
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**
**DEPARTMENT OF INTELLIGENT SYSTEMS**


# INSTRUMENTACE PROGRAMŮ PRO MĚŘENÍ POKRYTÍ

**PROGRAM INSTRUMENTATION ENABLING COVERAGE MEASUREMENT**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** **JAN VÁCLAVÍK**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Ing. ALEŠ SMRČKA, Ph.D.**
**SUPERVISOR**



**BRNO 2023**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

# Zadání bakalářské práce

142733

| | |
|---|---|
| Ústav: | Ústav inteligentních systémů (UITS) |
| Student: | **Václavík Jan** |
| Program: | Informační technologie |
| Specializace: | Informační technologie |
| Název: | **Instrumentace programů pro měření pokrytí** |
| Kategorie: | Analýza a testování softwaru |
| Akademický rok: | 2022/23 |

Zadání:

1. Nastudujte metody testování softwaru na základě modelů. Nastudujte kritéria pokrytí kódu.
2. Navrhněte nástroj pro instrumentaci programů při překladu s injekcí kódu pro měření vybraných kritérií pokrytí.
3. Implementujte nástroj pro instrumentaci programů v jazycích C/C++. Pro instrumentaci využijte infrastrukturu překladače clang nebo gcc. Implementujte kód pro počítání pokrytí pro kritéria zahrnující řádky kódu, rozhodovací logiku a datové toky.
4. Vytvořte demonstrační testovací sadu. Ověřte základní funkcionalitu automatickými testy.

Literatura:

- ISO/IEC/IEEE 29119-4:2015(E) Software and system engineering -- Software testing -- Test techniques.

Při obhajobě semestrální části projektu je požadováno:
První dva body zadání

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

| | |
|---|---|
| Vedoucí práce: | **Smrčka Aleš, Ing., Ph.D.** |
| Vedoucí ústavu: | Hanáček Petr, doc. Dr. Ing. |
| Datum zadání: | 1.11.2022 |
| Termín pro odevzdání: | 10.5.2023 |
| Datum schválení: | 3.11.2022 |

# Abstrakt

Tato práce se zabývá návrhem a implementací nástroje Ginstrum sloužící pro instrumentaci programů napsaných v jazyce C během překladu. Nástroj je implementován formou zásuvného modulu pro překladač GCC a instrumentuje místa programu, která přistupují do paměti, zapisují do paměti nebo volají funkce. Vzhledem k plánovanému použití nástroje pro testování a dynamickou analýzu umožňuje nástroj také vytvořit data z překladu, která slouží pro následnou analýzu pokrytí překládaného kódu testy.



# Abstract

This thesis deals with the design and the implementation of the Ginstrum tool for compile time instrumentation of C programs. The tool is implemented as a GCC Plugin and instruments places in program that access memory, write to memory or call functions. The tool also provides compile information that can be used for the code coverage measurement during testing and dynamic analysis.



# Klíčová slova

instrumentace, GCC, GENERIC, GIMPLE, graf toku řízení, abstraktní syntaktický strom, kritérium pokrytí, základní blok, GCC modul

# Keywords

instrumentation, GCC, GENERIC, GIMPLE, control flow graph, abstract syntax tree, coverage criterion, basic block, GCC plugin


# Citace

VÁCLAVÍK, Jan. *Instrumentace programů pro měření pokrytí.* Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

# Instrumentace programů pro měření pokrytí

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . . . .
Jan Václavík
7. května 2023

## Poděkování

Děkuji doktoru Aleši Smrčkovi za cenné rady a vedení této práce. Dále děkuji rodičům za podporu při studiu a Tobiáši Krupovi za korekturu textu.

# Obsah

# Seznam obrázků

# Kapitola 1

# Úvod

Ke tvorbě softwarových řešení dnes neodmyslitelně patří verifikace a validace. Verifikace se zabývá ověřování správnosti řešení a většinou je řešena pomocí systematického testování. Jednou z mnoha technik testování softwaru patří verifikace za běhu, která při testování analyzuje běh (trasu) programu. K této analýze je potřeba sbírat a zpracovávat události, které program za běhu vykonává.

Cílem této práce je vytvořit nástroj, který má za úkol přidat do kódu daného programu speciální volání (tzv. sondy), která dají analyzátoru vědět, že se v programu stala nějaká událost, například čtení z paměti, zápis do paměti nebo volání funkce. Tomuto procesu se říká instrumentace. Jakmile se instrumentovaný program spustí a narazí se na určitou událost, daná sonda se aktivuje a podá hlášení o tom, co přesně se stalo.

V kapitole 2 jsou shrnuty informace o grafu toku řízení, s jehož pomocí se dají měřit různá kritéria pokrytí. Dále jsou zde nastíněny nejdůležitější fáze překladu a také nástroje sloužící k překladu kódu a k měření pokrytí. Poslední část se do detailu zabývá uzly abstraktního syntaktického stromem GCC, jelikož pochopení jeho struktury je klíčové pro další části práce. Kapitola 3 začíná požadavky na to, co má nástroj umět, dále následuje seznam parametrů, které má nástroj ze spouštěného programu sbírat a také přístup, kterým je celý nástroj realizován. Nemalá část této kapitoly je poté věnována tomu, jakým způsobem tato speciální volání přidat do kódu pro různé konstrukce tak, aby to bylo co nejjednodušší a zároveň, aby nedošlo ke změně chování programu, který má být o tyto funkce obohacen. Poslední část se zabývá návrhem podkladů, které pomohou uživateli určit, jak dobře program testuje. Kapitola 4 se pak zabývá implementačními detaily celého nástroje. Je zde popsáno, jakým způsobem jsou tyto funkce tvořeny a jakým způsobem se přidají do stromu celého programu. Také je zde popsáno, jak je dosaženo toho, že je vidět správný řádek, kde se událost stala. Závěrem je také popsaná implementace, na základě které je vytvořen soubor obsahující informace, které pomohou při testování uživatelského programu. Poslední kapitola 5 nejprve nastiňuje, jak lze tento program ladit pomocí různých funkcí. Dále je popsáno, jak je testována funkčnost celého nástroje pomocí automatických testů a na kterých verzích GCC je nástroj schopný pracovat.

# Kapitola 2

# Současný stav

Tato kapitola přináší nezbytné informace pro pochopení celé práce jak z hlediska technického, tak i z hlediska jejího využití. Jelikož je díky instrumentačnímu nástroji možnost měřit různá kritéria pokrytí, je zapotřebí vědět, co je podstatou instrumentace a testování na základě modelů samotného. Cílem je také seznámit se s různými kritérii pokrytí, které je možné díky této práci měřit.

Jelikož program provádí instrumentaci za překladu, je zde také popsáno, jak překladače program překládají, jaké jsou jejich fáze a příklady překladačů, které se pro jazyk C používají.

V poslední části lze nalézt důkladnější popis uzlů abstraktního syntaktického stromu, který překladač GCC vytváří v rámci syntaktické analýzy, neboť princip fungování instrumentačního nástroje je založen právě na čtení, modifikaci a přidávání těchto uzlů do stromu.

## 2.1 Testování na základě modelů

Testování na základě modelů (angl. *model-based testing*) je způsob návrhu testů na základě abstraktního modelu, který reprezentuje nějakou vlastnost programu. Modely mohou být popsány formou UML diagramu, Petriho sítí nebo také grafem toku řízení [21]. Vlastnosti programu, které se mohou během testování sledovat, jsou například řádky kódu, větvení či použití všech proměnných.

Aby bylo možno tyto vlastnosti pomocí testování měřit, je potřeba při spuštění testu zjistit, kudy v testovaném kódu test prošel. Pro takovou detekci je potřeba testovaný kód instrumentovat.

### Instrumentace kódu

Jedná se o techniku, kdy je do původního zdrojového textu na určitá místa přidán nový kód. Jakmile program těmito místy projde, dojde k provedení tohoto přidaného kódu, který má obvykle za úkol podat informace o konkrétním místě v programu. Instrumentace může být prováděna manuálně uživatelem, který ji nejčastěji provádí jen pro účely ladění a poté z programu tento kód vymaže, nebo probíhá automaticky za pomoci jiného programu. Nově přidaný kód by však za žádných okolností neměl ovlivňovat původní chování programu, i když může být v některých případech záměr chování programu měnit [12]. Instrumentace kódu je velmi důležitá technika a má různá využití jako například:

- **analýza výkonnosti** (profiling) – díky instrumentaci je možné určit, jak dlouho trvá, než se vykonají určité části programu, které části trvají nejdéle a měly by se optimalizovat [12],
- **ladění programu** – výpisy, které instrumentace dokáže poskytnout, mohou pomoci programátorovi rychleji nalézt chybu a opravit ji [12],
- **sledování kódu** – cílené vypisování toho, co se v programu děje, může opět pomoct programátorovi s hledáním chyby, ale také může sloužit ke sběru statistik, případně může pomoct při testování, neboť z logů lze poznat kudy se v programu již prošlo a zacílit další testy na části, kterými se neprošlo s cílem dosáhnout nějakého kritéria pokrytí [12].

Instrumentace se poté dělí na statickou a dynamickou podle toho, kdy k ní dochází.

### Statická instrumentace

Statickou instrumentací je myšleno přidávání kódu ještě předtím, než je program spuštěn, a probíhá za překladu na úrovni zdrojového kódu případně mezikódu. Instrumentační nástroj vyvíjen v rámci této práce implementuje právě tento typ instrumentace.

Výhodou tohoto typu instrumentace je čitelnost výstupu, jelikož během překladu se lze dostat k informacím o zdrojovém kódu samotném. Lze z něj získat názvy proměnných, funkcí, souborů nebo také čísla řádků a lze tedy dobře ve výpisech dohledat, co se kde přesně stalo. Nevýhodou je pak to, že překladač je velmi složitý program s velmi složitou strukturou a může být obtížné instrumentovat s využitím struktur, které využívá samotný překladač. Dále také může dojít k tomu, že instrumentace neproběhne správně a program se ani nepřeloží nebo se úplně změní jeho chování [16].

### Dynamická instrumentace

Tato instrumentace probíhá až za běhu programu a může se využívat tam, kde statická instrumentace nestačí, a sice při instrumentaci knihovních funkcí a kódu, ke kterému se statická instrumentace nedostane, jelikož binární soubor obsahuje veškeré instrukce, které jsou potřeba k běhu a ne jen instrukce získané překladem vytvořených zdrojových textů.

Výhoda tkví v tom, že lze instrumentovat jakýkoliv kód a není potřeba přidávat nové průchody do překladu, který poté není pomalý. Nevýhodou však je, že instrumentace za běhu může program výrazně zpomalit [16]. Mezi příklady lze uvést například nástroj Valgrind[1] používaný pro analýzu vícevláknových programů, paměti a výkonnosti [28].

## Graf toku řízení

Graf toku řízení, zkráceně CFG (z angl. *Control flow graph*), je graf ukazující všechny různé cesty, kterými program může projít. Jedná se o orientovaný graf, kde uzly reprezentují jednotlivé základní bloky a hrany poté tyto bloky spojují.

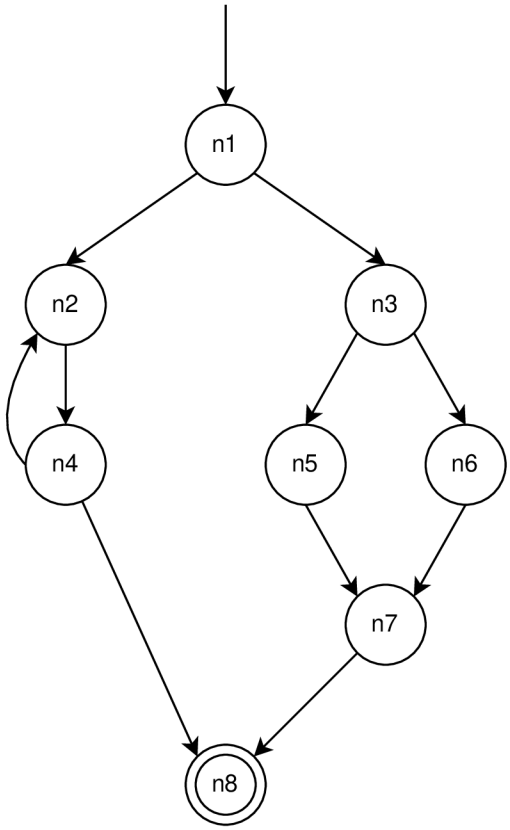
Formálně řečeno je graf toku řízení je čtveřice $G = (N, N_0, N_f, E)$, kde

- $N$ je konečná množina uzlů,
- $N_0 \subseteq N$, $N_0 \neq \emptyset$ je neprázdná množina počátečních uzlů,

[1] https://valgrind.org/

- $N_f \subseteq N$ je množina koncových uzlů,
- $E \subseteq N \times N$ je množina všech hran [24].

Uzlem je v tomto případě myšlen základní blok, který reprezentuje největší možnou posloupnost instrukcí, která bude programem vykonána, tedy pokud se program do tohoto bloku dostane, tak vykoná sekvenčně všechny instrukce, které se v něm nacházejí. Zároveň existuje pouze jediné místo, kudy může program do bloku vstoupit a pouze jediné místo, kterým z něj může vystoupit. Hrana poté spojuje uzly a vede vždy právě jedním směrem [20]. Základní blok, který je pro graf toku řízení konečný, se označuje dvojitým zakroužkováním. V případě obrázku 2.1




Obrázek 2.1: Příklad grafu toku řízení.

$$N = \{n1, n2, n3, n4, n5, n6, n7, n8\},$$
$$E = \{(n1, n2), (n1, n3), (n2, n4), (n4, n2), (n3, n5), (n3, n6),$$
$$(n4, n8), (n5, n7), (n6, n7), (n7, n8)\}.$$

### Cesta, délka cesty

Cesta je podgraf, který je tvořen neprázdnou sekvencí uzlů a hran, které se mohou opakovat. Délku cesty lze pak definovat jako počet hran v dané sekvenci nebo počet všech uzlů v dané cestě $n$ zmenšený o 1 [7].

Jako příklad cest lze na obrázku 2.1 zvolit $P_1 = [n1, n2, n4, n8]$, která má délku 3, nebo $P_2 = [n1, n3, n5, n7, n8]$, jejíž délka je 4.

### Kritérium pokrytí

Kritérium pokrytí je *pravidlo nebo soubor pravidel pro systematické generování požadavků na test* [24]. Určuje tedy, jakým způsobem jsou tvořeny testy pro daný program.

Aby bylo možné posoudit, jak dobře daná testovací sada testuje daný program, je počítáno tzv. pokrytí, které je udáváno většinou v procentech. Čím vyšší toto číslo je, tím jsou testy lépe testují daný program [4].

Existuje mnoho různých kritérií pokrytí. Některá jsou silnější než jiná, to znamená že testy splňující vyšší kritérium pokrytí testují program lépe než testy splňující nižší kritérium. Záleží však na aplikaci, jaké kritérium se zvolí, protože ne vždy je možné a potřebné použít právě to nejsilnější.

V následujících podkapitolách jsou uvedena typická kritéria pokrytí, podle kterých je možno generovat testy, pakliže známe graf toku řízení.

### Pokrytí všech uzlů a hran

Z anglického *node coverage* je pokrytí, při kterém jsou testy konstruovány tak, aby došlo k průchodu všech základních bloků. Jelikož se v základním bloku nachází sekvence instrukcí, které budou vždy provedeny, je toto pokrytí ekvivalentní s pokrytím všech řádků kódu. Toto pokrytí je však docela slabé, protože nám nezáleží na tom, jakým způsobem se do daného uzlu dostaneme a stačí pokrýt pouze jeden takový způsob [24].

Pro obrázek 2.1 by minimální testovací sada $T$ splňující toto kritérium pokrytí vypadala takto

$$T = \{[n1, n2, n4, n8], [n1, n3, n5, n7, n8], [n1, n3, n6, n7, n8]\}.$$

O něco silnější je pokrytí všech hran (*edge coverage*), kde musí testy zajistit průchod všemi hranami, což v praxi odpovídá pokrytí všech větví v kódu. Tedy nezajímá nás jakýkoliv jeden způsob, jak se dostat do základního bloku, ale zajímají nás všechny možné způsoby [24].

Pro obrázek 2.1 by testovací sada $T$ vypadala velmi podobně jako v případě pokrytí všech uzlů, ale jedna cesta by zde chyběla, a sice ta, kde by došlo k pokrytí hrany vedoucí z $n4$ do $n2$. Úplná testovací sada by tedy vypadala následovně:

$$T = \{[n1, n2, n4, n8], [n1, n2, n4, n2, n4, n8], [n1, n3, n5, n7, n8], [n1, n3, n6, n7, n8]\}.$$

### Pokrytí všech párů hran

Z anglického *edge-pair coverage* je kritérium pokrytí, v němž má testovací sada za úkol projít všemi dvojicemi hran, neboli projít každou podcestu mající délku menší nebo rovnu dvěma. Toto kritérium pokrytí je ještě silnější než kritérium pokrytí všech uzlů a hran [19].

Je-li opět využit obrázek 2.1 jako příklad, vypadala by testovací sada shodou okolností úplně stejně jako v případě kritéria pokrytí všech hran.

### Pokrytí datových toků

Tato skupina pokrytí se úplně neváže k pokrytí uzlů či hran, ale je spojená s proměnnými, přesněji řečeno se čtením a zápisem. V rámci této skupiny jsou zavedeny pojmy *def*, které reprezentuje zápis do proměnné, a *use* reprezentující její čtení. V této skupině lze nalézt 3 různá kritéria pokrytí.

1. *All-defs coverage*: Toto kritérium pokrytí je splněno na 100 %, pokryjí-li testy alespoň jednou zápis do každé proměnné.

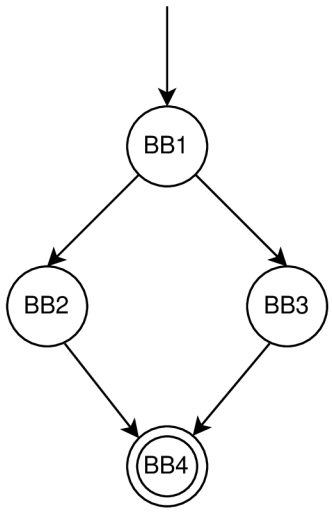
2. *All-uses coverage*: Zde je potřeba, aby testovací sada pokryla alespoň jednu cestu vedoucí od zápisu do proměnné k jejímu použití pro všechny proměnné.

3. *All-DU-Paths coverage*: Je velmi podobné jako *All-uses coverage* s tím rozdílem, že nestačí pouze jedna cesta od zápisu do proměnné ke čtení, ale jsou potřeba všechny cesty [10].

### Příklady CFG

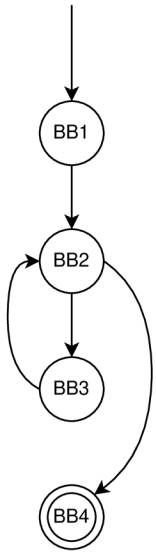
V této podkapitole jsou demonstrovány na obrázcích 2.2, 2.3 a 2.4 grafy toku řízení pro běžné konstrukce, se kterými se lze během programování a testování setkat.

```
1: int a = 5;
1: if (a == 5) {
2:     a += 1;
2: }
3: else {
3:     a += 2;
3: }
4: return a;
```



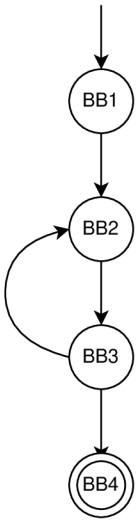

Obrázek 2.2: Graf toku řízení podmínky s korespondujícím kódem.

```
1: int a = 5;
2: while (a >= 0) {
3:     a--;
3: }
4: return a;
```




Obrázek 2.3: Graf toku řízení a kód pro cyklus `while`. Stejný graf toku řízení lze aplikovat i pro cyklus `for`.

```
1: int a = 5;
2: do {
2:     a--;
2: }
3: while (a >= 0);
4: return a;
```




Obrázek 2.4: Graf toku řízení a kód pro cyklus `do-while`.

## 2.2 Překlad zdrojových kódů a nástrojová podpora

Tato podkapitola popisuje, ze kterých částí se skládá překladač a co přesně dělají. Dále jsou pak představeny nejpopulárnější překladače pro jazyk C a existující nástroje umožňující měření pokrytí.

### Architektura překladače

V této podkapitole je lehce naznačeno, jak funguje překladač, jaké jsou jeho fáze a detailněji je popsána syntaktická analýza, jejíž abstraktní syntaktický strom hraje klíčovou roli při tvorbě instrumentačního programu. Překladač samotný lze rozdělit na 3 hlavní části, a sice přední část, střední část a zadní část.

### Přední část

Přední část (angl. *front-end*) je v překladači úsek, který jako první přijde do styku se zdrojovým kódem. Pokud je překladač schopen překládat více jazyků, jako například GCC nebo Clang, je potřeba pro ně více front-endů, neboť ty jsou závislé na jazyku, kterým je program napsán. Mezi fáze v této části překladu patří lexikální analýza, syntaktická analýza, sémantická analýza a generování mezikódu.

1. **Preprocessing** – u některých jazyků, například C nebo C++ je nejprve prováděn tzv. *preprocessing*, což je fáze, která má za úkol dosadit makra, která jsou definovaná pomocí `#define`, do zdrojového textu, vkládat soubory, které jsou označeny direktivou `#include`, nebo podmíněně přidávat či odebírat kód specifikovaný v direktivách `#ifdef` a `#ifndef`. Tento výsledný soubor poté předává lexikálnímu analyzátoru [5].

2. **Lexikální analýza** – fáze, jejíž činností je kromě odstranění komentářů také převést zdrojový kód na sekvenci tokenů, čemuž se říká *tokenizace*. Tokenem se rozumí každý prvek, který má pro daný jazyk nějaký význam, tedy například identifikátory, klíčová slova, literály nebo symboly aritmetických operací [27, kap. 3.1]. Příklad fungování lexikálního analyzátoru se nachází na obrázku 2.5.

3. **Syntaktická analýza** – jedna z nejdůležitějších a nejnáročnějších fází překladu. Jejím úkolem je zjistit, zdali jsou tokeny poskládány za sebou tak, že odpovídají gramatice daného jazyka. Výstupem je abstraktní syntaktický strom (AST), což je struktura skládající se z uzlů a hran, kdy uzly značí jednotlivé tokeny programu [27, kap. 4.1, 6.1].

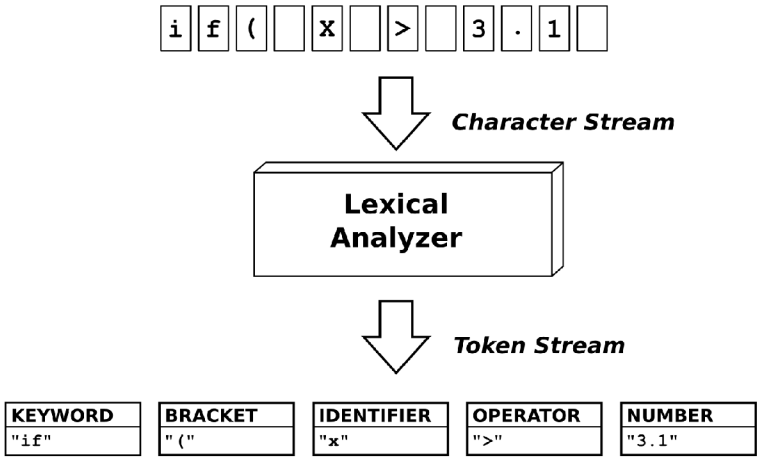
4. **Sémantická analýza** – fáze následující po syntaktické analýze, která má za úkol zjistit, zda má program smysl z hlediska významového. Stará se o to, aby proměnná měla správný typ, pokud s ní provádíme nějaké operace. Rovněž se stará o to, zdali nebyly náhodou použity nedefinované či neinicializované proměnné nebo nedefinované funkce [27, kap. 7].

5. **Generování mezikódu** – aby mohl překladač program lépe optimalizovat, nevytváří hned kód pro cílovou architekturu, nýbrž předtím generuje mezikód. Tento kód svou strukturou připomíná jazyk symbolických adres a je nezávislý na jazyku, v němž je napsán zdrojový text. Nad tímto kódem poté probíhají různé optimalizace, které mají daný program vylepšit, ať už z hlediska rychlosti nebo paměťového zatížení [13]. Ukázka mezikódu se poté nachází na obrázku 2.7.

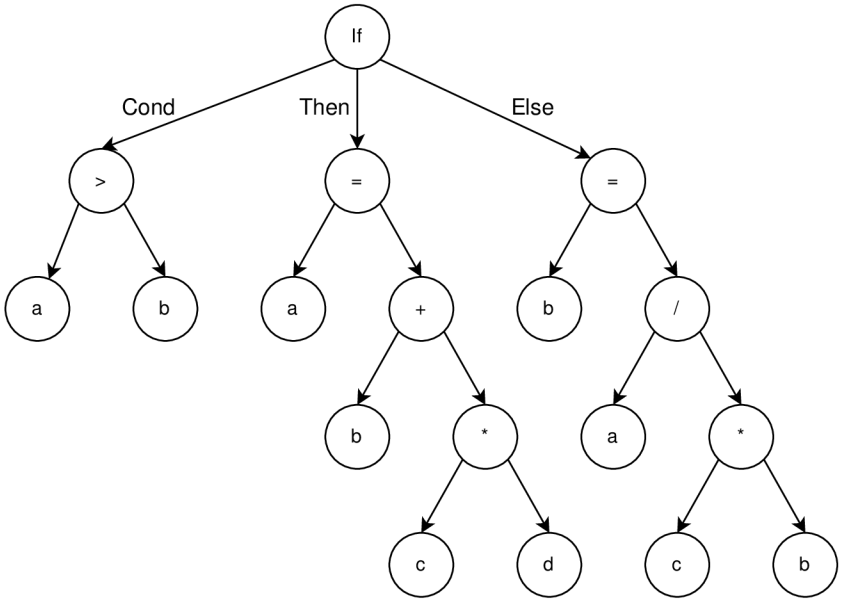
```
if (a > b) {
    a = b + c * d;
}
else {
    b = c * b / a;
}
```

Výpis 2.1: Demonstrační kód, na němž je ukázána tvorba AST.

Abstraktní syntaktický strom kódu 2.1 je zobrazen na obrázku 2.6.



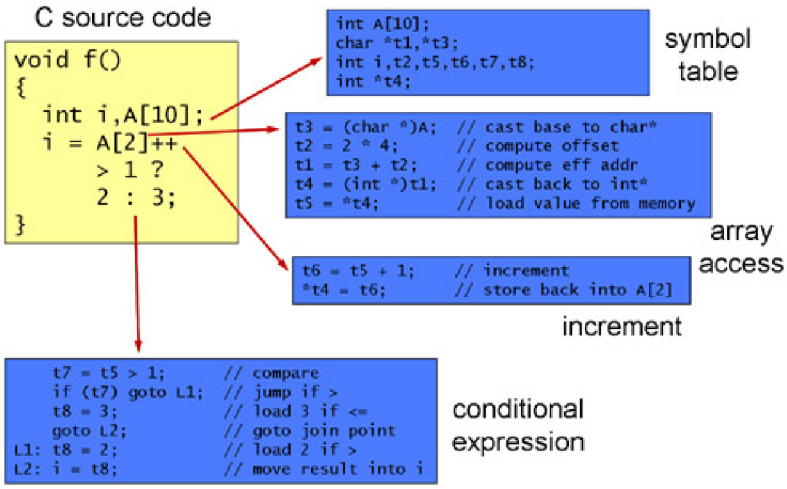

Obrázek 2.5: Proces lexikální analýzy, převzato z [23].



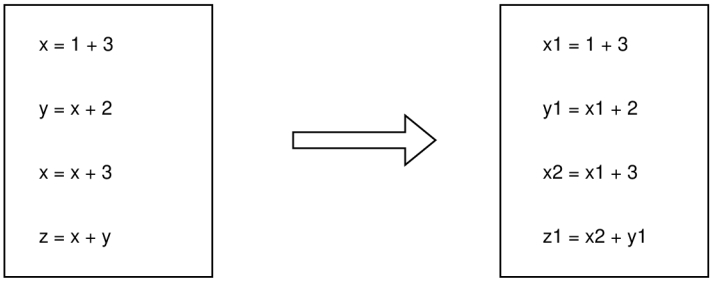

Obrázek 2.6: Abstraktní syntaktický strom pro kód 2.1. Je možné si povšimnout, že v těle podmínky pravá strana přiřazení začíná uzlem +. Důvodem je priorita početních operací. Procesor pojem priorita nezná a je tedy na překladači, aby strom sestavil tak, aby se v dalších fázích vygenerovaly instrukce, které zajistí správný výsledek.

### Střední část

Střední část, neboli *middle-end*, je část překladače zabývající se zejména optimalizací mezikódu a jeho přípravou na generování kódu pro cílovou architekturu. Forma kódu, která se často při optimalizaci používá, je tzv. *Static Single Assignment* (SSA), jejíž hlavní myšlenkou je pro každý zápis do proměnné provést zápis do nově vytvořené proměnné, aby tím bylo vyjádřeno, že se v proměnné nachází nová hodnota. Pomocí této techniky je poté možné provádět celou řadu optimalizací jako například vyčíslení výrazů s konstantní hodnotou, případně odstranění mrtvého kódu, tedy kódu, který se v programu nikdy neprovede [8]. Na obrázku 2.8 se nachází ukázka takového kódu.




Obrázek 2.7: Ukázka mezikódu. Převzato z [18].




Obrázek 2.8: Ukázka SSA.

### Zadní část

Účelem zadní části je vygenerovat z mezikódu kód pro architekturu, na které je program překládán. Musí rozhodnout, jaké instrukce použít a v jakém pořadí by za sebou měly jít. Zároveň provádí dodatečné optimalizace tak, aby byl kód pro architekturu co nejefektivnější [14].

## GCC

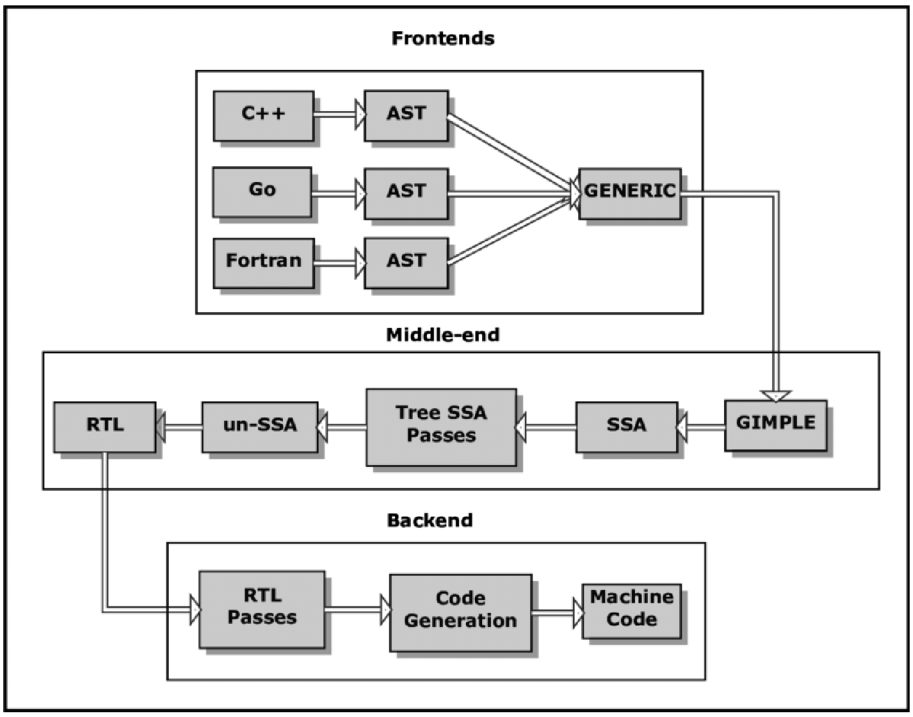
GCC je zkratka pro `GNU Compiler Collection` a jedná se o multiplatformní překladač, který dokáže přeložit programy v jazycích C, C++, Objective-C, Fortran, Ada, Go a D [6].

## Clang

Jedná se o další z překladačů, nicméně samotný Clang referuje pouze na přední část překladu, tedy lexikální a syntaktickou analýzu. Mezi jazyky, které Clang podporuje, patří C, C++, Objective C, OpenGL a Cuda [1].

## LLVM

Úzce souvisí s Clangem, ale je to soubor nástrojů umožňující práci s různými jazyky. LLVM se primárně zabývá střední a zadní částí překladu. Veškeré jazyky jsou převedeny do stejného




Obrázek 2.9: Schéma architektury překladače GCC, převzato z [29].

mezikódu, což poté velmi zjednodušuje práci. Tento jazyk je poté pomocí několika průchodů optimalizován a poté dochází ke generování kódu pro cílovou architekturu [3]. Za pomocí této infrastruktury již byl vytvořen program `tforc`[2], který dokáže instrumentovat přístupy do paměti a volání funkcí. Nicméně kvůli změnám verzí LLVM neprodukuje tento program nadále spolehlivé výsledky.

### Existující nástroje pro měření pokrytí

Tato podkapitola se zabývá nástroji, které jsou přímo specializované na měření pokrytí. Na trhu se nachází mnoho nástrojů, které tuto činnost dělají. Pro tuto práci byl vybrán nástroj Gcov, který open-source, a komerční BullseyeCoverage.

#### Gcov

Jedním z nejznámějších nástrojů pro měření pokrytí kódu je bezesporu *gcov*. Používá se především na měření pokrytí všech řádků kódu.

Nástroj funguje tak, že za překladu identifikuje základní bloky a do nich vloží volání svých funkcí, které způsobí zvýšení čítače průchodu. Jakmile instrumentovaný program projde základním blokem, čítač se zvýší [17]. Podle toho, které čítače zůstaly po konci běhu programu nulové, lze určit, jaké je pokrytí všech řádků kódu, případně větví.

Mezi jeho další výhody patří i instrumentace samotných funkcí, tudíž je možno zjistit, které funkce nebyly volané nebo také kterými větvemi program neprošel. Další výhodou je také to, že je zdarma. Nevýhodou budiž fakt, že tento nástroj úzce spolupracuje s GCC,

[2] https://pajda.fit.vutbr.cz/testos/tforc

takže ho není možné využít s jinými překladači. Rovněž gcov neumí instrumentovat čtení či zápisy do paměti [2].

#### BullseyeCoverage

Tento nástroj se zabývá nejen měřením pokrytí z hlediska toho, které funkce byly, nebo nebyly zavolány, ale zabývá i tzv. *Condition/Decision coverage*, tedy zdali se prošlo všemi větvemi všech řídicích konstrukcí. Mimo to také umí instrumentovat kód a lze specifikovat, který kód se má instrumentovat a který ne. Dále také vytváří zprávy o výsledcích pokrytí ve formátech HTML, XML a CSV. Zásadní nevýhoda je ta, že je jeho licence poměrně drahá. První nákup stojí 900 $ a každý další rok 200 $ [26].

## 2.3 Interní reprezentace AST v GCC

Tato interní struktura reprezentující abstraktní syntaktický strom každé funkce se nazývá GENERIC jehož účelem je jazykově nezávislá reprezentace tak, aby byl pro čitelný pro vývojáře a dalo se s ním dobře manipulovat. Každý strom je struktura typu `tree`. Jednotlivé uzly však zároveň mají i své vlastní jednoznačné označení (tzv. kód), aby se dalo určit, o jaký uzel se jedná, protože různé uzly slouží k různým účelům, mají odlišné vlastnosti a různě se s nimi pracuje [22, strana, 171]. Uzly také disponují svým typem, tedy informací o jejich datovém typu. Uzly reprezentující proměnné a přístupy do paměti jsou označeny typem, které mají u svých deklarací. Složitější uzly mají typy od proměnných odvozené. Například, jedná-li se o uzel přiřazení do proměnné celočíselného typu, tak uzel také bude mít celočíselný typ.

Mezi nejzákladnější datové typy uzlů patří:

- `integer_type_node` s variantami `unsigned`, `long`, `long long` a dalšími typy se specifickými bitovými šířkami,
- `float_type_node`,
- `double_type_node`,
- `ptr_type_node` – obecně ukazatel, například `integer_ptr_type_node` je ukazatel na celé číslo,
- `record_type` – použití u struktur,
- `function_type` – použití u funkcí [22, kap. 11.3].

V následujících částech jsou detailněji popsány uzly, které se vyskytují v programech nejčastěji. Součástí popisu každého uzlu bude kód, pod kterým v AST vystupuje, a případně seznam operandů, pokud uzel nějaké má.

#### Uzel rozsahu platnosti

Uzel s kódem `BIND_EXPR` reprezentuje nový rozsah platnosti. Typicky každá funkce (kromě funkce `main`) začíná tímto uzlem. Další výskyty lze nalézt například v těle konstrukce `if` nebo `for` nebo při umělém definování nového rozsahu platnosti pomocí složených závorek.

Tento uzel se skládá ze dvou částí. První z nich obsahuje seznam proměnných, které jsou v daném rozsahu platnosti deklarované. Druhou částí je poté podstrom, který do daného rozsahu platnosti spadá včetně vnořených rozsahů platnosti [22, kap. 11.7.2].

### Seznam příkazů

Uzel s kódem `STATEMENT_LIST` je dalším hojně využívaným, který sdružuje příkazy do dvousměrně vázaného seznamu, jenž přidává lepší možnosti průchodu, případně i přidání či odebrání jednotlivých prvků seznamu. Lze ho nalézt všude tam, kde sekvence sestává ze dvou a více příkazů. Jestliže je funkce nebo jakákoliv jiná konstrukce prázdná, tedy nenachází se v ní žádný kód, je tato prázdná část reprezentována právě prázdným seznamem příkazů [22, kap. 11.7.3].

### Deklarační a modifikační uzel

Uzel disponující kódem `DECL_EXPR` je uzel signalizující deklaraci proměnné. Opět se skládá ze dvou částí. První část obsahuje jméno deklarované proměnné. Druhá část může obsahovat buď výraz, který se má do dané proměnné přiřadit, jedná-li se i o definici, ale také nemusí obsahovat nic, pokud se jedná pouze o deklaraci [22, kap. 11.7.1].

Modifikační uzel s kódem `MODIFY_EXPR` se od toho deklaračního příliš neliší. Slouží nikoliv pro deklaraci, ale pro modifikaci proměnné. Opět má dva operandy. Prvním z nich je modifikovaná proměnná a druhým výraz, který se má do proměnné přiřadit. Prvním operandem nemusí být jen výraz či proměnná, ale může se také jednat o další modifikační uzel. S tímto se můžeme setkat nejčastěji při zřetězeném přiřazení [22, strana 192].

### Volání funkce

Uzel s kódem `CALL_EXPR` reprezentuje volání funkce. Jeho operandy jsou parametry seřazeny tak, jak se ve funkci nachází zleva doprava [22, strana 193].

### Přístupy do paměti

Jsou to zásadní uzly, neboť reprezentují místa v paměti, ze kterých se dá číst a do kterých se dá zapisovat. V této kategorii lze nalézt několik běžně používaných uzlů, a sice:

- `VAR_DECL` – uzel reprezentující použití proměnné,
- `PARM_DECL` – stejný jako použití proměnné, zde se však nejedná o proměnnou deklarovanou uvnitř dané funkce, ale je jí předán jako parametr,
- `COMPONENT_REF` – uzel představující přístup do struktury, prvním operandem je objekt (může se jednat o pole, nepřímou referenci[3] i další strukturu), druhým je vždy člen dané struktury,
- `INDIRECT_REF` – nepřímý přístup k proměnné, jeho jediným operandem je buď proměnná, ke které se nepřímo přistupuje, nebo místo v paměti, na které se lze dostat díky `POINTER_PLUS_EXPR` nebo `POINTER_DIFF_EXPR`, signalizující použití ukazatelové aritmetiky,
- `ARRAY_REF` – přístup do pole, prvním operandem může být jak proměnná, tak nepřímý přístup, tak i další přístup do pole, jedná-li se o vícerozměrné pole[4], druhý je vždy index a

[3] Zde ukázáno na obrázku 2.10.
[4] Ukázka stromu vícerozměrného pole se nachází na obrázku 2.11.

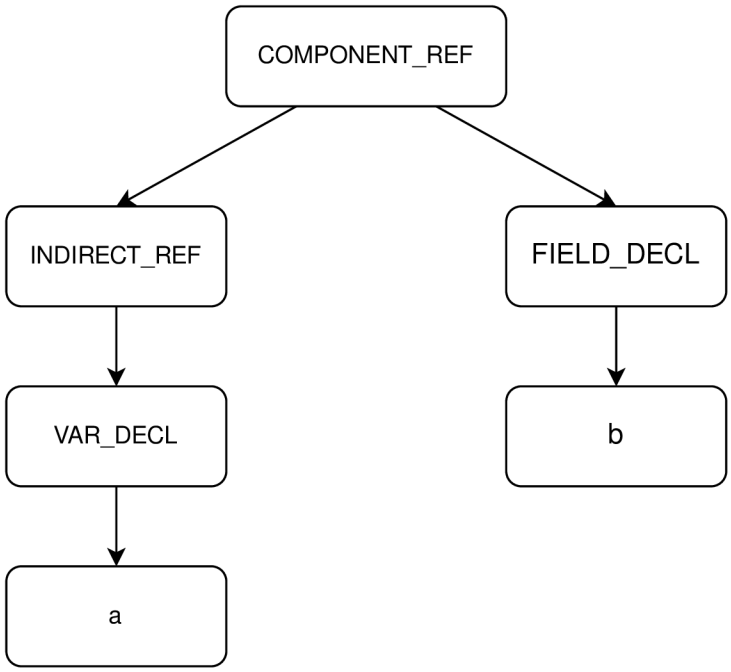
- `ADDR_EXPR` – reprezentuje adresu daného objektu (např. operátorem `&`), jediným operandem je objekt, jehož adresa má být získána [22, kap. 11.6.2].

### Konstruktor

Jedná se o typ uzlu, který lze použit pouze při deklaraci. Tento uzel reprezentuje deklaraci pole nebo struktur, tedy to, co se nachází uvnitř složených závorek, viz příklad níže [22, strana 193].

```
int arr[] = {1, 2, 3, 4, 5},
TStruct s = {.a = 1, .b = 2}.
```

Tyto uzly mohou v sobě být i vnořené, jedná-li se například o deklaraci vícerozměrného pole.



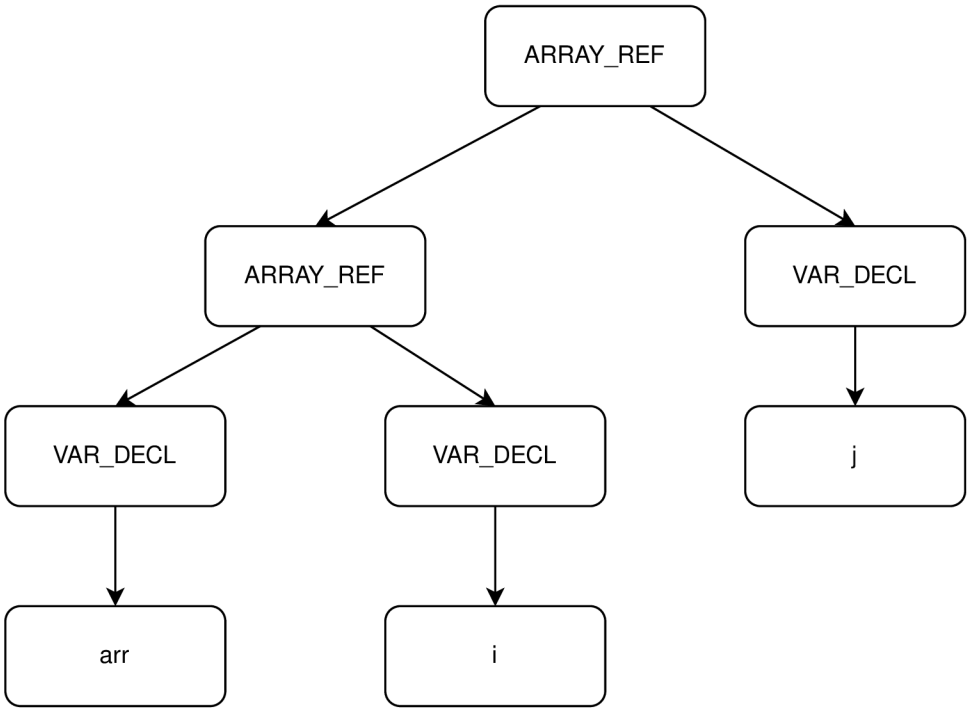

Obrázek 2.10: AST pro nepřímý přístup ke členu struktury `a->b`.

### Návratový uzel

Ten disponuje kódem `RETURN_EXPR` se ve srovnání s modifikačním uzlem liší pouze v tom, že reprezentuje návrat z funkce. Jestliže funkce skutečně něco vrací (není typu `void`), je jeho jediným operandem modifikační uzel, jehož prvním operandem je proměnná vygenerovaná překladačem a do druhého se přiřazuje návratová hodnota. Návratový uzel také nemusí mít žádný operand, pokud funkce nic nevrací a zároveň se v ní vyskytuje klíčové slovo `return` [22, strana 198].

### Podmínky a cykly

Podmínkový uzel s kódem `COND_EXPR` slouží k reprezentaci podmínky začínající klíčovým slovem `if`. Sestává ze 3 operandů. Prvním z nich je podmínka, která se má vyhodnotit, a ná-




Obrázek 2.11: AST pro přístup do dvourozměrného pole `arr[i][j]`.

sledující dva operandy poté reprezentují větev, která se provede, jestliže je podmínka pravdivá a větev, která se provede, je-li podmínka nepravdivá. Tato větev může být i prázdná, pokud takový blok neexistuje [22, strana 192].

Do příchodu GCC verze 10 byly tímto způsobem reprezentovány i cykly. V prvním operandu se vyskytovala podmínka cyklu a v následujících se poté ukrývaly skoky na část programu, jež se má provést v rámci cyklu, případně skok ven z cyklu.

Po příchodu verze 10 byly veškeré typy cyklů rozlišeny svým speciálním kódem. Cykly s kódy `WHILE_STMT` a `DO_STMT` reprezentující cykly `while`, respektive `do-while` mající jako první operand podmínku cyklu a druhý jeho tělo. Cyklus `for` s kódem `FOR_STMT` má operandy hned 4. První z nich uchovává inicializaci iterační proměnné, druhý podmínku, třetí co se má stát po vykonání těla cyklu (typicky inkrementace iterační proměnné) a čtvrtý samotné tělo cyklu [22, strana 217, 218].

### Složený uzel

Složený uzel využívající kód `COMPOUND_EXPR` slouží k reprezentaci operátoru čárka „,“. Ten funguje tak, že se nejprve vyhodnotí jeho první operand, jehož výsledek se zahodí, a poté se vyhodnotí jeho druhý operand, jehož typ a výsledek se použije dále [22, 25].

V jazyce C lze psát výrazy s operátorem čárka i s více operandy. Avšak uzel reprezentující tuto konstrukci má pouze dva operandy, tudíž dochází k převodu například z

```
(foo(), bar(), a = b + c)
```

na

```
((foo(), bar()), a = b + c).
```

V abstraktním syntaktickém stromě se tento uzel objeví pouze tehdy, obsahuje-li první operand konstrukci, která způsobuje nějaký vedlejší efekt, například volání funkce, přiřazení do proměnné, případně unární inkrementaci či dekrementaci. V opačném případě se v AST objeví pouze uzel druhého operandu, tedy jako by uživatel žádný výraz s operátorem čárka nenapsal.

### Aritmetické, logické a relační uzly

Tyto uzly lze ještě rozdělit na binární a unární, které se liší počtem operandů. Mezi zástupce binárních uzlů zde patří například `PLUS_EXPR`, `GT_EXPR` symbolizující operátor `>` nebo `BIT_AND_EXPR`. Z unárních uzlů lze zmínit například `NEGATE_EXPR` představující unární minus a `BIT_NOT_EXPR` pro bitovou negaci [22, kap. 11.6.3].

### Uzly unární inkrementace a dekrementace

Tyto uzly zastupují výrazy s operátory `++` a `--`. Jmenovitě se jedná o `PREINCREMENT_EXPR`, `POSTINCREMENT_EXPR`, `PREDECREMENT_EXPR` a `POSTDECREMENT_EXPR`, kde `PRE` a `POST` značí, zdali je operátor použitý před, nebo za daným objektem. Prvním operandem je tedy objekt, jehož hodnota se má změnit, druhým operandem je obvykle číslo 1 buď v podobě celého, nebo desetinného čísla [22, kap. 11.6.3].

### Konverzní uzly

Konverzní uzly zajišťují přetypování proměnných, ať už implicitní nebo explicitní. Jednou z kategorií jsou uzly `NOP_EXPR` a `CONV_EXPR`. Účelem `NOP_EXPR` je přetypování výrazu, které nevyžaduje další generování kódu. Příkladem budiž konverze z typu `char` na `int`. Uzel `CONV_EXPR` naopak zajišťuje přetypování, které může vyžadovat pozdější generování kódu. Například přetypování z typu `int` na `char`.

Další z uzlů, které zajišťují přetypování jsou `FLOAT_EXPR` a `FIX_TRUNK_EXPR`. První z nich slouží ke konverzi celého čísla na desetinné, druhý se stará o opak. Všechny konverzní uzly mají pouze jeden operand a to výraz, u nějž má dojít k přetypování [22, kap. 11.6.3].

# Kapitola 3

# Návrh nástroje pro instrumentaci programů za překladu

Tato kapitola se zabývá návrhem instrumentačního nástroje *ginstrum*. Nejprve následují požadavky na výsledný nástroj. Dále poté obsahuje popis zásuvného modulu (dále bude používáno slovo *plugin*) pro GCC, pomocí nějž je program realizován. Hlavní část je ovšem věnována návrhu instrumentací jednotlivých událostí.

## 3.1 Požadavky

Mezi hlavní požadavky programu patří instrumentace těchto událostí:

- **čtení z paměti** – signalizovat kdykoliv se čte z paměti a to nejen z proměnných, ale i z nepřímých přístupů,
- **zápis do paměti**,
- **volání funkce** – zahlásit, že se bude volat funkce těsně předtím, než se skutečně zavolá,
- **vstup do funkce** – oznámit, že se nacházíme ve volané funkci,
- **návrat z funkce** – co nejtěsněji před skutečným návratem oznámit, že se bude funkce opouštět,
- **pokračování po návratu z funkce** – oznámit, na kterém místě v kódu se dále pokračuje poté, co byla funkce zavolána a vykonána,
- **vstup do základního bloku** – ohlášení vstupu do základního bloku, které na rozdíl od výše uvedených událostí nemusí být úplně snadno rozpoznatelné.

Mezi další požadavky, které se neváží přímo k tomu, které události mají být instrumentovány, patří:

- **instrumentace pouze určitých typů událostí** – pomocí argumentů určit, které typy událostí mají být instrumentovány a které ne,
- **specifikace typu výpisu** – nabídnutí možnosti volby formátu výpisu a to jak formou prostého textu, tak i ve strukturovaném formátu JSON pro účely dalšího zpracování,

- **zachování původní funkcionality programu** – instrumentace jako taková může měnit chování programu, avšak cílem tohoto nástroje je instrumentovat program tak, aby se choval, jako by vůbec instrumentován nebyl a
- **kompatibilita napříč co nejvíce verzemi GCC** – verze GCC jsou samy o sobě velmi stabilní, avšak i ty se mezi sebou mohou trochu lišit, viz 2.3, úkolem tedy je, aby program fungoval napříč verzemi i navzdory těmto rozdílům.

Dalším požadavkem v zadání je vytvořit program pro počítání pokrytí pro kritéria zahrnující řádky kódu, rozhodovací logiku a datové toky. Avšak kvůli tomu, že programy mohou být vícevláknové, tak může být počítání pokrytí poměrně náročné, bylo tedy rozhodnuto, že nástroj kromě instrumentace vytvoří soubor obsahující informace, s jejichž pomocí a s pomocí instrumentačního nástroje je možné tato kritéria pokrytí měřit.

## 3.2 Přístup k provedení instrumentace

Původní myšlenkou bylo provádět instrumentaci těsně po fázi tzv. *gimplifikace*, tedy fázi, kdy dochází k přeměně abstraktního syntaktického stromu do mezikódu, tzv. GIMPLE a modifikací tohoto kódu docílit požadovaných výsledků. Nicméně po další analýze bylo zjištěno, že funkce pro tvorbu GIMPLE instrukcí jako parametry vyžadují i stromy daných prvků, tudíž by bylo potřeba projít i fází těsně po vytvoření AST, aby bylo možné získat korektní stromy funkcí sloužící k instrumentaci. Rovněž GIMPLE svojí strukturou připomíná spíše jazyk symbolických instrukcí a hledat v něm, které objekty se mají jak instrumentovat by bylo náročné. Naproti tomu abstraktní syntaktický strom se svojí strukturou více blíží původnímu zdrojového textu a je tak jednodušší identifikovat dané objekty a správně je modifikovat. Ovšem obrovskou výhodou GIMPLE je jazyková nezávislost, tedy nezáleží na tom, jestli se jedná o `C` nebo `C++` nebo jiný jazyk, mezikód bude vždy stejný.

Nejvhodnější a vlastně i jedinou možností realizace tohoto úkolu bylo instrumentovat kód pomocí tzv. zásuvného modulu, který je přímo pro úpravu překladu určen.

## 3.3 GCC Plugin

Koncepce zásuvných modulů pro překladač GCC je zde přítomná již od verze 4.5 a jedná se o programy psány v jazyce C++ umožňující vývojáři si upravit překlad podle vlastních potřeb, ať už je to pro přidávání nových průchodů, pro sběr statistik nebo modifikaci překládaného kódu. Plugin funguje tak, že si uživatel specifikuje, ve kterých fázích se má zapojit do práce, v daných fázích si převezme řízení od překladače, vykoná, co je potřeba, a poté předává řízení zpět překladači. Plugin je možno spustit i pro více fází.

Aby bylo možné s vývojem zásuvného modulu začít, je potřeba nainstalovat sadu nástrojů

`gcc-<gcc-verze>-plugin-dev`,

kde `gcc-verze` je hlavní verze překladače GCC, který je použit pro překlad na daném zařízení. Tato sada poskytne potřebné hlavičkové soubory pro vývoj. Důležitou věcí také je, aby byl plugin i program, na který bude plugin aplikován, přeložen stejnou verzí GCC, a aby i nástroje pro tvorbu zásuvných modulů byly ve stejné verzi, jinak může dojít k chybě v překladu, nebo k nečekanému chování výsledného projektu.

Při překladu lze zapnout funkcionalitu zásuvného modulu pomocí přepínače

`-fplugin=<cesta k zásuvnému modulu>`.

Stejně jako může uživatel upravit chování programu pomocí argumentů, může ho upravit i u zásuvného modulu. Argumenty jsou mu zadávány ve formátu

`-fplugin-arg-<název zásuvného modulu>-<klíč>[=<hodnota>]`.

Pakliže chce uživatel předat zásuvnému modulu více než 1 argument, stačí tento přepínač aplikovat znovu s jiným klíčem, případně hodnotou. Příklad překladu programu s využitím zásuvného modulu s argumentem by vypadal následovně:

```
gcc -fplugin=myplugin.so -fplugin-arg-myplugin-key myprogram.c.
```

## 3.4 Sledované parametry pro instrumentaci

Pro každou událost, kterou výsledný plugin sleduje, je potřeba zavolat funkci k tomu určenou. Každá událost však poskytuje různé informace, proto i funkce s nimi spjaté přijímají různé parametry. Tabulka 3.1 zobrazuje typy událostí a parametry, které daná událost sleduje.

| | Adresa | Hodnota | Typ | Funkce | Soubor | Řádek | Číslo základního bloku |
|---|---|---|---|---|---|---|---|
| Čtení z paměti | X | X | X | | X | X | |
| Zápis do paměti | X | X | X | | X | X | |
| Volání funkce | | | | X | X | X | |
| Vstup do funkce | | | | X | X | X | |
| Návrat z funkce | | | | X | X | X | |
| Pokračování po návratu z funkce | | | | X | X | X | |
| Vstup do základního bloku | | | | X | X | X | X |

Tabulka 3.1: Tabulka sledovaných parametrů k jednotlivým událostem.

U čtení a zápisu do paměti je potřeba znát adresu, která ačkoliv může být nečitelná, je jednoznačnou identifikací místa v paměti, ať už se program nachází ve kterékoliv funkci. Dále je důležité vědět, která hodnota se přečetla, případně zapsala. Typ proměnné zde nereprezentuje parametr, který by se sledoval, protože objekt nemůže měnit svůj typ za běhu, avšak slouží k tomu, aby instrumentační funkce pro čtení a zápis korektně vypisovaly hodnoty, ať už se jedná například o desetinná čísla nebo čísla zabírající více než standardní celé číslo. Konkrétně se jedná o

- `int`,
- `float`,
- `double`,
- `char`,

- `unsigned int`,
- `long int`,
- `long unsigned int`,
- `long long int` a
- `long unsigned int`.

Pro veškerou instrumentaci, která se týká funkcí, je zásadní vědět, o kterou funkci se jedná. Pro vstup do základního bloku se poté jedná o kombinaci jména funkce a indexu základního bloku, protože graf toku řízení se tvoří pro každou funkci zvlášť a samotný index by mohl být nejednoznačný. Pro každý typ instrumentace nás vždy zajímá název souboru, kde k dané událostí došlo, a také řádek, kde se událost stala.

## 3.5 Průchod funkcí

V hlavní fázi je program realizován jako jeden průchod AST. Po registraci této fáze se daná funkce spustí tolikrát, kolik definicí funkcí se nachází v daném zdrojovém textu. Případné deklarace funkcí a vlastních datových typů se ve stromě nenacházejí, i kdyby se vyskytovaly uvnitř nějaké funkce.

Pro spolehlivý průchod je nutné identifikovat, které uzly se mohou nacházet jak v těle funkce, tak i v těle dalších řídicích konstrukcí, např. podmínek a cyklů. Jedná se o následující uzly:

- `BIND_EXPR`,
- `CALL_EXPR`,
- `STATEMENT_LIST`,
- `DECL_EXPR`,
- `MODIFY_EXPR`,
- `COND_EXPR`,
- `SWITCH_EXPR`,
- `COMPOUND_EXPR`,
- `RETURN_EXPR`,
- `INDIRECT_REF`,
- unární inkrementace a dekrementace.

Mezi uzly patřící do této skupiny lze zařadit také `GOTO_EXPR` signalizující použití příkazu `goto` a `LABEL_EXPR` značící použití návěstí [22, strana 198], nicméně oba z nich nejsou pro instrumentaci jakékoliv z událostí zmíněných v podkapitole 3.1 důležité a lze je tedy přeskočit.

Později při vývoji bylo zjištěno, že součástí funkce může být i jakýkoliv výraz popsaný v podkapitole3.6, tyto výrazy sice většinou nemají v programu žádný význam, ale mohou

se v nich objevit i konstrukce s vedlejšími efekty, takže je potřeba i tyto výrazy projít a instrumentovat v nich přístupy do paměti, tak jako by tomu bylo při průchodu běžným výrazem.

Nevýhodou může být to, že pokud se tyto bezvýznamné výrazy v programu vyskytují, jsou pravděpodobně odstraněny překladačem. Jakmile se však přidají instrumentační funkce, tento kód odstraněn nebude. Obecně se však počítá s tím, že program tyto konstrukce bude obsahovat pouze minimálně, nebo vůbec.

## 3.6 Průchod výrazem

V tomto případě se jedná o uzly, které se mohou nacházet na pravé straně přiřazení. Některé uzly zde zmíněné se nacházejí i v podkapitole 3.5. Je to z toho důvodu, že tyto uzly mohou reprezentovat samotný příkaz a zároveň se mohou vyskytovat i ve výrazu. Je tomu tak u 5 posledních jmenovaných uzlů. Uzly, které se mohou nacházet ve výraze tedy jsou:

- binární a unární aritmetické operace,
- relační operace,
- bitové operace,
- unární inkrementace a dekrementace,
- konverzní uzly,
- `CONSTRUCTOR`,
- `CALL_EXPR`,
- `MODIFY_EXPR`,
- `COMPOUND_EXPR` a
- `COND_EXPR` reprezentující ternární operátor.

## 3.7 Návrh instrumentace všech typů událostí

Tato podkapitola se zabývá návrhem instrumentace všech typů událostí, které je potřeba monitorovat, a jsou popsané v 3.1. Zároveň se také zabývá návrhem `.covitems` souboru.

### Instrumentace čtení z paměti

Cílem této instrumentace je vložit funkci, které čtení z paměti signalizuje těsně před tím, než dojde ke skutečnému přečtení dané hodnoty[1].

Toho lze docílit pomocí složeného výrazu (v jazyce C operátor `,`), jehož prvním operandem je právě funkce, která se bude volat, a druhým operandem místo v paměti, z něhož se čte. Zápisem v jazyce C bychom dostali tento výsledek:

[1]Pro instrumentaci přístupů do paměti (čtení a zápis) je nutno dodat, že se jedná o instrumentaci L-hodnotových výrazů, tedy výrazů, které referují na místo v paměti, například proměnná, pole, struktura nebo nepřímý přístup do paměti přes ukazatel.

```
(cb()[2], a)
```

Nicméně, pokud by byl tento přístup použit při psaní klasického zdrojového textu, tak bychom pravděpodobně dostali jiné pořadí čtených proměnných, jelikož GCC všechny složené výrazy zabalí a jako 2. operand tohoto výrazu by byl původní výraz. Pro proměnné to nutně nemusí představovat problém, avšak horší by to bylo, kdyby se zde místo proměnných vyskytovaly funkce, které vracejí nějakou hodnotu. Poté by výpisy nemusely odpovídat tomu, co se v programu skutečně děje.

Kód nacházející se níže reprezentuje modifikaci proměnné, na jejíž pravé straně se nachází volání funkce[3]:

```
a = foo() * x + bar() * y
```

Ručně by byl zdrojový text instrumentován tak, aby se instrumentační volání nacházela co nejblíže objektům, které je potřeba sledovat:

```
a = (cb2(), foo()) * (cb1(), x) + (cb2(), bar()) * (cb1(), y)
```

Avšak GCC tento výraz přeskládá a výsledný výraz by poté vypadal takto:

```
a = ((cb2(), (cb1(), (cb2(), cb1())), foo * x + bar() * y)
```

Je patrné, že volání instrumentačních funkcí je přesunuto doleva, zatímco vše, co nesouvisí s instrumentací, vpravo. Chování programu jako takového by se nezměnilo, avšak díky tomuto jevu by docházelo k tomu, že by se nejprve zahlásilo veškeré události nacházející se v daném výrazu a až poté by přicházely výpisy z instrumentačních funkcí, které byly volány. Výpisy by sice korespondovaly s tím, co se v programu skutečně děje, nicméně by byly v nesprávném pořadí, a tedy neodpovídaly skutečnému toku programu a byly tedy i pro uživatele méně čitelné.

Pomocí zásuvného modulu tedy může dojít k instrumentaci přímo tam, odkud se z daného objektu čte, bez obav, že by překladač pořadí nějak změnil, neboť instrumentační nástroj operuje až ve fázi těsně po vytvoření AST. Stále ale nemusí být zaručeno, že budou výsledné události odpovídat pořadí, v jakém jsou objekty napsány ve zdrojovém textu například kvůli přeuspořádání výpočtu při optimalizaci.

To, co je pro čtení nejdůležitější sledovat, je hodnota, která je čtena, a adresa, ze které se čte. Bylo by možné použít i jméno, avšak ne všechny čtené objekty mohou být proměnné mající jméno. Někdy se může jednat i o přístup do paměti zprostředkován ukazateli. Pro vytvoření těchto parametrů stačí jen vytvořit uzel symbolizující adresu objektu, tedy `ADDR_EXPR`, což by pro proměnnou jako takovou bylo přímočaré řešení a výsledek by vypadal správně:

```
(cb(&a, a), a)[4]
```

Nejprve se zavolá instrumentační funkce, jejíž výsledek bude zahozen a při dalších výpočtech se využije právě proměnná vystupující jako druhý operand složeného výrazu. Tento přístup by šel využít pouze v případě, že by vícenásobné použití proměnné nemělo žádné další vedlejší efekty, což se například projeví u čtení dvourozměrného pole mající jako jeden index unární inkrementaci a volání funkce jako druhý:

[2] Pro volání instrumentační funkce bude použita zkratka `cb` (z angl. *callback*), případně varianty jako `cb1`, `cb2`, atd. pro odlišení různých instrumentačních funkcí.

[3] Kód je pouze demonstrační a je na něm ukázáno, jak by mohlo dojít k přeuspořádání instrumentačních volání, proto se zde nacházejí i funkce `foo` a `bar`, i když nereprezentují čtení z paměti.

[4] Tento přístup je v rámci nástroje skutečně použit, avšak pouze pro uzly `VAR_DECL` a `PARM_DECL`.

```
arr[i++][foo()].
```

Jestliže by byl použit stejný postup, instrumentace by proběhla takto:

```
cb(&(arr[i++][foo()]), arr[i++][foo()]), arr[i++][foo()].
```

Tento postup však nelze využít, jelikož je v něm na více místech použito jak volání funkce, tak unární inkrementace. Volání funkce samo o sobě nutně nemusí změnit stav programu, avšak unární inkrementace ho změní zcela jistě. Takže například pro pole nebo nepřímý přístup k proměnné nelze tento způsob použít, protože uvnitř se může nacházet libovolné množství těchto konstrukcí, které mohou změnit chování programu, což žádoucí.

Jako nejlepší řešení se ukázalo použití pomocných proměnných, jejichž popis tvorby se nachází v podkapitole 3.7. Postup pro instrumentaci `arr[i++]` by byl následující:

1. instrumentovat přístupy do paměti nacházející se v indexu pole v indexu pole:

   ```
   arr[(cb()[5], i++)].
   ```

2. vytvořit novou proměnnou typu ukazatel a přiřadit do ní adresu právě instrumentovaného přístupu do pole:

   ```
   __temp = &(arr[(cb(), i++)]).
   ```

3. přidat funkci pro instrumentaci čtení, jejímiž argumenty jsou hodnota ukazatele samotného a jeho dereference, tedy hodnota pole na daném indexu:

   ```
   __temp = &(arr[(cb(), i++)], cb(__temp, *__temp)).
   ```

4. přidat dereferenci ukazatele `__temp`. Tato hodnota je skutečným výstupem celého výrazu a je použita dále v programu:

   ```
   (__temp = &(arr[(cb(), i++)], cb(__temp, *__temp)), *__temp).
   ```

### Instrumentace čtení uvnitř adresy

Výstupem po použití operátoru `&` není hodnota objektu, nýbrž adresa, tudíž instrumentace zde není instrumentace nutná. Nicméně stále je potřeba vědět to, jakou hodnotu v sobě ukrývá místo v paměti, jehož adresu je potřeba získat. Co se nabízí, je instrumentovat přímo uvnitř operátoru `&`, avšak na úrovni zdrojového textu toto není možné. Postup, jakým lze instrumentaci uvnitř adresy řešit, lze ukázat na výrazu `&arr[i]` opět s využitím pomocné proměnné.

1. instrumentovat přístup do paměti v rámci indexi pole:

   ```
   &(arr[(cb(), i)]).
   ```

2. do pomocné proměnné přiřadit tento instrumentovaný výraz:

[5] Zde je pouze demonstračně použit tento zápis pro zjednodušení. Instrumentace unární inkrementace a dekrementace se nachází v podkapitole 3.7.

```
__temp = &(arr[(cb(), i)]).
```

3. za tento výraz přidat funkci instrumentující čtení. Jejími parametry je pomocná proměnná a dereference na ni:

```
(__temp = &(arr[(cb(), i)]), cb(__temp, *__temp)).
```

4. za tento výraz poté přidat pomocnou proměnnou, aby bylo možno adresu předat dále programu:

```
(__temp = &(arr[(cb(), i)]), cb(__temp, *__temp), __temp).
```

Řešení se skoro neliší od klasického čtení, avšak zásadní rozdíl se skrývá úplně na konci, kde místo dereference na pomocnou proměnnou používáme ji samotnou. Je to z toho důvodu, že se dále musí pracovat s adresou původního výrazu a nikoliv s jeho hodnotou.

### Instrumentace zápisu do paměti

Pro instrumentaci zápisu do paměti je na rozdíl od čtení nutné přidat funkci až za příkaz zápisu, aby bylo možné zjistit, co se skutečně zapsalo. Zároveň se musí přiřazení zkopírovat i do jiné proměnné tak, aby její výsledek mohl být použitý dále v programu, například, dojde-li k přiřazení do proměnné v rámci podmínky.

#### Tvorba globální proměnné

Aby se korektně instrumentoval nejen zápis, je potřeba přiřadit výsledný výraz do jiné proměnné. Není však možné použít jakoukoliv proměnnou ve funkci mající daný typ, protože tím by se mohlo změnit chování programu. Řešením je vytvořit novou proměnnou, do které by se výsledek zapsal. Možnostmi jsou vytvoření nové lokální, nebo globální proměnné. Z hlediska implementačního se řešení tvorby nové globální proměnné jeví jako jednodušší.

Ty mají název ve formátu `__temp<x>`, kde $x$ je celé číslo, které se zvyšuje s každým novým vygenerováním. Proměnné navíc musí být statické, aby byla zajištěna viditelnost pouze v modulu, v němž jsou deklarované a nedocházelo ke konfliktům[6].

Tento přístup je poměrně jednoduchý na implementaci, avšak nevýhoda tkví v tom, že se pro každou událost vyžadující pomocnou proměnnou musí vytvořit nová proměnná, která je použita pouze jednou, tudíž program samotný zabírá více paměti. Pro ilustraci, pro program mající asi 950 řádků bylo potřeba vytvořit 513 nových globálních proměnných[7].

Původní přístup k instrumentaci zápisu byl poměrně přímočarý, a sice, že pro příklad `i = 2` by se nejprve provedlo přiřazení do pomocné proměnné, následně se zavolala funkce s adresou a hodnotou a pomocná proměnná by se použila i na konci výrazu pro další výpočty:

```
(i = __temp = 2, cb1(&i, __temp), __temp)
```

Tento přístup by opět fungoval jen pro objekty, které nezpůsobují vedlejší efekty. Ve strukturách, dereferencích a polích se lze s těmito výrazy setkat.

Pro skutečně univerzální postup instrumentace opět přicházejí do hry ukazatele a zde je ukázán na `i = 2`.

[6] Kód pro vytvoření globální proměnné byl převzat z [11, strana 19, 21].

[7] Bylo vyzkoušeno pouze na jediném projektu tohoto rozsahu, nicméně i pro projekty poloviční rozsahu byly potřeba nízké stovky těchto proměnných.

1. vytvořit pomocnou proměnnou typu ukazatel a přiřadit do ní adresu prvního operandu modifikačního uzlu:

   ```
   __temp = &i.
   ```

2. za tento výraz přidat nový podobný tomu původnímu s tím rozdílem, že prvním operandem není proměnná `i`, ale dereference nově vytvořené pomocné proměnné:

   ```
   (__temp = &i, *__temp = 2).
   ```

3. přidat funkci určenou pro instrumentaci zápisu, jejíž hlavními argumenty jsou adresa původní proměnné (hodnota v ukazateli) a hodnota samotná (jeho dereference):

   ```
   (__temp = &i, *__temp = 2, cb1(__temp, *__temp)).
   ```

4. doplnit dereferenci na konec celého nově vzniklého výrazu, aby bylo možné výsledek přiřazení aplikovat dále, je-li modifikace samotná součástí nějakého komplexnějšího výrazu:

   ```
   (__temp = &i, *__temp = 2, cb1(__temp, *__temp), *__temp).
   ```

Jediné omezení tohoto přístupu je, že pomocí něj nelze instrumentovat zápis do členů struktur nebo prvků pole, pokud se nacházejí v konstruktoru, neboť to nelze ani na úrovni zdrojového kódu. Řešení tohoto problému jsem bohužel nenalezl, a proto instrumentační nástroj tuto funkcionalitu postrádá.

### Instrumentace deklarace

Technika uvedená výše se však dá použít pouze v případě modifikace proměnné. Proměnná již dříve musela být někde deklarována. Pro deklaraci proměnné s neprázdným druhým operandem (jedná se tedy o deklaraci a definici zároveň) je zapotřebí využít trochu jiného přístupu, neboť se na úrovni zdrojového nesmí deklarace nacházet uvnitř složeného výrazu.

Řešením je rozdělit deklaraci na dva příkazy, a to na deklaraci samotnou a na modifikaci proměnné. Pokud však deklarace nemá žádný druhý operand, není potřeba nic instrumentovat, protože se proměnné nepřiřazuje žádná hodnota.

Takže například pro `int i = 2;` vypadá upravený výraz následovně:

```
int i;
i = 2;
```

Nejprve se oddělí druhý operand od původní deklarace, ten se dosadí jako druhý operand do nově vytvořené modifikace proměnné a zde již není problém zápis instrumentovat.

### Instrumentace zřetězeného přiřazení

Jedná se o zápis ve formátu

```
a = b = c = d.
```

Ze zápisu je patrné, že některé z proměnných ve výrazu vystupují jak na levé, tak pravé straně, tedy je z nich čteno a je do nich zároveň i zapisováno. Vyhodnocení poté probíhá zprava doleva následovně:

1. `c = d`,
2. `b = c`,
3. `a = b`.

Tuto ne úplně běžnou konstrukci lze v AST detekovat tak, že druhým operandem modifikace proměnné je opět modifikace proměnné. Aby bylo dosaženo správné instrumentace, je nutné tuto konstrukci poněkud upravit, jelikož v tomto stavu to není možné právě proto, že je do některých proměnných zároveň zapisováno a zároveň je z nich čteno. Řešením je konstrukci rozdělit na několik menších výrazů, které již instrumentovat lze. Důležité je však dodržet pořadí vyhodnocení příkazu a začít přiřazením `c = d` a postupně se dopracovat až k `a = b`. Výsledný výraz poté vypadá následovně:

```
(c = d, b = c, a = b).
```

Po této upravě je již možné instrumentovat čtení i zápis jako u klasické modifikace proměnné.

### Unární inkrementace a dekrementace

Instrumentace těchto výrazů úzce souvisí jak se čtením, tak i zápisem, neboť tyto výrazy provádějí obě operace najednou, a jejich instrumentace je poněkud odlišná od instrumentace klasického čtení a zápisu, proto je jim věnovaná samostatná podkapitola.

Zde je však instrumentace o něco složitější, protože je potřeba instrumentovat jednak čtení původní hodnoty, ale také zápis nové hodnoty. Tato hodnota se však nikde ve stromě nenachází, proto jedinou možností zůstává vytvořit aritmetický výraz, jenž donutí předání změněné hodnoty instrumentační funkci.

Níže je naznačen postup instrumentace pro jednoduchý výraz `i++`.

1. do pomocné proměnné přiřadit adresu objektu, který se má inkrementovat:

   ```
   __temp = &i.
   ```

2. vytvořit volání pro instrumentaci čtení, neboť tato hodnota se skutečně přečte a přidat ho za předcházející výraz:

   ```
   (__temp = &i, cb(__temp, __temp)).
   ```

3. vytvořit nový modifikační uzel, na jehož levé straně se nachází 2. pomocná proměnná a na pravé straně původní unární inkrementace, jen je proměnná nahrazena dereferencí na 1. pomocnou proměnnou:

   ```
   (__temp = &i, cb(__temp, __temp), __temp2 = *__temp++).
   ```

4. vytvořit instrumentační funkci pro zápis, jejímž parametrem je kromě adresy, kam se zapisovalo také hodnota výrazu zvýšená o 1:

   ```
   (__temp = &i, cb(__temp, __temp), __temp2 = *__temp++,
       cb1(__temp, (__temp3 = __temp2 + 1, __temp3))).
   ```

5. za celý výraz přidat hodnotu objektu, která bude použita pro další výpočty:

```
(__temp = &i, cb(__temp, __temp), __temp2 = *__temp++,
cb1(__temp, (__temp3 = __temp2 + 1, __temp3)), __temp2).
```

Tento postup je již poněkud komplikovanější a výsledný výraz docela nepřehledný. Zároveň také vyžaduje 3 pomocné proměnné.

Co může být zvláštní je, proč se v kroku 4 používá přiřazení do nové proměnné v rámci volání `cb1`. Je to z toho důvodu, že funkce, které se volají při instrumentaci čtení a zápisu, vyžadují adresy proměnných jako parametry tak, aby uvnitř těchto funkcí mohlo dojít ke správnému přetypování a výpisu hodnoty, pokud by se jednalo o jiný než celočíselný typ. Z výrazu `__temp2 + 1` adresa získat nelze, proto je potřeba vytvořit novou proměnnou, která již adresu vlastní. Instrumentace probíhá úplně stejně i pro ostatní typy unární inkrementace a dekrementace.

### Instrumentace volání funkcí

Instrumentace volání funkcí je velmi podobné původnímu přístupu instrumentace čtení z paměti s tím rozdílem, že zde je potřeba rozlišit, kde v programu se volání nachází. Prvním případem je, když funkce není použita v žádném výrazu, tedy nic nevrací, nebo vrací, ale tato hodnota není dále využita. Zde je postup instrumentace stejný jako původní přístup u instrumentace čtení z paměti, jen zde dochází k volání jiné instrumentační funkce.

Ve druhém případě funkce vrací hodnotu, která je později někde použita. Tedy volání funkce je součástí nějakého výrazu. Zde je potřeba opět volat instrumentační funkci těsně před voláním dané funkce. Zároveň je však nutné do nově vygenerované proměnné přiřadit výsledek funkce a proměnnou s výsledkem použít při dalších výpočtech. Tím nejdůležitějším však zůstává to, aby funkce byla volána pouze jednou, protože kdyby docházelo k vícerému volání, mohlo by se změnit chování programu.

Pro funkci nevracející hodnotu vypadá instrumentace následovně:

```
(cb(), foo()).
```

Tento přístup je stejný jako původní přístup instrumentace čtení z proměnné s tím rozdílem, že tady lze použít vždy, neboť do instrumentační funkce se předává pouze její jméno, nikoliv i její výstup, takže se zavolá pouze jednou a nenaruší se chování programu.

Instrumentace funkce vracející nějakou hodnotu je obdobná, s tím rozdílem, že se musí výsledek volání přiřadit do pomocné proměnné, jejíž typ odpovídá návratovému typu funkce:

```
(cb(), __temp = foo(), __temp).
```

Instrumentace parametrů funkce pak probíhá stejně jako instrumentace čtení z paměti.

### Instrumentace návratu z funkce

Tato podkapitola se zabývá pouze instrumentací návratu, když při průchodu stromem detekováno klíčové slovo `return`. Jak probíha instrumentace návratu z funkce, dorazí-li program na konec definice funkce a `return` se zde nenachází, se lze dočíst v podkapitole 4.2.

Při této části instrumentace je důležité využít poznatky z podkapitoly 2.3, kde je řečeno, jakým způsobem může funkce vrátit hodnotu s klíčovým slovem `return`, tedy buď s nějakou hodnotou, nebo bez, jedná-li se o funkci typu `void`.

Poměrně jednoduchou myšlenkou by bylo před `return` příkazem přidat volání funkce, které by říkalo, že dojde k návratu z funkce:

```
cb();
return x;.
```

Avšak problém by byl v tom, že funkce může vracet hodnoty, jejichž čtení jsou instrumentována, a může také vracet výsledky jiných funkcí, tedy výpisy by byly opět v jiném pořadí a neodpovídaly by skutečnému toku programu. Cílem tedy je signalizovat návrat co nejpozději je to možné. Jestliže se však jedná o `return` bez operandu, tak se v něm nemůže nacházet nic dalšího, co je třeba instrumentovat, a pak je řešení, kdy se volání přidá před klíčové slovo `return`, dokonce jediné možné.

Pro případy, kdy funkce vrací nějakou hodnotu se chováme velmi podobně jako je tomu u instrumentace zápisu do paměti. Předtím je však podstatné říci, jak takový `return` vypadá v AST. Tedy například pro `return y;` vypadá návratový uzel takto:

```
D.xxxx = y;,
```

kde `D.xxxx` je proměnná vytvořena překladačem a má speciální kód `RESULT_DECL`. Za `x` lze poté dosadit libovolné celé číslo. Pokud by došlo k instrumentaci stejně jako při zápisu, tak by došlo k chybě přístupu do paměti během překladu. Takže je nutné, aby se s návratovou proměnnou nijak nehýbalo. Postup při instrumentaci návratu s hodnotou je tedy následující.

1. Instrumentovat čtení nacházející se na pravé straně modifikace:

   ```
   D.xxxx = (cb(), y).
   ```

2. Vytvořit novou proměnnou, do které se přiřadí celá pravá strana návratu, která se dále naváže do stávajícího výrazu:

   ```
   D.xxxx = (__temp = (cb(), y)).
   ```

3. Přidat instrumentační funkci návratu do stávajícího výrazu:

   ```
   D.xxxx = (__temp = (cb(), y), cb2()).
   ```

4. Přidat proměnnou na konec výrazu, aby se do návratové proměnné přiřadila naše nová proměnná nesoucí výsledek, který se má vrátit:

   ```
   D.xxxx = (__temp = (cb(), y), cb2(), __temp).
   ```

### Instrumentace pokračování po volání funkce

Může se zdát, že když je instrumentován návrat z funkce, tak je zřejmé, kde v kódu se pokračuje, což ale není pravda. V jazyce C totiž existují funkce `setjmp` a `longjmp`, s jejichž pomocí lze v rámci programu skočit kamkoliv nejen v rámci funkce, ale i napříč funkcemi. Tedy, kdyby program zavolal funkci, v rámci níž by se skočilo někam jinam, kde by se znovu volala ta samá funkce, změnilo by se i místo, kde bude program dále pokračovat, jakmile dojde k návratu z dané funkce. Postup pro instrumentaci tohoto typu události není nijak složitý, za volání funkce je pouze přidáno volání instrumentační funkce:

```
(foo(), cb()).
```

### Instrumentace vstupu do funkce

Tuto událost je potřeba instrumentovat z toho důvodu, že kdyby bylo instrumentováno pouze volání funkce, tak lze pouze zjistit, že došlo k jejímu zavolání. Nicméně to ještě neznamená, že se skutečně volá daná funkce.

Může se stát, že je z různých důvodů potřeba upravit například nějakou knihovní funkci, která ale bude dělat něco navíc a bude pojmenovaná stejně. Pokud by tedy fungovala pouze instrumentace volání, nebylo by jasné, zdali se volá knihovní funkce, nebo nově implementovaná. Díky instrumentaci vstupu do funkce to lze snadno poznat, jelikož u knihovních funkcí k instrumentaci nedochází.

Toto je první z událostí, jejíž instrumentace probíhá až ve fázi, kdy je dokončen graf toku řízení. Samotná detekce začátku funkce není složitá a původně byl tento typ instrumentace realizován ve fázi AST. Bylo však zjištěno, že tento způsob mění řádek, na kterém začíná první základní blok funkce, který by měl začínat řádkem první instrukce, ale začíná tam, kde začíná definice funkce. To znamená, že by se skutečný začátek základního bloku mohl o pár řádků lišit, což samo o sobě nepředstavuje zásadnější problém, nicméně je dobré, když řádek základního bloku koresponduje s řádkem první instrukce (prvního příkazu) ve funkci. Toho by šlo dosáhnout i v rámci fáze AST, avšak ve fázi CFG je implementace o poznání jednodušší.

### Instrumentace vstupu do základního bloku

Tento typ instrumentace je podobně jako vstup do funkce instrumentován až po fázi AST. Důvodem je, že v této fázi by byla instrumentace poněkud složitější a rozhodně ne spolehlivá. Z AST se sice dají rozpoznat jednotlivé konstrukce, avšak situace se komplikuje, když do hry přijde ternární operátor nacházející se v nějakém výrazu. Navíc CFG, které generuje překladač, nemusí nutně odpovídat definici CFG popsané v podkapitole 2.1, protože si vytváří i základní bloky, které jsou navíc.

Jedním z rysů fáze CFG je, že se lze dostat k základním blokům dané funkce, ale také ke GIMPLE instrukcím, které se k danému bloku vážou. To umožňuje na začátek každého bloku vložit funkci signalizující vstup do základního bloku i s parametrem indexu daného bloku a názvu funkce, protože CFG se skládají pro každou funkci zvlášť. Tímto lze zajistit, že se funkce signalizující vstup do základního stane první instrukcí, která se v rámci daného základního bloku provede. Jestliže však dojde k situaci, kde nastane, že je potřeba instrumentovat vstup do bloku a zároveň vstup do funkce, je volání signalizující vstup do funkce před funkcí signalizující vstup do základního bloku.

### Pozice událostí v kódu

To, že je vidět, která událost se instrumentuje, případně v jaké funkci, je dobrý začátek, ale při větším programu může být počet výpisů tak vysoký, že při jejich zpětném procházení nemusí být úplně jasné, kde k události v programu došlo. Z tohoto důvodu instrumentační funkce přijímají i parametry obsahující název souboru a číslo řádku.

```
<mult_expr 0x7f80f7af2708
   type <integer_type 0x7f80f79355e8 int public SI
       size <integer_cst 0x7f80f791cee8 constant 32>
       unit-size <integer_cst 0x7f80f791cf00 constant 4>
       align:32 warn_if_not_align:0 symtab:0 alias-set -1 canonical-type
```

```
    0x7f80f79355e8 precision:32 min <integer_cst 0x7f80f791cea0
    -2147483648> max <integer_cst 0x7f80f791ceb8 2147483647>
    pointer_to_this <pointer_type 0x7f80f793d9d8>>

arg:0 <var_decl 0x7f80f868eea0 x type <integer_type 0x7f80f79355e8 int>
    addressable used read SI prog.c:17:9 size <integer_cst
    0x7f80f791cee8 32> unit-size <integer_cst 0x7f80f791cf00 4>
    align:32 warn_if_not_align:0 context
    <function_decl 0x7f80f7af5000 main>
    chain <var_decl 0x7f80f868ef30 x type <integer_type 0x7f80f79355e8
    int> addressable used read SI prog.c:17:16
    size <integer_cst 0x7f80f791cee8 32> unit-size <integer_cst
    0x7f80f791cf00 4> align:32 warn_if_not_align:0 context
    <function_decl 0x7f80f7af5000 main> chain
    <var_decl 0x7f80f7af6000 c>>>
arg:1 <var_decl 0x7f80f868ef30 y>
prog.c:18:11 start: prog.c:18:9 finish: prog.c:18:13>
```

Výpis 3.1: Textová vizualizace stromu pro `x * y`.

Ve výpisu 3.1 lze kromě toho, jak vypadá AST výrazu po použití interní funkce `debug_tree`, která je pro ladicí účely velmi užitečná, vidět i informace o lokaci nacházející se na posledním řádku. Je zde možné vidět název souboru, což je `prog.c`. Výraz poté začíná na řádku 18 a sloupci 11. Končí poté na řádku 18 a sloupci 13.

Lze si povšimnout, že i u proměnných samotných existuje jakési označení souboru, řádku a sloupce, avšak tyto informace slouží pro určení místa deklarace dané proměnné, nikoliv použití proměnné. Z toho vyplývá, že u výrazů, kde jsou použity proměnné nebo jiné objekty nelze zjistit jejich přesné místo použití, proto také nelze do instrumentační funkce předávat i sloupec, který by ještě zpřesnil místo, kde se daná událost nachází. Jelikož však proměnné neobsahují informace o jejich poloze, nelze také ani zjistit ve kterém souboru a řádku se nacházejí.

Jedinou možností, která zůstává, je dědit číslo řádku a soubor z nejbližšího uzlu vyšší úrovně, který již svou pozici v programu obsahuje. Je-li použit obrázek 3.1, tak pro proměnnou `x` deklarovanou na řádku 17 lze přisoudit lokaci z nejbližšího uzlu vyšší úrovně, což je `x * y`, tedy `MULT_EXPR` nacházející se na řádku 18.

Omezení však tkví v tom, že je-li příkaz víceřádkový, může se číslo řádku kvůli dědění lokace lišit od skutečného řádku, kde k události došlo. Jestliže výraz `x * y` není zapsán takto v jednom řádku, ale je zapsán takto:

```
1: x
2: *
3: y,
```

je zděděné číslo řádku to, kde se nachází znak `*`, tedy řádek 2. Čtení z `x` tedy je o jeden řádek níže než ve skutečnosti a čtení z `y` o jeden řádek výše. Číslo řádku také nemusí odpovídat, nachází-li se na začátku základního bloku například deklarace neinicializované proměnné. V takovém případě může překladač v rámci optimalizace tento kus kódu modifikovat a začátkem se tak může stát další příkaz v sekvenci příkazů.

## 3.8 Návrh podkladů pro měření pokrytí

Tato část musí být opět vykonána v rámci CFG fáze a to kvůli tomu, že jsou zde opět potřeba informace o základních blocích, zejména pak o předcích a následovnících konkrétních bloků. Původním plánem bylo, že by se vytvoření tohoto souboru s podklady dělo v jediném překladu instrumentovaného programu společně s instrumentací. Avšak kvůli velkým zásahům, které se kvůli instrumentaci dějí v AST, to není reálné. Proto bylo rozhodnuto, že nad cílovým programem proběhnou 2 překlady.

V prvním dochází k úplnému vypnutí fáze AST a vypnutí instrumentace ve fázi CFG. Plugin tedy má za úkol pouze vytvořit soubor s podklady. Ve druhém poté dojde k vypnutí tvorby podkladů a plugin pouze instrumentuje daný program.

### Formát podkladů

Ke každému programu je vytvořen soubor `.covitems`, tedy například pro program `myprog.c` se vytvoří soubor `myprog.c.covitems`. Soubor je ve formátu JSON kvůli strojovému zpracování informací a obsahuje následující informace:

- **bb** – obsahuje číslo základního bloku, které si zvolí překladač,
- **function** – název funkce, ve které se blok nachází (více funkcí může mít základní bloky se stejnými indexy),
- **pred** – předchůdci, neboli základní bloky, jejichž hrana vede do základního bloku **bb**,
- **succ** – následovníci, neboli bloky, ke kterým je připojena hrana vycházející z bloku **bb**,
- **defs** – proměnné, do kterých bylo v bloku **zapsáno**,
- **uses** – proměnné, ze kterých bylo v **bb** čteno.

U `defs` a `uses` se počítá s tím, že zde budou všechna čtení i zápisy, takže se proměnné mohou opakovat, pokud v daném bloku nastanou tyto události vícekrát.

# Kapitola 4

# Implementace instrumentačního nástroje

Tato kapitola se zabývá implementací instrumentačního nástroje. Nejprve je demonstrováno, jak program samotný začíná a jak vypadá inicializace jednotlivých stromů pro instrumentační funkce. Dále jsou popsány detaily, které se týkají tvorby nových uzlů a jejich navázání do stromu. V neposlední řadě jsou popsány některé zajímavější části u konkrétních instrumentovaných událostí, kde by z návrhu nemuselo být úplně jasné, jak se řeší.

## 4.1 Zahájení běhu programu

První funkce, která se v rámci zásuvného modulu spustí je funkce `plugin_init`, beroucí parametry `struct plugin_name_args` a `struct plugin_gcc_version`. Parametr `plugin_name_args` reprezentuje argumenty zásuvného modulu a parametr `plugin_gcc_version` poté obsahuje informace o verzi GCC, aby se případně předešlo překladu se špatnou verzí překladače. Velmi důležité je, aby byla v rámci zásuvného modulu deklarovaná proměnná `plugin_is_GPL_compatible`, jinak nedojde k zavedení zásuvného modulu do programu.

Pro to, aby se mohl plugin připojit do určité fáze a vykonat potřebné kroky, tak je potřeba říct, v jakých fázích má plugin fungovat a co se má dít. K tomu slouží funkce `register_callback`. Instrumentační nástroj vyžaduje registrace do dvou fází pro správné fungování.

Funkce `register_callback` přijímá jako parametry jméno zásuvného modulu, kód dané fáze, funkci, která se má zavolat, je-li dosaženo dané fáze, a uživatelská data. Pro AST fázi je kód fáze `PLUGIN_PRE_GENERICIZE` a nepředávají se žádná uživatelská data. Fáze CFG naopak nevyžaduje funkci, která by měla danou fázi obsloužit, ale v uživatelských datech vyžaduje popis průchodu, neboli strukturu `register_pass_info`. Ta vyžaduje mnoho informací a jedná se zejména o instanci průchodu, což je v tomto případě třída dědící z průchodu GIMPLE. V této třídě se poté nachází virtuální metoda `execute`, která se spustí s každou momentálně zpracovávanou funkcí.

Struktura `register_pass_info` rovněž vyžaduje název referenčního průchodu, kterému je náš průchod nejblíže. Jelikož je potřeba instrumentovat po vytvoření grafu toku řízení, je jeho název `cfg`. Poslední důležitou informací je, jestli se náš průchod bude konat před, nebo po referenčním průchodu. Zde potřebujeme, aby byl graf již vytvořen, takže průchod bude realizován po této fázi[1].

[1] Kód pro korektní inicializaci `register_pass_info` byl převzat z [15].

Poslední fáze, která se registruje, fáze s kódem `PLUGIN_FINISH`, která spustí až na úplném konci běhu samotného zásuvného modulu. V této fázi dochází k uzavírání a mazání souborů důležitých pro správný chod zásuvného modulu.

Poté dochází ke zpracování argumentů pluginu. Argumenty lze specifikovat kombinací písmen `a`, `b` a `c` a jejich význam je následující:

- `a` (accesses) – instrumentace čtení z paměti a zápisu do paměti,
- `b` (basic blocks) – instrumentace vstupu do základního bloku a
- `c` (calls) – instrumentace volání funkce, vstupu do funkce, návratu z funkce, pokračování po volání funkce.

## 4.2 Fáze AST

Tato podkapitola popisuje implementační detaily fáze těsně po vytvoření abstraktního syntaktického stromu.

### Inicializace instrumentačních funkcí

Pro lepší uchování všech funkcí sloužících k instrumentaci byla zřízena struktura `instrumenting_functions`, jejímiž položkami jsou právě stromy těchto funkcí. Pro každý typ instrumentace definovaný v podkapitole 3.1 je použita samostatná funkce a žádná z nich se nepoužívá pro více typů instrumentací.

Strom deklarace funkce se vytváří za použití funkce `get_identifier`, která vytvoří identifikační uzel podle řetězce, který je jí zadán. Nezáleží tedy na tom, jestli se jedná o funkci, nebo proměnnou, nebo o něco, co se v programu vůbec nenachází. To, co pak vytváří samotný deklarační uzel, je funkce `lookup_name`, která se již dívá na to, co za objekt má najít a vrátí uzel podle toho, co najde ve svých tabulkách. Pokud není nic nalezeno, vrací se `NULL_TREE`, což je jen jinak vyjádřený `NULL`.

Funkce, která signalizuje vstup do základního bloku je však použita až ve fázi CFG a přesto je inicializována již zde. Prvním z důvodů je to, že je i z hlediska přehlednosti lepší, aby byly veškeré funkce inicializovány na jednom místě. Dalším poněkud závažnějším důvodem je to, že `lookup_name` přestává fungovat ve fázi CFG. Pravděpodobně je to způsobeno tím, že se tabulky během fází mění a v této fázi již není možné deklarační uzel funkce tímto způsobem získat.

Pokud dojde k tomu, že nějaká z instrumentačních funkcí není definovaná, třeba kvůli tomu, že chybí jedna funkce nebo chybí celý hlavičkový soubor s jejich deklaracemi, dojde k ukončení programu. Kdyby se pokračovalo, pravděpodobně by došlo k chybě přístupu do paměti během překladu, jelikož by byl použit prázdný strom deklarace funkce.

### Struktura nástroje

Tato fáze je realizována jako jediný rekurzivní průchod AST a skládá se ze dvou hlavních funkcí. První z nich je `determine_node`, která funguje jako přepínač mezi jednotlivými dalšími funkcemi schopné instrumentovat konkrétní uzly. Funkce `determine_node` má tedy na starost hledat uzly, které jsou popsány v podkapitole 3.5.

Uzly, které se nacházejí ve výrazu a jsou zmíněné v podkapitole 3.6, obsluhuje funkce `traverse_expr`, která již funguje jako průchod stromu v režimu preorder. To znamená,

že se v rámci výrazu program zanoří co nejhlouběji a dělá to tak dlouho, dokud nenarazí na konečný uzel, což je ve většině případů nějaká proměnná, nebo jiný uzel, který nevede k žádnému dalšímu větvení. Zde dochází k samotné instrumentaci dané události. Nový uzel je postupně navracen výše a tak je například uzel reprezentující čtení z proměnné nahrazen uzelm čtení z proměnné, před kterým je volána instrumentační funkce. Obecně lze říci, že se `determine_node` zabývá příkazy jako celky a `traverse_expr` pravými stranami přiřazení.

### Průchod řídicími konstrukcemi

Funkce `determine_node` je schopna detekovat veškeré řídicí konstrukce, které se v programu mohou nacházet. Jedná se o posloupnost příkazů, podmínky, vícenásobné podmínky (tzv. `switch`) a všechny typy cyklů, tedy `for`, `do-while` a `while`.

Pro průchod posloupností příkazů je k dispozici speciální typ `tree_stmt_iterator` umožňující nejen snadný průchod seznamem, ale také přidávání a mazání uzlů. Na každý z těchto příkazů je poté aplikována funkce `determine_node`, jelikož každý z těchto příkazů bude vždy jeden z uzlů, které jsou zmíněny v podkapitole 3.5.

Pro průchod ostatními řídicími konstrukcemi jsou poté vytvořeny samostatné funkce, kromě všech cyklů, které jsou řešeny dohromady. Obecně jsou tyto funkce velmi podobné, liší se pouze makry, která jsou použita pro přístup k jednotlivým operandům.

Funkce pro instrumentaci podmínek zároveň slouží i pro instrumentaci cyklů ve starších verzích GCC, neboť jak zaznělo v podkapitole 2.3, podmínky a cykly jsou reprezentovány stejným uzlem `COND_EXPR`. Makrem `COND_EXPR_COND` se lze dostat k podmínkové části podmínky a jelikož se jedná o část, která nemůže obsahovat žádné řídicí konstrukce, lze na ni aplikovat funkci `traverse_expr`. Na ostatní části je poté potřeba aplikovat `determine_node`, neboť zde se opět mohou nacházet řídicí konstrukce. Jen je potřeba nejprve zkontrolovat, zda blok `else` není prázdný, protože ne každá podmínka ho obsahuje.

### Tvorba nových uzlů

Hlavičkové soubory obsažené v nástroji pro tvorbu zásuvných modulů obsahují mimo jiné i funkce sloužící pro tvorbu nových uzlů různých typů.

Prvním typem funkcí, které se dají použít pro tvorbu uzlů, jsou funkce `buildn`, případně `buildn_loc`. Signatura těchto funkcí vypadá následovně:

```
buildn(enum tree_code code, tree type, ...)
buildn_loc(location_t loc, enum tree_code, ...).
```

Znak `n` zde zastupuje celá čísla od 0 do 5 a znamená, kolik operandů tvořený uzel má. Reálná signatura funkce je tedy například

```
build2(enum tree_code code, tree type, tree op1, tree op2),
```

Podobně je na tom i `buildn_loc`, který přijímá navíc i pozici uzlu.

Dalším parametrem je `code`, který umožňuje vytvoření jakéhokoliv uzlu. Podstatným parametrem je také `type` reprezentující datový typ daného uzlu. Posledními parametry jsou operandy pro daný uzel, které jsou rovněž typu `tree`.

Jinou možností je použít dedikované funkce pro tvorbu konkrétních uzlů. Jejich výhodou je to, že nevyžadují datový typ uzlu a zároveň přijímají i jeho lokaci. Nevýhoda tkví v tom, že tyto funkce nejsou k dispozici pro všechny uzly, tedy je potřeba využít funkci `build`.

V programu se ve většině případů vytvářejí uzly pro složený výraz reprezentující operátor `,` a uzel reprezentující volání jedné z instrumentačních funkcí. Pro složený výraz, lze použít `build_compound_expr` vyžadující pozici uzlu v kódu a stromy obou operandů.

Pro vytvoření volání funkce je potřeba použít funkci `build_call_expr_loc`, která kromě lokace vyžaduje i počet parametrů, které má dané volání obsahovat, a také jméno funkce, které ovšem nemůže být funkci předáno jako čisté jméno, nýbrž jako deklarační uzel dané funkce. Ten lze získat pouze ve fázi AST použitím nejdříve funkce `get_identifier`, jejíž výstup je použit v `lookup_name`. Dalšími argumenty této funkce jsou pak parametry, které má dané volání obsahovat, avšak opět ve formě stromu. Tedy chceme-li funkci předat nějakou celočíselnou konstantu, je i pro tu nutné vytvořit uzel, který ji reprezentuje. Konkrétně pro celočíselnou konstantu by se jednalo o

```
build_int_cst(integer_type_node, num),
```

kde `num` je celé číslo, pro které chceme uzel vytvořit.

Tyto funkce rovněž slouží i pro tvorbu uzlů reprezentující určitý datový typ, neboli `TREE_TYPE` popsán v podkapitole 2.3. V mé implementaci je použita funkce `build_pointer_type`, jelikož jsou vytvářeny proměnné typu ukazatel. Rovněž pro vytvoření dereferenčního uzlu byla použita funkce `build_indirect_ref`, která kromě lokace a uzlu, na který má být dereference vytvořena, vyžaduje speciální konstantu sdělující, o jaký konkrétní typ dereference se jedná. V našem případě je použita konstanta `RO_UNARY_STAR` reprezentující operátor `*` sloužící v jazyce C právě pro vyjádření dereference.

Dalším důležitým uzlem, který je v rámci programu často vytvářen je uzel `NOP_EXPR`. Funkce instrumentující čtení a zápis totiž disponují argumenty typu `void *`. Společně s číslem datového typu, který je funkci také předáván, lze poté hodnotu korektně přetypovat a vypsat, to umožňuje tisknout jak celá, tak i desetinná čísla.

### Navazování uzlů do stromu

Uzly typu `tree` jsou ve své podstatě ukazatele, tímto způsobem jsou také předávány do funkcí. Avšak při instrumentaci dochází k tvorbě nových uzlů, které sice obsahují ty staré, nicméně samotné nikam nepatří. Proto je potřeba nově vytvořené uzly obsahující nezbytné funkce pro instrumentaci navázat tam, kde se nacházel původní uzel, aby došlo k promítnutí této změny dále při překladu a poté i v samotném instrumentovaném programu.

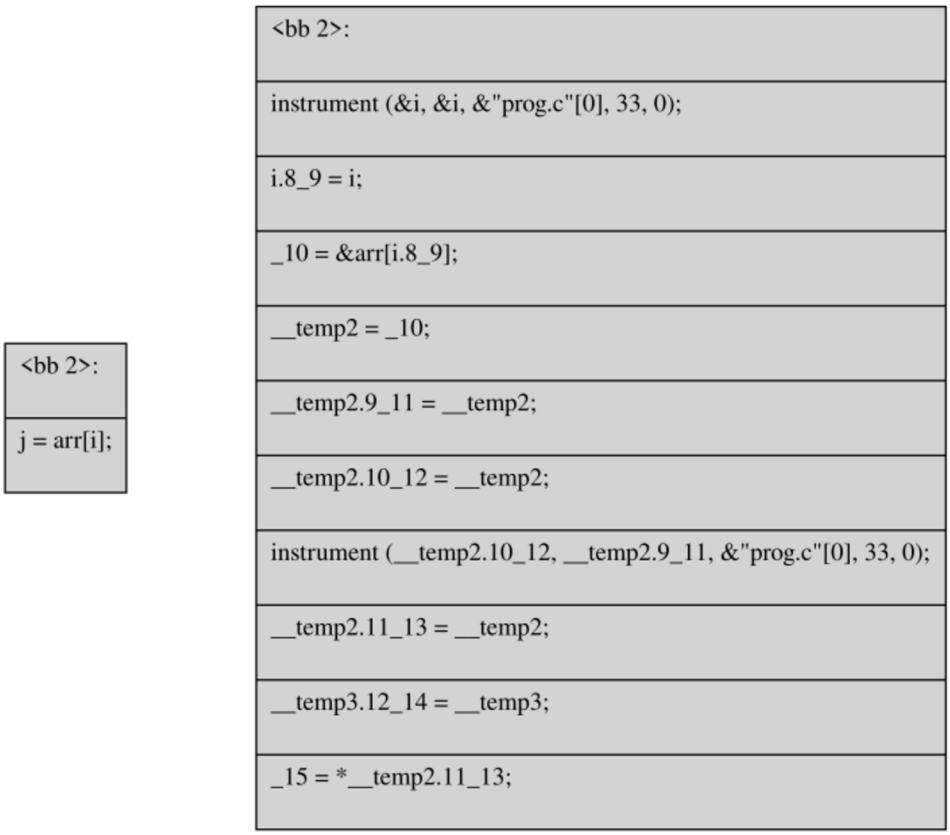
Během průchodu výrazem je přiřazení vyřešeno tak, že se rekurzivně volá `traverse_expr` pro každý operand a instrumentovaný uzel je ve výsledku navrácen a přiřazen do operandu uzlu předešlého.

Avšak u příkazů samotných je průběh malinko jiný. Nacházíme-li se například v těle podmínky, které samo o sobě může obsahovat další podmínky a další příkazy a je zde jasné, že se bude volat funkce `determine_node`, která strom dále zpracovává, je potřeba jí dát vědět, jakým způsobem se má navázat na uzel, ze kterého byla volána. K tomuto účelu funkce bere kromě uzlu, který se má projít i parametr `has_statement_list`, který nám říká, jestli je uzel seznamem příkazů, či nikoliv.

Pro oba z těchto případů je důležité se chovat odlišně. Pokud se nejedná o uzel seznamu příkazů, ale o uzel jakéhokoliv jiného příkazu, je potřeba jeho instrumentovanou variantu navrátit zpátky, jinak by uzel zůstal neinstrumentován. Naopak jedná-li se o `STATEMENT_LIST`, je potřeba nově instrumentovaný uzel korektně navázat do daného seznamu příkazů a není potřeba nic vracet, jelikož uzel je již v seznamu příkazů pevně navázán. Samotná práce s uzlem seznamu příkazů se provádí pomocí speciálního typu `tree_stmt_iterator`, který

umožňuje jednoduchý průchod tímto seznamem, jelikož je implementován ve formě dvousměrně vázaného seznamu, což bylo popsáno také v podkapitole 2.3. Avšak každý příkaz, na který může tento iterátor během průchodu narazit, je sám o sobě struktura obsahující ukazatele na následný a předešlý uzel. Informace o uzlu samotném jsou v něm sice také ukryty a dal by se modifikovat přímo, nicméně interní struktury mohou být změněny, proto jsou použity speciální funkce na navázání nového příkazu do seznamu příkazů, které překladač nabízí.

S iterátorem je uchována i jeho globální kopie ukazující na stejné místo jako původní iterátor, proto lze ve funkci na korektní navázání příkazu do stromu použít tento globální iterátor. Nejprve je použita funkce `tsi_delink`, která vymaže konkrétní příkaz ze seznamu a iterátor nyní ukazuje na následující příkaz. Díky této informaci je poté možné navázat nově instrumentovaný příkaz na stejné místo, kde se dříve nacházel původní příkaz pomocí funkce `tsi_insert_before`, která uzel vloží před místo, kam iterátor ukazuje.




Obrázek 4.1: Příklad instrumentace čtení z paměti. Na levém obrázku se nachází čtení z `i` a z `arr[i]` (přiřazení do `j` je zde jen pro úplnost příkazu). Na pravém obrázku se poté nachází instrumentované čtení z těchto proměnných (nikoliv však zápis do `j`).

### Identifikace konce definice funkce

Instrumentace návratu z funkce je prováděna přesně tak, jak byla popsána v podkapitole 3.7. V rámci návratového uzlu je kontrolováno, zda je první operand prázdný, nebo ne, aby

bylo možné rozlišit, zda se jedná o první, či druhý případ. Pro detekci funkce, která na konci žádný `return` nemá, je potřeba počkat, až se projde celá funkce. Poté se lze detekovat její konec. Není potřeba kontrolovat všechny funkce, ale jen ty, které jsou typu `void`, neboť funkce jiných typů musí obsahovat `return` na všech místech, kde by potenciálně mohly skončit. Jedině u typu `void` může program dojít až na konec její definice.

Dále je nutné rozlišit, jestli tělo funkce sestává z jednoho, nebo více příkazů. Jinými slovy, zda obsahuje uzel typu `STATEMENT_LIST`, či nikoliv. Pokud `STATEMENT_LIST` obsahuje, stačí pomocí makra `tsi_last` získat poslední příkaz a pokud ten není návratovým uzlem, lze vytvořit uzel s funkcí signalizující návrat z funkce a navázat ho do seznamu příkazů. V případě, že by se ve funkci nacházel jen jeden příkaz, je zapotřebí vytvořit nový seznam příkazů, jehož prvním příkazem bude původní příkaz, který se ve funkci nacházel, a druhým bude právě naše instrumentační funkce.

### Rozšířené přiřazení

Rozšířené přiřazení umožňuje programátorovi zkrátit některé přiřazovací výrazy. Například pro

```
x = x + 2
```

lze použít

```
x += 2.
```

Oba tyto výrazy jsou svým chováním identické. V jazyce C pro podobné operace existují kromě `+=` i další operátory realizující podobné operace. Jsou to například `-=`, `*=`, `&=` a další. V abstraktním syntaktickém stromě jsou tyto operace reprezentovány stejně, takže vskutku nezáleží na tom, který přístup se použije. Když se však instrumentují tyto uzly, ať už se jedná o zápis pro uzel nacházející se na levé straně přiřazení nebo o čtení pro uzel na pravé straně přiřazení, tak je potřeba rozlišit, zda se jedná o rozšířené přiřazení, nebo o klasické.

Čím se rozšířené přiřazení liší od toho klasického na úrovni AST je, že uzel na levé straně přiřazení a na pravé ukazují na stejné místo v paměti. Jinými slovy, změní-li se něco uvnitř uzlu vlevo, projeví se to i v uzlu napravo.

Představme si, že bychom výraz `arr[i] += 1` přepsali na `arr[i] = arr[i] + 1`, aby bylo vidět, co přesně se stane. Nejprve je instrumentován index uzlu na pravé straně. Pro demonstrační účely je použito pouze volání funkce před použitím proměnné. Jak se skutečně instrumentuje čtení z paměti lze nalézt v podkapitole 3.7.

```
arr[(cb(), i)] = arr[(cb(), i)] + 1.
```

Byl instrumentován pouze index pravé strany, avšak díky tomu, že uzly spolu sdílejí paměť, tak se změna promítla i do druhého uzlu. Kdyby byl použit klasický způsob přiřazení, nic takového by se nestalo, neboť uzly by spolu paměť nesdílely.

Kdyby poté došlo k instrumentaci toho, co se nachází na pravé straně, nástroj by narazil na pole s indexem, které je již instrumentované. Proběhla by tedy instrumentace kódu, který tam byl uměle dodán, což rozhodně není dobře.

Jedním z řešení by bylo přeskakovat instrumentaci funkcí, které zde byly přidány v rámci instrumentace nebo detekovat, zdali uzel na levé straně je identický s korespondujícím uzlem na straně pravé. Řešení, které bylo v programu použito, je jiné, a sice vytvořit identickou kopii uzlu na levé straně. Tím vznikne stejný uzel, avšak na jiném místě v paměti, takže je možné instrumentovat uzly na levé a pravé straně nezávisle. K tomu existuje funkce

`copy_node`, která je schopna vytvořit bohužel pouze mělkou kopii, takže nedochází ke zkopírování operandů daného uzlu. S její pomocí však byla vytvořena funkce `deep_copy_node`, která již umí vytvářet i kopie operandů.

### Kompatibilita se standardními knihovnami

Při zahrnutí knihovny `stdlib.h` v programu, který měl být instrumentován došlo k tomu, že program instrumentoval kromě uživatelských funkcí i jiné funkce nepocházející od uživatele. Konkrétně se jedná o:

- `__bswap16`,
- `__bswap32`,
- `__bswap64`,
- `__uint16t_identity`,
- `__uint32t_identity`,
- `__uint64t_identity`.

Tyto funkce by mohly být instrumentovány špatně a i kdyby ne, tak by docházelo k výpisům, které nejsou pro uživatele žádoucí. Seznam těchto funkcí se nachází v poli `stdlib_functions`. Pokud program zaregistruje, že by měla být instrumentována jedna z těchto funkcí, dochází k jejich přeskočení a pokračuje dále s jinou funkcí. Znamená to tedy, že program nepodporuje instrumentaci uživatelských funkcí mající stejné jméno. Pokud by v budoucích verzích GCC došlo k přidání dalších podobných funkcí, bylo by je potřeba je do pole přidat. V opačném případě není zaručen správný chod zásuvného modulu. Při použití jiných standardních knihoven k takovému problému nedošlo. Instrumentační nástroj také nepodporuje knihovnu `complex.h`. Instrumentace čtení z komplexních čísel a zápis do nich je přeskočen, neboť obsahují speciální typy uzlů.

## 4.3 Fáze CFG

V této podkapitole jsou popsány některé zajímavé implementační detaily z fáze těsně po vytvoření grafu toku řízení.

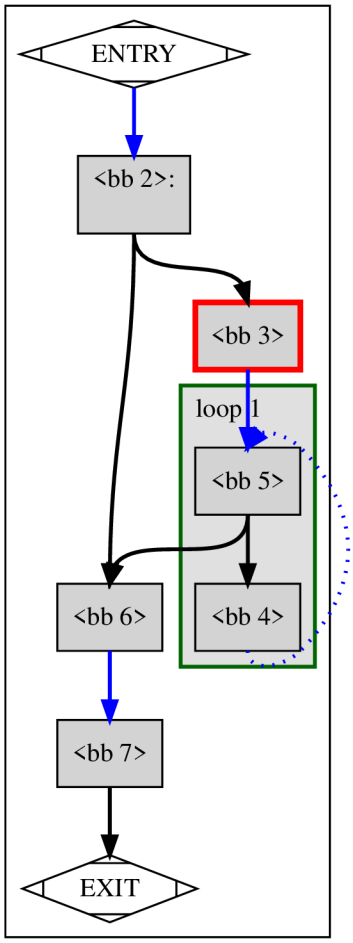
### Vstup do základního bloku

K instrumentaci této události nedochází ve fázi AST, tudíž je potřeba využít jiné funkce než ty, které byly použity ve fázi po vytvoření AST, protože ty by zde nefungovaly, neboť nyní již nepracujeme s AST, ale s instrukcemi `GIMPLE`. Funkce z AST fáze jsou přeci jen využity a to pro úpravu parametrů argumentů instrumentační funkce. Pro průchod základními bloky lze využít makro `FOR_EACH_BB_FN(bb, fn)`, které pro každou funkci `fn` projde veškeré její základní bloky. Základní blok, v němž se cyklus právě nachází, je poté uložen v proměnné `bb`, která mimo jiné uchovává i sekvenci `GIMPLE` instrukcí, jež daný blok obsahuje. Jelikož se nejedná o strom, nelze využít `build_call_expr` pro sestavení `CALL_EXPR` uzlu. Zde lze využít funkce `gimple_build_call_from_tree`, která, jak název napovídá, vytváří volání funkce v `GIMPLE` podobě ze stromu. To je jediný důvod, proč byl tento uzel sestaven již ve fázi AST.

Když je k dispozici základní blok se všemi instrukcemi, zbývá pouze přidat příslušnou funkci do sekvence instrukcí, a jelikož se jedná o vstup do základního bloku, patří na začátek této sekvence. Zde je však nutné si uvědomit, že základní blok může začít instrukcí, která se vykoná, ale také instrukcí typu `LABEL`, na kterou se může skočit. Jinými slovy, pokud by i v tomto případě byla instrumentační funkce vložena na první místo, pravděpodobně by se nikdy nevykonala.

Překladač také nevytváří graf toku řízení tak, jako by ho člověk tvořil na papíře. Někdy si pomůže a vytváří umělé základní bloky, které se mohou nacházet například před nebo za cyklem, za účelem pozdější optimalizace. Tyto bloky neobsahují žádné instrukce a jejich instrumentace tudíž nemá smysl.

Postup pro navázání do sekvence je velmi podobný jako navazování nového uzlu do seznamu příkazů.




Obrázek 4.2: Ukázka umělého základního bloku (zde označen červeně). Jedná se o upravený graf toku řízení, který lze získat při překladu za použití přepínače `-fdump-tree-cfg-graph`.

### Vstup do funkce

Implementace je velmi podobná vstupu do základního bloku s tím rozdílem, že instrumentační funkce pro tento typ události se nachází za funkcí instrumentující vstup do základního

bloku. Trochu jsem měl dilema, která z funkcí by měla být v sekvenci na prvním místě, ale nakonec jsem usoudil, že vstup do bloku by měl být přímo na začátku základního bloku, a proto se nachází na začátku sekvence. Dále je potřeba opět ze stejných důvodů jako při instrumentaci vstupu do bloku vytvořit uzel volání instrumentační funkce již ve fázi AST.

Dalším rozdílem je fakt, že by se funkce instrumentující vstup do funkce měla nacházet pouze v prvním základním bloku dané funkce, protože další bloky začátek funkce nereprezentují. Naštěstí makro `FOR_EACH_BB_FN` prochází základními bloky s indexem od nejnižšího po nejvyšší s tím, že nejnižší index reprezentuje blok nacházející se na samém začátku funkce. Existuje také makro `FOR_EACH_BB_REVERSE_FN`, které základními bloky prochází v opačném pořadí.

### Přidávání údajů o poloze do nových uzlů

Původní implementace byla taková, že z důvodu neznalosti chování typu `location_t`, který v sobě uchovává informace o poloze jednotlivých konstrukcí ve zdrojovém kódu, byly pro vytváření nových uzlů používány funkce `buildn(code, type, ...)`. Tyto funkce v rámci své vnitřní implementace používají speciální konstantu `UNKNOWN_LOCATION`, která znamená, že neznáme lokaci daného uzlu ve zdrojovém kódu.

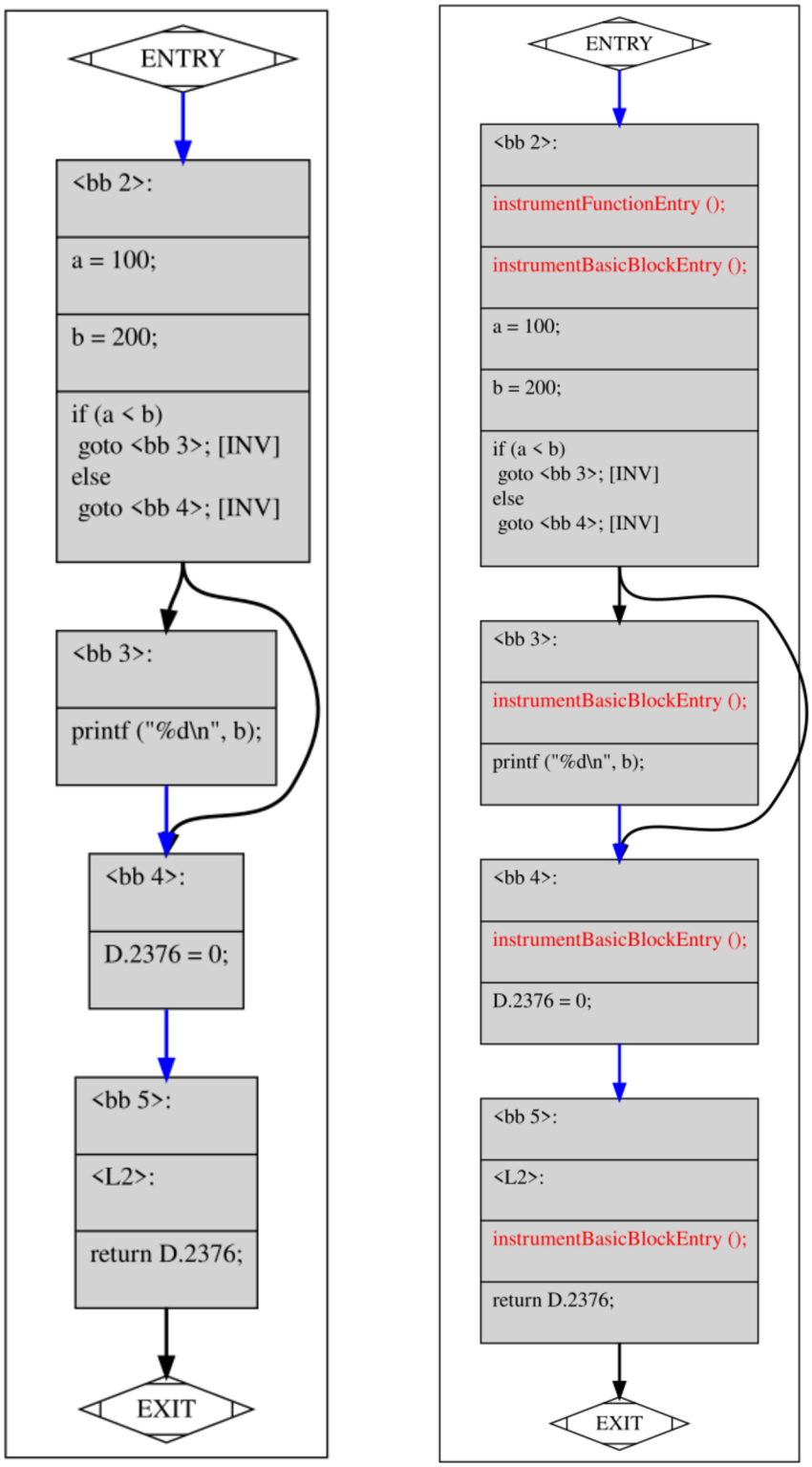
Původně bylo předpokládáno, že neznámá pozice nově tvořených uzlů nebude mít vliv na číslo řádku, kde došlo k nějaké události. Nakonec se ukázalo, že tento přístup působí problémy při instrumentaci vstupu do základního bloku, kde se řádek získává pomocí makra `gimple_location`, které vrací polohu dané `GIMPLE` instrukce zděděnou z dříve probíhajících fází. Ovšem po instrumentaci v rámci fáze AST se v základních blocích objeví instrukce, které tam dříve nebyly, a jestliže některé z nich mají neznámou lokaci, může to poté ovlivnit řádek, který by byl pomocí tohoto makra získán. V praxi se pozice základních bloků v některých případech o několik řádků lišila, což může být problém, protože identifikovat řádek, na kterém základní blok začíná pouze ze zdrojového textu, může být někdy obtížné.

Proto jsem začal využívat funkce, které jako parametr přijímají i lokaci, například místo `build_call_expr`, který vytváří uzel `CALL_EXPR`, byla použita funkce `build_call_expr_loc`, která dělá to samé, jen přijímá lokaci jako parametr navíc. Tam, kde takové funkce neexistují, nebo je jejich použití příliš komplikované, lze přiřadit lokaci manuálně pomocí makra `SET_EXPR_LOCATION`. Cílem je tedy každému nově vytvořenému uzlu předat lokaci tak, aby ve fázi CFG nedocházelo k posunutí řádků, ať už se jedná o vstup do základního bloku nebo do funkce.

Z toho však vyplývá, že nově vytvořeným uzlům je přidělena stejná pozice jako pozice původního uzlu, tedy jak soubor, tak řádek, tak i sloupec, protože jiné informace o poloze nejsou dostupné. Jelikož dochází pouze k výpisu řádků, nemusí nás sloupce příliš zajímat. Výhodu to má také v tom, že nedochází k ovlivnění lokací pro CFG fázi.

## 4.4 Tvorba podkladů pro měření pokrytí

Jak již bylo zmíněno v podkapitole 3.8 jsou potřeba 2 překlady k tomu, aby došlo k vytvoření `.covitems` souboru a následné instrumentaci. Tvorba tohoto souboru se děje první, protože je potřeba mít na vstupu neinstrumentovaný kód pro nejpřesnější výsledky. K tomu jsou uzpůsobeny i argumenty zásuvného modulu, kdy klíč `covitems` vypíná veškerou instrumentaci a zapíná tvorbu souboru.




Obrázek 4.3: Porovnání instrumentovaného a neinstrumentovaného kódu. Na levé straně se nachází původní neinstrumentovaný kód, na pravé poté instrumentovaný kód s přidanými instrumentačními funkcemi. Z důvodu velikosti obrázku jsou instrumentační funkce bez parametrů.

### Předci a následovníci základního bloku

Podkladový soubor sice ještě vyžaduje informace ohledně bloku, ve kterém se právě nachází, a funkci, to jsou však stejné parametry, které se použijí při instrumentaci vstupu do základního bloku. Aby je bylo možné zjistit, musíme se podívat na hrany, které do něj vstupují a které z něj vychází. K tomu lze použít makro `FOR_EACH_EDGE(e, ei, bb->succs)`, kde `e` je instance hrany a `ei` je hranový iterátor. Konkrétně toto použití makra iteruje hranami, které z daného bloku vycházejí. Pro iteraci hranami, které do něj vstupují stačí pouze zaměnit `bb->succs` za `bb->preds`.

Bylo by logické, aby každá funkce začínala základním blokem 0, nebo 1. To však neplatí v případě GCC, neboť každá funkce začíná základním blokem 2. Je to dáno tím, že základní bloky 0 a 1 jsou speciální bloky, které reprezentují vstup do funkce a výstup z ní [22, kap. 15.1]. Při využití těchto maker se může stát, že nějaký základní blok je hranou připojen k jednomu z těchto speciálních. Tyto hrany je nutné přeskočit, neboť tyto speciální bloky se v grafu toku řízení jako klasické bloky nenacházejí a byly by tak v souboru navíc. Tyto bloky ani nejsou ze stejných důvodů instrumentované, takže by se uživatel ani nedozvěděl, že se těmito bloky prošlo.

### Čtení a zápisy proměnných

Aby šlo získat tyto informace, tak už je potřeba projít instrukci po instrukci. Existují 3 typy GIMPLE instrukcí, které z proměnných čtou nebo do nich zapisují:

- `GIMPLE_ASSIGN` je podobné uzlu `MODIFY_EXPR` a reprezentuje přiřazení. Na levé straně se nachází proměnná, do které se přiřazuje a na pravé straně se mohou nacházet pouze 1 – 3 další proměnné, ze kterých se čte.
- `GIMPLE_CALL` je velmi podobné uzlu `CALL_EXPR`, avšak s tím rozdílem, že zde mají tento kód i instrukce, kde je na levé straně proměnná a na pravé volání funkce.
- `GIMPLE_COND` se opět velmi podobá uzlu `COND_EXPR`. Zde nás však zajímá pouze samotná podmínka, která může obsahovat proměnné, ze kterých se čte.

Komplikace však nastává v momentě, kdy se k proměnným dostaneme, protože GIMPLE samotný si vytváří pomocné proměnné, aby mohl jeden složitý výraz rozložit na více menších. Naštěstí se tyto proměnné od klasických liší tím, že mají kód uzlu `SSA_NAME`, takže je lze odlišit od těch, které se zde vyskytují již od samotného začátku překladu.

Dále je potřeba se vyhnout návratové proměnné popsané v podkapitole 3.7. Jelikož má uzel této proměnné kód `VAR_DECL`, je od ostatních proměnných k nerozeznání. Lze však použít makro `DECL_ARTIFICIAL`, které vrací informaci o tom, zdali je proměnná uměle deklarována, což návratová proměnná je.

To, co do souboru rozhodně nepatří, jsou konstanty, které program sice považuje za proměnné, nicméně ve výsledném souboru by nedávaly smysl. Na to stačí použít makro `CONSTANT_CLASS_P`, které identifikuje, zdali se jedná o konstantu, či nikoliv. Speciálním případem jsou řetězcové literály, které jsou ještě zabaleny do uzlu `ARRAY_REF`.

Přístupy do paměti, které nelze vyjádřit pouze jedním identifikátorem jako například přístupy do pole nebo struktur, využívají funkci `print_generic_expr_to_str`, která tento složený název vrací jako řetězec. Bohužel GCC 8 tuto funkci nemá a má pouze alternativu tisknoucí do souboru. Z tohoto důvodu je v rámci programu používán dočasný soubor `nodename.tmp`, do kterého se tento název ukládá. Program vzápětí tento soubor otevírá

a jeho obsah dále zpracovává. Soubor je po dokončení všech činností zásuvného modulu smazán.

Jediným omezením může být to, že se do souboru v rámci `defs` a `uses` může objevit například

```
arr[_3].
```

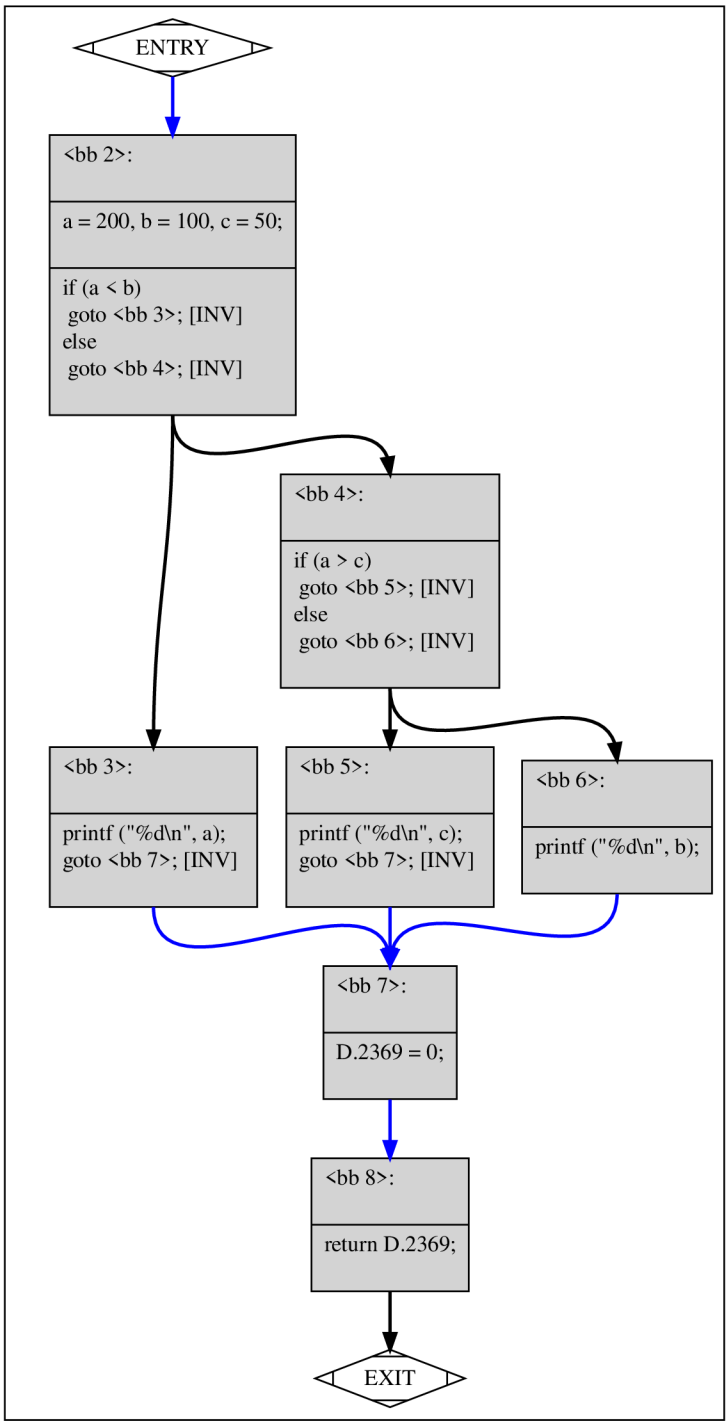
To je způsobeno tím, že index v poli mohl být nějaký složitější výraz, který GIMPLE musel rozložit na menší podvýrazy, a do proměnné `_3` umístil výsledek celého výrazu. Podobně k tomu může dojít i u nepřímých přístupů a jiných konstrukcí.

```
int main() {

    int a = 200;
    int b = 100;
    int c = 50;
    if (a < b) {
        printf("%d\n", a);
    }
    else if (a > c) {
        printf("Ending in else if block %d\n", c);
    }
    else {
        printf("%d\n", b);
    }
    return 0;
}
```

Výpis 4.1: Příklad kódu pro nějž má být vytvořen soubor `.covitems`.

Na obrázku 4.4 se poté nachází graf toku řízení za použití přepínače `-fdump-tree-cfg--graph` při překladu pro tento kód.




Obrázek 4.4: CFG pro kód 4.1 vygenerovaný překladačem.

Soubor `.covitems` pro kód 4.1 a graf toku řízení 4.4 poté vypadá takto:

```
{"bb": 2, "function": "main", "pred:": [], "succ": [3, 4],
    "defs": ["a", "b", "c"], "uses": ["a", "b"]}
{"bb": 3, "function": "main", "pred:": [2], "succ": [7],
    "defs": [], "uses": ["a"]}
{"bb": 4, "function": "main", "pred:": [2], "succ": [5, 6],
    "defs": [], "uses": ["a", "c"]}
{"bb": 5, "function": "main", "pred:": [4], "succ": [7],
    "defs": [], "uses": ["c"]}
{"bb": 6, "function": "main", "pred:": [4], "succ": [7],
    "defs": [], "uses": ["b"]}
{"bb": 7, "function": "main", "pred:": [3, 5, 6], "succ": [8],
    "defs": [], "uses": []}
{"bb": 8, "function": "main", "pred:": [7], "succ": [],
    "defs": [], "uses": []}.
```

Výpis 4.2: Obsah souboru `.covitems`.

# Kapitola 5

# Ověření funkcionality instrumentačního nástroje

Tato kapitola se detailněji zabývá tím, jakým způsobem bylo kontrolováno, zda instrumentační nástroj pracuje, jak má a nedochází u něj k chybám. Je zde popsáno, které prostředky lze využít pro ladění na úrovni AST a CFG. Dále je součástí, jak probíhalo ověření funkcionality nástroje pomocí automatických testů.

## 5.1 Testování a ladění během vývoje

Automatické testy dokáží velmi rychle vyzkoušet mnoho předdefinovaných příkladů a zjistit, jestli se některé výstupy neliší od očekávaných. Jestliže však chceme zjistit, kde přesně se chyba nachází, je potřeba se projít celý strom nebo GIMPLE instrukce. Automatické testy nám v tomto moc nepomohou, pouze nám mohou říci, který z výpisů se odchyluje od očekávaného.

Infrastruktura GCC naštěstí nabízí funkce, díky nimž lze ověřit, zda instrumentace ať už na úrovni abstraktního syntaktického stromu nebo na úrovni grafu toku řízení. Na úrovni AST se jedná o funkce `debug` a `debug_tree`, které slouží k zobrazení AST, ať už jde o celou funkci nebo jen například o modifikaci proměnné. Tyto funkce se liší pouze tím, v jakém formátu produkují výstup.

Výstup funkce `debug` se skoro neliší od původního zdrojového textu. To, co je zde navíc, jsou pouze doplněné informace o přetypování, pokud se jedná o implicitní přetypování, a uzávorkování výrazů tak, aby bylo zajištěno jejich správné vyhodnocení. Tato funkce je vhodná k použití v případě, že chceme vidět, jak vypadá instrumentovaný kód jako celek, a nezajímají nás detaily.

Jestliže chceme znát detaily, které se vážou k jednotlivým prvkům daného programu, je vhodné využít funkci `debug_tree`, která výstup zobrazí v podobě, která více připomíná strukturu stromu. Výpis stromu však funguje pouze do určité hloubky, takže pro další detaily pak lze aplikovat tuto funkci na další poduzly, abychom se dozvěděli více.

Pro výpisy instrumentovaných instrukcí ve fázi CFG existuje více variant. Ty, které jsem používal já, byly funkce `debug_gimple_stmt` vypisující konkrétní instrukci GIMPLE a `gimple_dump_bb`, která vypisuje veškeré instrukce nacházející se v daném základním bloku.

## 5.2 Automatické testování

Automatické testování se zabývá celkovou funkčností instrumentačního nástroje, tedy, zdali výpisy, které získáme při spuštění instrumentovaného programu, odpovídají očekávaným.

### Struktura testů

Každý test má svoji složku pojmenovanou podle toho, co je hlavním cílem testu. Kromě programu, který má být otestován, se ve složce ještě nacházejí soubory `expected.trace` a `actual.trace`, které v sobě uchovávají očekávanou a získanou trasu, kterou daný program prošel, tedy výpisy způsobené voláním jednotlivých instrumentačních funkcí. Na každém řádku se nachází jeden záznam ve formátu JSON reprezentující událost, která byla vyvolána. Kvůli problémům s adresami popsané v následující podkapitole nelze výstupy porovnávat pouhým porovnáním obsahu obou souborů.

Pro ověření standardního výstupu testovaného programu, tedy jestli instrumentace náhodou nezměnila chování testovaného programu, se v adresáři nacházejí i soubory `expected.stdout` a `actual.stdout`.

Poslední co se v adresáři každého testu nachází, jsou soubory `sameAddresses.txt` a `samePtrValues.txt`, jejichž struktura a účel je blíže popsána v následující podkapitole.

### Adresy paměti a hodnoty ukazatelů

Pro to, aby se dal program správně testovat, je důležité, aby testovaný program pro více běhů produkoval stále stejný výstup. Toho však nelze v tomto případě dosáhnout, neboť výpisy obsahují adresy daných paměťových míst a mohou obsahovat i hodnoty ukazatelů, které se rovněž běh od běhu liší. Děje se tak kvůli prevenci před možnými útoky, které mohou způsobit přetečení programu do jiného místa v paměti, které nemá vyhrazeno (tzv. *buffer overflow*). Tomuto se v angličtině říká *Address space layout randomization*, zkráceně *ASLR*, a znamená to, že při každém běhu programu se náhodně zvolí rozsah paměti, který je danému programu přidělen [9]. Existují sice příkazy, které vypínají toto náhodné přidělování adres pro konkrétní spuštění programu jako například

```
setarch `uname -m` -R ./prog,
```

nicméně tyto adresy nejsou stejné, pokud dojde k vypnutí a zapnutí počítače.

Původním řešením bylo testovat nikoliv výpisy, které instrumentovaný program poskytne po jeho běhu, ale přímo testovat, jestli se instrumentovaný zdrojový text shoduje s očekávaným. Tento výstup lze získat pomocí funkce `debug`, která je popsána v podkapitole 5.1. Nemusela by však být zajištěna kompatibilita testů s novějšími verzemi GCC, neboť `debug` je interní funkcí překladače, jejíž výstup se může s novými verzemi měnit.

Proto je součástí každého testu ještě soubor `sameAddresses.txt`. Ten vznikne tak, že se z očekávaného výpisu odstraní všechny události, které se netýkají čtení a zápisu do paměti, a z tohoto zbylého výpisu se identifikují čísla řádků, které obsahují stejné adresy pro všechny možné adresy, které se ve výpisu nacházejí. Všechna čísla řádků, kde se konkrétní adresa nachází, pak reprezentují jeden řádek `sameAddresses.txt`. Celkový počet těchto řádků je poté dán celkovým počtem různých adres, které se ve výpisu nacházejí. Testovací skript poté zpracuje tyto seznamy a zkontroluje, zda se skutečně na daných řádcích výpisu tyto adresy nacházejí. Soubor `samePtrValues.txt` pak funguje na velmi podobném principu s tím rozdílem, že se v původním výpisu ponechají veškeré události.

## 5.3 Kompatibilita s různými verzemi GCC a omezení testů

Jedním z požadavků popsaných v podkapitole 3.1 bylo to, aby byl nástroj podporován napříč různými verzemi GCC. Ve většině času byl nástroj vyvíjen s GCC ve verzi 9. Občas však byla použita i verze 12 kvůli změnám, které zachycuje podkapitola 2.3.

Nástroj byl testován na verzích 7 – 12. Nicméně u verze 7 se nacházely některé změny v signaturách funkcí, které by bylo nutné řešit podmíněným překladem. V některých případech také docházelo k optimalizacím i na úrovni syntaktické analýzy. Chování programu se sice nezměnilo, avšak docházelo k odstranění některých proměnných, jejichž záznam o čtení by ve výsledném výpisu chyběl. Z tohoto důvodu se použití nástroje doporučuje pro verze 8 – 12.

Pro testy je klíčové, aby se pořadí výpisů po každém běhu testovaného programu neměnilo. Může se však stát, že v případě paralelních programů nebo novější verze GCC se pořadí výpisů změní a testy tak mohou selhat. Proto je doporučeno testy spouštět pouze na jednovláknových programech a verzích překladače, které instrumentační nástroj podporuje.

Testování výstupu `.covitems` bylo provedeno manuálně na verzích GCC 8 a 12, neboť graf toku řízení a některé vnitřní proměnné, které se také mohou dostat do souboru, se mohou měnit.

# Kapitola 6

# Závěr

Cílem této práce bylo vytvořit nástroj, který má za úkol instrumentovat různé události, ať už je to čtení z paměti, zápis do paměti, volání funkcí nebo vstup do základního bloku. Dále bylo za úkol vytvořit program počítající různá kritéria pokrytí, což bylo upraveno na vytvoření souboru obsahující informace, díky kterým je možné tato kritéria pokrytí měřit. Vytvořil jsem plně funkční program, který obě tyto funkcionality splňuje. Funkcionalita byla ověřena demonstrační testovací sadou.

V první části jsem popsal, jakým způsobem probíhá testování na základě modelů a zmínil některá kritéria pokrytí, která instrumentační nástroj umožňuje měřit. Dále také byly popsány uzly abstraktního syntaktického stromu překladače GCC, bez jejichž znalostí by instrumentační nástroj nemohl vůbec vzniknout.

Následně jsem identifikoval požadavky, které jsou na nástroj kladeny, na základě nichž jsem navrhl, jakým způsobem instrumentace jednotlivých událostí probíhá, jak jsou modifikovány jednotlivé uzly a jak instrumentací nezměnit chování programu. Poté jsem navrhl strukturu podkladového souboru, díky němuž má uživatel dostatek informací pro to, aby mohl měřit daná kritéria pokrytí.

Díky této práci jsem se dozvěděl, jak se tvoří abstraktní syntaktický strom a jaké informace obsahuje. Dále jsem si prohloubil znalosti ohledně mezikódu překladače GCC, dozvěděl se, jak vypadá a jaké konstrukce může obsahovat. Rovněž jsem se seznámil s tím, jak překladač tvoří graf toku řízení. A hlavně jsem se naučil se všemi těmito strukturami pracovat.

Ačkoliv je tento nástroj plně funkční, stále na něm lze pracovat a vylepšovat ho. Například by mohlo dojít ke znovupoužití některých dočasných proměnných, které se kvůli instrumentaci tvoří, a snížit tak jejich počet, neboť některé typy instrumentace vyžadují až 3 pomocné proměnné. Kdybych měl znalosti o této problematice na začátku tvorby této práce, zvážil bych použití fáze CFG pro instrumentaci všech událostí a fázi AST bych použil pouze pro inicializaci instrumentačních funkcí.

# Literatura


[1] *Clang: a C language family frontend for LLVM* [online]. [cit. 2023-04-19]. Dostupné z: https://clang.llvm.org/.

[2] *Gcov Intro (Using the GNU Compiler Collection (GCC))* [online]. [cit. 2023-04-24]. Dostupné z: https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html.

[3] *The LLVM Compiler Infrastructure* [online]. [cit. 2023-04-19]. Dostupné z: https://llvm.org/.

[4] ISO/IEC/IEEE International Standard - Software and systems engineering —Software testing —Part 1:Concepts and definitions. *ISO/IEC/IEEE 29119-1:2013(E)*. Sep. 2013, s. 1–64.

[5] *Preprocessor* [online]. Listopad 2020 [cit. 2023-04-18]. Dostupné z: https://en.cppreference.com/w/cpp/preprocessor.

[6] *GCC, the GNU Compiler Collection* [online]. Duben 2023 [cit. 2023-04-19]. Dostupné z: https://gcc.gnu.org/.

[7] Allen, F. E. Control Flow Analysis. In: *Proceedings of a symposium on Compiler optimization*. New York, NY, USA: Association for Computing Machinery, 1970, s. 1–19. ISBN 9781450373869. Dostupné z: https://dl.acm.org/doi/10.1145/390013.808479.

[8] Arnott, C. *How do compilers work 2 – Middle end* [online]. Březen 2017 [cit. 2023-04-19]. Dostupné z: https://medium.com/@ChrisCanCompute/how-do-compilers-work-2-middle-end-c4c8cff80b90.

[9] Beshkov, M. *What is ASLR (Address Space Layout Randomization)?* [online]. Březen 2023 [cit. 2023-04-19]. Dostupné z: https://www.wallarm.com/what/what-is-aslr-address-space-layout-randomization.

[10] Clarke, L., Podgurski, A., Richardson, D. a Zeil, S. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*. 1989, sv. 15, č. 11, s. 1318–1332. DOI: 10.1109/32.41326.

[11] Dejdar, J. *Podpora pro tvorbu semestrální práce z předmětu Programovací jazyky a překladače*. 2017. Bakalářská práce. České vysoké učení technické v Praze. Vypočetní a informační centrum. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/69680/F8-BP-2017-Dejdar-Jan-thesis.pdf?sequence=1&isAllowed=y.

[12] Filaret, I. *Using code instrumentation for debugging and constraint checking.* Pullman, Wash. :, 2009. Diplomová práce. Washington State University. Dostupné z: https://rex.libraries.wsu.edu/esploro/outputs/graduate/Using-code-instrumentation-for-debugging-and/9990052516220184 2#file-0.

[13] GeeksForGeeks. *Intermediate Code Generation in Compiler Design* [online]. [cit. 2023-04-27]. Dostupné z: https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/.

[14] Hjort Blindell, G. Inside the Compiler. In: Hjort Blindell, G., ed. *Instruction Selection : Principles, Methods, & Applications.* 2016, s. 4–5. ISBN 978-3-319-34019-7. Dostupné z: https://www.diva-portal.org/smash/get/diva2:951540/FULLTEXT01.pdf.

[15] Ibáñez, R. F. *A simple plugin for GCC – Part 2* [online]. Srpen 2015 [cit. 2023-05-01]. Dostupné z: https://thinkingeek.com/2015/08/16/simple-plugin-gcc-part-2/.

[16] Kempf, T., Karuri, K. a Gao, L. Software Instrumentation. In: *Wiley Encyclopedia of Computer Science and Engineering.* John Wiley & Sons, Ltd, 2008, s. 1–11. ISBN 9780470050118. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470050118.ecse386.

[17] Larson, P., Hinds, N., Ravindran, R. a Franke, H. Improving the Linux Test Project with kernel code coverage analysis. In: Andrew J. Hutton, C. C. R., ed. *Proceedings of the Linux Symposium.* Ottawa: [b.n.], červenec 2003, s. 1–12.

[18] Leupers, R. *Intermediate Representation* [online]. [cit. 2023-04-27]. Dostupné z: https://www.lancecompiler.com/intermediate.htm.

[19] Li, N., Praphamontripong, U. a Offutt, J. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage. In: *2009 International Conference on Software Testing, Verification, and Validation Workshops.* April 2009, s. 220–229.

[20] Liu, D. *15.7 Application: Control Flow Graphs* [online]. Computer Science University of Toronto [cit. 2023-04-18]. Dostupné z: https://www.cs.toronto.edu/~david/course-notes/csc110-111/15-graphs/07-control-flow-graphs.html.

[21] Paul Ammann, J. O. Coverage Criteria. In: Paul Ammann, J. O., ed. *Introduction to Software Testing.* 2. vyd. Cambridge: Cambridge University Press, 2017, kap. 2.4. ISBN 978-1-107-17201-2.

[22] Richard M. Stallman, G. D. C. *GNU Compiler Collection Internals.* 2023 [cit. 2023-20-04]. Dostupné z: https://gcc.gnu.org/onlinedocs/gccint.pdf.

[23] Schaefer, F.-R. *Lexical Analysis At Your Fingertips* [online]. Prosinec 2015 [cit. 2023-04-18]. Dostupné z: https://www.codeproject.com/Articles/1067166/Lexical-Analysis-At-Your-Fingertips.

[24] Smrčka, A. *Automatizované testování a dynamická analýza.* 2023. Texty k přednáškám.

[25] Standardization, I. O. for. *Programming languages – C.* Standard. Prosinec 2010.

[26] Technology, B. T. *BullseyeCoverage - C++ Code Coverage Tool* [online]. [cit. 2023-04-22]. Dostupné z: https://www.bullseye.com/index.html.

[27] Thain, D. *Introduction to compilers and language design.* 2. vyd. 2020. 247 s. ISBN 979-8-655-18026-0.

[28] "ValgrindDevelopers". *Valgrind: About* [online]. [cit. 2023-04-25]. Dostupné z: https://valgrind.org/info/about.html.

[29] Wicht, B. *Cache-Friendly Profile Guided Optimization.* Fribourg, 2013. Disertační práce. University of Applied Science. Dostupné z: https://www.researchgate.net/publication/307545338_Cache-Friendly_Profile_Guided_Optimization.

# Příloha A

# Výstupy ladicích funkcí

```
1   #include <stdio.h>
2   #include "callbacks.h"
3
4   int main() {
5       int a = 1, b = 2, c = 3, x;
6       x = a - c * b / a;
7       printf("%d\n", x);
8
9       return 0;
10  }
```

Výpis A.1: Kód, na němž jsou ukázany výstupy ladicích funkcí GCC.

Kód A.1, respektive jeho AST lze pomocí funkce `debug` vytisknout do podoby trochu modifikovaného zdrojového textu, který ale zobrazuje veškeré uzly, které tam syntaktická analýza dodala:

```
{
    int a = 1;
    int b = 2;
    int c = 3;
    int x;

        int a = 1;
        int b = 2;
        int c = 3;
        int x;
    x = a - (c * b) / a;
    printf ((const char * restrict) "%d\n", x);
    return 0;
}
return 0;
```

Výpis A.2: Výpis po použití funkce `debug`.

```
<statement_list 0x7f122fbfa820
    type <void_type 0x7f122fad7f18 void VOID
        align:8 warn_if_not_align:0 symtab:0 alias-set -1 canonical-type
        0x7f122fad7f18 pointer_to_this <pointer_type 0x7f122fadf000>>
    side-effects head 0x7f122fc6c708 tail 0x7f122fc6c720 stmts
    0x7f122fc8cc30 0x7f122fbfade0

    stmt <bind_expr 0x7f122fc8cc30 type <void_type 0x7f122fad7f18 void>
        side-effects
        vars <var_decl 0x7f1230830ea0 a type <integer_type
        0x7f122fad75e8 int> used read SI myprog.c:17:9
            size <integer_cst 0x7f122fabeee8 constant 32>
            unit-size <integer_cst 0x7f122fabef00 constant 4>
            align:32 warn_if_not_align:0 context <function_decl
            0x7f122fc97000 main> initial <integer_cst 0x7f122fadd060 1>
            chain <var_decl 0x7f1230830f30 b>>
        body <statement_list 0x7f122fbfa840 type <void_type
            0x7f122fad7f18 void>
            side-effects head 0x7f122fc6c648 tail 0x7f122fc6c6f0 stmts
            0x7f122fbfaca0 0x7f122fbfacc0 0x7f122fbface0 0x7f122fbfad00
            0x7f122fc94780 0x7f122fad21c0 0x7f122fbfadc0

            stmt <decl_expr 0x7f122fbfaca0 type <void_type 0x7f122fad7f18
            void> side-effects arg:0 <var_decl 0x7f1230830ea0 a>
                myprog.c:17:9 start: myprog.c:17:9 finish:
                myprog.c:17:9>
            stmt <decl_expr 0x7f122fbfacc0 type <void_type 0x7f122fad7f18
            void> side-effects arg:0 <var_decl 0x7f1230830f30 b>
                myprog.c:17:16 start: myprog.c:17:16 finish:
                myprog.c:17:16>
            stmt <decl_expr 0x7f122fbface0 type <void_type 0x7f122fad7f18
            void> side-effects arg:0 <var_decl 0x7f122fc98000 c>
                myprog.c:17:23 start: myprog.c:17:23 finish:
                myprog.c:17:23>
            stmt <decl_expr 0x7f122fbfad00 type <void_type 0x7f122fad7f18
            void> side-effects arg:0 <var_decl 0x7f122fc98090 x>
                myprog.c:17:30 start: myprog.c:17:30 finish: myprog.c:17:30>
            stmt <modify_expr 0x7f122fc94780 type <integer_type
                0x7f122fad75e8 int>
                side-effects arg:0 <var_decl 0x7f122fc98090 x>
                arg:1 <minus_expr 0x7f122fc94758 type
                    <integer_type 0x7f122fad75e8 int>
                    arg:0 <var_decl 0x7f1230830ea0 a>
                    arg:1 <trunc_div_expr 0x7f122fc94730 type
                        <integer_type 0x7f122fad75e8 int>

                        arg:0 <mult_expr 0x7f122fc94708 type
```

```
                    <integer_type 0x7f122fad75e8 int>
                    arg:0 <var_decl 0x7f122fc98000 c>
                    arg:1 <var_decl 0x7f1230830f30 b>
                    myprog.c:18:15 start: myprog.c:18:13 finish:
                    myprog.c:18:17>
                arg:1 <var_decl 0x7f1230830ea0 a>
                myprog.c:18:19 start: myprog.c:18:13
                finish: myprog.c:18:21>
            myprog.c:18:11 start: myprog.c:18:9
            finish: myprog.c:18:21>
        myprog.c:18:7 start: myprog.c:18:5 finish: myprog.c:18:21>
    stmt <call_expr 0x7f122fad21c0 type <integer_type
        0x7f122fad75e8 int>
        side-effects
        fn <addr_expr 0x7f122fbfad80 type <pointer_type
            0x7f122fc93b28>
            constant arg:0 <function_decl 0x7f122fb72a00 printf>
            myprog.c:19:5 start: myprog.c:19:5
            finish: myprog.c:19:10>
        arg:0 <nop_expr 0x7f122fbfada0 type <pointer_type
            0x7f122fc29dc8>
            readonly constant
            arg:0 <addr_expr 0x7f122fbfad40 type <pointer_type
                0x7f122fc93a80>
                readonly constant
                arg:0 <string_cst 0x7f122fbfad20 type <array_type
                    0x7f122fc25348>
                    readonly constant static "%d\012\000">
                myprog.c:19:12 start: myprog.c:19:12
                finish: myprog.c:19:17>
            myprog.c:19:12 start: myprog.c:19:12
            finish: myprog.c:19:17>
            arg:1 <var_decl 0x7f122fc98090 x>
        myprog.c:19:5 start: myprog.c:19:5 finish: myprog.c:19:21>
    stmt <return_expr 0x7f122fbfadc0 type <void_type
        0x7f122fad7f18 void> side-effects
        arg:0 <modify_expr 0x7f122fc947f8 type <integer_type
            0x7f122fad75e8 int>
            side-effects arg:0 <result_decl 0x7f122fc4bd20 D.2899>
            arg:1 <integer_cst 0x7f122fadd048 constant 0>
            myprog.c:39:12 start: myprog.c:39:12
            finish: myprog.c:39:12>
        myprog.c:39:12 start: myprog.c:39:12
        finish: myprog.c:39:12>>
block <block 0x7f122fbeea20 used vars <var_decl 0x7f1230830ea0 a>
supercontext <function_decl 0x7f122fc97000 main>>
myprog.c:16:12 start: myprog.c:16:12 finish: myprog.c:16:12>
stmt <return_expr 0x7f122fbfade0 type <void_type 0x7f122fad7f18 void>
```

```
side-effects
arg:0 <modify_expr 0x7f122fc94820 type <integer_type
    0x7f122fad75e8 int>
    side-effects arg:0 <result_decl 0x7f122fc4bd20 D.2899>
    arg:1 <integer_cst 0x7f122fadd048 0>
    <built-in>:0:0 start: <built-in>:0:0 finish: <built-in>:0:0>
<built-in>:0:0 start: <built-in>:0:0 finish: <built-in>:0:0>>
```

Výpis A.3: Výpis kódu A.1 po použití funkce `debug_tree`.

# Příloha B

# Příklad výstupu instrumentovaného programu

```c
#include <stdio.h>
#include "callbacks.h"

int main() {

    int a = 200;
    int b = 100;
    int c = 50;

    if (a < b) {
        printf("%d\n", a);
    }
    else if (a > c) {
        printf("%d\n", c);
    }
    else {
        printf("%d\n", b);
    }

    return 0;
}
```

Výpis B.1: Demonstrační kód pro předvedení výstupu po jeho instrumentaci.

Pro správnou funkci je třeba v instrumentovaném programu používat hlavičkový soubor `callbacks.h`, který obsahuje deklarace instrumentačních funkcí, které jsou implementovány v souboru `callbacks.c`. Výpisy, které instrumentovaný program vyprodukuje, jdou poté na standardní chybový výstup.

```
void _instr_log_memory_reading(void *varAddress, const
   char *filename, int line, int type) {

   // Get the value in the correct format
   char *valueToPrint = _instr_getValueFromType(type, varAddress);
   char *envVariable = getenv("OUTPUT_FORMAT");
   if (envVariable == NULL ||
       strcmp(envVariable, "plain") == 0 ||
       strcmp(envVariable, "json") != 0) {
       fprintf(stderr, "Reading from address %p value %s in file
           %s on line %d\n", varAddress, valueToPrint, filename, line);
   }
   else {
       fprintf(stderr, "{\"type\": \"reading\", \"address\": \"%p\",
           \"value\": %s, " "\"filename\": \"%s\",
           \"line\": %d}\n", varAddress, valueToPrint, filename, line);
   }
}
```

Výpis B.2: Implementace funkce, která se zavolá, dojde-li ke čtení z paměti. Podobným způsobem jsou implementovány i zbylé funkce jen s jinými parametry.

Jestliže spustíme program po instrumentaci bez specifikace proměnné prostředí `OUTPUT_FORMAT`, získáme následující výpis:

```
Entering basic block 2 in function main in file myprog.c on line 6
Entering function main in file myprog.c on line 6
Writing to address 0x7ffedcd4fa2c value 200 in file myprog.c on line 6
Writing to address 0x7ffedcd4fa28 value 100 in file myprog.c on line 7
Writing to address 0x7ffedcd4fa24 value 50 in file myprog.c on line 8
Reading from address 0x7ffedcd4fa2c value 200 in file myprog.c on line 10
Reading from address 0x7ffedcd4fa28 value 100 in file myprog.c on line 10
Entering basic block 4 in function main in file myprog.c on line 13
Reading from address 0x7ffedcd4fa2c value 200 in file myprog.c on line 13
Reading from address 0x7ffedcd4fa24 value 50 in file myprog.c on line 13
Entering basic block 5 in function main in file myprog.c on line 14
Calling function printf in file myprog.c on line 14
Reading from address 0x7ffedcd4fa24 value 50 in file myprog.c on line 14
Ending in else if block 50
Continuing in executing function main after function printf was called in
file myprog.c.c on line 14
Entering basic block 7 in function main in file myprog.c on line 20
Returning from function main in file myprog.c on line 20
Entering basic block 8 in function main in file myprog.c on line 0
```

Výpis B.3: Výpis událostí pro kód B.1 ve formátu prostého textu.

Pakliže spustíme instrumentovaný program s proměnnou `OUTPUT_FORMAT` nastavenou na `json`, získáme tento výpis:

```
{"type": "basic block entry", "index": 2, "function": "main",
"filename": "myprog.c", "line": 6}
{"type": "function entry", "function": "main",
"filename": "myprog.c", "line": 6}
{"type": "writing", "address": "0x7ffedcd4fa2c", "value": 200,
"filename": "myprog.c", "line": 6}
{"type": "writing", "address": "0x7ffedcd4fa28", "value": 100,
"filename": "myprog.c", "line": 7}
{"type": "writing", "address": "0x7ffedcd4fa24", "value": 50,
"filename": "myprog.c", "line": 8}
{"type": "reading", "address": "0x7ffedcd4fa2c", "value": 200,
"filename": "myprog.c", "line": 10}
{"type": "reading", "address": "0x7ffedcd4fa28", "value": 100,
"filename": "myprog.c", "line": 10}
{"type": "basic block entry", "index": 4, "function": "main",
"filename": "myprog.c", "line": 13}
{"type": "reading", "address": "0x7ffedcd4fa2c", "value": 200,
"filename": "myprog.c", "line": 13}
{"type": "reading", "address": "0x7ffedcd4fa24", "value": 50,
"filename": "myprog.c", "line": 13}
{"type": "basic block entry", "index": 5, "function": "main",
"filename": "myprog.c", "line": 14}
{"type": "function call", "function": "printf",
"filename": "myprog.c", "line": 14}
{"type": "reading", "address": "0x7ffedcd4fa24", "value": 50,
"filename": "myprog.c", "line": 14}
{"type": "continuation after function call", "called function": "printf",
"current function": "main", "filename": "myprog.c", "line": 14}
{"type": "basic block entry", "index": 7, "function": "main",
"filename": "myprog.c", "line": 20}
{"type": "function return", "function": "main", "filename":
"myprog.c", "line": 20}
{"type": "basic block entry", "index": 8, "function": "main",
"filename": "myprog.c", "line": 0}
```


Výpis B.4: Výpis událostí pro kód B.1 ve formátu JSON.