This has a number of drawbacks:

- If this metadata is kept together with the data it refers to, it changes the representation of data structures. This is undesirable in systems programming where, as mentioned before, control over memory layout is often necessary. This also introduces interoperability problems with external code, such as libraries and operating system calls, which do not expect such metadata to be present in data structures.
- If the metadata is kept in a separate data structure to avoid these problems, there is an increased lookup cost to obtain the metadata associated with a given buffer.
- Unlike higher-level programming languages, C has pointers, which can point into the middle of an object. Whereas in a language like Java the runtime can efficiently retrieve the metadata associated with an array by looking at a fixed position from the beginning of the array, C pointers do not necessarily point to the beginning of the array, and it is not generally possible to find the beginning of the array given a pointer to an arbitrary part of it. Therefore, rather than associating metadata with arrays, every pointer must have associated lower and upper bounds information associated with it (known as a *fat pointer* representation), which brings a large memory overhead.

A different approach relies on the observation that, in a memory-safe C program, all the metadata required is *already* present, in the form of the idioms that C programmers use to keep track of bounds information. However, since this information is kept in an ad-hoc way by the programmer without language support, the compiler cannot verify the correctness of its usage. If the programmer had a way to inform the compiler how this information is being used, then the compiler could mechanically verify that memory accesses are within bounds by checking against the programmer-provided metadata, and that the metadata provided is consistent with the data it is associated to. This can be achieved with dependent types.

Dependent types (ASPINALL; HOFMANN, 2004) are types indexed by expressions. In the same way that languages with parametric polymorphism, also known as generics in the object-oriented world, allows parameterizing a type with another type (e.g., **Array<Int>**, an array of integers), a language with dependent types allow parameterizing types with expressions (e.g., **Array<Int, n>**, an array of **n** integers, where **n** is some program variable). Now, if **p** is a C pointer to an array of integers whose length is stored in a variable **len**, we might give **p** a type like **Ptr<int, 0, len>**, meaning a pointer to a region of integers whose lower bound is 0 and whose upper bound is **len**. If the programmer annotates the variable **p** with such a type, the compiler has enough information to know where to look for the bounds of the region associated with **p**. When the program tries to access memory through this pointer using an expression like **p[i]**, the compiler can then insert a run-time check like **assert(i$\geq$0 && i$<$len)** to ensure that the index **i** is valid before performing the access. This provides a solution to the problems mentioned before:

- Because the system uses the metadata already present in the program as supplied by the programmer, there is no need to change the layout of data to carry extra information. In this way, programmer's control over memory layout and compatibility with external libraries is retained.
- Because no extra metadata other than that already present in the program is introduced, the impact in memory usage is minimal.
- Finally, because the checks inserted by the compiler are in terms of the same variables used by the program, rather than using separately maintained metadata, many of the inserted checks can be proven redundant by the compiler and optimized away by standard compiler optimization techniques.

The last point requires some clarification. Consider a C function to compute the

sum of the elements of an array of integers, using the typical idiom of passing the length of the array along with a pointer to it as separate arguments (Figure 1.1a). The function uses an index variable **i** initialized with **0**, and at the end of each iteration the index is incremented. Finally, the iteration stops when the condition **i < len** becomes false. This is a typical idiom for iterating over the elements of an array in C.

Now consider that we give this function a dependent type like:

sum: Fn int (array: Ptr(int, 0, len), len: int)

meaning that **sum** is a function taking two arguments: **array**, a pointer to integers whose bounds are **0** and **len**; and **len**, an integer. Given this information, the compiler can instrument the code such that, prior to the point where **array[i]** is accessed, a run-time check is inserted to verify that **i** is within the pointer's bounds. This will result in code like that in Figure 1.1b. However, because the inserted check is written in terms of **i** and **len**, the same variables used in the loop control flow, standard compiler optimizations are able to tell that the check is actually redundant in this case: it is always the case that **i>=0 && i<len** within the body of the loop, since the loop iterates for **i** from **0** to **len** (exclusive). Therefore, the compiler can optimize this check away. This elimination of redundant checks would be harder to perform if the checks inserted by the compiler were written in terms of implicit metadata maintained separately by the instrumentation, as it would not be evident from the control flow that the check is always true inside the loop. Thus, checks that can be proven true at compile-time can be removed, reducing the performance impact of the instrumentation, and checks that cannot be proven true at compile-time are left in as checks to be performed at run-time, thus ensuring that memory accesses are safe. For instance, suppose, in Figure 1.1a, that the programmer had mistakenly written **for (int i=0; i<=len; i++)**, substituting **<=** for **<**, thus leading to an off-by-one error where a position one past the last valid index of the array is accessed. In this case, the inserted check would not have been optimized away by the compiler, and would ensure execution is interrupted before the invalid access occurs.

Deputy (CONDIT et al., 2007) introduces such a dependent type system for C, which allows programmers to express such dependent pointer types by means of annotations in the C source code. However, Deputy was implemented using CIL (NECULA et al., 2002), a framework for program analysis and transformation specific to the C language. Other languages, such as C++, present the same memory safety problems of C, so it would be beneficial to be able to apply the same techniques in a more language-independent way.

Figure 1.1: Example C code and resulting instrumented program

(a) Original program

```
int sum(int *array, int len) {
    int result = 0;
    for (int i=0; i<len; i++) {
        result += array[i];
    }
    return result;
}
```

(b) Instrumented program

```
int sum(int *array, int len) {
    int result = 0;
    for (int i=0; i<len; i++) {
        assert(i>=0 && i<len);
        result += array[i];
    }
    return result;
}
```

Source: The author

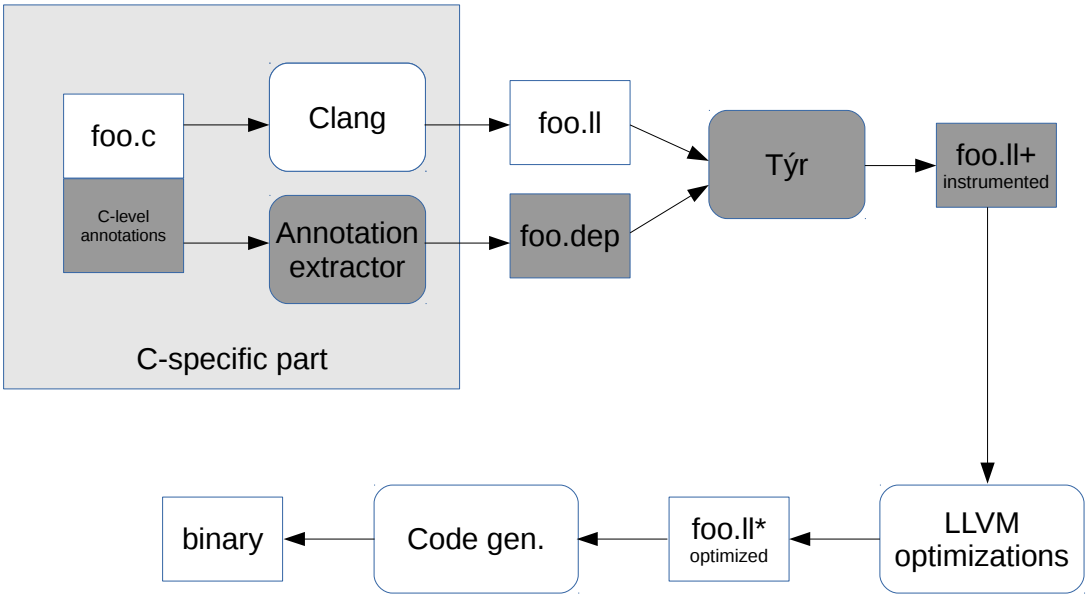
1.2 Proposal

This works proposes Týr, a program transformation based on dependent types for the LLVM Intermediate Representation language. LLVM (LATTNER; ADVE, 2004) is a language-agnostic framework for code analysis, transformation and compilation, designed around a well-defined intermediate language. Clang (CLANG, 2015) is a C/C++ compiler which emits LLVM IR code, and employs the LLVM infrastructure to perform optimizations and emit machine code. By targeting the lower-level LLVM IR language, Týr can be applied to ensure spatial memory safety of various languages that can be compiled to LLVM IR, such as C and C++. Týr introduces a rich type language for describing bounds information in LLVM IR programs, and implements a code transformation that instruments the program with run-time checks to ensure conformance with the described bounds.

Figure 1.2 shows the flow of the compilation process of a program in the Týr framework. The components in light gray are part of the normal compilation process in Clang and LLVM: a C program is given as input to the Clang compiler, which emits the corresponding lower-level LLVM IR code. This code is then fed to the pipeline of LLVM optimizations, generating an optimized LLVM IR program, which is finally passed on to the LLVM code generation step, which emit machine code for a given architecture.

The components in dark gray are those added by Týr. First, the C source code must be augmented with dependent type annotations provided by the programmer, such as that provided for the **sum** function above. The C program is fed to Clang as usual for LLVM IR code generation; in addition, the dependent type annotations are extracted from the C program and mapped into equivalent type annotations expressed in terms of the LLVM IR level program. Then the LLVM IR program generated by Clang and the extracted annotations are fed into Týr proper, which sweeps the LLVM IR program checking that

Figure 1.2: Flow of the compilation process in Týr



Source: The author

the program conforms to the provided annotations. It also instruments the program with run-time checks needed to ensure that the constraints expressed in the annotations are enforced at run-time, thus generating an instrumented LLVM IR program. This program is then fed to the usual pipeline of optimizations, with possible elimination of redundant checking code, and passed on to code generation to emit an instrumented binary.

It should be noted that the annotation extractor, which maps dependent types expressed in terms of C into equivalent types expressed in terms of LLVM IR, is the only component in the architecture of Týr that is specific to the C programming language. In principle, Týr can be adapted to any other language that compiles to LLVM IR, as long as a mapping can be defined from the high level language types into LLVM IR level types, by replacing the annotation extractor with one appropriate to the language one wishes to support.

1.3 Scope and limitations

The goal of the present work is to focus on the LLVM IR level component of the architecture. We define a set of dependent type based annotations which can be used to describe relationships between pointers and their associated metadata in LLVM IR, a set of rules that govern the usage of values and the insertion of run-time checks guided by those

types, and an algorithm for transforming an input LLVM IR program plus programmer-provided type annotations into an output LLVM IR program instrumented with run-time checks. We implement a prototype of the system, and apply it to a number of benchmark programs to evaluate the performance impact of the instrumentation and its effectiveness in catching errors related to memory safety.

The proposed system is aimed at *spatial* memory safety (i.e., checking that objects in memory are accessed within valid bounds), rather than *temporal* memory safety (checking that objects in memory are not accessed after they have been deallocated). These concepts will be described in more detail in Chapter 2. In general, the mechanisms for addressing spatial and temporal memory safety are distinct and orthogonal.

The C-level annotation extractor has been left outside the scope of the present work. In our current prototype, we have to provide annotations at the level of the LLVM IR program, rather than at the level of the higher level C source program (i.e., we annotate the LLVM IR program emitted by Clang, rather than the C program itself). This is a limitation of the current prototype which can be addressed by future work.

Another limitation of the present system is that it trusts the programmer to provide correct annotations in certain cases. This is partly due to limitations in the current implementation, and partly because Týr allows an instrumented program to interact with non-instrumented external code, such as system libraries. If the programmer provides correct annotations for the external functions and data, Týr can enforce the correct usage of those functions and data from the instrumented part of the code. However, since Týr has no control over the non-instrumented code, Týr has to trust that the external code actually conforms to the the annotations provided by the programmer.

1.4 Usage

The instrumented binary contains code that ensures that certain conditions hold when pointers or their associated bounds are manipulated. If a condition fails to hold at run-time, execution is aborted and an error message is presented. If the code is compiled with debug information on, the emitted messages will also contain the source code file, function name, line and column of the program where the condition was violated.

For example, consider a C source code file **example.c** (Figure 1.3a), which consists of the **sum** function presented before, plus a **main** function which defines an array of 3 numbers, calls **sum** with it, and prints the result. Consider also a Týr dependent type

Figure 1.3: Example C program and dependent type annotation file

(a) example.c

```
1. #include <stdio.h>
2.
3. int sum(int *array, int len) {
4.     int result = 0;
5.     for (int i=0; i<len; i++) {
6.         result += array[i];
7.     }
8.     return result;
9. }
10.
11. int main() {
12.     int a[] = { 10, 20, 30 };
13.     int result = sum(a, 3);
14.     printf("%d\n", result);
15.     return 0;
16. }
```

(b) example.dep

```
sum: Fn i32 (array: Ptr(i32, 0, len), len: i32)
```

Source: The author

Figure 1.4: Execution of the example program with no run-time memory safety errors

```
1. $ ./tyrcc.sh -g example.c
2. $ ./example-tyr
3. 60
```

Source: The author

annotation file **example.dep** (Figure 1.3b), containing the LLVM IR level annotation for the **sum** function.[1]

If we compile this program with Týr and run the resulting binary, we get the resulting sum printed out as expected (Figure 1.4). However, consider that we modify the test in line 5 from **i<len** to **i<=len**, thus creating a program which would access a memory position one past the end of the array. If we recompile and run it, Týr will catch the attempt to access the invalid memory position and abort execution with an error (Figure 1.5). The error message includes a lot of LLVM IR level information which is not directly relevant to the programmer, but is useful to understand the functioning of the prototype, such as the line of LLVM IR code which caused the violation (lines 3 and 4 of the printout) and a description of the performed check in terms of the internal functioning of the Týr prototype (line 5). However, because we compiled the program with debug information (Clang option **-g**), Týr is also able to tell us (line 6) that the problematic access lies in function **sum()**, line 6, column 19, i.e., the out-of-bounds **array[i]** access.

Consider now that rather than changing the check in line 5 of the example code (Figure 1.3a), we instead change the function call in line 13 from **sum(a, 3)** to **sum(a, 4)**, passing an incorrect size along with the array. After recompiling and running the

[1] As noted in the previous section, in the current implementation we have to provide the dependent type annotations at the LLVM IR level rather than the C level. For this example, the mapping between the two levels is straightforward, the only noticeable difference between C level and LLVM IR level types being the use of **i32** instead of **int**.

Figure 1.5: Execution of the program with an error inside the function **sum**

```
1. $ ./tyrcc.sh -g example.c
2. $ ./example-tyr
3. ERROR: LLVM line 36:
4.     %12 = load i32, i32* %11, !dbg !35
5. Check (sub %.tyr.deref80 %9) sgt 0# violated (%.tyr.fail94, type-deref/3)
6. example.c: sum() line 6, column 19
```

Source: The author

Figure 1.6: Execution of the program with an error in the call to **sum**

```
1. $ ./tyrcc.sh -g example.c
2. $ ./example-tyr
3. ERROR: LLVM line 68:
4.     %4 = call i32 @sum(i32* %3, i32 4), !dbg !50
5. Check 3# sge 4# violated (%.tyr.fail214, fits/3)
6. example.c: main() line 13, column 18
```

Source: The author

program again, we get a different error (Figure 1.6). This time, the error lies in the LLVM IR instruction which calls **sum** (reported in line 3 of the printout). The check (line 5) indicates a conflict between the provided integer value (4) and the upper bound of the array (3), which was expected to be greater or equal (**sge**) than the provided integer. The source code location of the violation (line 6) is in function **main()**, line 13, column 18, i.e., the incorrect call to **sum**. Týr aborts execution before the function **sum** is called, since the provided value for the second argument does not match the pointer given as the first argument.

1.5 Results

We performed a set of benchmarks to measure the performance impact of the instrumentation inserted by Týr in terms of execution time and memory consumption relative to the non-instrumented programs. In our benchmarks, we observed an average overhead of 25.6% in CPU time, with a median of 12.1%. For all but one of the benchmarks, the observed overhead was below 27%. In the one benchmark which showed a greater overhead, we actually encountered two memory safety bugs, both of which have been caught by Týr's instrumentation. In all cases, the memory consumption overhead was near zero. These figures are similar to the ones reported for Deputy (CONDIT et al., 2007), and generally better than other software-based approaches which rely on keeping separate metadata.

The contributions of this work can be summarized as:

- The design of a dependent type language for LLVM IR for expressing dependencies between pointers and their bounds, including dependent types for pointers, functions, arrays and structures. By aiming at the lower LLVM IR level language, this approach is more generally applicable to any language which can be compiled to LLVM IR, as long as the types in the source language are mapped to the corresponding dependent types at the LLVM IR level.
- A LLVM IR to LLVM IR code transformation which ensures spatial memory safety by inserting run-time checks directed by the type information provided by the programmer.
- A prototype implementation of the proposed system, with experimental evaluation demonstrating that the system can be used to ensure spatial memory safety of C programs with generally reasonable performance overhead.

1.6 Outline

The rest of this work is organized as follows. Chapter 2 presents the conceptual and technical background of this work. Chapter 3 describes the Týr dependent types and code transformation. Chapter 4 presents experimental results demonstrating the efficacy of the system. Chapter 5 presents related work. Chapter 6 presents a conclusion and directions for future work.

2 BACKGROUND

This chapter presents the conceptual and technical background of the current work. Sections 2.1 and 2.2 present the concept of memory safety and an overview of the mechanisms used to ensure it. Section 2.3 deals with dependent types. Section 2.4 presents an overview of how pointers and arrays are used to manipulate memory in C. Section 2.5 describes the LLVM IR language and the subset of LLVM IR used in this work. Section 2.6 discusses the mapping between C and LLVM constructs. Section 2.7 presents a summary.

2.1 Defining memory safety

There is no single definition of memory safety among authors. Broadly speaking:

> **Broad definition.** *A program is said to be memory-safe if it only makes access to regions of memory allocated to it. A language is said to be memory-safe if its semantics guarantee that valid programs written in it are memory-safe.*

This is a very lax definition, however. For instance, by this definition, if two arrays, of ten integers each, are allocated contiguously in memory, a program attempting to access the eleventh position of the first array would still be considered memory safe, because, although the access is out of the bounds of the first array, it still falls within a region of memory allocated to the program (specifically, to the second array). Such a definition is useful in the context of guaranteeing that multiple programs sharing the same memory space do not step over each other's memory (KUMAR; KOHLER; SRIVASTAVA, 2007), but it does not help in preventing buffer overflows and other memory corruption within a single program.

Usually, we are interested in a stricter definition of memory safety which accounts for these situations. It is harder to make such a definition without talking about concepts specific to a particular programming language, but one might generalize definitions such as in Criswell, Geoffray e Adve (2009) as:

> **Stricter definition.** *A program is said to be memory safe if every access to memory happens through a reference to a previously allocated object, the object has not been deallocated, and the region of memory accessed through such a reference has been allocated to that specific object.*

Such a definition is still open to multiple interpretations. For instance, two contiguous arrays belonging to a single data structure might be considered part of a single object, and therefore out-of-bounds accesses to the first array which still fall within the region allocated to the object as a whole might not be considered a violation of memory safety. Some works aim for memory safety in this sense (BERGER; ZORN, 2006), but others rule out such out-of-bounds accesses. In this work, we are interested in this even stricter variant, where fields in a single data structure are treated as individual objects with respect to memory safety.

Memory safety has a *spatial* and a *temporal* aspect. Spatial memory safety refers to ensuring that no out-of-bounds access to memory is performed (with what is considered out of bounds varying with the definition of memory safety used), whereas temporal memory safety refers to ensuring that no access is performed to memory (or to an object) which has already been deallocated or has not been allocated yet.

2.2 Mechanisms for ensuring memory safety

The mechanisms for ensuring each form of memory safety are distinct. For instance, guaranteeing spatial memory safety might involve storing bounds information for arrays and adding run-time checks to ensure that indices are within bounds, while guaranteeing temporal memory safety might involve employing automatic memory managent mechanisms, such as reference counting or garbage collection, which ensure that an object in memory is only deallocated after no references to it remain.

Higher-level programming languages usually enforce both kinds of memory safety, whereas proposed solutions for memory-safe low-level programming vary in what they provide. Some, such as CCured (NECULA et al., 2005) and Cyclone (JIM et al., 2002), attempt to provide both. Others, such as Deputy (CONDIT et al., 2007), provide only spatial memory safety, and can be used together with a complementary solution for temporal memory safety, such as a conservative garbage collector (BOEHM; WEISER, 1988). Like Deputy, this work addresses specifically spatial memory safety.

The remainder of this section presents an overview of various mechanisms used for ensuring temporal and spatial memory safety.

2.2.1 Temporal memory safety

Temporal memory safety violations arise from the use of references to objects that are not present in memory, either because they have never been allocated, or because they have already been deallocated. The first case surfaces in the form of uninitialized pointers; this can be avoided by ensuring, when a pointer is created, that it is initialized either with the address of a valid object in memory, or with a special null value, and that attempts to dereference a null pointer will be detected and handled in some way by the environment; languages like Java do precisely this. The case of access after deallocation poses a greater difficulty. In C and C++, the region of memory allocated to an object may be released (by explicit request from the programmer) even though references to it remain in the program; such references are known as *dangling pointers*. Solutions to this problem ensure no memory access through dangling references are performed, either by taking over control of memory deallocation to ensure that memory is not released while references remain to it, or by constraining the creation of new references to allocated objects.

For instance, in languages such as Java and Haskell, there is no mechanism for the programmer to manually release memory allocated by objects. Rather, the runtime memory management system keeps track of memory allocations. Periodically, a *garbage collector* looks for objects which are allocated in memory but are not accessible by the program, because all references to it are gone, and reclaims the memory taken by those objects. Because memory is only released after no references remain to them, no temporal memory safety violations are possible. A drawback of this approach is that the programmer loses control over when memory is to be deallocated, which may be a problem in memory-constrained systems and when the timing of such operations is important, such as in real-time systems.

A different approach, taken for instance by ATS (CUI; DONNELLY; XI, 2005), is using *linear types* to enforce a policy in the creation and usage of references to objects, such that the compiler knows statically what is the last use of an object, and can emit code to reclaim the memory used by it immediately after.

2.2.2 Spatial memory safety

Spatial memory safety violations arise from accessing a region of memory outside the bounds of an allocated object (or, more strictly, accessing through a reference to an object a region of memory not pertaining to it). The most common case of this is in the indexing of arrays, but it can happen in other situations where a reference may point to objects of variable size, such as unions of differently-sized objects, or in object-oriented languages which allow downcasting an object to a subtype with more fields.

Dynamic checks. One way to solve this problem for arrays is to store the array in memory as a header, containing its length and possibly other metadata, and a payload, consisting of the elements themselves. When an access is performed to the array, run-time checks are performed to ensure that the index of the requested position is a valid index in the array by comparing it with the stored length; if this turns out to be false, a run-time exception is signaled. Languages like Java, Python and Scheme work like this. If implemented naively, this incurs an overhead on every array access. However, it is often possible to prove statically that an access is within bounds, and avoid the run-time check in those cases. For example, given a loop bounded by the length of the vector, like

```
for (i=0; i<vector.length; i++) {
    vector[i] = i*i;
}
```

a compiler can easily verify that the variable **i** only assumes valid indices to **vector**, and therefore a run-time check is not necessary.

Similarly, unions may be stored with a tag indicating which element of the union is active, and checking it at run-time before accessing its contents. Downcasts can also be similarly checked for validity.

Static enforcement. Another way to solve the problem is to ensure that code that might violate memory safety does not get compiled or run, or designing language constructs that make it impossible to express memory-unsafe operations. For instance, tagged unions in ML and Haskell allow the definition of unions of differently-sized types, but it is only possible to access the contents of such values by pattern-matching against the tag of the value; there are no linguistic facilities in these languages to perform an access using the wrong fields of the tag. More generally, strongly-typed languages guarantee that the memory used by a value of one type cannot be reinterpreted as a value of another type, as is possible for instance in C.

Some languages employ dependent types to ensure that only valid indices are used to index an array. Purely static dependent type systems, such as in Idris (BRADY, 2011) and ATS (CUI; DONNELLY; XI, 2005), effectively ensure that the programmer performs a bounds check to verify that an index is valid before trying to access an array, and reject programs when they cannot prove that this is the case. Systems which mix static and dynamic checking, such as Deputy, accept programs when they cannot either prove or disprove that an access is valid, but insert run-time checks in such cases to ensure that an out-of-bounds access is not performed.

2.3 Dependent types

This section presents a theoretical account of dependent types. This information is not strictly necessary for understanding the Týr type system, but is presented here for completeness.

A dependent type system (ASPINALL; HOFMANN, 2004) is one in which types can be indexed by expressions. For instance, one might define a type **Array** ***n*** of arrays of ***n*** elements. By allowing types to be parameterized by expressions, 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 (at compile-time) or dynamically (at run-time). At the same time, this gain in expressiveness may lead to undecidability problems in the type system, if the kinds of expressions that are allowed to appear in types are unconstrained.

An inspiration for dependent type systems comes from the Curry–Howard isomorphism (SØRENSEN; URZYCZYN, 2006), which relates propositions in constructive logic to types, and logical proofs of proposition to terms pertaining to the corresponding types. For instance, a type $A \to B$ of functions from A to B can be interpreted as the corresponding to the logical implication $A \to B$, and a function inhabiting that type can be interpreted as a proof of that implication. The intuition behind this is that a function of type $A \to B$ can be seen as a procedure that takes a proof of A and produces a proof of B, and thus behaves as a proof that given A, one may obtain B.

In this view, a dependent type system is the type-theoretical equivalent of first-order predicate logic: predicates over terms are mapped to types indexed by value terms. The type-theoretical equivalent of universal quantification is the *dependent product type*,

a generalization of arrow types: whereas $A \to B$ denotes a function taking a value of type A and returning a value of type B, the dependent product type $\Pi x : A.\ B$ does the same, but gives a name, x, to the input value of type A, thus allowing it to be referenced within the output type B. This corresponds to the logical proposition $\forall x \in A.\ B$, where x can appear in B. For instance, a function taking an integer n and producing an array of size n might be given the type $\Pi n : Int.\ Array\ n$. The simple arrow type $A \to B$ is a special case of $\Pi x : A.\ B$ where x does not appear in B.

Likewise, the type-theoretical equivalent of existential quantification is the *dependent sum type*, $\Sigma x : A.\ B$, a generalization of pairs where the type of the second component of the pair can be dependent on the value of the first component, i.e., x can appear in B. The intuition behind this correspondence is that the fact that there exists a value x of type A such that $P(x)$ is true, or $\exists x : A.\ P(x)$, can be represented by a pair of one such x and a proof of $P(x)$, i.e., a term inhabiting the type $P(x)$. Analogously to arrow types, plain cartesian pairs are a special case of $\Sigma x : A.\ B$ where x does not appear in B.

Types and kinds. In the context of type systems with parametric types, it is often useful to classify types into *kinds*, in the same way that terms are classified into types (PIERCE, 2002, ch. 29). Kinds allow specifying which type expressions represent complete types, and which are type constructors which have to be applied to other types to make a complete type. Complete types such as Int or $Bool$ or $Pair\ Int\ Bool$ are given the kind $*$, whereas a type constructor such as $Pair$ is given the kind $* \Rightarrow * \Rightarrow *$, meaning it is a type expression which, when applied to two type expressions of kind $*$ (i.e., two complete types), yields a type of kind $*$ (i.e., a complete type, such as $Pair\ Int\ Bool$).

In type systems with conventional parametric polymorphism, kind expressions always only involve the kind $*$ and the kind constructor $\Rightarrow$. In a dependent type system, types can also be parameterized by terms (which have a type), and therefore types such as Int can also appear in the kind of a type expression. For instance, a type constructor like $Array$ might have a kind $* \Rightarrow Int \Rightarrow *$, meaning it takes a type ($*$) and an integer expression (Int) to produce a complete type ($*$).

Application examples. The prototypical example of application for dependent types is the `Vector` type family of lists of a fixed length. For simplicity of exposition, we shall consider vectors with elements of a fixed type $Elem$, rather than polymorphic vectors. One might define the type family of vectors as follows, where :: stands for the kinding relationship, much in the same way as : stands for the typing relationship:

$$Vector :: Nat \Rightarrow *$$

Now we could define a function $concat$, which concatenates two vectors, as having the type:

$$concat : \Pi m : Nat.\, \Pi n : Nat.\, Vector\ m \rightarrow Vector\ n \rightarrow Vector\ (m + n)$$

indicating that $concat$ is a function that takes vectors of sizes m and n, for any natural numbers m and n, and returns a vector whose size of size $m + n$. This allows expressing constraints on the size of vectors at the type level. For instance, a function returning the first element of a vector might be given the type:

$$first : \Pi n : Nat.\, Vector(n + 1) \rightarrow Elem$$

which specifies at the type level that $first$ can only be applied to vectors with at least one element (since $n + 1$ is always greater than or equal to 1 for a natural n). That is, rather than relying on a run-time exception mechanism to catch applications of $first$ to empty vectors, we forbid at the type level the application of $first$ to such vectors. Dependent types thus allow the specification of more complex program invariants at the type level than is possible in simpler type systems.

Expressivity and decidability. 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 (AUGUSTSSON, 1998). 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.

In this work, we will employ dependent types to describe the relationships between pointers and their bounds in LLVM IR programs. The remainder of this chapter will

discuss memory manipulation in C, the LLVM IR language, and the relationship between C and LLVM IR. In Chapter 3, we will define a set of dependent types to describe bounds in pointers, arrays, functions and data structures, and use this information to define a code transformation to ensure spatial memory safety of LLVM IR programs.

2.4 Memory in C: pointers and arrays

Pointers are a typed abstraction of memory addresses, and are the main mechanism for dealing with memory in C. We write $ty*$ for the type of pointers to objects of type ty; we say that ty is the *base type* of the pointer. A pointer may be *null* – meaning it has a distinguished value (all bits zero in most modern architectures) which is considered not to point to any valid object – or it can point to an object of its base type.

A pointed object can be part of an *array*, i.e., a sequence of contiguous objects of the same type in memory. Pointers to adjacent elements of the array can be obtained by performing *pointer arithmetic*: if **a** is a pointer to objects of type ty, and **n** is an integer, then **a+n** is a pointer to the **n**th object of type ty after the one pointed to by **a**. The integer **n** can also be negative, which results into a pointer to an object before **a** in memory. Alternatively, one may subtract an integer from a pointer. In terms of the memory address underlying the pointer, addition and subtraction happens in multiples of the size of the base type. For instance, if **a** is a pointer to 32-bit (4 byte) integers, then **a+1** points 4 bytes after **a**.

If a pointer **a** points to an object of type ty, then it can be *dereferenced* (***a**), thus retrieving the object pointed to. Array indexing is just a convenient notation for performing pointer arithmetic followed by dereference: **a[i]** is syntactic sugar for ***(a+i)**. One can also overwrite the object pointed to with a new value, with either the dereference syntax ***(a+i) = val**, or with its equivalent array indexing syntactic sugar **a[i] = val**.

C enforces no constraints on pointer arithmetic: it is possible to add or subtract an integer to a pointer such that the resulting pointer points to a region of memory outside the bounds of the array the original pointer pointed to. Creating such an out-of-bounds pointer is considered *undefined behavior* in C: “behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which [the C language] International Standard imposes no requirements” (ISO/IEC, 2011, §3.4.3). In principle, this means the compiler is free to emit any code under such circumstances. In practice, what usually happens is that a pointer is created to a region of memory that does not belong to the region the

base pointer was associated with. Dereferencing such an out-of-bounds pointer may yield garbage data, data from other parts of the program, or it may crash the program. Storing into the out-of-bounds pointer may corrupt other data in the program, or may also lead to a crash.

As a special exception, constructing – but not dereferencing – a pointer to the region of memory immediately after a valid object is defined behavior according to the C language standard. This is because it is a common idiom to iterate over an array by incrementing a pointer until it reaches the end of the array, i.e., until it points immediately past the last element of the array.

Arrays are sequences of contiguous objects in memory. In C, arrays are not first-class values: they cannot be passed as arguments or returned from functions directly. Instead, pointers to arrays are passed or returned. Array indexing, as seen above, is a form of pointer arithmetic; a reference to an array is equivalent to a pointer to its first element. Nevertheless, arrays do constitute a data type distinct from pointers in C. Arrays have a base type and a size, the number of elements it has. The compile-time **sizeof** operator, which returns the size in bytes of an object or type, when applied to an array, returns the size of the base type times the number of elements of the array. For instance, given a declaration like:

```
int a[10];
```

sizeof(a) yields 40 (assuming 32-bit integers), even though **a** usually behaves like a pointer. However, when arithmetic is performed with an expression with an array type, the result has a pointer type: **sizeof(a+0)** yields the size of a pointer (4 in a 32-bit architecture, 8 in a 64-bit architecture), even though **a** and **a+0** point to the same address.

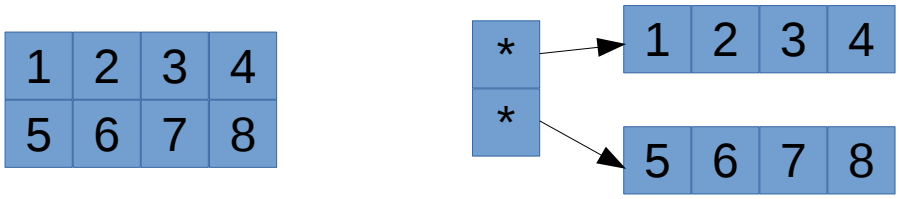
These distinctions are important because at the LLVM IR level, there is a clear separation between arrays and pointers. Both are first-class values, and the conversion from an array to a pointer to its first element, which is implicit in C, is an explicit operation in LLVM IR.

Multidimensional arrays are treated conceptually as arrays of arrays. For instance, given a declaration like:

```
int m[5][10];
```

m will have the type **int [5][10]**, an array of 5 elements where each element is itself an array of 10 elements. **sizeof(m)** will be $5 \times 10 \times$ **sizeof(int)**. **m** will be equivalent to the address of the beginning of the array.

Figure 2.1: A 2-dimensional array (left) and an array of pointers to two 1-dimensional arrays (right). C accesses both kinds of structures using the same syntax.



Source: The author

C uses the syntax **m[i][j]** to access the element in position **j** within the sub-array **i**. Remember, however, that the array index syntax is syntactic sugar to pointer arithmetic plus dereference. To make multidimensional indexing work, C makes it so that **m[i]** (or, equivalently, ***(m+i)**) yields a pointer to the **i**th sub-array, so that **m[i][j]** gets the **j**th element within that sub-array. Since **m+i** (without the dereference) would already be a pointer to the **i**th sub-array, **m+i** and ***(m+i)** represent the same address. Their types, however, differ: **m+i** is a pointer to arrays of 10 integers, whereas **m[i]** is an array of 10 integers itself.

In this way, C uses the same syntax to access the elements in a true multidimensional array and to access elements through an array of pointers to other arrays. That is, data structures like those in Figure 2.1 are accessed in the same way syntactically: C will convert arrays to pointers in such a way as to make it possible to access each sub-array of the 2-dimensional array as if they were individual arrays in a transparent way. This is not the case in LLVM IR, where multidimensional array indexing is a completely distinct operation from access through an array of pointers.

2.5 The LLVM Intermediate Representation language

LLVM is a widely used language-agnostic framework for program compilation, analysis and transformation designed around a uniform Intermediate Representation (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 2.2 shows an example code snippet in C and how it might be represented

Figure 2.2: Sample C function and equivalent LLVM IR code

```
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 = icmp slt i32 %i, 0
  br i1 %1, label %then, label %else

 then:
  %2 = add i32 %i, %i
  store i32 %2, i32* %result
  br label %end

 else:
  %3 = getelementptr i32, i32* %x, i32 %i
  %4 = load i32, i32* %3
  store i32 %4, i32* %result
  br label %end

 end:
  %5 = load i32, i32* %result
  ret i32 %5
}
```

Source: The author

in LLVM.

The LLVM IR language is in Static Single Assignment (SSA) form (ALPERN; WEGMAN; ZADECK, 1988) with respect to its registers: each register is assigned exactly once, and each definition dominates all of its uses, i.e., a register can only be used in a given instruction if the control flow graph of the code ensures that the definition will be reached before the use.

Memory access is done through typed pointers. Pointers are even more prominent in LLVM IR than in C, as all memory other than registers is referenced through pointers. This includes functions in function calls, global variables (which are accessed through pointers to the location where the value of the global is stored), and local variables other than registers (which are accessed through pointers to the stack). The Static Single Assignment restriction applies only to registers, not to memory, i.e., there is no restriction on the number of assignments to a given memory location.

The full LLVM IR language has a large number of instructions, optional flags and other features not directly relevant to this work. For simplicity of exposition, we define a relevant subset of the LLVM IR language, which covers the most important operations related to memory safety. In Chapter 3, we will describe the Týr code transformation in terms of this subset, although the actual implementation of the system operates on the full LLVM IR language. Figure 2.3 shows a grammar of this subset. The next sections describe it in more detail. A description of the full LLVM IR language can be found in LLVM (2015a).

Figure 2.3: Subset of LLVM IR considered in this work. An overline indicates a sequence of zero or more of the overlined element.

Module:	mod	::=	$\overline{tydef}\ \overline{prod}$
Type definition:	$tydef$	::=	id **=** **type** $\{\overline{ty}\}$
Product:	$prod$	::=	gid **=** **global** $const$
		|	gid **=** **constant** $const$
		|	**define** $ty\ gid\ (\overline{ty\ id})\{\overline{blk}\}$
		|	**declare** $ty\ gid\ (\overline{ty})$
Basic block:	blk	::=	$label$**:** $\overline{cmd}\ term$
Label:	$label$	::=	id
LLVM Types:	ty	::=	isz | ty***** | $[n \times ty]$ | **void**
		|	$\{\overline{ty}\}$ | $ty(\overline{ty})$ | id
Global ids:	gid	::=	**@a** | **@b** | ...
Local ids:	id	::=	**%a** | **%b** | ...
Constants:	$const$	::=	$ty\ n$ | ty **zeroinitializer** | $ty\ [\overline{const}]$ | gid | **void**
Values:	val	::=	$const$ | id
Binary ops.:	bop	::=	**add** | **sub** | **mul** | **sdiv** | **udiv**
Cast ops.:	$cast$	::=	**bitcast** | **inttoptr** | **ptrtoint**
		|	**sext** | **zext** | **trunc**
Compare ops.:	cmp	::=	**eq** | **ne** | **ult** | **ule** | **ugt** | **uge**
		|	**slt** | **sle** | **sgt** | **sge**
Instructions:	$inst$	::=	cmd | $term$
Commands:	cmd	::=	id **=** $bop\ val_1, val_2$
		|	id **=** **icmp** $cmp\ val_1, val_2$
		|	id **=** **load** val
		|	**store** val_1, val_2
		|	id **=** **getelementptr** val, val_{idx}
		|	id **=** **getelementptr** $val,$ i$sz\ 0, val_{subidx}$
		|	id **=** **alloca** ty, val
		|	id **=** $cast\ val$ **to** ty
		|	id **=** **call** $val\ (\overline{val})$
Terminator:	$term$	::=	**ret** val
		|	**br** $val, label_1, label_2$
		|	**unreachable**

Source: The author

2.5.1 Top-level structure

The basic unit of compilation in LLVM is the *module*. A module contains type definitions, which allow assigning names to structure types, and declarations of global variables, constants and functions, collectively known as *products*.

A global variable or constant definition contains the name of the global, the type of the contents, and an initializer. Note, however, that globals in LLVM IR are treated as pointers to the memory location where the value of the global is stored. Therefore, given a declaration of a global 32-bit integer variable like **@a = global i32 42**, **@a** will have the type **i32***, i.e., a *pointer* to the place where the integer **42** is stored. To access or change the value of the global, one must use the **load** and **store** instructions for memory manipulation.

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 consists of a label and a sequence one or more instructions. The last instruction, and only the last instruction, in a basic block, must be a *terminator*, an instruction which finishes exection of the current block, such as a transfer of control to another basic block (branch) or to the function caller (return). Non-terminator instructions are collectively called *commands*.

A function *declaration* is like a function definition, but does not include parameter names or a function body. They are used to declare the availability of functions declared in other modules, or builtin LLVM functions known as *intrinsics*.

A type definition consists of a name on the left-hand side, and a structure type on the right hand side. Only structure types can appear on the right-hand side.[1] The named type thus created is considered distinct from any other type, even one with an identical definition.

2.5.2 Types

LLVM IR has integer types isz for arbitrary positive integer sizes sz. The most frequently ocurring ones in LLVM IR code are: the 1-bit boolean type i1, which is used

[1] In the full LLVM IR language, any type can appear on the right hand side, but named non-structure types are essentially equivalent to their anonymous equivalents, whereas named structure types are considered distinct types.

for instance for the result of comparison operators; and $\mathrm{i32}$ and $\mathrm{i64}$, which correspond to the most common integer sizes in C code. Unlike C, LLVM IR does not have separate types for signed and unsigned integers; rather, distinct instructions are used for signed and unsigned operations. However, LLVM assumes two's complement representation for integers; therefore, operations that produce the same result under two's complement whether the values are viewed as signed or not, such as addition and subtraction, don't have separate signed and unsigned instructions. Also unlike C, LLVM IR never performs implicit casts between integer types (or any other implicit casts, for that matter): every cast must be performed by an explicit operation in the IR.

ty$*$ is the type of pointers to values of type *ty*.

$[n \times ty]$ is the type of arrays of n elements of type *ty*; n must be a constant. Unlike C, arrays are first-class values in LLVM IR, and they don't behave like pointers. The implicit conversion in C from an array to a pointer to its first element is an explicit operation at the IR level, performed by the **getelementptr** instruction.

$\{ty_1, \ldots, ty_n\}$ is the type of a structure whose fields have types $ty_1, \ldots, ty_n$. Note that this type is anonymous; a type definition can be used to assign a name to such a structure type.

$ty(ty_1, \ldots, ty_n)$ is the type of a function taking arguments of types $ty_1, \ldots, ty_n$ and returning a type *ty*. Functions are not a first-class type in LLVM IR; where a function value is needed, for instance in the **call** instruction, a pointer to the function is used instead.

void is used as the return type of functions which don't return a meaningful value. For simplicity of exposition, in our simplified LLVM IR grammar, we consider **void** as a type containing a single value, also named **void**, similar to the unit or () type in languages like Haskell, Rust, and Standard ML. Functions returning **void** are considered to return this dummy value. This allows us to treat calls to void-returning functions in the same way as calls to other functions; otherwise, we would need different production rules for **call** instructions which assign their return value to an identifier vs. those that don't return a value, and likewise for **ret** instructions.

Finally, *id* represents a named type, i.e., one defined with a type definition.

2.5.3 Values

Values in LLVM IR are either constants or registers. Because a bare constant such as **0** does not have a specific type (i.e., it could be an i32 or i64 or any other integer type),

all constants are preceded by an explicit type (e.g., **i32 0**).[2]

The constant **zeroinitializer** represents a value where all bits are zero. For integers, that is equivalent to **0**, but **zeroinitializer** can be used as a zero value for any first-class type, including aggregate types like arrays and structures, and is often used as an initializer, as its name suggests. Like integers, **zeroinitializer** is always preceded by an explicit type.

$[n \times ty]$ $[const_1, \ldots, const_n]$ represents an array of type $[n \times ty]$ whose elements are $const_1, \ldots, const_n$. Note that only constants can be used as initializers.

Global variables, constants and functions are written with a preceding @. Because the name of a global is actually a pointer to the location where the value of the global is stored, globals are considered (link-time) constants, because the *pointer* represented by a global name never changes during the execution of the program.

Registers names are written with a preceding %. They represent function-local definitions in SSA form.

As a convenience, **false** and **true** are used as aliases for **i1 0** and **i1 1**, i.e., constants of the boolean i1 type. In this work, we also consider **null** as an alias for **zeroinitializer**. This assumes that the null pointer has all bits set to zero, which is the case in modern architectures. This assumption simplifies initializing arrays and structures containing pointers, as we can initalize the entire aggregate value to **zeroinitializer** and assume that all contained pointers will be null.

2.5.4 Instructions

Arithmetic instructions (**add**, **sub**, **mul**, **sdiv**, **udiv**) work as expected. **sdiv** and **udiv** represent signed and unsigned division, respectively.

The **icmp** instruction performs a given comparison (e.g., **ult** for *unsigned less than*) between two given integers, yielding a boolean.

The **load** instruction takes a pointer of some type $ty*$ and yields the value of type *ty* stored at the location referenced by the pointer. Likewise, **store** takes a value of some type *ty* and a pointer of type $ty*$ and stores the value in the location referenced by the pointer.

[2] In the actual LLVM IR language, type annotations are placed differently depending on the specific instruction, and register values also carry type annotations. In our simplified grammar, only constant values carry annotations, and they are always placed immediately before the constant, regardless of the instruction.

getelementptr is the instruction responsible for computing addresses from a base address, i.e., for performing pointer arithmetic and computing addresses of sub-elements of aggregate values. In its most general form, as it appears in the full LLVM IR language, this instruction has the form id **= getelementptr** $p, idx_0, \overline{idx_i}$. **getelementptr** takes a base pointer p and one or more indices. The first index idx_0 is used as an offset from the base pointer, and corresponds directly to the C notion of pointer arithmetic: idx_0 times the size of the base type of p is added to the address represented by p. The remaining indices can only be used when the base type of p is a (potentially nested) aggregate type. Each successive index descends one level into the aggregate value, computing the address of a sub-element of it. For instance, consider a pointer p of type $[5 \times [10 \times i32]]*$, i.e., a pointer to a 5×10 array of integers (which in LLVM IR is represented as an array of arrays of integers). In an instruction like:

%q = getelementptr $[5 \times [10 \times$ **i32**$]]*$ **p, i32 0, i32 5, i32 7**

the first index, **i32 0**, indicates an offset from **p** in multiples of the base type, $[5 \times [10 \times$ **i32**$]]$. This index would be useful if we were dealing with a pointer to a region containing multiple $[5 \times [10 \times$ **i32**$]]$ arrays. Since this is not the case, we use an offset of zero. The second index, **i32 5**, moves the pointer to the sixth sub-array (since indices are zero-based). The third index, **i32 7**, moves the pointer to the eighth integer within that sub-array. The result is a pointer of type **i32**$*$.

getelementptr is often used to perform the implicit C conversion from an array to a pointer to its first element. For instance, whenever a string constant is used as an argument to a function taking **char ***, the resulting LLVM IR code uses **getelementptr**. For example, consider C code like:

```
puts("Hello");
```

In the corresponding LLVM IR code, the string will be declared as a global constant:

```
@.str = constant [6 x i8] c"Hello\00"
```

where **c"Hello\00"** is syntactic sugar for a six-element array of **i8** (8-bit integers) containing the characters, and **\00** is the null terminator at the end of the string, which is implicit in C but explicit in LLVM IR. Then, when performing the function call, first the code must obtain a pointer into the array, and then pass the pointer to the **puts** function:

```
%p = getelementptr [6 x i8]* @.str, i32 0, i32 0
call i32 @puts(i8* %p)
```

The first zero index is the "horizontal" offset from the **@.str** pointer – globals are always pointers to their contents, so **@.str** is a pointer to an array of characters, and the first zero index tells we want the first (and only) such array in the pointed region. The second zero index descends into the array, selecting the first element within it.

To simplify the description of the code transformation in the next chapter, we will consider that code is normalized in such a way that there are only two forms of the **getelementptr** instruction: either a single index is used (for C-style pointer arithmetic), or two indices are used and the first is always zero (for descending a single level into a data structure). Any more complex **getelementptr** instruction can be broken down into sequences of those two forms.

alloca takes a type and a quantity, allocates that many elements in the stack, and returns a pointer to the allocated region. This is often used to allocate storage for local variables. For instance, to allocate a local 64-bit integer variable, code might be emitted like:

```
%x = alloca i64, i32 1
```

where **i32 1** is how many integers we want to allocate. As is the case with globals, **%x** is a pointer to the location where the integer is stored, not the integer itself. One must **load** from the pointer to access the value of the integer.

The casting instructions are used to perform various kinds of type conversions. **sext**, **zext** and **trunc** signal-extend, zero-extend and truncate integer values, respectively. **inttoptr** and **ptrtoint** convert between integers and pointers. **bitcast** converts between types with the same size. The source and destination types must either be both pointers or both non-pointers.

call is the function call instruction. It takes a pointer to the function to be called and a list of arguments, and yields the value returned by the function.

The three terminator instructions are **ret**, **br** and **unreachable**. **ret** returns from a function, yielding a value to the function caller. **br** is the branching instruction. It takes a boolean value and two labels, and transfers control to the block labeled by the first label if the boolean is true, and to the second one if false. In our simplified grammar, there is no special form of this instruction for unconditional branching; unconditional branches can be written as **br true**, $label$, $label$. **unreachable** is a special terminator indicating that control should never reach the end of the basic block. This is used for instance after a call

to a function that never returns, like **exit**, because LLVM IR always requires a terminator at the end of each block.

2.5.5 Omissions from the full language

The language described here is meant to be a core for the description of the rules of Týr, which are mostly concerned with memory safety. As such, a number of features of the full LLVM IR language which were not directly relevant to this work have been omitted.

Features omitted in the core grammar, but supported by the implementation:

- Floating point types, since they are never involved in pointer computation, as well as other types not related to memory safety, such as packed structures.
- Control flow instructions other than **br** and **ret**.
- Constant expressions for integer and pointer arithmetic, comparison and casting, since they are mostly equivalent to their corresponding instructions.

Features omitted from the core, but partially supported by the implementation:

- Variable-arity functions are partially supported by the implementation: the system allows the definition and use of variable-arity functions, such as **printf**, but does not perform type-checking for the non-fixed arguments.

Featured omitted from the core and currently not supported by the implementation:

- Structure values, as opposed to pointers to structures: in our experience, code emitted by Clang always works with structures by means of pointers: global variables, as seen above, are always pointers, and so are locals created with **alloca**. Function parameters may contain structure values, but those are copied to locals in the stack as soon as the function starts. The fields of these structures in memory are always accessed by first computing a pointer to the desired field, then accessing the pointer. Because of this, the instructions dealing with structure *values*, as opposed to pointers to structures, have been omitted.
- Instructions dealing with vectorized data, such as SIMD instructions, and their related types.
- Instructions related to threads and synchronization.
- Instructions and features related with casting across multiple address spaces, which

Figure 2.4: Example of ϕ-node in SSA-form

(a) Original program

```
if (x > 0)
  y = 1;
else
  y = 2;

f(y);
```

(b) SSA-form program

```
if (x > 0)
  y1 = 1;
else
  y2 = 2;

y = phi(y1, y2)
f(y);
```

Source: The author

are used in contexts of GPUs and other alternative architectures.

Although the features not supported are not particularly important for the present work, they should be addressed by future work for the sake of completeness.

2.5.6 Phi instructions

In SSA languages, ϕ-nodes are used in situations where a register must acquire different values depending on where control is coming from. For instance, suppose we have a C program like that in Figure 2.4a. This code assigns different values to **y** depending on the branch taken by the **if**. This cannot be directly translated into an SSA-form language, because SSA requires that a name is assigned exactly once in the code. To work around this restriction, in SSA-form code, each branch of the **if** uses a different variable, and a ϕ-node is used at the point where control flow from both branches merges (i.e., after the **if**) to assign a value to **y** depending on which branch reaches the ϕ-node. Figure 2.4b shows the resulting program in pseudo-code.

The **phi** instruction is the LLVM IR implementation of ϕ-nodes. The code emitted by Clang, however, is mostly devoid of **phi** instructions; instead, all local variables are created in the stack using the **alloca** instruction, and, as noted before, LLVM IR is not in SSA-form with respect to memory. Later during compilation, at the LLVM level, a pass called **mem2reg** converts those stack-allocated values into registers and adds **phi** instructions accordingly. Since Týr is run on the code emitted by Clang before it is passed on to the LLVM level for optimization and compilation, we do not have to deal with **phi** instructions directly. Ocasionally, **phi** instructions are emitted by Clang in special circumstances. To make sure that no **phi** instructions are present when code reaches Týr, we run an LLVM pass, called **reg2mem**, which does the opposite of **mem2reg**, converting all **phi** instructions into stack-allocated locals, before Týr is applied to the code. This is

done automatically as part of the Týr compilation routine.

2.6 Correspondence between C and LLVM IR

Although in principle there is no single way to translate a C program into LLVM IR (just like there is no single way to translate C into assembly language), the LLVM IR language was designed with C-like languages in mind, and the Clang compiler does perform a very straightforward translation of C code to LLVM IR. Therefore, it is generally possible to define a correspondence between constructs in each language. This section presents the most important aspects of this correspondence.

2.6.1 Types

The C integer types are mapped to integer types of the appropriate size in LLVM IR. Whereas the sizes of C integer types may vary with the architecture and platform, LLVM integer types always have a fixed size, therefore the mapping is architecture-dependent. For instance, **long int** may be mapped to $\mathrm{i}32$ or $\mathrm{i}64$.

LLVM IR does not have a void pointer type. C's **void *** is mapped to the type of pointers to bytes, $\mathrm{i}8*$.

LLVM IR arrays are used for translating fixed-size arrays in C, such as global and local arrays declared with a constant size. Array constants in C, such as string constants and array initializers, are represented by global array constants in LLVM IR. For instance, consider a C function which declares and initializes a local array like **int a[] = {23, 42, 81}**. In LLVM, the initializer will become a global constant, and the code for the function will allocate a local array in the stack and then copy the value of the constant into the newly allocated array. This is because each instantiation of the function must manipulate a fresh copy of the array, not the initializer which is shared by the whole program.

As mentioned before, arrays are first-class values entirely distinct from pointers in LLVM IR, and the implicit conversion between arrays and pointers which happens in C is performed explicitly at the LLVM IR level by means of the **getelementptr** instruction.

Definition of a structure named *name* in C is mapped to a definition of a type **%struct.*name*** in LLVM IR.

2.6.2 Variables and functions

Global variables in C are directly mapped to LLVM IR globals. Local variables are mapped to a stack allocation through the **alloca** instruction. For example, if the source contains the declaration of a local variable like **int x**, the translated code will allocate an integer on the stack and assign a pointer to the integer to a local register, usually with the same name as the original variable: **%x = alloca i32, i32 1**, where **i32 1** is the size of the allocation.

C functions are straightforwardly mapped to LLVM IR functions. However, the parameters of LLVM IR functions are considered registers, and therefore are immutable due to the SSA restriction, unlike their C equivalents. Therefore, at the beginning of the function, the translated code will allocate a local variable for each parameter and copy the values of the parameters to these locals. For instance, if the C source has a function **void f(int x)**, the corresponding LLVM IR code will look like:

```
define void @f(i32 %x) {
    %1 = alloca i32
    store i32 %x, i32* %1
    ...
}
```

where a new integer is allocated in the stack, and the **%x** function parameter is copied to the new location at the beginning of the function. This is done to preserve the mutability of the **x** variable at the LLVM IR level.

2.7 Summary

In this chapter we have seen an overview of memory safety, dependent types, the LLVM IR language and its relationship with C. We also have defined a subset of LLVM IR which includes the core features of the language involved in pointer manipulation. In the next chapter, we will define a set of dependent types for describing the relationships between pointers and their bounds in LLVM IR, and a code transformation guided by those types to ensure spatial memory safety of LLVM IR programs.

3 THE TÝR CODE TRANSFORMATION

We propose Týr, a code transformation for LLVM IR based on dependent types for pointers, functions, arrays and structures that ensures the correct usage of these constructs with respect to spatial memory safety. This chapter describes the approach in detail and is organized as follows. Section 3.1 presents some considerations taken into account in the design of the algorithm. Section 3.2 presents the set of dependent types and expressions used by Týr. Section 3.3 presents an overview of the instrumentation algorithm, considering its global (module-level) and the local (function-level) aspects. Section 3.4 presents the rules for assigning Týr types to LLVM IR values and instrumenting LLVM IR functions according to those types. Section 3.5 revisits the global aspect of the instrumentation, giving a more complete explanation about the construction of the global environment. Section 3.6 details how instrumentation actions are emitted as actual LLVM IR instructions. Section 3.7 discusses the soundness of the algorithm with respect to spatial memory safety. Section 3.8 presents some efficiency considerations. Section 3.9 provides a summary.

3.1 Design considerations

Týr is conceived as a type-based code transformation for the LLVM IR language. Although in this work we focus on supporting those features of LLVM IR necessary for ensuring spatial memory safety of C programs, we intend Týr to be more generally useful as a *foundation* for spatial memory safety in other languages targeting LLVM IR with a similar memory model, such as C++ and Objective C.

The design of the system has to balance a number of constraints. The type language adopted for the annotations must be expressive enough to describe the common idioms used in C programs to keep track of bounds information, while at the same time avoiding the problem of undecidability. We addressed these problems in two ways. First, to guarantee termination, the expressions which can appear in dependent types comprise a restricted sub-language. This sub-language allows constants, arithmetic operations, and references to local variables, function parameters and structure fields, which is generally sufficient to describe bounds in C programs, but excludes other operations such as function calls, loops and side effects. Second, checks which depend on evaluation of expressions are postponed to run-time: rather than performing the check at compile-time,

the code is instrumented to perform the check and abort execution if the check fails. This makes Týr more flexible, by allowing bound expressions to depend on information only available at run-time.

Another aspect which influenced the design of Týr is efficiency. Because the run-time checks inserted by the instrumentation have a cost in execution time, it is desirable to avoid inserting unnecessary checks, and to try to make the checks that are inserted optimizer-friendly, making them easier to be eliminated by further compilation passes. A number of decisions have been made to make this possible, as will be seen in Section 3.8.

Týr acts as an LLVM pass, taking LLVM IR code as input (together with type annotations) and producing instrumented LLVM IR code as output. Rather than changing the LLVM IR language to augment it with a new type system, the Týr type-based code transformation takes the dependent type information it needs as a separate input, leaving the LLVM IR source unchanged. There is some precedent for that within the Clang/LLVM framework itself: *type-based alias analysis* (TBAA) of C/C++ code is performed by an LLVM IR pass which takes type information from Clang in the form of LLVM IR metadata: "In LLVM IR, memory does not have types, so LLVM's own type system is not suitable for doing TBAA. Instead, metadata is added to the IR to describe a type system of a higher level language." (LLVM, 2016) In the current Týr implementation, we use a separate input file rather than embedded metadata, but the principle is similar: a richer set of types described on the top of LLVM IR without modifying the LLVM IR language. This makes it possible to reuse the entire Clang/LLVM infrastructure with no modifications to accomodate the new types.

3.2 Týr types

This section will introduce the type language used by Týr. Section 3.2.1 will present an overview of each Týr type. Section 3.2.2 will discuss the correspondence between LLVM IR types and Týr types. Section 3.2.3 will introduce the concept of compatibility between Týr types.

Figure 3.1: Grammar of Týr types

Týr types:

$$
\begin{array}{rcl}
TY & ::= & \textbf{isz} \mid \textbf{void} \mid id \\
& \mid & Ptr^{\pm}(TY, \delta_1, \delta_2) \\
& \mid & SPtr^{\pm}(TY, \delta_1, \delta_2) \\
& \mid & Local(TY) \\
& \mid & Array(n, TY) \\
& \mid & SArray(n, TY) \\
& \mid & Fn^{\pm}\ TY\ (\overline{id : TY}) \\
& \mid & Struct\ id\ (\overline{id : TY}) \\
& \mid & Field(TY, val, id)
\end{array}
$$

Non-nullness:

$$
\pm \quad ::= \quad + \mid -
$$

Dependent type expressions:

$$
\begin{array}{rcl}
\delta & ::= & val \\
& \mid & \delta_1\ op\ \delta_2 \\
& \mid & *val \\
& \mid & val_{TY}.id \\
& \mid & sizeof(ty) \\
op & ::= & + \mid - \mid * \mid /
\end{array}
$$

Source: The author

3.2.1 Overview of Týr types

Figure 3.1 presents a grammar of Týr types. In this work, we use the convention of writing *ty* to represent LLVM IR types and TY to represent Týr types. This section will present an overview of all Týr types. In further sections, their properties will be discussed in greater detail.

Týr has a variety of pointer types. $Ptr^{\pm}(TY, \delta_1, \delta_2)$ is the more general pointer type. It has a base type TY, and lower and upper bounds δ_1 and δ_2. The lower and upper bounds are integer expressions indicating the range of indices which are within the bounds of the region referenced by the pointer. The lower bound is inclusive, while the upper bound is exclusive, or, in other words, the lower bound is the first valid index into the referenced region, and the upper bound is the first index after the valid region.

Ptr pointers also have a *non-nullness tag*, written as a sign superscript after the Ptr constructor. A pointer is *non-nullable* (Ptr^{+}) if it is known at compile-time not to be null; it is *nullable* (Ptr^{-}) if it is not known to be non-null (i.e., it may or may not be null). This feature is used to avoid inserting unnecessary checks for nullness: a null check may be necessary when *constructing* a Ptr^{+} pointer to ensure it is not null, but once a pointer is tagged Ptr^{+}, null checks are not necessary when accessing it.

Most operations are allowed on Ptr pointers, subject to bounds and nullness checking, but they cannot appear in dependent type expressions, and their bound expressions δ_1 and δ_2 cannot refer to local variables. The exact rules will be presented in Section 3.4.2.

$SPtr^{\pm}(TY, \delta_1, \delta_2)$ is the type of *string pointers*, a variant of Ptr used for regions

of memory delimited by a null terminator, like a C string delimited by the ASCII NUL character. Their bounds are handled specially, allowing accesses past the declared upper bound as long as the null terminator is not overstepped. String pointers will be discussed in more detail in Section 3.4.3.

$Local(TY)$ is a more restricted pointer type, used for local variables in the stack. $Local$ pointers can appear in dependent type expressions, and their base type TY can contain references to other local variables, but there are some restrictions in their usage which effectively ensure that they are never aliased, i.e., the region of memory accessible through a $Local$ pointer is not accessible through any other pointer. Local pointers will be discussed in more detail in Section 3.4.4.

$Field(TY, val, id)$ is used for pointers to structure fields. They keep track of the structure value (val) and field (id) they came from, their base type TY can contain references to other fields of the same structure, but they have usage restrictions resembling those of $Local$ pointers. They will be discussed in more detail in Section 3.4.6.

This variety of pointer types arises from the necessity to accomodate the usage patterns in the LLVM IR language while ensuring safety, as will be seen in Section 3.4. This can be contrasted to Deputy (CONDIT et al., 2007), which has a single pointer type (with some modifiers), but disallows pointers to local variables and structure fields involved in type dependencies. This kind of restriction is not an option in Týr, because in LLVM IR code as emitted by Clang, access to local variables and structure fields is always performed by means of pointers. Therefore, the same restrictions which are performed at the syntactical level in systems like Deputy, are realized as type-level restrictions in Týr.

$Array(n, TY)$ is the type of arrays of n elements of type TY. This type corresponds to the LLVM $[n \times ty]$ type. $SArray(n, TY)$ is the type of string arrays, arrays whose last element is guaranteed to be a null terminator. Array elements are always accessed by first constructing pointers to the elements, so they will be discussed together with Ptr and $SPtr$ pointers in Section 3.4.2.

$Fn^{\pm}\ TY_{ret}\ (id_1 : TY_1, \ldots, id_n : TY_n)$ is the type of pointers to functions taking arguments named $id_1, \ldots, id_n$ of types $TY_1, \ldots, TY_n$, respectively, and returning a value of type TY_{ret}. The parameter types and the return type are allowed to depend on parameter values, which is why the parameters are named in the type. For instance, a function taking an integer len and a pointer p to a region of len integers might be given the type Fn^{+} **void** $(len : \text{i32}, p : Ptr^{-}(\text{i32}, 0, len))$. Like Ptr and $SPtr$ pointers, Fn pointers also come in two flavors: nullable (Fn^{-}) and non-nullable (Fn^{+}). Function pointers will

be discussed in Section 3.4.7.

$Struct\ id\ (id_1 : TY_1, \ldots, id_n : TY_n)$ is the type of a structure named id with fields $id_1, \ldots, id_n$ of types $TY_1, \ldots, TY_n$, respectively. The name id may be absent (written $\varnothing$) when the structure is anonymous. As with functions, the types of the fields may depend on the values of other fields. For instance, a structure containing a pointer to a buffer and its length might be given the type $Struct\ buf\_t\ (data : Ptr^{-}(\mathrm{i8}, 0, len), len : \mathrm{i32})$. Structures will be discussed together with structure field pointers in Section 3.4.6.

Integer, void, and named types are exactly equivalent to their corresponding LLVM IR types. The size of the integers used for bounds depends on the size of pointers in the target architecture of the program being compiled, typically the size of the architecture word (e.g., $\mathrm{i32}$ for a 32-bit processor and $\mathrm{i64}$ for a 64-bit one). We write iWORD to refer to that integer type in the architecture of interest without specifying a concrete architecture.

3.2.2 Correspondence between LLVM IR and Týr types

Týr obtains dependent type information from type annotations provided by the programmer. However, it would be impractical to require the programmer to annotate *every* expression in the LLVM IR program: not only that would require a huge amount of work by the programmer, but it also would require the programmer to give types to temporary values generated by the compiler, which the programmer usually is not aware of. To avoid this problem, Týr allows the type annotation for any identifier to be omitted, in which case a default Týr type is inferred based on the LLVM IR type.

Because the Týr types are richer than the LLVM IR ones, for a given LLVM IR type there are usually many possible corresponding Týr types. For instance, an LLVM IR pointer type like $\mathrm{i8}*$ can correspond to a Týr $Ptr^{-}(\mathrm{i8}, lo, hi)$ for infinitely many possible expressions lo and hi. In these cases, Týr has to choose a sensible default. For pointers, Týr uses the default bounds $[0, 1)$, which mean the pointer points to exactly one object of the given base type. Figure 3.2 shows the default mapping from LLVM IR to Týr types in absence of further type information. We write $\lceil ty \rceil$ to denote this "lifting" from LLVM IR type *ty* to a corresponding Týr type. Note that the Týr function and structure types require naming the parameters and fields, so that they can be referenced in dependent type expressions, whereas the corresponding LLVM IR level function and structure types do not give names to parameters and fields. Therefore, we have to fill in dummy names when mapping the LLVM IR types to Týr types. The concrete names used are immaterial

Figure 3.2: Default mapping $\lceil \cdot \rceil$ from LLVM IR types to Týr types

$$
\begin{array}{ll}
\lceil \mathrm{i}sz \rceil & = \mathrm{i}sz \\
\lceil id \rceil & = id \\
\lceil \textbf{void} \rceil & = \textbf{void} \\
\lceil ty_{ret}(\overline{ty_i})* \rceil & = Fn^{-}\ \lceil ty_{ret} \rceil\ (\overline{id_i : \lceil ty_i \rceil}), \text{ for freshly-created } id_i \\
\lceil ty* \rceil & = Ptr^{-}(\lceil ty \rceil, 0, 1) \\
\lceil [n \times ty] \rceil & = Array(n, \lceil ty \rceil) \\
\lceil \{\overline{ty_i}\} \rceil & = Struct\ \varnothing\ (\overline{id_i : \lceil ty_i \rceil}), \text{ for freshly-created } id_i
\end{array}
$$

Source: The author

Figure 3.3: Mapping $\lfloor \cdot \rfloor$ from Týr types to LLVM IR types

$$
\begin{array}{ll}
\lfloor \mathrm{i}sz \rfloor & = \mathrm{i}sz \\
\lfloor id \rfloor & = id \\
\lfloor \textbf{void} \rfloor & = \textbf{void} \\
\lfloor Fn^{\pm}\ TY_{ret}\ (\overline{id_i : TY_i}) \rfloor & = \lfloor TY_{ret} \rfloor (\lfloor \overline{TY_i} \rfloor) * \\
\lfloor Ptr^{\pm}(TY, lo, hi) \rfloor & = \lfloor TY \rfloor * \\
\lfloor SPtr^{\pm}(TY, lo, hi) \rfloor & = \lfloor TY \rfloor * \\
\lfloor Local(TY) \rfloor & = \lfloor TY \rfloor * \\
\lfloor Field(s, val, fld) \rfloor, \text{ where } s = Struct\ \_\ (\ldots, fld : TY_{fld}, \ldots) & \\
\qquad & = \lfloor TY_{fld} \rfloor * \\
\lfloor Array(n, TY) \rfloor & = [n \times \lfloor TY \rfloor] \\
\lfloor SArray(n, TY) \rfloor & = [n \times \lfloor TY \rfloor] \\
\lfloor Struct\ \varnothing\ (\overline{id_i : TY_i}) \rfloor & = \{\lfloor \overline{TY_i} \rfloor\} \\
\lfloor Struct\ id\ (\overline{id_i : TY_i}) \rfloor & = id
\end{array}
$$

Source: The author

(provided that they are unique), since the types of parameters and fields in these default Týr types do not contain references to each other.

Conversely, when instrumenting the code, Týr must be able to translate the Týr-level types back to LLVM IR ones in order to correctly emit instructions. Therefore, a mapping from Týr to LLVM IR types is necessary. There is always a single LLVM IR type corresponding to a Týr type; the mapping consists essentially of dropping all dependent type information. Figure 3.3 shows this mapping. We write $\lfloor TY \rfloor$ to denote this lowering from Týr type TY to the corresponding LLVM IR type.

3.2.3 Type compatibility

An important concept in the system is that of *type compatibility*. Type compatibility tells which Týr types can be used when a value of a given Týr type is expected,

and which conditions must be satisfied for that to be possible. This is akin to a subtyping relationship, except that most cases of interest in Týr will require some type of run-time check to be performed when a type conversion is performed. For example, a pointer with bounds $[lo, hi)$ can be passed as an argument to a function expecting a pointer with different bounds $[lo', hi')$, as long as the bounds of the provided pointer entirely contain the expected bounds, i.e., $lo \leq lo' \wedge hi \geq hi'$.

There are two kinds of compatibility rules in the system. *Value–type* rules specify when a value val (of some type TY_{val}) can be used when a value of some type TY is expected. This is used, for example, when checking if a value val can be passed as an argument to a function expecting a parameter of type TY. *Type–type* rules specify when a type TY_1 can be used when another type TY_2 is expected, without reference to a concrete value of type TY_1. This is used, for example, when checking if two function signatures are compatible (say, when using a function pointer of one type when a function pointer of another type is expected): since the functions are not being *called*, there are no concrete values for the parameters, so the compatibility check must be based on the parameter types alone.

This distinction is important because some type conversions are only possible when a concrete value is available to perform a check against. For example, consider the case where a Ptr^- (nullable) pointer is being used where a Ptr^+ (non-nullable) pointer is expected. If we are not talking about a specific Ptr^- value, we must consider whether this conversion is valid for *all* possible Ptr^- values. Since there is at least one Ptr^- value which cannot be converted to Ptr^+ (namely, **null**), this conversion must be rejected.

On the other hand, if we are considering whether a specific value val of type Ptr^- can be used where a value of type Ptr^+ is expected, we do not have to consider all possible Ptr^- values, only that specific value. Therefore, this conversion *can* be allowed, as long as we check first that $val \neq$ **null**.

We write $compat_\Gamma(val, TY) = \gamma$ to mean that under environment Γ (a mapping from identifiers to their corresponding Týr types), value val is compatible with type TY, provided that the condition γ is satisfied. Likewise, we write $compat_\Gamma(TY_1, TY_2) = \gamma$ to mean that under environment Γ, type TY_1 is compatible with type TY_2, provided that condition γ is satisfied. Context should always make it clear whether we are talking about the value–type or the type–type relationship.

We write $\top$ for the trivial condition, i.e., $compat_\Gamma(x, y) = \top$ means that the relationship holds with no further checks required. Analogously, we write $\bot$ to mean a

condition that cannot be satisfied, i.e., $compat_\Gamma(x, y) = \bot$ means that the relationship *never* holds.

The compatibility relationship will be defined by cases along Section 3.4. If the type–type relationship is not explicitly defined for a given pair of types, it defaults to $\bot$, i.e., the types are not compatible. Analogously for the value–type relationship.

The conditions generated by the compatibility relationship will be emitted as run-time checks by the instrumentation algorithm. The instrumentation actions will be described in Section 3.4, and how conditions are emitted as concrete LLVM IR instructions will be described in Section 3.6.

3.3 Instrumentation overview

The next sections will describe in detail the Týr instrumentation algorithm: how it processes the input LLVM IR program, how it assigns types to each program value, and how it instruments the program with checks based on those types. We will begin with an outline of the algorithm and an explanation of some notational conventions used along this chapter.

Týr takes as inputs an LLVM IR module mod, and programmer-provided type annotations for identifiers appearing in the module. We represent these annotations as a mapping A from identifiers to their provided Týr type annotations. If identifier id is given type TY by the programmer, we say that $A(id) = TY$; if no annotation is provided for identifier id, we say that $A(id) = \varnothing$. For notational simplicity, we consider A to map both global and local identifiers to their corresponding annotations, ignoring nesting; we assume that identifiers are not reused within a module, i.e., the same identifier will not be used in two different functions. Since function-local identifiers can be renamed with no change in semantics, there is no loss of generality in assuming so; this is just a convention to simplify notation.

The algorithm makes two passes through mod. The first pass visits each global variable, function and type definition and builds a *global environment* Γ_{global}, a mapping from each global identifier to a Týr type. The second pass visits the body of each function definition, assigning a Týr type to each local identifier and instrumenting the code as it goes through each instruction. Two passes are necessary because, unlike C, LLVM IR allows a function to refer to global identifiers defined afterwards in the module, without requiring a prototype definition.

Figure 3.4: Global environment generation pass

$GlobalEnvPass(mod, A)$:
  $\Gamma_{global} \leftarrow \varnothing$
  For each global identifier gid defined in mod:
    If there is a programmer-provided annotation $A(gid)$:
      If $A(gid)$ is compatible with gid's definition in the module:
        $\Gamma_{global} \leftarrow_+ gid : A(gid)$
      Else:
        Fail
    Else:
      $\Gamma_{global} \leftarrow_+ gid$: *a default Týr type that is compatible with the definition*

Source: The author

3.3.1 Global environment construction pass

Figure 3.4 describes in rough lines the global environment generation pass. It initializes the global environment as empty, and adds a binding for each global definition. We write $\Gamma \leftarrow_+ id : TY$ as an abbreviation of $\Gamma \leftarrow \Gamma \cup \{id : TY\}$, i.e., adding a binding to an existing environment.

We have not defined here what it means for an annotation to be compatible with its definition, and how a default type is given for definitions without an annotation. Because it will be easier to explain the choices of types used for globals after we have seen each type in more detail in Section 3.4, we will postpone a more detailed account of their usage at the global level to Section 3.5.

3.3.2 Local pass

The local pass (Figure 3.5) takes a function definition and yields a new instrumented function definition. It initializes the local environment by augmenting the global environment with bindings for each function parameter. It also initializes the Δ environment of automatically managed bounds for local pointers, which will be discussed in more detail in Section 3.4.4, and a list $blks^*$ of instrumented blocks. Then it visits each block of the function, using an applicable instrumentation rule (see Section 3.3.3 below) for each instruction. Each applied rule will emit LLVM IR instructions to produce an instrumented block, and also possibly affect the local environments Γ and Δ. Each instrumented block is appended to $blks^*$, and at the end a new function definition is returned, replacing the

Figure 3.5: Local instrumentation pass

$LocalPass(\Gamma_{global}, \textbf{define}\ ty_{ret}\ gid\ (\overline{ty_i\ id_i})\ \{blks\})$:
    Destructure $\Gamma_{global}(gid)$ as $Fn^{\pm}\ TY_{ret}\ (\overline{id_i : TY_i})$
    $\Gamma \leftarrow \Gamma_{global} \cup \{\overline{id_i : TY_i}\}$
    $\Delta \leftarrow \varnothing$
    $blks^* \leftarrow []$

    For each block blk_i in $DepthFirstVisit(blks)$:
        Destructure blk_i as $label$**:** $\overline{inst_i}$
        $blk^* \leftarrow [label\textbf{:}]$
        For each $inst_i$:
            If there is an instrumentation rule which applies to $inst_i$:
                Apply rule to $inst_i$, appending emitted code to blk^*
            Else:
                Fail
        $blks^* \leftarrow_+ blk^*$

    Return **define** $ty_{ret}\ gid\ (\overline{ty_i\ id_i})\ \{blks^*\}$

Source: The author

original blocks with the instrumented ones.

Because the local environment is built as the algorithm traverses the instructions of the function, the order in which instructions are visited is important: the definition of an identifier must be visited before uses of that identifier. Because the input code is in SSA-form, definitions must dominate their uses, which means that every possible path of control flow which reaches a use of an identifier will reach its definition first. This means that *any* traversal order which is consistent with the possible control flow paths through the function will reach definitions before their uses. Therefore, a depth-first visit of the blocks, beginning with the first function block and following the destination of branch instructions, marking visited blocks to avoid revisiting them, will ensure that identifiers are available in the environment when we reach instructions which use them.

3.3.3 Instrumentation rules

In the following sections we will define rules for instrumenting the various kinds of instructions. Each rule will have a form similar to the following example:

Instruction: $id =$ **load** ptr, when $type_\Gamma(ptr) = Ptr^-(TY, lo, hi)$

Action:

Emit check $ptr \neq null$

Emit check $lo \leq 0 \wedge hi \geq 1$

Emit the instruction

$\Gamma \leftarrow_+ id : TY$

The "Instruction" part specifies the form of the instructions matched by the rule (in this case, a **load** instruction), optionally followed by "when" and some specific conditions which must be satisfied for the rule to match (in this case, that the loaded pointer be of a specific type, using the notation that will be presented in Section 3.4.1). The "Action" part specifies what code is to be emitted when the rule is applied, and how the environment is to be updated.

"Emit the instruction" means to emit the instruction being matched itself. Typically, some instrumentation code will be emitted *before* emitting the instruction being processed, to ensure some conditions are met at run-time before execution reaches the instruction. "Emit check γ" means to emit code which checks that condition γ is satisfied at run-time, and aborts execution if it is not. Likewise, in some cases it will be necessary to emit code computing some expression δ. In these cases, we write "Emit $val \leftarrow \delta$", meaning that code to compute the value of δ is to be emitted, and whatever value is computed will be refered to as val in the subsequent action lines. The procedures for generating these run-time checks and computations as actual LLVM IR instructions will be presented in Section 3.6.

3.4 The Týr typing and instrumentation rules

This section presents the rules used by the Týr instrumentation algorithm to process each kind of value and instruction. Section 3.4.1 presents the rules for typing simple LLVM IR values. Sections 3.4.2, 3.4.3 and 3.4.4 presents the rules for common pointers, string pointers, and local pointers, the instrumentation of instructions manipulating them, and the compatibility relationship between the pointer types. Section 3.4.5 explains the rationale for the invariance of the compatibility relationship of pointers with respect to the pointer's base type. Section 3.4.6 explains the rules for handling structure types and pointers to structure fields, and Section 3.4.7 deals with function pointers. Section 3.4.8

deals with the **bitcast** instruction which converts between pointer types. Section 3.4.9 deals with the other LLVM IR instructions not covered by the previous sections.

3.4.1 Typing values

Values in LLVM IR are either constants or identifiers. Constants are always have an explicit type in the program code, and identifiers have the type assigned to them by the current environment Γ. There are no complex expressions in LLVM IR. Therefore, rules for giving types to individual values are simple. We write $type_\Gamma(val) = TY$ to mean that, under environment Γ, value val has the Týr type TY:

$$type_\Gamma(id) = \Gamma(id)$$
$$type_\Gamma(gid) = \Gamma(gid)$$
$$type_\Gamma(ty\ x) = \lceil ty \rceil, \text{ where } ty\ x \text{ is a constant}$$

3.4.2 Common pointers

$Ptr^{\pm}(TY, lo, hi)$ is the type of pointers to a region of memory containing elements of type TY, indexable from lo (inclusive) to hi (exclusive). Ptr^{+} is the type of *non-nullable* pointers, pointers that are guaranteed to be different from **null**. Ptr^{-} is a common pointer which may or may not be null. In this sense, Ptr^{+} pointers are a subset of Ptr^{-} pointers.

The main operations on pointers are *loading* a value from the location it points to, *storing* a new value into that location, and *pointer arithmetic*, which computes a new pointer from an existing one.

The **load** instruction takes a pointer and loads the first element (the one at index 0) of the region pointed to. From this we can conclude that the bounds of the referenced region must contain at least the range [0, 1). Moreover, the pointer cannot be null. The result of the **load** instruction is a value of the base type of the pointer.

Instruction: $id =$ **load** ptr, when $type_\Gamma(ptr) = Ptr^-(TY, lo, hi)$

Action:

Emit check $ptr \neq null$

Emit check $lo \leq 0 \wedge hi \geq 1$

Emit the instruction

$\Gamma \leftarrow_+ id : TY$

The **store** instruction takes a value and a pointer, and stores that value in the place of the first element of the region pointed to by the pointer. Like **load**, the pointed region must contain at least the [0, 1) bound and not be null. Moreover, the value being stored must be compatible with the base type of the pointer.

Instruction: **store** val, ptr, when $type_\Gamma(ptr) = Ptr^-(TY, lo, hi)$

Action:

Emit check $ptr \neq null$

Emit check $lo \leq 0 \wedge hi \geq 1$

Emit check $compat_\Gamma(val, TY)$

Emit the instruction

The **getelementptr** instruction performs pointer arithmetic. As mentioned in Section 2.5.4, we consider two cases of pointer arithmetic. The first case is horizontal pointer arithmetic, which takes a pointer ptr and an index idx and yields a pointer to the element idx positions after the one currently pointed to by ptr. The result is a pointer of the same base type as the original. The bounded region is also the same, but since the bounds are expressed relative to the pointed location, they need to be updated after the pointer is offset.

Instruction: $id =$ **getelementptr** ptr, idx,

when $type_\Gamma(ptr) = Ptr^-(TY, lo, hi)$

Action:

Emit check $ptr \neq null$

Emit the instruction

$\Gamma \leftarrow_+ id : Ptr^+(TY, lo - idx, hi - idx)$

Note that the resulting pointer does not have to be in-bounds: it is perfectly valid, for example, to take a pointer with bounds $[0, 10)$ and add an offset of 20 to it. The

resulting pointer will have bounds $[0 - 20, 10 - 20)$, i.e., $[-20, -10)$. This does not pose a memory safety violation because such a pointer cannot be dereferenced: both **load** and **store** check that the pointer being accessed has bounds $[lo, hi)$ such that $lo \leq 0 \wedge hi \geq 1$, which is false in this case, and therefore an attempt to load from or store to the pointer will abort with a run-time error instead.

Although one might expect the system to rule out even constructing such an out-of-bounds pointers, we have seen in Section 2.4 that the C language standard allows constructing, but not dereferencing, a pointer to the element immediately after a region of memory. Therefore, we have to allow this case of out-of-bounds pointer anyway, and we have to check for this case before loading and storing from a pointer. Since we have to check bounds when dereferencing the pointer anyway, checking them when constructing the pointer would be redundant, and amount to an unnecessary instrumentation overhead.

We do require the source pointer to be non-null, though. As will be seen below, a null pointer can be given *any* bounds; since the system disallows dereferencing them, this does not pose a memory safety violation. This is convenient because a null pointer can be used whenever a Ptr^- pointer with any bounds is expected, which is consistent with the idea of a nullable pointer. However, adding an offset to a null pointer could produce a non-null pointer within the declared bounds, which could lead to a memory safety violation. Therefore, pointer arithmetic is not allowed on null pointers. A side effect of this is that the *result* of a pointer arithmetic operation can be given a non-null Ptr^+ type, since we have already checked for non-nullness when constructing the pointer. This significantly reduces the number of nullness checks that appear in the instrumented program, and thus the performance impact of the instrumentation, as will be seen in Section 3.8.

The second case of pointer arithmetic is downwards pointer arithmetic, which takes a pointer to an aggregate data type, such as an array or structure, and yields a pointer to a sub-element within the aggregate data type. Unlike the first case, we do require the source pointer to be within bounds: since we are producing a pointer to a sub-element of the first element of the pointed region, we require that first element to exist. When the aggregate data type is an array, the result is a pointer to the base type of the array, whose bounds are those of the array, offset by the sub-element index. Structures will be dealt with in Section 3.4.6.

Instruction: $id =$ **getelementptr** $ptr, \mathrm{i}sz\ 0, idx$,

when $type_\Gamma(ptr) = Ptr^-(Array(n, TY), lo, hi)$

Action:

Emit check $ptr \neq null$

Emit check $lo \leq 0 \wedge hi \geq 1$

Emit the instruction

$\Gamma \leftarrow_+ id : Ptr^+(TY, 0 - idx, n - idx)$

The rules for **load**, **store** and **getelementptr** for Ptr^+ pointers are exactly the same as those for Ptr^- pointers, except that the check $ptr \neq null$ is omitted, since the pointer is guaranteed to be non-null.

Type compatibility rules. In general, for a pointer type $Ptr^\pm(TY, lo, hi)$ to be able to be used where another pointer type $Ptr^\pm(TY, lo', hi')$, the bounds of the first type must be wider than the second type, i.e., it must be the case that $lo \leq lo' \wedge hi \geq hi'$. Put another way, it is always safe to use a pointer to a given region of memory as if it pointed to a subportion of that region.

There are a few more details concerning null pointers. First, a null pointer can be given *any* bounds; since null pointers cannot be dereferenced, as seen above, no memory safety violation arises from allowing arbitrary bounds for null pointers. This allows null pointers with any given bounds to be used interchangeably wherever a null pointer is expected. Second, a Ptr^+ pointer must be guaranteed not to be null. When using a Ptr^- pointer where a Ptr^+ one is expected, a check must be emitted to ensure that the pointer is not null.

We can write these constraints down as the following three rules:

$compat_\Gamma(val, Ptr^-(TY, lo', hi'))$, when $type_\Gamma(val) = Ptr^-(TY, lo, hi)$

$= (val = null) \vee (lo \leq lo' \wedge hi \geq hi')$

$compat_\Gamma(val, Ptr^+(TY, lo', hi'))$, when $type_\Gamma(val) = Ptr^-(TY, lo, hi)$

$= (val \neq null) \wedge (lo \leq lo' \wedge hi \geq hi')$

$compat_\Gamma(val, Ptr^\pm(TY, lo', hi'))$, when $type_\Gamma(val) = Ptr^+(TY, lo, hi)$

$= (lo \leq lo' \wedge hi \geq hi')$

The first rule says that when converting a pointer value val from a Ptr^- type to another, either the pointer must be null, in which case any bounds are acceptable, or the

destination bounds must be narrower than the source ones. The second rule says that when converting a Ptr^- to a Ptr^+ one, the pointer must *not* be null, *and* the bounds must be compatible. Finally, the third one says that when converting a Ptr^+ to either a Ptr^- or Ptr^+ pointer, we already know that the pointer is not null, so we only have to check the bounds.

The type–type relationship is a more conservative version of the above. Since we do not have an actual value to check for non-nullness, type–type compatibility rules for Ptr^- pointers must work under the assumption that the pointer may or may not be null. Converting from a Ptr^- type to another is only allowed if the bounds match (because the source pointer might be not null), and converting from a Ptr^- to a Ptr^+ type is ruled out altogether (because the source pointer might be null).

$$\begin{aligned} &compat_\Gamma(Ptr^+(TY, lo, hi), Ptr^\pm(TY, lo', hi')) \\ &\qquad = (lo \leq lo' \wedge hi \geq hi') \end{aligned}$$

$$\begin{aligned} &compat_\Gamma(Ptr^-(TY, lo, hi), Ptr^-(TY, lo', hi')) \\ &\qquad = (lo \leq lo' \wedge hi \geq hi') \end{aligned}$$

3.4.3 String pointers and arrays

$SPtr^\pm(TY, lo, hi)$ is the type of *string pointers*, pointers to null-delimited regions of memory like C strings. The region of memory accessible through a $SPtr^\pm(TY, lo, hi)$ pointer is divided in two parts:

- A freely readable and modifiable part, delimited by the bounds $[lo, hi)$;
- A *null-terminated* region, starting at position hi and extending until the first occurrence of an element whose bits are all zero (which might be right at the hi position). The non-zero elements in this region can be read and modified; the null terminator itself can be read but not overwritten.

In other words, an $SPtr^\pm(TY, lo, hi)$ pointer has access to the same region as the corresponding $Ptr^\pm(TY, lo, hi)$ pointer, *plus* an arbitrarily long region after the declared upper bound, until a null terminator is hit. Because the system forbids overwriting the null terminator, such a pointer can be given to functions like **strlen**, which take a pointer to a region with no explicit bounds, and iterate over the region until a null terminator is hit. If

the system were not able to enforce the invariant that there is a null terminator at the end of the region, we would not be able to use functions like **strlen** without compromising spatial memory safety.

It may seem strange at first for the pointer to have both explicit $[lo, hi)$ bounds and a null-delimited part. To understand the reasoning behind this, consider the use case of a pointer s to a 100-byte buffer meant to store strings. If we gave s the type $Ptr^{+}(\mathrm{i8}, 0, 100)$, we would not be able to pass it to string-processing functions like **strlen**, because we would have no guarantee that the region is null-terminated. On the other hand, if we gave it a string pointer type with no unrestricted $[lo, hi)$ part, using only the null terminator to tell where the region ends, then whenever we wrote a zero byte to the buffer we would lose access to the region beyond that byte. This would preclude initializing the buffer with zeros, for example, or storing strings of different lengths during the buffer's lifetime, since whenever we wrote a null-terminated string to it we would lose access to the region after the string. By having both an explicitly-bounded region which we can read and write without constraints, and a part where null-termination is enforced, we can give s the type $SPtr^{+}(\mathrm{i8}, 0, 99)$, meaning that the first 99 bytes of the buffer can be used in any way, but beyond that region, null termination is enforced, meaning that the null terminator cannot be overwritten, thus ensuring we can pass s to functions like **strlen**.

String pointers enforce one more invariant: when performing pointer arithmetic with an $SPtr$ pointer, the resulting pointer must not step over the null terminator. Unlike Ptr pointers, which can be safely incremented past their valid region because the bounds can be checked when the pointer is dereferenced, finding the limits of the region referenced by an $SPtr$ pointer relies on the ability of looking ahead for the null terminator, therefore we must ensure that the null terminator is always ahead (or at), not behind, the current position. Since we already must take up the overhead of bounds checking for the upper bound when performing pointer arithmetic, we also check that the pointer is not decremented past its lower bound either. The flip side of these restrictions is that when *accessing* the pointer, we don't have to check bounds, since all $SPtr$ pointers are in-bounds by construction.

> Instruction: $id =$ **load** ptr, when $type_{\Gamma}(ptr) = SPtr^{-}(TY, lo, hi)$
>
> Action:
>
> Emit check $ptr \neq null$
>
> Emit the instruction
>
> $\Gamma \leftarrow_{+} id : TY$

When storing to the pointer, we must ensure that either we are in the freely-modifiable $[lo, hi)$ region, or we are not overwriting the null terminator. That means that either the value we are overwriting is not the null terminator, or the new value we are writing is itself a null terminator. As with normal stores, we also have to check that the type of the value being stored is compatible with the base type of the pointer.

> Instruction: **store** val, ptr, when $type_\Gamma(ptr) = SPtr^-(TY, lo, hi)$
> Action:
>
> Emit check $ptr \neq null$
>
> Emit check $compat_\Gamma(val, TY)$
>
> Emit check $(lo \leq 0 \wedge hi \geq 1) \vee (*ptr \neq null) \vee (val = null)$
>
> Emit the instruction

As with Ptr pointers (Section 3.4.2), horizontal pointer arithmetic of $SPtr$ pointers must ensure that the pointer is within bounds. This means either that the resulting pointer is within the $[lo, hi)$ region, or it is at or beyond hi but not beyond the null terminator. In other words, if the resulting pointer would point to index idx and idx is above hi, then the region $[hi, idx)$ must consist entirely of non-zero elements. Note that the element at position idx itself may be the null terminator. We write $nonZero(ptr, start, end)$ to represent the condition that the elements in range $[start, end)$ from the region pointed to by ptr are non-zero.

> Instruction: $id =$ **getelementptr** ptr, idx,
>
> when $type_\Gamma(ptr) = SPtr^-(TY, lo, hi)$
>
> Action:
>
> Emit check $ptr \neq null$
>
> Emit check $(idx \geq lo) \wedge (idx \leq hi \vee nonZero(ptr, hi, idx))$
>
> Emit the instruction
>
> $\Gamma \leftarrow_+ id : SPtr^+(TY, lo - idx, hi - idx)$

Downwards pointer arithmetic into an aggregate value is possible provided that the pointer does not point to the null terminator itself. If we allowed pointing to sub-parts of the null terminator, we would be able to corrupt the null terminator by writing to the sub-parts. (Alternatively, we could make the resulting pointer an $SPtr$ pointer with an empty freely writable part; this would preserve the null terminator but the resulting pointer would not be particularly useful.)

Instruction: $id =$ **getelementptr** $ptr, \mathrm{i}sz\ 0, idx$,

when $type_\Gamma(ptr) = SPtr^-(Array(n, TY), lo, hi)$

Action:

Emit check $ptr \neq null$

Emit check $*ptr \neq null$

Emit the instruction

$\Gamma \leftarrow_+ id : Ptr^+(TY, 0 - idx, n - idx)$

The rules for $SPtr^+$ pointers are the same as those for $SPtr^-$ pointers, with the $ptr \neq null$ checks omitted.

Type compatibility rules. Like common pointers, string pointers can safely have their bounds narrowed. Unlike common pointers, string pointers can also have their upper bound *widened*, as long as the new bounds do not reach the null terminator.

$compat_\Gamma(val, SPtr^\pm(TY, lo', hi'))$, where $type_\Gamma(val) = SPtr^+(TY, lo, hi)$

$= (lo \leq lo') \wedge (hi \geq hi' \vee nonZero(val, hi, hi'))$

Like Ptr pointers, any bound is valid for null $SPtr$ pointers, and converting from $SPtr^-$ to $SPtr^+$ requires a nullness check.

$compat_\Gamma(val, SPtr^-(TY, lo', hi'))$, where $type_\Gamma(val) = SPtr^-(TY, lo, hi)$

$= (val = null) \vee ((lo \leq lo') \wedge (hi \geq hi' \vee nonZero(val, hi, hi')))$

$compat_\Gamma(val, SPtr^+(TY, lo', hi'))$, where $type_\Gamma(val) = SPtr^-(TY, lo, hi)$

$= (val \neq null) \wedge ((lo \leq lo') \wedge (hi \geq hi' \vee nonZero(val, hi, hi')))$

It is possible to convert an $SPtr$ pointer into a corresponding Ptr pointer; this amounts to dropping access to the null-terminated region after the declared higher bound, and so it is a form of bound narrowing. Converting a pointer of type $SPtr^\pm(TY, lo, hi)$ to type $Ptr^\pm(TY, lo', hi')$ is effectively equivalent to first converting it to an $SPtr$ pointer with the new bounds, and then drop the null-terminated part (which requires no particular check).

$compat_\Gamma(val, Ptr^{\pm'}(TY, lo', hi'))$, where $type_\Gamma(val) = SPtr^\pm(TY, lo, hi)$

$= compat_\Gamma(val, SPtr^{\pm'}(TY, lo', hi'))$

The opposite conversion, from a common pointer to a null-terminated one, is not possible. Even if we checked that the referenced region did end with a null terminator at the time of the conversion, there would be nothing to prevent *other* pointers to the same region from overwriting the null terminator in the future, so this conversion cannot be allowed.

Since checking for absence of null terminators when widening the upper bounds of a string pointer requires an actual string pointer to read from, bound widening is only allowed in the value–type compatibility relationship. Type–type conversions can only narrow the declared bounds, and thus are similar to those for common pointers.

$$\begin{aligned} &compat_\Gamma(SPtr^+(TY, lo, hi), SPtr^\pm(TY, lo', hi')) \\ &\qquad = (lo \leq lo' \wedge hi \geq hi') \end{aligned}$$

$$\begin{aligned} &compat_\Gamma(SPtr^-(TY, lo, hi), SPtr^-(TY, lo', hi')) \\ &\qquad = (lo \leq lo' \wedge hi \geq hi') \end{aligned}$$

$$\begin{aligned} &compat_\Gamma(SPtr^\pm(TY, lo, hi), Ptr^{\pm'}(TY, lo', hi')) \\ &\qquad = compat_\Gamma(SPtr^\pm(TY, lo, hi), SPtr^{\pm'}(TY, lo', hi')) \end{aligned}$$

String arrays. $SArray(n, TY)$ is the type of string arrays, arrays of size n whose last element is guaranteed to be a null terminator. The semantics is slightly different from that of $SPtr$ pointers: whereas the null terminator is at an unspecified position *beyond* the declared bounds of an $SPtr$ pointer, it is contained *within* the declared size of a string array, exactly at the last position. The reason for this is that while the bounds of a pointer are only relevant for the instrumentation, and not part of the underlying LLVM IR pointer type, the size of an array *is* part of the underlying LLVM IR type, and affects the size in bytes of the type. The size of an array type should always reflect the actual type of the underlying array.

In terms of usage, the difference between a common array and a string array is that when an element of a string array is indexed via **getelementptr**, the resulting pointer is an $SPtr$ pointer. The bounds of the pointer exclude the last element, since the null terminator must be outside the explicit bounds of an $SPtr$ pointer. We present below the rule for Ptr^- pointers to string arrays. The rules for Ptr^+ and $SPtr^\pm$ pointers to string arrays are analogous, adapting the nullness and bounds checks accordingly.

Instruction: $id =$ **getelementptr** $ptr, \mathrm{i}sz\ 0, idx,$

when $type_\Gamma(ptr) = Ptr^-(SArray(n, TY), lo, hi)$

Action:

Emit check $ptr \neq null$

Emit check $lo \leq 0 \wedge hi \geq 1$

Emit the instruction

$\Gamma \leftarrow_+ id : SPtr^+(TY, 0 - idx, n - 1 - idx)$

3.4.4 Local pointers

$Local(TY)$ values are pointers to exactly one stack-allocated element of type TY. Local pointers can only ever arise as the result of an **alloca** instruction, which allocates space in the stack and returns a pointer to the allocated memory.

Bound expressions appearing within a $Local$ type can depend on other $Local$ variables. For example, if len is a $Local(\mathrm{i32})$ value (i.e., a pointer to an integer allocated on the stack), then we can have another value p with type $Local(Ptr^-(\mathrm{i8}, 0, *len))$, i.e., a pointer to a stack-allocated pointer whose lower bound is 0 and whose upper bound is whatever value is stored in the place pointed to by len.[1]

Local pointers can be loaded from and stored to (i.e., the value the pointer points to can be read and overwritten), but the pointer itself cannot be copied, passed around, or used in pointer arithmetic. This ensures that the pointer is the *only* reference to the value it points to, i.e., the pointer is never aliased. This means we can always track modifications to that place, because modifications always happen through that single pointer.

This property of local pointers makes it possible use them as bounds of other (local) pointers in a safe way. Suppose we were not able to track modifications to a value like $*len$. Then we would not be able to safely make a pointer p with bounds like $Local(Ptr^-(\mathrm{i8}, 0, *len))$, because $*len$ could be changed after the pointer was constructed, potentially expanding p's bounds beyond the valid region, which would allow spatial memory safety violations. Because the usage restrictions of local pointers ensure we *can* keep track of all modifications to the value of $*len$, the system can then instrument such modifications in such a way that only modifications which do not break the safety of p's bounds are allowed.

[1] Remember that the equivalents of C local variables in LLVM IR are accessed through pointers to the stack, so a local pointer variable in C will be accessed through a pointer to the pointer stored in the stack.

It should be noted that local variables are not *required* to have a $Local$ type, only variables involved in type dependencies are. Pointers to local variables not involved in type dependencies can be given a more general pointer type, which don't have the usage restrictions of $Local$ pointers, but cannot be dereferenced in type expressions.

The specific pointer type assigned to the result of an **alloca** instruction depends on whether the programmer has provided an annotation for the pointer, the allocation quantity, and the allocation base type. If an annotation is given, it must provide a $Local$, Ptr or $SPtr$ type consistent with the LLVM IR type and quantity used by the **alloca** instruction; otherwise, the annotation is rejected.

A $Local$ type annotation can only be given to allocations with a quantity of 1, and the base type must be compatible with the initial (zero[2]) value of the object. This means, for instance, that the base type of a $Local$ pointer cannot be a Ptr^+ type, since the stored value is initially null.

> Instruction: $id =$ **alloca** $ty, \mathrm{i}sz\ 1$, when $A(id) = Local(TY)$
> Action:
>
> Emit check $compat_\Gamma(ty$ **zeroinitializer**$, TY)$
> Emit the instruction
> Emit $zeroFill(id, 0, 1)$
> $\Gamma \leftarrow_+ id : Local(TY)$

Here, $zeroFill(id, lo, hi)$ means to emit code to store zeros in the positions in range $[lo, hi)$ relative to the pointer id.

If a Ptr type annotation is provided by the programmer, then the bounds must be compatible with the allocation quantity, and the base type must be similarly compatible with the initial value of the region.

> Instruction: $id =$ **alloca** ty, qtd, when $A(id) = Ptr^{\pm}(TY, lo, hi)$
> Action:
>
> Emit check $lo \geq 0 \wedge hi \leq qtd$
> Emit check $compat_\Gamma(ty$ **zeroinitializer**$, TY)$
> Emit the instruction
> Emit $zeroFill(id, lo, hi)$
> $\Gamma \leftarrow_+ id : Ptr^{\pm}(TY, lo, hi)$

[2]We assume here that the memory allocated by **alloca** is zero-initialized. This is enforced by emitting code to zero-fill the region immediately after the **alloca** instruction.

If an $SPtr$ type is provided, then the declared upper bound must be strictly less the quantity of the allocation. This ensures that the last element of the allocated region (which is initialized with zeros) will remain outside the declared bounds, and therefore that a null terminator will be present past the declared upper bound, as required by the rules of $SPtr$ pointers (as seen in Section 3.4.3).

Instruction: $id =$ **alloca** ty, qtd, when $A(id) = SPtr^{\pm}(TY, lo, hi)$
Action:
Emit check $lo \geq 0 \wedge hi < qtd$
Emit check $compat_{\Gamma}(ty$ **zeroinitializer**$, TY)$
Emit the instruction
Emit $zeroFill(id, lo, hi)$
$\Gamma \leftarrow_{+} id : SPtr^{\pm}(TY, lo, hi)$

When no annotation is provided, in general the allocation is given a Ptr type with appropriate bounds based on the allocation quantity, and the base type is inferred from the LLVM IR one given in the instruction.

Instruction: $id =$ **alloca** ty, qtd,
when $A(id) = \varnothing \wedge (qtd \neq 1 \vee ty$ *is not a pointer type*$)$
Action:
Emit the instruction
Emit $zeroFill(id, 0, qtd)$
$\Gamma \leftarrow_{+} id : Ptr^{+}(\lceil ty \rceil, 0, qtd)$

The exception is allocations with a quantity of 1 and a ponter base type, i.e., those corresponding to local pointer variables in C. To reduce the annotation burden on the programmer, and to allow the bounds of a pointer to vary during execution (for example, when incrementing the pointer in C), we implicitly allocate two integer local variables to hold the bounds of the pointer, which will be automatically updated whenever the pointer is modified. We initialize all variables with zeros, set the bounds of the created pointer to be the newly created local integers, and add a mapping in the automatic bounds environment Δ to record that this pointer has automatically managed bounds.

Instruction: $id =$ **alloca** $ty*, qtd$, when $A(id) = \varnothing \wedge qtd = 1$

Action:

Emit $lo =$ **alloca** iWORD, iWORD 1, for fresh identifier lo

Emit $hi =$ **alloca** iWORD, iWORD 1, for fresh identifier hi

Emit the instruction

Emit $zeroFill(id, 0, 1), zeroFill(lo, 0, 1), zeroFill(hi, 0, 1)$

$\Gamma \leftarrow_+ id : Local(Ptr^-(\lceil ty \rceil, *lo, *hi))$

$\Gamma \leftarrow_+ lo : Local(\text{iWORD}), hi : Local(\text{iWORD})$

$\Delta \leftarrow_+ id : (lo, hi)$

Loading from a $Local$ pointer requires no checks: since the $Local$ type is only given to pointers returned by **alloca** with a quantity of 1, we know that the bounds of the referenced region are $[0, 1)$, and therefore safe to load. However, if the base type contains pointer dereferences (e.g., $Local(Ptr(\text{i8}, 0, *len))$), these must be replaced with the actual values of the pointers at the time of the load. The intuition behind this is that after you load the content of a local variable x into a register id, id is fixed, i.e., a snapshot of the value of x at that moment. Therefore, the bounds of id must also be a snapshot of the values of the bounds at the time of the load, and it makes no sense for further modifications to the values of the bounds of x to affect the snapshot stored in id. To achieve this, each pointer dereferenced in the base type is loaded into a register, and the dereference is replaced by the register in the resulting type.

Instruction: $id =$ **load** ptr, when $type_\Gamma(ptr) = Local(TY)$

Action:

Emit $id_i \leftarrow *x_i$ for each $*x_i$ appearing in TY and fresh id_i

Emit the instruction

$\Gamma \leftarrow_+ id : TY[*x_i \mapsto id_i]_{\text{for each } i}$

Storing into a $Local$ pointer is trickier. First, we need to check that the type of the value being stored is the same as the base type of the local variable. But since the type of the variable may depend on the variable itself, the type may change upon the assignment. Therefore, this compatibility check must happen against the new type, i.e., replacing all dereferences of the variable with the would-be new value. Second, because the types of other $Local$ pointers (as well as the type of the pointer itself) may depend

on the variable being modified, their types may change. We need to check that the new value that they would acquire if the assignment happens is compatible with their current type. For instance, if p is a $Local(Ptr(\mathrm{i8}, 0, *len))$, changing the value stored at len will change the bounds of p. If the new bounds are compatible with the old ones (for example, if we are decrementing len, thus narrowing the bounds of p), then the assignment to len is okay. If not (for example, if we are incrementing len, thus widening the bounds of p, thus enabling access to a memory region which should not be accessible), then the assignment to len must not be allowed.

> Instruction: **store** val, ptr, when $type_\Gamma(ptr) = Local(TY) \wedge \Delta(ptr) = \varnothing$
> Action:
> Emit check $compat_\Gamma(val, TY[*ptr \mapsto val])$
> For each $(id : TY_{id})$ in Γ where $*ptr$ appears in TY_{id}
> Emit check $compat_\Gamma(id, TY_{id}[*ptr \mapsto val])$
> Emit the instruction

For $Local$ pointers whose base type is a pointer with automatic bounds, we must also update the bounds based on those of the new pointer being stored onto it.

> Instruction: **store** val, ptr,
> when $type_\Gamma(ptr) = Local(Ptr^-(TY_{ptr}, *lo, *hi)) \wedge \Delta(ptr) = (lo, hi)$
> Action:
> Let $(TY_{val}, lo_{val}, hi_{val}) = DestructurePointerType(type_\Gamma(val))$
> Emit check $compat_\Gamma(TY_{val}, TY_{ptr})$
> Emit **store** lo_{val}, lo, **store** hi_{val}, hi
> For each $(id : TY_{id})$ in Γ where $*ptr$ appears in TY_{id}
> Emit check $compat_\Gamma(id, TY_{id}[*ptr \mapsto val])$
> Emit the instruction
> Where:
> $DestructurePointerType(Ptr^\pm(TY, lo, hi)) = (TY, lo, hi)$
> $DestructurePointerType(SPtr^\pm(TY, lo, hi)) = (TY, lo, hi)$

In principle, we would not want to allow pointer arithmetic on $Local$ pointer: $Local$ pointers only ever point to a single object, so pointer arithmetic is not particularly useful on them. Moreover, if we allowed pointer arithmetic with a zero offset, the result would be a new pointer to the same element as the existing $Local$ pointer, which would

violate the non-aliasing invariant of $Local$ pointers. However, as we will see below, $Local$ pointers can be converted into general pointers with empty bounds, so it is possible to perform pointer arithmetic on $Local$ pointers, so long as the result is empty-bounded.

> Instruction: $id =$ **getelementptr** ptr, idx, when $type_\Gamma(ptr) = Local(TY)$
> Action:
>
> Emit the instruction
> $\Gamma \leftarrow_+ id : Ptr^+(TY, 0, 0)$

Type compatibility rules. In general, $Local$ pointers are not compatible with anything, not even with themselves: since they cannot be passed around or copied, it does not make sense to speak of type compatibility for local pointers. There is one exception to this rule, to accomodate a common use case in LLVM IR: adding a call to a "dummy" LLVM function passing a pointer to a local variable as a means of associating metadata with it. For example, Clang inserts calls to the functions **@llvm.lifetime.start** and **@llvm.lifetime.end** to associate liveness information with variables. These are essentially dummy functions, present in the code only as a way to represent information which further LLVM passes may use, and not present in the resulting binary. Because these functions do not read or write to the pointer being passed, they do not pose a violation of memory safety. To account for this case, Týr allows converting $Local$ pointers into common Ptr pointers, as long as the resulting bounds are empty. This allows the address of the local variable to be passed as an argument to those calls, but disallows using the *content* of the variable, thus preserving safety.

> $compat_\Gamma(Local(TY), Ptr^\pm(TY, lo, hi)) = (lo = hi)$

3.4.5 Invariance of pointer types

Although Ptr pointers have a compatibility relationship based on the pointer bounds (i.e., two pointers with different bounds can be compatible, subject to run-time checks on the bounds), there is no such relationship based on the base type (i.e., two pointers with different base types are never compatible). In the terminology of subtyping, Ptr is *invariant* with respect to the base type. That is, if S and T are two different types, then $Ptr^\pm(S, lo, hi)$ and $Ptr^\pm(T, lo, hi)$ are never compatible with (or subtypes of) each

other, even if S is compatible with T. The reason is that the region the pointer points to is both readable and writable. Consider the following example. As we have seen before, $Ptr^{+}(\text{i8}, 0, 10)$ is compatible with $Ptr^{+}(\text{i8}, 0, 5)$, because a value of the wider type can be used wherever a value of the narrower one is expected (in other words, every pointer with bounds $[0, 10)$ also has the $[0, 5)$ region). Now, consider a pointer *to* a region containing a pointer of that type, $Ptr^{+}(Ptr^{+}(\text{i8}, 0, 10), 0, 1)$. If we considered it a compatible with $Ptr^{+}(Ptr^{+}(\text{i8}, 0, 5), 0, 1)$, then it would allow code to store a $Ptr^{+}(\text{i8}, 0, 5)$ inside the region. Afterwards, the code which had the $Ptr^{+}(Ptr(\text{i8}, 0, 10), 0, 1)$ pointer to that region would read the pointer in the region back as $Ptr^{+}(\text{i8}, 0, 10)$, thus giving it wider bounds than it actually has, which would allow it to access invalid memory through it. Therefore, the compatibility/subtyping relationship cannot be covariant with respect to the base type.

On the other hand, if we allowed the relationship to be contravariant, i.e., if $compat_{\Gamma}(S, T)$ then $compat_{\Gamma}(Ptr^{+}(T, lo, hi), Ptr^{+}(S, lo, hi))$, then it would allow casting a pointer $Ptr^{+}(Ptr^{+}(\text{i8}, 0, 10), 0, 1)$ to $Ptr^{+}(Ptr^{+}(\text{i8}, 0, 20), 0, 1)$, which is clearly bad because then the pointer contained in the region could have its bounds arbitrarily expanded when read back. Therefore, the compatibility relationship must be invariant with respect to the base type. The same applies to $SPtr$ pointers.

3.4.6 Structures

The handling of structures presents some conceptual similarities with the handling of local variables, in that structures define a scope within which the types of the components are allowed to depend on the values of other components (or even the same component) of the same structure.

Structure fields are always accessed through pointers obtained via the **getelementptr** instruction. Given a pointer p to a structure of type $Struct\ id\ (\overline{id_i : TY_i})$, **getelementptr** $p, 0, i$ yields a pointer to the ith field of the structure (counting from zero). Unlike array indices, LLVM IR requires the structure field index i to be a constant. If the field is not involved in type dependencies (i.e., if its type does not depend on any field and the type of no other field depends on it), then the pointer can be given a common pointer type with no usage restrictions other than those imposed by bounds.

Instruction: $id =$ **getelementptr** $ptr, \mathrm{i}sz\ 0, idx$,

when $type_\Gamma(ptr) = Ptr^-(S, lo, hi)$

$\wedge\ S = Struct\ id_S\ (fld_0 : TY_0, ..., fld_{n-1} : TY_{n-1})$

$\wedge\ TY_{idx}$ *does not contain references to any* $fld_{0...n-1}$

$\wedge$ *No* $TY_{0...n-1}$ *contains references to* fld_{idx}

Action:

Emit check $ptr \neq null$

Emit check $lo \leq 0 \wedge hi \geq 1$

Emit the instruction

$\Gamma \leftarrow_+ id : Ptr^+(TY_{idx}, 0, 1)$

If the field is involved in type dependencies (i.e., its type depends on other fields or the types of other fields depend on it), the resulting pointer is given a $Field$ type. This is analogous to a $Local$ pointer, in that it has similar usage restrictions which allow the system to track modifications to the fields of a structure. Rather than carrying the base type of the referenced value, the field pointer type has the form $Field(S, val, fld)$, where S is the type of the structure the field is part of, val is the structure pointer from which the field pointer originated, and fld is the name of the field within that structure. With all this information, we can both recover the base of the pointer (by consulting the type of field fld in S), and fetch the values of the other fields in the structure to deal with dependencies (by emitting **getelementptr** instructions for the other fields of val).

Instruction: $id =$ **getelementptr** $ptr, \mathrm{i}sz\ 0, idx$,

when $type_\Gamma(ptr) = Ptr^-(S, lo, hi)$

$\wedge\ S = Struct\ id_S\ (fld_0 : TY_0, ..., fld_{n-1} : TY_{n-1})$

$\wedge\ (TY_{idx}$ *contains references to any* $fld_{0...n-1} \vee$

Some $TY_{0...n-1}$ *contains references to* $fld_{idx})$

Action:

Emit check $ptr \neq null$

Emit check $lo \leq 0 \wedge hi \geq 1$

Emit the instruction

$\Gamma \leftarrow_+ id : Field(S, ptr, fld_{idx})$

Loading from a $Field$ pointer does not require any checks, since a $Field$ pointer can only be derived from a valid structure. Much like loading from a $Local$ pointer takes a

snapshot of the current value of the local variable, and thus any references to other locals in the type of the loaded value must be replaced by their actual values at the point of the **load**, so does loading from a $Field$ pointer require taking a snapshot of the values of the fields it depends on in the resulting type. In the "Emit" statements, we will write $ptr_S.fld$ to mean fetching the value of field fld from a structure of type S pointed to by ptr.

Instruction: $id =$ **load** ptr,
when $type_\Gamma(ptr) = Field(S, ptr, fld)$
$\wedge\ S = Struct\ id_S\ (fld_0 : TY_0, \ldots, fld_{n-1} : TY_{n-1})$
Action:
Emit check $ptr \neq null$
Emit $id_i \leftarrow ptr_S.fld_i$ for each fld_i appearing in TY_{fld} and fresh id_i
Emit the instruction
$\Gamma \leftarrow_+ id : TY_{fld}[fld_i \mapsto id_i]_{\text{for each } i}$

Again like $Local$ pointers, storing into a $Field$ requires checking that the value being stored is compatible with the field type, replacing any references to the field itself with the would-be new value, and references to other field names fld_i with $ptr_S.fld_i$ so we can fetch their values. We must also check that other fields whose types depend on the field being modified are compatible with the new value, again performing the same substitutions on the types.

Instruction: **store** val, ptr,
when $type_\Gamma(ptr) = Field(S, ptr, fld_{idx})$
$\wedge\ S = Struct\ id_S\ (fld_0 : TY_0, \ldots, fld_{n-1} : TY_{n-1})$
Action:
Emit check $compat_\Gamma(val, subst(TY_{idx}))$
For each $fld_i : TY_i$ where fld_{idx} appears in TY_i and $fld_i \neq fld_{idx}$
Emit check $compat_\Gamma(ptr_S.fld_i, subst(TY_i))$
Emit the instruction
Where:
$subst(TY) = TY[fld_{idx} \mapsto val][fld_i \mapsto ptr_S.fld_i]_{\text{for each } i}$

Type compatibility rules. Because field pointers have similar usage restrictions as local pointers, type compatibility is also limited for them. We allow converting field

pointers to common pointers with empty bounds, for the same reasons as we do for local pointers (see Section 3.4.4).

$$compat_\Gamma(Field(S, ptr, fld), Ptr^{\pm}(TY, lo, hi)),$$
$$\text{where } S = Struct\ id_S\ (..., fld : TY, ...)$$
$$= (lo = hi)$$

3.4.7 Functions

$Fn^{\pm}\ TY_{ret}\ (id_1 : TY_1, ..., id_n : TY_n)$ is the type of pointers to functions taking arguments $id_{1...n}$ of types $TY_{1...n}$ and returning type TY_{ret}. Like Ptr and $SPtr$ pointers, Fn pointers may be nullable (Fn^-) or non-nullable (Fn^+). The types of parameters and the return type can depend on the values of the parameters.

Functions are always handled through function pointers in LLVM IR; functions themselves are not first-class values. Function pointers cannot be loaded from or stored to, nor is it meaningful to perform pointer arithmetic on them, since the pointer base type does not have a size. Therefore, **load**, **store** and **getelementptr** are not applicable to Fn pointers.

The function pointed to by an Fn pointer can be invoked with the **call** instruction. **call** is passed the function pointer and a list of arguments, one for each parameter of the function. For the call to be allowed, the types of each of the arguments must be compatible with the types declared for the corresponding parameters. Because the types of the parameters may depend on the values of the parameters, references to parameters within a parameter type must be replaced with the values being used in the call when checking compatibility, in much the same way as references to a local variable are replaced with the local's new value when storing to a $Local$ pointer, and references to fields in structures are replaced with the actual structure values when storing to a $Field$ pointer. The **call** instruction yields a value of the function's return type, also with parameters replaced with argument values.

Instruction: $id = \textbf{call}\ fn\ (val_1, ..., val_n)$,

when $type_\Gamma(fn) = Fn^-\ TY_{ret}\ (id_1 : TY_1, ..., id_n : TY_n)$

Action:

Emit check $fn \neq null$

Emit check $compat_\Gamma(val_i, TY_i[id_j \mapsto val_j]_{\text{for each } j})$ for each val_i

Emit the instruction

$\Gamma \leftarrow_+ id : TY_{ret}[id_j \mapsto val_j]_{\text{for each } j}$

The **ret** instruction is used to return a value from the current function to its caller. The type of the value being return must be compatible with the declared return type. Here we do not have to replace parameter names in the return type, because when the **ret** instruction is running, all parameters are in scope and have actual values.

Instruction: **ret** val,

within a function whose signature is $Fn^\pm\ TY_{ret}\ (...)$

Action:

Emit check $compat_\Gamma(val, TY_{ret})$

Emit the instruction

3.4.8 Bitcast

The **bitcast** instruction is used to cast a value to a different type which has the same size in bytes as the original type. For instance, it can be used to cast an **i32** value to **float**, because both are 32-bits wide, but not an **i32** to **double**, because the sizes differ. The result is a value which has the exact same bit pattern as the original, but reinterpreted as the new type.

Because pointers typically have all the same size (the size of a machine address), it is generally possible in LLVM IR to cast between any two pointer types. Therefore, it is possible, for example, to cast an **i32*** to **double***, even though **i32** and **double** don't have the same size, because **i32*** and **double*** do have the same size.

LLVM IR requires that the source and destination types are both non-pointers or both pointers; there are separate instructions for converting from integers to pointers and vice-versa. Casting between non-pointer types requires no special treatment from the perspective of memory safety, because memory is only accessed through pointer types.

Instruction: $id =$ **bitcast** val **to** ty_{new},

when $type_\Gamma(val) = TY_{old}$

$\wedge \lfloor TY_{old} \rfloor$ *is not a pointer type*

$\wedge\ sizeof(\lfloor TY_{old} \rfloor) = sizeof(ty_{new})$

Action:

Emit the instruction

$\Gamma \leftarrow_+ id : \lceil ty_{new} \rceil$

Casts between pointer types require more attention. First, because pointer bounds are expressed in terms of indices, the actual bounds they represent are relative to the size of the base type. For instance, if a pointer p has the type $Ptr^+(\text{i32}, 0, 10)$ and it points to memory address 1000, then the upper bound of the accessible region is address 1040 ($1000 + 4 \times 10$), because i32 is 4 bytes long. Therefore, if the pointer is cast to **i64***, for instance, the resulting pointer must have type $Ptr^+(\text{i64}, 0, 5)$, with bounds shrunk to account for the wider base type (i.e., because the base type is now 8 bytes long, the upper bound index must be 5 so that $1000 + 8 \times 5$ is still at or below 1040). Computing the new bounds is done by multiplying the number of elements by the size of the old base type to obtain a number of bytes, and then dividing the number of bytes by the size of the new base type, thus obtaining the number of elements in the new type.

Instruction: $id =$ **bitcast** val **to** $ty_{new}*$,

when $type_\Gamma(val) = Ptr^\pm(TY_{old}, lo, hi)$

Action:

Let $lo' = (lo * sizeof(\lfloor ty_{old} \rfloor))/sizeof(ty_{new})$

Let $hi' = (hi * sizeof(\lfloor ty_{old} \rfloor))/sizeof(ty_{new})$

Emit the instruction

$\Gamma \leftarrow_+ id : Ptr^\pm(\lceil ty_{new} \rceil, lo', hi')$

An $SPtr$ pointer cannot be cast to another $SPtr$ type with a new base type in this way, because the null terminator of the old type may not be recognized as a null terminator when interpreted in the new type. For instance, if a sequence of 8-bit characters were to be reinterpreted as a sequence of 32-bit integers, the single-byte null terminator would not be recognized as a null value anymore, but rather would be interpreted as part of an integer. This would make the system unable to find the null terminator of the region, thus allowing a memory safety violation. However, we can allow $SPtr$ pointers to be cast

using the same rule presented above if we let the resulting pointer have a Ptr rather than $SPtr$ type (effectively dropping access to the null-delimited region).

There is a second problem with casting between pointers: if the destination pointer type has a base type that is itself a pointer type or somehow contains pointers, then the cast may lead to non-pointer values being interpreted as pointers. For example, if a pointer to a buffer of bytes (e.g., **i8***) is reinterpreted as being a pointer to a structure containing pointer fields (e.g., **{ i32, i8* }***), then the pointers within the structure will be filled with the values of the bytes in the buffer, which may or may not constitute valid pointers. To ensure safety, we would have to disallow such casts. However, many C programs rely on the ability to do this. A common case is the use of the **void*** type (which is translated to **i8*** in LLVM IR) as a kind of "generic" pointer type, which is then cast to an appopriate type when using the pointer. For instance, the POSIX **pthread_create** function, which spawns a new thread, takes (among other arguments) a pointer to a function **start_routine** to be executed within the new thread, and an argument **arg** to pass to that function. To allow values of different types to be used, **start_routine** is defined as taking a **void*** argument, and **arg** has the type **void***. In typical usage, the actual function passed as **start_routine** will then cast its parameter to the actual type of the **arg** argument, and then use it. Unlike the case with pointer bounds, where typically the bounds are passed as extra arguments to the C function, in this case there is no information passed to the **start_routine** function which would tell it the actual type of the argument: the function simply assumes that an argument of the proper type was passed and casts it. This makes it more difficult to ensure safety of such casts. In this work, we opted to allow such casts, even though they are potentially unsafe, to allow this usage in C programs. There are possible solutions to this problem, such as introducing a form of true parametric polymorphism to replace the usage of **void*/i8*** as a make-shift generic type. This should be addressed by future work.

3.4.9 Other instructions

ptrtoint converts pointers to integers. This is a safe operation, because the resulting integer cannot be used to access memory. **inttoptr** does the opposite conversion, from integers to pointers. This allows arbitrary integers to be turned into pointers, which would lead to memory unsafety if the resulting pointers could be dereferenced. To avoid this problem, we allow the pointer to be created, but give it empty bounds. This allows idioms

like casting an integer to a pointer for use as a sentinel value (e.g., using **((void *) -1)** as a distinguished pointer value which is known not to point to any valid object, similarly to the way **null** is used), but disallows pointing to an arbitrary address and then reading from or writing to it.

> Instruction: $id =$ **ptrtoint** val **to** isz, when $type_\Gamma(val)$ *is a pointer type*
> Action:
>
> Emit the instruction
> $\Gamma \leftarrow_+ id : \text{i}sz$

> Instruction: $id =$ **inttoptr** val **to** $ty*$, when $type_\Gamma(val) = \text{i}sz$
> Action:
>
> Emit the instruction
> $\Gamma \leftarrow_+ id : Ptr^-(\lceil ty \rceil, 0, 0)$

The other instructions do not yield or access pointers and have trivial rules. Arithmetic instructions expect integer operands of the same type and yield an integer with the same type as the operands. The comparison instruction expects two integers or two pointers and yields a boolean. Comparing pointers is safe, because the region referenced by the pointers is not accessed. **sext**, **zext** and **trunc** convert integer operands into other integer types.

Because the Týr rules for those instructions do not add anything new over the type checks LLVM already does, we will omit them here. Likewise, the rules for the control flow instructions **br** and **unreachable** do not have any memory-safety related restriction.

3.5 Global environment, revisited

We can now revisit the procedure for constructing the global environment (Figure 3.4). The types given to global variables, constants and functions depend on the type annotations provided by the programmer. As with locals, an annotation, if provided, must be consistent with the LLVM IR type and the initializer used with the element; unlike locals, the initializer of a global variable or constant need not be **zeroinitializer**.

Because globals and constants are pointers to their storage, and the $Local$ type cannot be used for values outside functions, any annotation provided for a global must

be of the form $Ptr^{\pm}(TY, 0, 1)$, representing a pointer to the location where the global is stored. The base type TY must be compatible with the global's initializer.

Another difference between globals and locals is that, because globals occur at the module top-level rather than within a function block, we cannot emit run-time compatibility checks for globals. This means that any compatibility check required to ensure that the initializer is compatible with the given annotation base type must depend only on information available at compile-time, so we can check it without emitting any run-time check. This is not a problem in practice, since the given types cannot contain references to $Local$ variables anyway (since there are none at the global scope), and globals cannot be used in bound expressions (since they do not have the aliasing guarantees of $Local$ values).

In other words, if an annotation $A(gid)$ is given for a global variable or constant gid initialized with value $const$, then it must be the case that $A(gid) = Ptr^{\pm}(TY, 0, 1)$, and $compat_{\Gamma}(const, TY)$. Although any nullness tag can be specified for the annotation pointer, there is no reason for specifying Ptr^{-}, since the pointer to the global allocated region is known to be non-null.

If an annotation is not given for a global variable or constant, a default type $Ptr^{+}(type_{\Gamma}(const), 0, 1)$ is assigned, where $const$ is the global's initializer. As we have seen, $type_{\Gamma}$ uses the default lifting $\lceil\cdot\rceil$ from LLVM IR types to Týr types presented in Figure 3.2 to give types to constants.

For functions, the annotation must have the form $Fn^{\pm}\ TY_{ret}\ (id_1 : TY_1, \ldots, id_n : TY_n)$. The quantity and names of the parameters must match, and the Týr types must be consistent with the LLVM IR level types, i.e., if parameter id_i has LLVM IR type ty_i, then the Týr type TY_i given to it in the annotation must be such that $\lfloor TY_i \rfloor = ty_i$. The same applies to the return type. If an annotation is not given, a default type $Fn^{+}\ \lceil ty_{ret} \rceil\ (\overline{id_i : \lceil ty_i \rceil})$ is assigned to the function, where ty_{ret} and $\overline{ty_i}$ are the types appearing in the LLVM IR function declaration.

3.6 Emitting instrumentation code

In Section 3.4, we have described which checks must be emitted and which computations must be performed for each kind of instruction. In this section, we describe how those checks and computations are turned into actual LLVM IR instructions.

There are four kinds of "Emit" actions appearing in the instrumentation rules:

- "Emit $inst$", where $inst$ is one or more LLVM IR instructions, means simply to emit the given instructions in the resulting code. "Emit the instruction" is an abbreviation for "Emit $inst$" when $inst$ is the instruction being currently instrumented itself.
- "Emit $zeroFill(id, \delta_{lo}, \delta_{hi})$" means to emit instructions to zero out the region within the range $[\delta_{lo}, \delta_{hi})$ relative to the pointer id. This can be done by computing the start and length of the region, and calling the C **memset** function to fill it with zeros.[3] This is equivalent to the sequence:

  Emit $start \leftarrow id + \delta_{lo}$
  Emit $len \leftarrow (\delta_{hi} - \delta_{lo}) * sizeof(ty)$, where ty is the base type of id
  Emit $ptr =$ **bitcast** $start$ **to i8***
  Emit $v_{void} =$ **call @memset** $(start, \text{i32 } 0, len)$

- "Emit $v \leftarrow \delta$" means to emit code to compute the value of expression δ, and that the LLVM IR value representing the result of the computation will be referred to as v in the subsequent actions. The rules for emitting code for each kind of expression will be discussed in Section 3.6.1.
- "Emit check γ" means to emit code to check that condition γ is true at run-time, and abort execution if it is not. The rules for emitting code for each kind of check will be discussed in Section 3.6.2.

3.6.1 Computing expressions

Dependent type expressions are evaluated for their value rather than for side effects. Simple expressions, such as identifiers and constants, do not require emitting any instructions to compute them; they can be used directly as LLVM IR values. Composite expressions, on the other hand, require instructions to be emitted to compute their values. For example, an expression like $2 + 3$ will result in an instruction like $v =$ **add** iWORD 2, iWORD 3 being emitted, where v is a fresh identifier. The instruction itself is not the value of the expression; rather, the instruction is emitted, and the value ends up assigned to register v. Therefore, computing expressions generates as a result both a (potentially empty) sequence of instructions, and an LLVM IR constant or reg-

[3] In the actual implementation, the LLVM IR **@llvm.memset.p0i8.iWORD** intrinsic function is called instead. **memset** is used here for the sake of simplicity since it has a simpler function signature.

ister which will contain the value of the expression after the sequence of instructions is evaluated.

In this section, we will describe the rules for computing each kind of expression. We will use a rule format using "Expression" and "Action" parts, similar to the one used for instrumentation rules in Section 3.4.

The action "Yield val" will be used to mean that val is the resulting value of the expression. As we have seen in Section 3.3.3, we write "Emit $v \leftarrow \delta$" to mean that expression δ is to be computed and the result will be referred to as v in subsequent action lines. In effect, this will mean to apply one of the rules in this section to compute δ. The rule will potentially emit some instructions, add some registers to the environment, and finally "Yield" some value val. This value is what the v in "Emit $v \leftarrow \delta$" will refer to afterwards.

The rules presented here may create auxiliary registers to hold intermediate values. Although for simplicity of exposition we do not always explicitly add every intermediate register to the environment in the rules, we do always explicitly add registers appearing in the "Yield" action of a rule. This means that after an "Emit $v \leftarrow \delta$" action, it is always possible to use $type_\Gamma(v)$ to obtain the type of the computed expression.

The grammar of dependent type expressions δ has been presented in Figure 3.1 (Section 3.2.1). We shall now consider each kind of expression.

Simple expressions. For simple constants and identifiers (val), no code has to be emitted; only the equivalent LLVM IR value must be returned.

> Expression: val
> Action:
> Yield val

Likewise, $sizeof(ty)$ is a constant expression whose value can be determined at compile-time.

> Expression: $sizeof(ty)$
> Action:
> Yield iWORD n, where n is the statically-known size of LLVM IR type ty

Binary operations. For binary operations $\delta_1\ op\ \delta_2$, we must first compute each side. Afterwards, different actions are taken depending on the type of each side.

Expression: $\delta_1\ op\ \delta_2$

Action:

Emit $v_1 \leftarrow \delta_1$

Emit $v_2 \leftarrow \delta_2$

$ComputeBinop(v_1\ op\ v_2)$

We define $ComputeBinop$ by cases, depending on the types of the operands. When both operands are integers, we must first ensure that they both have the same integer size; if they do not (for example, if one is i32 and the other is i64), we cast the smaller operand to the size of the larger. Then we emit the instruction to perform the appropriate arithmetic operation.

$ComputeBinop(v_1\ op\ v_2)$,

when $type_\Gamma(v_1) = \text{i}sz_1 \wedge type_\Gamma(v_2) = \text{i}sz_2$

Action:

Let $sz_{max} = max(sz_1, sz_2)$

Emit $v_1' \leftarrow$ **sext** v_1 to sz_{max}

Emit $v_2' \leftarrow$ **sext** v_2 to sz_{max}

Emit $v = op{\rightarrow}inst[op]\ v_1', v_2'$

$\Gamma \leftarrow_+ v : \text{i}sz_{max}$

Yield v

Where $op{\rightarrow}inst$ is the mapping:

$$op{\rightarrow}inst = \{ + \mapsto \textbf{add}, - \mapsto \textbf{sub}, * \mapsto \textbf{mul}, / \mapsto \textbf{sdiv}\}$$

If the left side is a common (Ptr) pointer and the right side is an integer, we use the **getelementptr** instruction to perform pointer arithmetic. It should be noted that the bounds of the pointer are irrelevant here, since Ptr pointers appearing in dependent type expressions cannot be dereferenced. Likewise, the resulting pointer is given empty bounds $[0, 0)$.

$ComputeBinop(v_1 + v_2)$,

when $type_\Gamma(v_1) = Ptr^{\pm}(TY, lo, hi) \wedge type_\Gamma(v_2) = \mathrm{i}sz$

Action:

Emit $v =$ **getelementptr** v_1, v_2

$\Gamma \leftarrow_+ v : Ptr^+(TY, 0, 0)$

Yield v

For subtracting two pointers, we must first convert them to integers, then subtract the result, and finally divide it by the size of the base type. Again, the bounds of the pointers do not matter, but both pointers must have the same base type.

$ComputeBinop(v_1 - v_2)$,

when $type_\Gamma(v_1) = Ptr^{\pm}(TY, lo, hi) \wedge type_\Gamma(v_2) = Ptr^{\pm}(TY, lo', hi')$

Action:

Emit $v_{size} \leftarrow sizeof(\lfloor TY \rfloor)$

Emit $v_1' =$ **ptrtoint** v_1 **to** iWORD

Emit $v_2' =$ **ptrtoint** v_2 **to** iWORD

Emit $v_3 =$ **sub** v_1, v_2

Emit $v =$ **sdiv** v_3, v_{size}

$\Gamma \leftarrow_+ v :$ iWORD

Yield v

Local pointer dereference. For $Local$ pointer dereference $*\delta$, we just load from the pointer. There is no need to compute the pointer first, because $Local$ pointer expressions are always simple identifiers, never composite expressions.

Expression: $*\delta$

when $type_\Gamma(\delta) = Local(TY)$

Action:

Emit $v =$ **load** δ

$\Gamma \leftarrow_+ v : TY$

Yield v

Structure field dereference. For loading the value of a structure field, we need to first get a pointer into the desired field, then load from it.

Expression: $val_S.fld_k$

when $S = Struct\ id_S\ (..., fld_k : TY, ...)$

Action:

Emit $v_{ptr} =$ **getelementptr** $val, \text{iWORD}\ 0, \text{iWORD}\ k$

Emit $v =$ **load** v_{ptr}

$\Gamma \leftarrow_+ v : TY$

Yield v

3.6.2 Checks

Checks insert a control flow change in the program. An action "Emit check γ" means to emit code to check whether a boolean condition γ is true, continue execution normally if it is, and abort execution otherwise. The grammar of boolean conditions γ is given in Figure 3.6.

Figure 3.6: Grammar of boolean check conditions

Condition: $\gamma ::= \top \mid \bot \mid \delta_1\ cmp\ \delta_2 \mid nonZero(\delta_{ptr}, \delta_{lo}, \delta_{hi}) \mid \gamma_1 \wedge \gamma_2 \mid \gamma_1 \vee \gamma_2$

Comparison: $cmp ::= = \mid \neq \mid < \mid \leq \mid > \mid \geq$

Source: The author

At the point where a check is to be performed, a boolean condition is computed. If true, execution continues normally to the code after the check. If false, execution is diverted to a failure block, which prints some diagnostic message to the user and aborts execution. In the actual implementation, a different failure block with an appropriate diagnostic message is created for each check, but for the purposes of exposition we can consider that each function has a single block named **fail** which calls the C standard library function **abort** to finish program execution with an error.

To describe check code generation, we will use an auxiliary procedure $Check$ of three arguments: the boolean check to be performed, the label to jump to in case of

success, and the label to jump to in case of failure. Whenever an "Emit check γ" action appears, we create a new success label, and emit code to evaluate the check γ, jump to the created success label if the check is successful, and to the function's **fail** label if the check fails.

> Definition: Emit check γ
> Action:
>
> Let ℓ_{succ} be a freshly generated label
> $Check(\gamma, \ell_{succ}, \textbf{fail})$
> Emit ℓ_{succ}**:**

We can now describe the $Check$ procedure for each case in Figure 3.6. The $\top$ check is trivial: we simply emit code to unconditionally jump to the success label.

> $Check(\top, \ell_{succ}, \ell_{fail})$:
>
> Emit **br true**, ℓ_{succ}, ℓ_{succ}

Likewise, for $\bot$ we emit code to unconditionally jump to the failure label.

> $Check(\bot, \ell_{succ}, \ell_{fail})$:
>
> Emit **br true**, ℓ_{fail}, ℓ_{fail}

For comparison checks, we emit instructions to compute each side of the comparison operator, and then the appropriate instruction to compare the results. Finally, we branch to the success or failure label depending on the result of the comparison.

> $Check(\delta_1\ cmp\ \delta_2, \ell_{succ}, \ell_{fail})$:
>
> Emit $v_1 \leftarrow \delta_1$
> Emit $v_2 \leftarrow \delta_2$
> Emit $v_{bool} =$ **icmp** $cmp{\rightarrow}op[cmp]\ v_1, v_2$
> Emit **br** $v_{bool}, \ell_{succ}, \ell_{fail}$

Where $cmp{\rightarrow}inst$ is the mapping:

$$cmp{\rightarrow}inst = \{ = \mapsto \textbf{eq}, \neq \mapsto \textbf{ne}, < \mapsto \textbf{slt}, \leq \mapsto \textbf{sle}, > \mapsto \textbf{sgt}, \geq \mapsto \textbf{sge} \}$$

A $nonZero(val, \delta_{lo}, \delta_{hi})$ check verifies that, within the range of indices $[\delta_{lo}, \delta_{hi})$ relative to a pointer val, there are no elements whose bytes are all zero (i.e., there are

no **zeroinitializer** elements in the given range). To perform this check, we emit code to compute the start and length of the region, call an auxiliary function **zerofind** that sweeps the region looking for zero elements[4], and then branch to the success or failure label depending on the result. The **zerofind** function must be provided by the implementation at run-time.

> $Check(nonZero(val, \delta_{lo}, \delta_{hi}), \ell_{succ}, \ell_{fail})$:
>
> Emit $start \leftarrow ptr + \delta_{lo}$
>
> Emit $len \leftarrow \delta_{hi} - \delta_{lo}$
>
> Emit $elemsize \leftarrow sizeof(\lfloor type_\Gamma(val) \rfloor)$
>
> Emit $v_{bool} =$ **call @zerofind**$(start, len, elemsize)$
>
> Emit **br** $v_{bool}, \ell_{succ}, \ell_{fail}$

For logical conjunction, we generate an auxiliary label ℓ_{then}. We emit code for the left side of the conjunction, making it so that if it succeeds, it jumps to the ℓ_{then} label, which will point to the code for the right side of the conjunction. If the right side succeds, we jump to the success label; if either side fails, we jump to the failure label.

> $Check(\gamma_1 \wedge \gamma_2, \ell_{succ}, \ell_{fail})$:
>
> Let ℓ_{then} be a freshly generated label
>
> $Check(\gamma_1, \ell_{then}, \ell_{fail})$
>
> Emit ℓ_{then}**:**
>
> $Check(\gamma_2, \ell_{succ}, \ell_{fail})$

Logical disjunction is analogous, but we jump to ℓ_{succ} if either side succeeds, and the if the left side fails we jump to the right side.

> $Check(\gamma_1 \vee \gamma_2, \ell_{succ}, \ell_{fail})$:
>
> Let ℓ_{else} be a freshly generated label
>
> $Check(\gamma_1, \ell_{succ}, \ell_{else})$
>
> Emit ℓ_{else}**:**
>
> $Check(\gamma_2, \ell_{succ}, \ell_{fail})$

[4]This function can be thought of as similar to the C standard library **memchr** function, but working with arbitrary-sized elements rather than just bytes.

3.7 Soundness

Soundness in the context of Týr refers to the ability of the system to prevent spatial memory safety violations. There are two known sources of unsoundness in the system. The first one is the pointer bitcast rule, which, as discussed in Section 3.4.8, allows some forms of unsafe pointer casts to enable typical C usage of such pointers. As already mentioned, addressing the problem of allowing such casts in a safe way is an interesting question for future work.

The other source of unsoundness is external code not instrumented by Týr. As mentioned in the introduction, one of the advantages of the Týr approach is to keep binary compatibility, thus allowing external modules and libraries to be used unmodified with a program compiled by Týr. Naturally, Týr cannot ensure that such external code will respect the invariants represented by the Týr type system.

Týr relies on the programmer providing annotations which accurately describe the functions, variables and data types provided by external code. For example, if the programmer is going to use the C library function **memcpy**, which takes two pointers to buffers and their length in bytes and copies the contents from one buffer to the another, the programmer must supply an annotation like:

$$memcpy : Fn\ \textbf{void}\ (dest : Ptr(\text{i8}, 0, len), src : Ptr(\text{i8}, 0, len), len : \text{i64})$$

Because Týr does not have access to the definition of the **memcpy** function in the standard library, it cannot verify that the annotation is correct with respect to the bounds; rather, it must *trust* the programmer to provide an accurate annotation for the function.

In the opposite direction, the functions, variables and data types defined in a Týr module may be used by external code not compiled with Týr, which therefore is not constrained to use these elements only in the ways allowed by the Týr type system. For example, consider a function defined by the programmer with an annotation like that provided for **memcpy** above. Even though code compiled by Týr will only be able to call the function with pointers respecting the constraints imposed by that annotation, external code which has access to the function will be able to call it with arbitrary pointers, thus introducing unsafety.

Apart from these sources of unsoundness, we expect the system to be sound. More specifically, we should be able to show that if a program P not interacting with unsafe

external code can be successfully type-checked in Týr (without using the pointer bitcast rule), yielding an instrumented program P', then: (1) P' does not contain spatial memory safety violations; and (2) given the same inputs, P' either produces the same output as P or aborts with a run-time type violation.

Informally, we can argue (1) by showing that all pointers created by the program will be consistent with the bounds of the referenced region and with the nullness of the pointer, since all memory access is done through pointers. New regions are allocated in an LLVM IR program in two ways: through the **alloca** instruction, and when creating global variables and constants. The rules which give a type to the pointer returned by **alloca** always require the pointer type to be consistent with the size of the allocation and with the initial value of the region, so the consistency of those pointers with their associated region is enforced. Likewise, the rules which give type for the pointers to global variables and constants ensure the consistency of the pointer type with the initial value and the bounds of the storage for the global. Then, we must show that only consistent pointers can be derived from consistent pointers, i.e., that whenever a pointer type is used in a place expecting another pointer type, the conversion will only be allowed if the resulting type is still consistent with the referenced region. Finally, we must show that no operation can cause a spatial memory safety violation when provided consistent pointers. This is enforced by the rules for **load** and **store**. We can argue (2) by showing that the inserted instrumentation does not alter any program values other than those created by the instrumentation itself, that the instrumentation only inserts code rather than modifying the instructions or the execution order of the instructions in the program, and that the inserted code does not produce any externally visible effect other than aborting when a run-time type violation occurs.

A formal proof of soundness is not provided in this work. Such a proof would likely go along the lines of the one given for SoftBound in Zhao (2013), by (1) defining an extended semantics for LLVM IR with the properties enforced by the Týr system built into the rules of the semantics; (2) proving that a program P does not commit spatial memory safety violations when run in this extended LLVM+Týr semantics; (3) formally defining the instrumentation pass performed by Týr as a mapping from LLVM IR to LLVM IR programs; and (4) showing that if program P is mapped to program P' by the instrumentation, then the execution of P' on the conventional LLVM IR semantics simulates the execution of the original P in the LLVM+Týr semantics, and therefore, by (2), the instrumented program does not commit spatial memory safety violations. The

formalization of such a proof is also a direction for future work.

Experimentally, we show that the type system is effective at preventing spatial memory safety violations and does not change the behavior of programs apart from aborting on run-time type violations by applying it to a set of benchmarks, as will be seen in Chapter 4.

3.8 Efficiency considerations

If every check inserted by the instrumentation were performed at run-time, the system would have a huge overhead. To make the system practical to use, checks that can be proven true at compile-time should be removed, thus making the overhead of the system more manageable. Whereas Deputy (CONDIT et al., 2007) included an optimization pass specially crafted for optimizing the checks it inserted, we rely solely on the standard optimization passes performed by LLVM during compilation. As we show in Chapter 4, this is quite effective in most cases. However, for this to work, we had to take the ability of LLVM to identify and optimize different kinds of expressions in account when designing the type system.

Two features of the type system have been designed specifically with this in mind. The original implementation of Týr followed Deputy in using addresses rather than indices for pointer bounds, i.e., the bounds of a pointer were the addresses of the lower and upper limits of the referenced region. Thus, a pointer p to a region of len integers would be given the type $Ptr(\text{i32}, p, p + len)$. This had the advantage that pointer arithmetic did not change the bounds of the pointer: $p + 1$ still had bounds $[p, p + len)$, because the bounds are relative to the original pointer, not to the pointer resulting from the pointer arithmetic. However, it turned out that LLVM was not generally able to optimize checks based on pointers. For example, consider the following code fragment using pointer p:

```
for (i=0; i<len; i++) {
  do_something_with(p[i]);
}
```

As explained before, **p[i]** is syntactic sugar for ***(p+i)**, i.e., pointer arithmetic followed by a dereference. Because p had bounds $[p, p + len)$, and $p + i$ *also* had bounds $[p, p + len)$, to check if the dereference was allowed (i.e., to see if $p + i$ points to at least one element), the system would add a check that $p+i \geq p \wedge (p+i)+1 \leq p+len$. LLVM was not generally able to remove such a pointer-based check, even though $0 \leq i < len$

within the body of the **for** loop. When applied to a test benchmark peforming 1000 iterations of a 100×100 matrix multiplication, that version of the system had an execution time overhead of 454% over the non-instrumented program.

After observing this, we changed the system to use integer indices rather than pointers for bounds. Then, the type of p was $Ptr(\mathrm{i8}, 0, len)$ rather than $Ptr(\mathrm{i8}, p, p{+}len)$. Now bounds are always relative to the pointer they are associated to, so when the pointer arithmetic $p + i$ is performed, the bounds of the resulting pointer must be adjusted to $[0 - i, len - i)$, rather than $[0, len)$. To check that the resulting pointer points to at least one element (i.e., that its bounds encompass at least the range $[0, 1)$), the system will add a check that $(0 - i) \leq 0 \wedge 1 \leq (len - i)$. LLVM is able to prove this integer-based check to be true within the body of the loop[5], and thus remove it from the program. This change alone dropped the overhead in the test benchmark from 454% to 176%.

The second observation we made was that most of the rest of the overhead was caused by nullness checks. The original system did not make a distinction between Ptr^- and Ptr^+ pointers, so every pointer access required a check to determine if the pointer was not null before use. LLVM was able to optimize simple cases of null checks, such as those involving pointers to globals (which are known to be not null), but not those involving non-constant pointers, such as those created by pointer arithmetic. In code like the fragment above, a test to check that $(p + i) \neq null$ would be inserted before the pointer was dereferenced, and that check would not be eliminated by LLVM. This is due to the fact that the result of the pointer arithmetic could in fact be null in case of overflow; however, as discussed in Section 3.4.2, such an overflowed pointer would necessarily be out-of-bounds, so the Týr system would not allow it to be dereferenced whether it were null or not. Therefore, the check for nullness in this case is superfluous within the context of Týr, but the LLVM optimization passes do not have enough information to know that.

To address this problem, we introduced the non-nullness tag to pointers. In general, a pointer is assumed to be nullable (Ptr^-) by default, so dereferences still require the nullness check. However, if pointer arithmetic is performed on a Ptr^- pointer, the source pointer is checked for nullness, but the *result* of pointer arithmetic is marked Ptr^+. This means that in the code fragment above, when computing $p + i$, the system will insert a check that $p \neq null$, but the resulting $p + i$ pointer is Ptr^+, so no check $(p{+}i) \neq null$ needs to be inserted before the dereference. Moreover, because the inserted

[5] $(0 - i) \leq 0$ is true because it is equivalent to $0 \leq i$, which is true inside the loop. $1 \leq (len - i)$ is true because this is equivalent (in integer arithmetic) to $0 < (len - i) \iff i < len$, which is also true inside the loop.

check $p \neq null$ does not change across the iterations of the loop (i.e., it does not depend on i), LLVM is able to move it out of the loop, thus drastically reducing the impact of the check in execution time. After this change, the overhead in the test benchmark dropped from 176% to the 9% reported in Chapter 4.

In this way, we were able to obtain reasonably low overhead without requring a specially crafted analysis pass to look for redundant checks.

3.9 Summary

In this chapter, we have presented in detail the Týr type-based code instrumentation. We have seen the kinds of types used to keep track of bounds and nullness information for pointers, and how the instrumentation rules handle each instruction to ensure that the invariants represented by the Týr types are maintained. We have seen how the conditions imposed by those rules are realized as actual LLVM IR instructions emitted in the instrumented program. We have discussed the soundness of the system, sources of unsoundness, and design decisions taken to reduce the performance impact of the instrumentation.

In the next chapter, we will present experimental results obtained with a prototype implementation of the system presented here, analyzing its efficacy with a number of example C programs.

4 EXPERIMENTAL RESULTS

This chapter presents experimental results obtained with a prototype implementation of the Týr code transformation. Section 4.1 discusses typical usage of the implementation to protect C programs against spatial memory safety violations. Section 4.2 presents the benchmarks used in the experiments. Section 4.3 presents the results obtained.

The source code of the implementation discussed in this chapter has been made available on GitHub[1]. A virtual machine with an environment prepared to run the benchmarks presented here is also available[2]. This virtual machine is installed with the versions of LLVM, Clang and other system dependencies used to perform the experiments, making it easier to replicate the results.

4.1 Usage

Using the implementation consists of running the **tyrcc** script, passing as arguments the C program to be compiled and, optionally, the file containing the dependent type annotations. The script performs the tasks of running Clang on the input C program, passing the LLVM IR output emitted by Clang to Týr for instrumentation, and feeding the instrumented result back into the LLVM compilation pipeline for optimization and machine code generation. If a dependent type annotations file is not provided, the script looks for a file with the same name as the C program being compiled but with a **.dep** extension. If no such file is found, no programmer-supplied dependent type information is used, and the system assumes default Týr types based on the LLVM types in the input program, as explained in the previous chapter.

In typical usage, the programmer may either provide dependent type annotations up front, or may opt to first apply Týr to the program without supplying any dependent type information. In the latter case, this will typically produce an instrumented binary with overly conservative run-time checks which will impede normal execution. For instance, in absence of annotations, Týr will normally assume that pointers have bounds $[0, 1)$, which will cause run-time checks to fail when the pointer is indexed with an index other than 0. Based on the reported errors, the programmer will then supply appropriate type annotations which will specify how bounds are expressed in the program, and run

[1] <https://github.com/vbuaraujo/tyr>
[2] <http://inf.ufrgs.br./~vbuaraujo/tyr>

Týr a second time to test the new set of annotations.

As an example, consider a C program which takes one or more command-line arguments, the first of which is an integer. The program prints the argument indexed by the integer:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
  int i = atoi(argv[1]);
  puts(argv[i]);
  return 0;
}
```

Note that this program is not memory-safe: if an improper index is passed in the command line, the program will perform an out-of-bounds access to the **argv** array, which may lead to a segmentation fault or reading an invalid region of memory.

If this program is compiled with **tyrcc** with no provided annotations, the generated binary will fail:

```
1.  $ tyrcc.sh -g argv.c
2.  WARNING: No depfile specified and argv.dep does not exist
3.  $ ./argv-tyr 2 foo bar
4.  ERROR: LLVM line 20:
5.      %6 = load i8*, i8** %5, !dbg !21
6.  Check (sub %.tyr.deref39 1#) sgt 0# violated (%.tyr.fail53, type-deref/3)
7.  argv.c: main() line 5, column 16
```

The print error message has four lines (lines 4–7 in the printout above):

- An indication of the location of the error in the input LLVM IR program;
- The line of LLVM IR code which caused violation of Týr's rules;
- A description of the check which failed; and
- The C source file, function, line and column, obtained from debug information emitted by Clang.[3]

Of these, the first three lines are of interest primarily for debugging the prototype itself. The last one, however, if of interest to the programmer: it says that the point in the source program which caused a violation is line 5, column 16, which refers to the **argv[1]** access to get the first argument from the command line.[4] This happens because Týr was

[3] Týr can only show this information if the program is compiled with debugging information (Clang flag **-g**). Note that enabling debugging is orthogonal to the optimization level: "LLVM debug information does not prevent optimizations from happening". (LLVM, 2015b)

[4] The element 0 in the **argv** array is the name of the currently executing program. The first supplied command line argument is element 1, i.e., the second element in the array.

not given enough information to know the bounds of the region pointed to by **argv**, which it then conservatively assigned the bounds $[0, 1)$.

We can then supply an appropriate annotation indicating that the upper bound for the **argv** pointer is given by the **argc** argument, and that the elements pointed to by **argv** are themselves string pointers (i.e., pointers to a sequence of characters (**i8**) delimited by a null terminator):

```
main: Fn i32 (argc: i32, argv: Ptr(SPtr(i8, 0, 0), 0, argc))
```

After this annotation is supplied, the example file can be recompiled and will successfully run when given correct arguments. Moreover, it will catch the invalid accesses performed when incorrect arguments are given to the program. Notice that there are three ways in which the program may finish:

- The provided index is valid: in this case, execution is successful.

```
$ ./argv-tyr 2 foo bar
foo
```

- The provided index is invalid: in this case, execution fails, and the error message indicates line 6, column 8 (the **argv[i]** access) as the problem.

```
$ ./argv-tyr 5 foo bar
ERROR: LLVM line 27:
    %12 = load i8*, i8** %11, !dbg !24
Check (sub %argc %9) sgt 0# violated (%.tyr.fail99, type-deref/3)
argv.c: main() line 6, column 8
```

- No index is provided: in this case, execution also fails, and the error message indicates line 5, column 16 as the problem (the **argv[1]** access to get the index).

```
$ ./argv-tyr
ERROR: LLVM line 20:
    %6 = load i8*, i8** %5, !dbg !21
Check (sub %argc 1#) sgt 0# violated (%.tyr.fail45, type-deref/3)
argv.c: main() line 5, column 16
```

4.2 Benchmarks

We performed a set of benchmarks to measure the performance impact of the instrumentation inserted by Týr in terms of execution time and memory consumption, as well as the effort taken to provide the dependent type information required by Týr. Most of the benchmarks used were taken from the Computer Language Benchmarks Game

(CLBG, 2016). Although these benchmarks were originally intended as a way to compare the performance of different programming languages, they have also been used in a number of works as a general benchmark suite, in contexts such as evaluating performance of language implementation techniques (WILLIAMS; MCCANDLESS; GREGG, 2010; WRIGSTAD et al., 2010; BRUNTHALER, 2010) and parallelization (SHIRAKO et al., 2009; GERAKIOS; PAPASPYROU; SAGONAS, 2010). Additionally, we included a matrix multiplication benchmark written by ourselves. All benchmarks used, as well as the scripts used to run them, are included with the Týr source code, in the **examples/benchmarksgame** directory.

Environment. The benchmarks were run on a quad-core Intel Core i5-2410M CPU with 8GB of RAM, configured to run fixed at its nominal frequency of 2.30GHz, with frequency scaling and Intel Turbo Boost disabled. The benchmarks were run on Debian GNU/Linux (kernel 4.3.0), running in single user mode (**init=/bin/bash**) to avoid interference from daemons running in background. The benchmarks were compiled with Clang/LLVM 3.7.1.

Procedure. For each benchmark, one binary with and one without Týr instrumentation was compiled. In a first run, their output was saved to a file and compared to ensure that the instrumentation did not change the output of the program. Afterwards, each version was run 30 times, with output redirected to **/dev/null** to avoid variability caused by I/O. The user CPU time (i.e., the time spent by the process actually scheduled in CPU in user mode, as opposed to time where the process was not scheduled or was running system calls) and peak memory consumption for each execution were collected using the GNU time utility (not the shell **time** builtin, which does not capture memory usage information). Then we computed the mean of the collected measures.

Annotation effort. Table 4.1 shows the number of non-blank lines of code in each benchmark program and in the corresponding dependent type annotations file which we wrote to be able to successfully compile and execute it with Týr. In most cases, only the top-level elements (functions and global variables) had to be annotated; for local variables, Týr's automatic inference and automatic bounds have been sufficient. The main exception was local pointers to strings, since the current implementation is not able to infer pointer null-terminatedness automatically except for pointers directly derived from null-terminated constants (such as C string constants).

The benchmark requiring the most programmer-supplied annotations is "regex-dna", with 32 lines. These include annotations for three external functions from the Perl-

Table 4.1: Non-blank lines of code in source file and in type annotations file

Benchmark	Code	Annotations	Ratio
binarytrees	118	7	5.9%
fannkuchredux	86	1	1.2%
fasta	166	12	7.2%
fastaredux	112	8	7.1%
knucleotide	145	25	17.2%
mandelbrot	49	2	4.1%
matrix	26	0	0.0%
nbody	128	5	3.9%
pidigits	45	1	2.2%
regexdna	105	32	30.5%
revcomp	72	14	19.4%
spectralnorm	40	4	10.0%
Mean			9.06%
Median			6.53%
Std. deviation			9.01%

Source: The author

Compatible Regular Expressions library which receive a large number of pointer arguments that require annotation. It also includes annotations for a number of local string pointers, since the current implementation cannot infer $SPtr$ pointers automatically.

The benchmark requiring the least annotations is “matrix”, which requires no annotations at all because all arrays used are global arrays with a statically declared size, and Týr can infer correct types for those automatically.

4.3 Results

Table 4.2 shows the CPU time and memory consumption of each benchmark running without and with Týr instrumentation, and the overhead of the instrumentation relative to the non-instrumented program. The largest CPU time overhead was shown by the “revcomp” benchmark, an outlier at 203.2%. For all other benchmarks, the overhead was below 27%, in a number of cases being close to zero.

The lowest overhead was shown by the “binarytrees” benchmark, which actually presented a speedup compared to the non-instrumented version. “binarytrees” works primarily with pointers to single objects rather than arrays, and therefore virtually no overhead was introduced in terms of bounds checking. Additionally, virtually all null checks inserted by Týr in this program can easily be proven redundant by the optimizer, since the program itself performs all the required null checks in the process of detecting the end of

Table 4.2: Benchmark results

	CPU time (seconds)			Peak memory usage (kilobytes)		
Benchmark	Plain	Týr	Overhead	Plain	Týr	Overhead
binarytrees	193.930	182.858	−5.7%	219 998	220 016	0.0%
fannkuchredux	55.661	67.658	21.6%	1 314	1 321	0.6%
fasta	4.972	5.247	5.5%	1 622	1 631	0.5%
fastaredux	1.748	2.012	15.1%	1 203	1 228	2.0%
knucleotide	31.997	37.022	15.7%	129 480	129 488	0.0%
mandelbrot	29.256	33.493	14.5%	32 476	32 569	0.3%
matrix	0.909	0.998	9.7%	1 199	1 180	−1.6%
nbody	10.278	13.004	26.5%	1 653	1 672	1.1%
pidigits	1.601	1.603	0.1%	2 281	2 320	1.7%
regexdna	29.944	30.104	0.5%	131 617	131 627	0.0%
revcomp	0.628	1.904	203.2%	249 490	249 500	0.0%
spectralnorm	15.490	15.510	0.1%	1 785	1 764	−1.2%
Global overhead statistics:						
Mean			25.6%			0.3%
Median			12.1%			0.1%
Std. deviation			56.8%			1.0%
Excluding “revcomp”:						
Mean			9.4%			0.3%
Median			9.7%			0.3%
Std. deviation			10.2%			1.1%

Source: The author

a binary tree. We hypothesize that the slight speedup was observed because the inserted checks, which abort execution when the checked conditions are false, allow the compiler to assume they are true after control flow passes through the check, thus enabling more optimizations, while at the same time very few bounds checks have been added by the instrumentation since the code performs almost no array access.

Further analysis of the "revcomp" outlier revealed some interesting facts. The first observation is that the non-instrumented "revcomp" program contains spatial memory safety violations when given incorrect inputs. The "revcomp" benchmark takes as input a text file consisting of sequences of records containing an identification header followed by a DNA sequence string. It performs a translation on those strings, and outputs a text file structured in the same way, with the translated DNA sequences. There are two memory safety violations in the benchmark code. First, it assumes that the header which separates each record in the input file is terminated by a newline character; if the newline is not present, the program reads beyond the buffer where it stores the input, leading to a segmentation fault. Second, it uses characters read from input as indices into a global 128-element array; if the input contains characters outside the range $[0, 128)$, the program performs an out-of-bounds access to this array, reading garbage data. In the version instrumented by Týr, both violations are caught.

The second observation is that the "revcomp" program uses start and end pointers as bounds when manipulating the input buffer rather than numeric bounds, and uses incrementing and decrementing of pointers to walk over the input buffer, rather than a fixed pointer plus an index. This is interesting because, as discussed in Section 3.8, the original design of Týr used pointer addresses as bounds rather than numeric indices, but we changed to numeric lower and upper bounds after we observed that LLVM is not generally able to optimize checks based on pointer arithmetic, whereas it is very good at optimizing index-based checks. In this case, however, we might have obtained better results by using pointer-based checks. From the point of view of the type language, these two ways of expressing bounds are equivalent: a pointer **p** to a region whose end is delimited by another pointer **q** can be expressed as having bounds $[p, q)$ in terms of addresses, or $[0, q - p)$ in terms of indices. Likewise, relative to the **q** pointer, the region can be given the bounds $[p - q, 0)$. Týr does support this style of bounds using pointer subtraction for function parameters; however, the LLVM IR code that Clang emits transfers function parameters to local variables in the stack at the beginning of the function, where they get automatic bounds, which are always stored as simple integers. Giving appropriate bounds to the **p**

Table 4.3: Benchmark results for modified versions of revcomp

	CPU time (seconds)			Peak memory usage (kilobytes)		
Version	Plain	Týr	Overhead	Plain	Týr	Overhead
original	0.628	1.904	203.2%	249 490	249 500	0.0%
safe	0.628	1.769	181.6%	249 485	249 522	0.0%
safe+indices	0.640	1.200	87.5%	249 494	249 518	0.0%

Source: The author

and **q** pointers (**from** and **to** in the relevant code in "revcomp") would require mutually dependent types – **p** has bounds $[0, q - p)$ and **q** has bounds $[p - q, 0)$ –, which is not supported for local variables in the current implementation of the system. This is more a matter of implementation, rather than a theoretical limitation of the system.

As an experiment, we produced two alternative versions of the "revcomp" program. In the first, we simply fixed the memory safety problems in the original program in a straightforward way. In the second, we also replaced the pointer-based accesses to the input buffer with index-based accesses. Table 4.3 shows the performance effects of these modifications. We observe that fixing the safety problems in the original reduced the overhead from 203.2% to 181.6%. Switching to using indices instead of pointers, the overhead is further reduced to 87.5% relative to the same program without instrumentation (91.0% relative to the original). This is still high when compared to the other benchmarks, but already shows a significant improvement relative to the pointer-based version. We believe the overhead is still high because the halting conditions in loops which walk over the buffer are not directly related to the bounds of the buffer (essentially, the code increments **p** and decrements **q** and stops when they meet), which makes optimization more difficult. Clearly this is an area which should be addressed by further work.

Overall, the mean CPU time overhead in the benchmarks was 25.6%, with a median of 12.1%. Excluding "revcomp", the mean drops to 9.4%, with a median of 9.7%. This shows that in general, standard compiler optimizations are able to remove most of the overhead of the instrumentation with run-time checks.

The memory usage overhead was negligible, being close to zero in most cases. The largest relative overhead was 2.0% in the "fastaredux" benchmark, but this corresponds to an absolute overhead of 24 kilobytes. The largest absolute overhead was 93 kilobytes in the "mandelbrot" benchmark. Because Týr does not change the representation of data, all overhead is due to an increase in code size due to the inserted checks, and additional usage of stack space for storing the bounds of automatic variables, and possibly for storing the temporary results of computing bound expressions at run-time. In two benchmarks, the

overhead was negative, but they represent an absolute reduction of 18 and 24 kilobytes, respectively; we believe that this can be accounted to normal execution variability.

In the next chapter, we will discuss related work and how the results presented here compare with other proposed systems addressing spatial memory safety.

5 RELATED WORK

This chapter presents a discussion of related work on memory safety of low-level programs. Section 5.1 presents an overview of other works addressing memory safety of C programs. Section 5.2 discusses tools for debugging memory safety problems in C. Section 5.3 discusses examples of programming languages designed to support low-level systems programming while provising some guarantees regarding memory safety. Section 5.4 presents a summary.

5.1 Memory safety in C

Deputy. Deputy (CONDIT et al., 2007; CONDIT, 2007) is a dependent type system for ensuring spatial memory safety in C, which is conceptually closest to Týr. Like Týr, Deputy is concerned only with spatial memory safety; complementary techniques, such as conservative garbage collection (BOEHM; WEISER, 1988), can be used with either system to address temporal memory safety.

As mentioned in Chapter 1, Deputy is based on the CIL framework (NECULA et al., 2002) for code analysis and transformation, which is tied to the C programming language, whereas Týr operates on LLVM IR, thus being more generally applicable to other languages which can be compiled to LLVM IR, such as C++ and Objective C.

Deputy supports dependent C-style unions, by allowing each clause of the union to be parameterized by a boolean condition that tells which element of the union is active at a given moment. This is currently not supported by Týr. Deputy also supports a form of parametric polymorphism to address the problem of arbitary casts from and to **void*** in C code, mentioned in Section 3.4.8. Deputy does not allow the unsafe type casts allowed by Týr's bitcast rule (Section 3.4.8) by default, but it does include constructs to allow the programmer to force an unsafe type cast, overriding the type system rules, which were required for them to be able to run certain benchmarks.

Condit (2007) reports execution time overheads from zero to 81% in a set of benchmarks, with an outlier at 3880% for a benchmark where the null checks were difficult to optimize away. Their average execution time overhead (calculated by the present author from the numbers in the table presented in Condit (2007, p. 90)) is 25%, excluding the outlier, which about the same number observed for Týr.

Deputy has a mode where null checks are disabled, under the assumption that null

pointer dereferences will be caught by the hardware and operating system. This option is not available to us with an unmodified LLVM, because LLVM treats null dereference as undefined behavior, and therefore enables optimizations which are unsafe in the presence of null pointer dereferences. Therefore, to ensure safety, we must guard null pointer dereferences with checks in code instrumented by Týr. With null checks disabled, Condit (2007) reports an overhead of 86% in the worst case, and a slowdown of less than 40% in all remaining tests. Týr already shows a slowdown under 27% for all but one of the benchmarks performed, so it is not clear whether the ability to disable null checks would significantly improve performance in Týr, or if the nullness analysis built into the type system in the form of the non-nullness tag in Ptr pointers is already sufficient to reduce the impact of null checks. In any case, the ability to disable null checks in Týr would require modifying LLVM to not optimize code under the assumption that null pointer dereference is undefined, to ensure that such dereferences are caught by the hardware. No figures are provided for the memory consumption overhead in Deputy.

SoftBound. SoftBound (NAGARAKATTE et al., 2009) is a compile-time transformation for enforcing spatial memory safety in C. Like Týr, SoftBound on its own does not address temporal memory safety, although it has been combined with CETS (NAGARAKATTE et al., 2010), a system for enforcing temporal memory safety from the same authors, to produce the SoftBoundCETS (NAGARAKATTE, 2012) system. Like Týr, SoftBound is implemented as an LLVM IR transformation.

SoftBound works with unmodified, unannotated C/C++ code. Unlike the approaches based on dependent types, it keeps its own metadata separately from the programmer visible data. For pointers in local variables, it uses extra local variables to store lower and upper bounds information, which is checked when the pointer is loaded from or stored to, and updated when the pointer itself is modified. In function calls, it passes lower and upper bounds information along with pointer arguments by changing the function signature to take the bounds as extra arguments. For in-memory pointers, it keeps a table data structure which is consulted when a pointer is loaded into a local register and updated when a pointer is stored in memory.

Nagarakatte (2012) reports an average execution time overhead of 74% for enforcing only spatial memory safety. For combined spatial and temporal memory safety, they report an average 108% overhead, which drops to 81% when applying a special optimization for type-safe programs which do not perform arbitrary pointer casts (the same casts which trigger the unsafe behavior of the bitcast rule described in Section 3.4.8). No

figures are provided for the overhead of only spatial memory safety with the type-safe optimizations enabled. In contrast, Týr presented an average 26% overhead in the performed benchmarks, with a median of 12%.

SoftBound has a "store-only" mode which only performs bounds checks when *storing* data via a pointer, but not when loading. In this mode, the reported average overhead is 41%, which is still higher than Týr. Moreover, whereas store-only checking can prevent a number of security vulnerabilities, it is not capable of preventing buffer overreads such as the one responsible for the Heartbleed vulnerability (DURUMERIC et al., 2014).

With respect to memory consumption, an average overhead of 94% is reported for SoftBound, whereas Týr presented an average of 0.3%. This is a result of SoftBound keeping its own metadata in a separate data structure, whereas Týr reuses the metadata already present in the program.

CCured. CCured (NECULA et al., 2005) is a type system and program transformation that ensures temporal and spatial memory safety for C programs. CCured classifies C pointers into SAFE (those not involved in casts or pointer arithmetic, and require only null checks), SEQ (those involved only in pointer arithmetic, and thus require bounds checking), and WILD (those involved in arbitrary casts, and thus require extra metadata to ensure the validity of the casts). It uses a whole-program analysis to infer the appopriate kind to assign to each pointer.

By default, CCured uses a fat pointer representation, storing lower and upper bounds together with pointers, thus changing the representation of data. To call external code, CCured either requires the programmer to define wrapper functions which specify the checks and conversions to be performed between CCured and external code, or it allows an alternative representation in which metadata is stored in a separate data structure which mirrors the shape of the program data; in the latter case, binary compatibility is preserved, but performance is reduced.

Temporal memory safety is enforced in two ways. For heap allocations, the standard C memory allocation functions are replaced with the Boehm-Demers-Weiser conservative garbage collector (BOEHM; WEISER, 1988), a technique which can also be used with Týr or the other systems for spatial memory safety. For stack allocations, CCured imposes a restrictive policy which forbids pointers to the stack from being stored into the heap or global variables, and only allows stack pointers in a stack frame to point to frames above it, thus ensuring that the target of the pointer will not be deallocated before

the pointer.

Necula et al. (2005) reports an execution time overhead between 3% and 87% in a set of benchmarks, with an outlier at 891%. The mean of the reported overheads for each of the benchmarks (calculated by the present author from the table in Necula et al. (2005, p. 36)) is 79% (31% excluding the outlier), with an average memory consumption overhead of 75% in average (there was no analogous outlier in memory consumption).

5.2 Memory debugging tools

Memcheck. Memcheck (SEWARD; NETHERCOTE, 2005) is a tool for detecting a wide range of memory errors in programs as they run, built on the top of the Valgrind (NETHERCOTE; SEWARD, 2007) dynamic binary instrumentation framework. Memcheck is capable of detecting some kinds of spatial and temporal memory safety violations, such as accesses past heap blocks, stack overflows, and uses of uninitialized and already-freed memory. However, unlike Týr, Memcheck is not able to tell the boundaries between contiguous objects in memory: for example, an out-of-bounds array access will not be detected if the out-of-bounds address happens to fall within another valid program object. Memcheck also incurs a large overhead: programs typically run about 20 to 30 times slower. Therefore, it is more useful as a debugging tool rather than a general tool for protecting programs against spatial memory safety violations.

AdressSanitizer. AdressSanitizer (SEREBRYANY et al., 2012) is an instrumentation for detecting some kinds of spatial and temporal memory safety violations in C/C++. It works by marking certain regions of memory as poisoned, and instrumenting loads and stores so that accesses to poisoned regions abort program execution. Spatial memory safety violations are detected by reserving a red zone around allocated objects and marking it as poisoned. Temporal memory safety violations are detected by marking the whole freed region as poisoned and putting it in quarantine for a period.

AddressSanitizer does not detect all spatial or temporal memory safety violations. For instance, out-of-bound accesses with offsets large enough to reach past the red zone and into another valid program object are not detected. The reported average CPU time overhead is 73%, with an increase of 3.37x (237% overhead) in memory consumption.

5.3 Safe systems programming languages

Cyclone. Cyclone (JIM et al., 2002) is a safe dialect of C which replaces C's unsafe constructs which safe variants. It imposes some restrictions on C features which are sources of unsafety, for instance restricting pointer arithmetic, arbitrary casts, memory freeing, and inserting null checks before dereferences. In their place, it adds a number of new features which reintroduce much of the same functionality in a safe way, such as fat pointers which allow pointer arithmetic with run-time bounds checking, never-null pointers (equivalent to Týr's Ptr^+ pointers, with similar behavior when casting from and to nullable pointers), parametric polymorphism, and region-based memory management. It does not aim to be compatible with C (i.e., one cannot generally compile an unmodified C program with Cyclone), but rather aims to facilitate porting C programs to Cyclone. Reported execution time in a set of benchmarks varies between 0% and 185% relative to the corresponding C programs.

Rust. Rust (RUST, 2016) is a systems programming language aiming to ensure type, memory and thread safety. Spatial memory safety is generally addressed through run-time bounds checking, and forbidding unsafe casts by default. Temporal memory safety is addressed through a scheme of pointer ownership which constrains the copying of pointers (and therefore accessibility of the same region of memory by multiple parts of the program) in a way that makes it possible for the system to ascertain when a region can be freed. The same mechanism also plays a role in thread safety, by constraining access of the same data by multiple threads. Rust is an entirely new language, which does not aim compatibility of ease of portability from C.

Rust includes an unsafe subset, which allows raw pointer manipulation in a way that is otherwise precluded by the language. This is intended to allow writing bindings to code written in C and other languages, and to allow safe abstractions to be implemented in an efficient way which would be normally precluded due to the type system being too conservative. Unsafe code must be explicitly marked with the **unsafe** keyword. The idea is to isolate unsafe code, and that functionality implemented using **unsafe** be exposed through safe interfaces to the rest of the program. Since Rust compiles to LLVM IR, it might be interesting to investigate the applicability of Týr to enforce spatial memory safety in those parts of Rust code which use unsafe features.

5.4 Summary

With an average CPU time overhead of 25% and near zero memory overhead, Týr is competitive in terms of performance with other approaches for spatial memory safety. The numbers are similar to those reported for Deputy, and generally better than other approaches for memory safety in C. The tools also vary in what kinds of errors they protect against (spatial vs. temporal memory safety), what coverage they have in detecting them (i.e., whether they have false negatives), and whether they are specific to C (such as Deputy and CCured) or more language-agnostic (such as Týr and SoftBound). Some tools are more appropriate for debugging, rather than for use in production code. Finally, some works adress the problem of memory-safe systems programming by designing new languages where programs are safe by construction, at the expense of backwards compatibility with existing code.

6 CONCLUSION

This work presented Týr, a dependent type system and associated program transformation for ensuring spatial memory safety of C programs at the LLVM IR level, by allowing programmers to describe at the type level the relationships between pointers and bounds information already present in C programs. In this way, Týr ensures spatial memory safety by checking the consistent usage of this pre-existing metadata, through a combination of static type checking and run-time checks inserted in the program. We have shown that the resulting system is effective at protecting against spatial memory safety violations, with a reasonably low execution time overhead and nearly zero memory consumption overhead. We have also shown that by designing the type system with the LLVM optimization passes in mind, we were able to emit run-time checks in a form that is more easily optimizable by LLVM, thus achieving performance competitive with other systems for spatial memory safety without requiring a specially crafted optimization pass.

There is a number of possible directions for future work. Some possibilities of interest are listed below.

Integration with Clang. Currently, Týr stands as a completely independent piece of software relative to the Clang/LLVM framework, taking type annotations as an input separate from the program to be instrumented. It might be interesting to provide better integration with the Clang/LLVM infrastructure. For instance, the Clang compiler could be modified to allow dependent types to be written as annotations within the C/C++ code, as in Deputy, rather than being provided in a separate file. Such annotations could then be passed on to LLVM as metadata embedded in the LLVM IR program. The source-level annotations would be written in terms of the C level types and mapped to the LLVM IR level types used by Týr. Finally, Týr could be reimplemented as a pass internal to LLVM, rather than being an external program. This integration might improve the usability of the system by programmers.

Extension to languages other than C. In this work, we focused on C programs. However, one of the main advantages of targeting LLVM IR is the possibility of extending the system for other languages using the LLVM infrastructure, such as C++, Objective C, and Rust. This might involve extending the type system to account for features of those languages not present in C, as well as mapping the constructs of those languages to the types provided by Týr.

Extending Týr to C++ would require the creation of a mapping from C++ level

dependent types to the LLVM IR levle dependent types used by the instrumentation. This mapping would mirror the way Clang maps C++ level constructs, such as classes, methods and templates, into LLVM IR. It might be necessary to create new Týr types to accomodate they way those constructs are represented at the LLVM IR level.

Formal soundness proof. As discussed in Section 3.7, we currently do not have a formal proof of correctness of the system. Such a proof would entail defining a formal semantics of LLVM IR augmented with the properties enforced by Týr, and demonstrating that the program transformation performed by Týr is sufficient to enforce that semantics in LLVM IR programs.

Support for the whole LLVM IR language. As discussed in Section 2.5.5, there are some features of LLVM IR which are currently not fully supported by Týr, especially those concerned with threads and synchronization, and vectorized data and instructions (SIMD). It would be desirable to extend the implementation to cover all of the LLVM IR language.

Experiments with larger benchmarks. The experimental validation in this work has been performed with relatively small benchamrk programs. This has been done in part because of the annotation effort involved in applying Týr to the programs to be instrumented. With better integration of the dependent type annotations in Clang at the source language level, as described above, it is expected that the the effort to annotate programs would be smaller, therefore making it more feasible to apply Týr to larger programs. This would give a better idea of how the system behaves with a wider variety of constructs and idioms used in real-world programs, both in terms of performance, and in the ability of the type language to describe bounds information as it appears in a wider range of C programs. In this process, we may find it necessary to extend the type language with newer types to describe other situations not anticipated in the current work.

Treatment of arbitrary casts. C allows arbitrary type casts between pointers, and C programs use such casts to implement forms of polymorphism. As explained in Section 3.4.8, currently Týr allows such casts to allow these programs to be compiled, at the expense of safety. Allowing such casts while ensuring safety is a hard problem. As seen in Chapter 5, other systems for ensuring spatial memory safety in C all encounter problems when dealing with arbitrary casts and employ different solutions: Deputy (CONDIT et al., 2007) introduces parametric polymorphism, and the ability for the programmer to explicitly override the type system when a cast cannot be proven safe; SoftBound (NAGARAKATTE et al., 2009) has a greater overhead in the presence of ar-

bitrary casts, since some optimizations are only enabled for programs not using arbitrary casts; likewise, CCured classifies pointers involved in arbitrary casts as WILD, which carry extra metadata and have a high associated overhead. Finding the best way of handling this problem in Týr is a problem to be considered, the Deputy solution being the easiest to adopt, given the conceptual similarity of Týr and Deputy.

Annotating the standard C library. Correct usage of external libraries from instrumented programs requires the programmer to provide correct annotations for the external functions and variables. It would be interesting to provide annotations for a representative portion of the standard C library out of the box, since most C programs will use functions and variables from it. Doing so not only improves the usability of the system, by reducing the amount of annotations the programmer has to provide, but also presents another opportunity to test the ability of the current type language to describe bounds information in a wider variety of real-world situations.

REFERENCES

ALPERN, B.; WEGMAN, M. N.; ZADECK, F. K. Detecting equality of variables in programs. In: ACM. **Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. [S.l.], 1988. p. 1–11.

ASPINALL, D.; HOFMANN, M. Dependent types. In: PIERCE, B. C. (Ed.). **Advanced Topics in Types and Programming Languages**. [S.l.]: MIT Press, 2004. ISBN 0262162288.

AUGUSTSSON, L. Cayenne—a language with dependent types. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 1998. v. 34, n. 1, p. 239–250.

BERGER, E. D.; ZORN, B. G. Diehard: probabilistic memory safety for unsafe languages. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 2006. v. 41, n. 6, p. 158–168.

BOEHM, H.-J.; WEISER, M. Garbage collection in an uncooperative environment. **Software: Practice & Experience**, John Wiley & Sons, Inc., New York, NY, USA, v. 18, n. 9, p. 807–820, set. 1988. ISSN 0038-0644. Disponível em: <http://dx.doi.org/10.1002/spe.4380180902>.

BRADY, E. C. Idris: systems programming meets full dependent types. In: ACM. **Proceedings of the 5th ACM workshop on Programming Languages Meets Program Verification**. [S.l.], 2011. p. 43–54.

BRUNTHALER, S. Inline caching meets quickening. In: D'HONDT, T. (Ed.). **Proceedings of ECOOP 2010 – Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010.** Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 429–451. ISBN 978-3-642-14107-2. Disponível em: <http://dx.doi.org/10.1007/978-3-642-14107-2_21>.

CLANG. **Clang: a C language family frontend for LLVM**. 2015. <http://clang.llvm.org>. Accessed in July 2015.

CLBG. **The Computer Language Benchmarks Game**. 2016. <https://benchmarksgame.alioth.debian.org/>. Accessed in February 2016.

CONDIT, J. et al. Dependent types for low-level programming. In: **Programming Languages and Systems**. [S.l.]: Springer, 2007. p. 520–535.

CONDIT, J. P. **Dependent types for safe systems software**. Tese (Doutorado) — University of California, Berkeley, 2007.

CRISWELL, J.; GEOFFRAY, N.; ADVE, V. S. Memory safety for low-level software/hardware interactions. In: **USENIX Security Symposium**. [S.l.: s.n.], 2009. p. 83–100.

CUI, S.; DONNELLY, K.; XI, H. ATS: A language that combines programming with theorem proving. In: **Proceedings of the 5th International Workshop on Frontiers of Combining Systems (FroCoS)**. [S.l.]: Springer, 2005. p. 310–320.

DURUMERIC, Z. et al. The matter of Heartbleed. In: **Proceedings of the 2014 Conference on Internet Measurement Conference**. New York, NY, USA: ACM, 2014. (IMC '14), p. 475–488. ISBN 978-1-4503-3213-2. Disponível em: <http://doi.acm.org/10.1145/2663716.2663755>.

GERAKIOS, P.; PAPASPYROU, N.; SAGONAS, K. Race-free and memory-safe multithreading: Design and implementation in cyclone. In: **Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation**. New York, NY, USA: ACM, 2010. (TLDI '10), p. 15–26. ISBN 978-1-60558-891-9. Disponível em: <http://doi.acm.org/10.1145/1708016.1708020>.

ISO/IEC. **International Standard ISO/IEC 9899:2011 – Programming languages – C (Committee Draft – April 12, 2011)**. [s.n.], 2011. Disponível em: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>.

JIM, T. et al. Cyclone: A safe dialect of C. In: **USENIX Annual Technical Conference, General Track**. [S.l.: s.n.], 2002. p. 275–288.

KUMAR, R.; KOHLER, E.; SRIVASTAVA, M. Harbor: software-based memory protection for sensor nodes. In: ACM. **Proceedings of the 6th international conference on Information processing in sensor networks**. [S.l.], 2007. p. 340–349.

LATTNER, C.; ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In: IEEE. **Code Generation and Optimization, 2004. CGO 2004. International Symposium on**. [S.l.], 2004. p. 75–86.

LLVM. **LLVM Language Reference Manual**. 2015. <http://llvm.org/releases/3.7.0/docs/LangRef.html>. Accessed in January 2018.

LLVM. **Source Level Debugging with LLVM**. 2015. <http://releases.llvm.org/3.7.0/docs/SourceLevelDebugging.html>. Accessed in January 2018.

LLVM. **Type-Based Alias Analysis [source code comments]**. 2016. <http://llvm.org/docs/doxygen/html/TypeBasedAliasAnalysis_8cpp_source.html>. Accessed in May 2016.

NAGARAKATTE, S. et al. SoftBound: highly compatible and complete spatial memory safety for c. In: ACM. **ACM Sigplan Notices**. [S.l.], 2009. v. 44, n. 6, p. 245–258.

NAGARAKATTE, S. et al. CETS: compiler enforced temporal safety for c. In: ACM. **ACM Sigplan Notices**. [S.l.], 2010. v. 45, n. 8, p. 31–40.

NAGARAKATTE, S. G. **Practical low-overhead enforcement of memory safety for C programs**. Tese (Doutorado) — University of Massachusetts Amherst, 2012.

NECULA, G. C. et al. CCured: type-safe retrofitting of legacy software. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 27, n. 3, p. 477–526, 2005.

NECULA, G. C. et al. CIL: Intermediate language and tools for analysis and transformation of C programs. In: SPRINGER. **Compiler Construction**. [S.l.], 2002. p. 213–228.

NETHERCOTE, N.; SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: **Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2007. (PLDI '07), p. 89–100. ISBN 978-1-59593-633-2. Disponível em: <http://doi.acm.org/10.1145/1250734.1250746>.

PIERCE, B. C. **Types and Programming Languages**. [S.l.]: MIT Press, 2002.

RUST. **The Rust Programming Language**. 2016. <https://doc.rust-lang.org/book/>. Accessed in June 2016.

SEREBRYANY, K. et al. AddressSanitizer: A fast address sanity checker. In: **USENIX Annual Technical Conference**. [S.l.: s.n.], 2012. p. 309–318.

SEWARD, J.; NETHERCOTE, N. Using valgrind to detect undefined value errors with bit-precision. In: **USENIX Annual Technical Conference, General Track**. [S.l.: s.n.], 2005. p. 17–30.

SHIRAKO, J. et al. Phaser accumulators: A new reduction construct for dynamic parallelism. In: **Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on**. [S.l.: s.n.], 2009. p. 1–12. ISSN 1530-2075.

SØRENSEN, M. H.; URZYCZYN, P. **Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)**. New York, NY, USA: Elsevier Science Inc., 2006. ISBN 0444520775.

WILLIAMS, K.; MCCANDLESS, J.; GREGG, D. Dynamic interpretation for dynamic scripting languages. In: **Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization**. New York, NY, USA: ACM, 2010. (CGO '10), p. 278–287. ISBN 978-1-60558-635-9. Disponível em: <http://doi.acm.org/10.1145/1772954.1772993>.

WRIGSTAD, T. et al. Integrating typed and untyped code in a scripting language. In: **Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 2010. (POPL '10), p. 377–388. ISBN 978-1-60558-479-9. Disponível em: <http://doi.acm.org/10.1145/1706299.1706343>.

ZHAO, J. **Formalizing an SSA-based compiler for verified advanced program transformations**. Tese (Doutorado) — Princeton University, 2013. Disponível em: <http://www.cis.upenn.edu/~stevez/vellvm/Zhao13.pdf>.