Web-basierte Systeme

07: WebAssembly

Wintersemester 2024

Rüdiger Kapitza

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Vorläufiger Vorlesungsplan

16. Oktober	Einführung und Darstellung von Webseiten
23. Oktober	HTML und CSS
30. Oktober	Hypertext Transfer Protocol
6. November	Browser Schnittstellen
13. November	**Kommunikationsschnittstellen im Browser**
20. November	WebAssembly
27. November	Architektur moderner Browser
4. Dezember	Clientseitige Architekturmuster und serverseitige Implementierung von Web-basierten Systemen
11. Dezember	Vorbereitung Papieranalyse
8. Januar	Papieranalyse
15. Januar	Lastverteilung durch Zwischenspeicher
22. Januar	Web3
29. Januar	Aspekte von Web Sicherheit
5. Februar	Zusammenfassung und Ausblick

WebAssembly

Zielsetzung der Lerneinheit

- Ermittlung der Motivation für die Entwicklung von WebAssembly
- Bestandsaufnahme bisheriger Ansätze für clientseitigen Code
- Grundkenntnisse der Struktur & Funktionsweise von WASM
- Verständnis wie WebAssembly eingebunden wird

Motivation

- Webanwendungen werden immer komplexer & anspruchsvoller
  - Interaktive Visualisierung von 3D Objekten
  - Audio- und Videosoftware
  - Spiele!
- Obwohl JavaScript im Laufe der Jahre an Geschwindigkeit gewonnen hat, ist die Ausführungsgeschwindigkeit oft deutlich langsamer als bei nativem Programmcode!

Status Quo clientseitiger Code

- JavaScript aktuell/bisher alternativlos
  - Ursprünglich schlechte Performanz wurde durch Just-in-Time-Compiler (JIT-Compiler) verbessert
  - Der Einsatz eines JIT-Compilers kostet aber auch Zeit und teilweise müssen die Laufzeitoptimierungen wieder verworfen werden
- Alternative Programmiersprachen oder Codeformate (mittels Plug-Ins) konnten sich nicht durchsetzen
  - Vgl. ActiveX, Java und Flash
- Letzter Trend: Übersetzung von anderen Sprachen in JavaScript
  - Besseren Sprachkonzepte in der Ausgangssprache
    - z.B. TypeScript - für bessere Typüberprüfung und schnellere Fehlererkennung beim Programmieren
  - Höhere Ausführungsgeschwindigkeit
    - Transcompiler wandelt C/C++ in `asm.js` um, eine Untermenge von JavaScript, die eine bessere (Vor)optimierung ermöglicht

WebAssembly

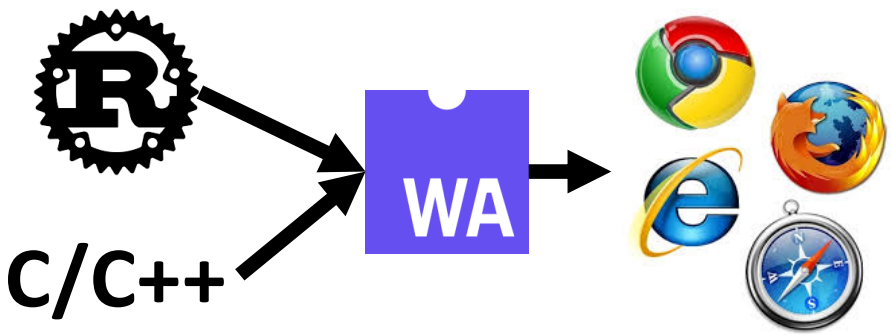
Was ist WebAssembly?

- WebAssembly stellt ein **sicheres, schnell ausführbares, portables Bytecode-Format für Webanwendungen** dar
- Sicherheit wurde bisher durch Laufzeitumgebungen für entsprechende Programmiersprachen erzielt, diese sind aber nicht auf portablen *low-level* Code ausgerichtet
  - Bsp.: C/C++ Binärcode ist schnell aber schwerer *abzusichern*
- Die Performance von *ahead-of-time* kompiliertem Code ist oft nicht vergleichbar mit Programmiersprachen, die in einer Laufzeitumgebung (mit Sandbox) ausgeführt werden
- Portabilität & determ. Verhalten ist wichtig und folglich muss das Format hardware- und plattformunabhängig sein!
- Ein kompaktes Format ist für das schnelle Laden von Code für komplexe Anwendungen unerlässlich

WebAssembly

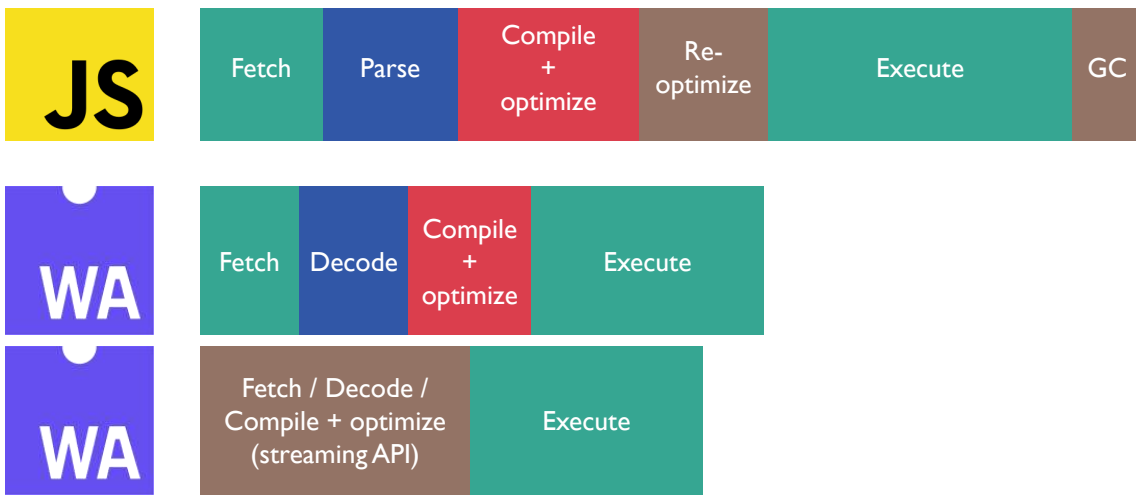
Was ist WebAssembly?

- WebAssembly realisiert eine virtuelle ISA (Instruction Set Architecture), diese kann das Übersetzungsziel verschiedener Programmiersprachen sein
- Beliebt und gut unterstützt: C/C++ und Rust
  - Sprachen mit statischer Speicherverwaltung!



WebAssembly versus JavaScript: Laden und ausführen

- Im Vergleich zu JavaScript [1] kann WebAssembly schneller geladen und ausgeführt werden
- Auch ein nebenläufiges Laden und Dekodieren ist möglich!



[1] Phasen laufen teilweise mehrfach und nebenläufig ab.

Codeformat(e)

- Low-level Bytecode-Format mit **binär und textueller Darstellung**
- Verlustfrei ineinander überführbar
- Typen: 32-bit und 64-bit Integer werden unterstützt (i32, i64), sowie Fließkommazahlen (f32, f64)

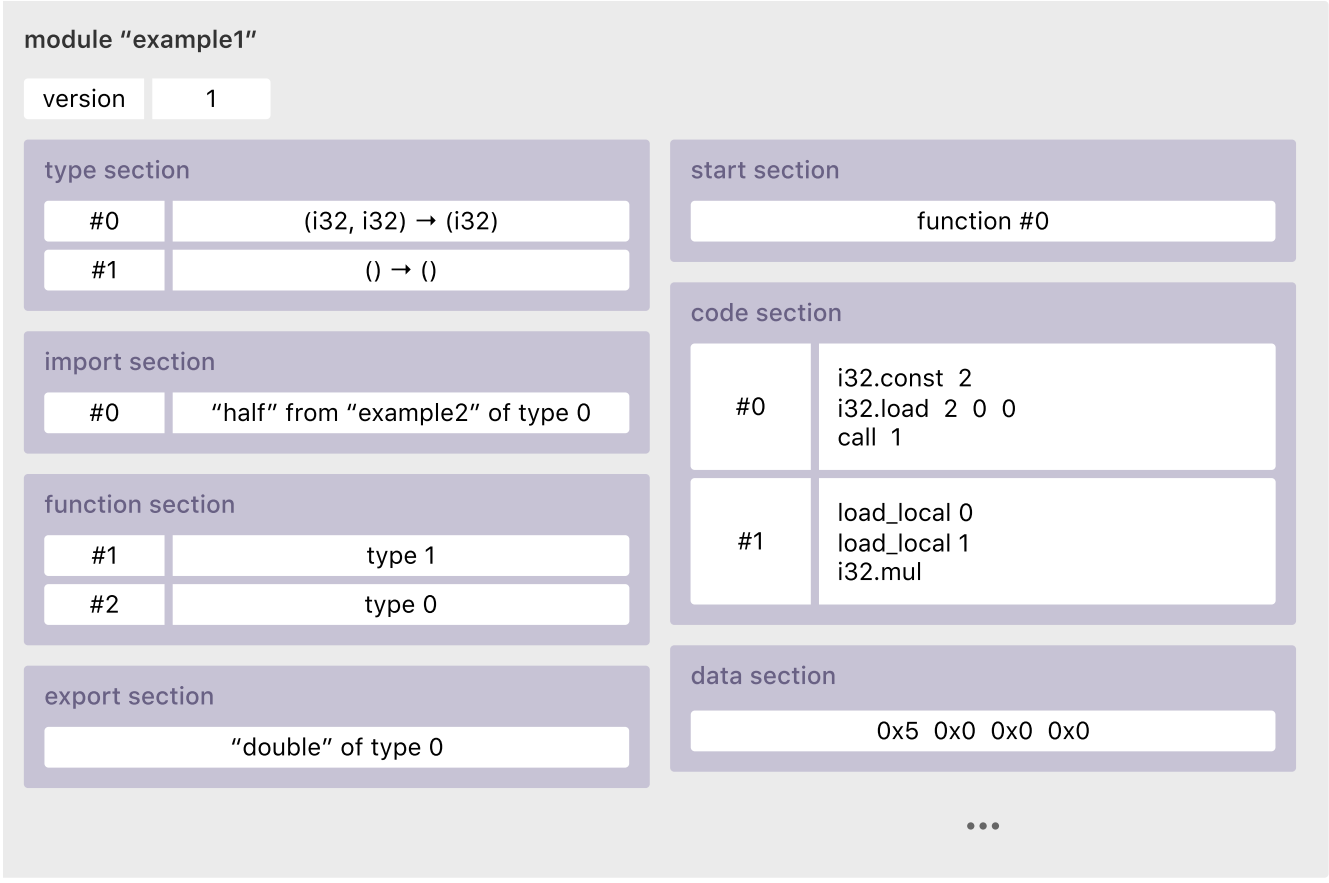
Binärformat

```
0061 736d 0100 0000 0187 8080 8000 0160
027f 7f01 7f03 8280 8080 0001 0004 8480
8080 0001 7000 0005 8380 8080 0001 0001
0681 8080 8000 0007 9080 8080 0002 066d
656d 6f72 7902 0003 6164 6400 000a 8d80
8080 0001 8780 8080 0000 2001 2000 6a0b
```

Textformat

```
1  (module
2    (table 0 anyfunc)
3    (memory $0 1)
4    (export "memory" (memory $)
5    (export "add" (func $add))
6    (func $add (; 0 ;) (param $0 i32) (param $1 i32) (
           result i32)
7      (i32.add
8        (get_local $1)
9        (get_local $0)
10     )
11   )
12 )
```

Anatomie eines WebAssembly Moduls (entspricht oft einem Program)



Anatomie eines WebAssembly Moduls

1. **Type** — Deklaration der Funktionssignaturen
   - Enthält alle im Modul verwendeten Signaturen

```
(i32 i32 -> i32)  // func_type #0
(i64 -> i64)      // func_type #1
( -> )            // func_type #2
```

2. **Import** — Importierte Funktionen
   - Führt alle externen Abhängigkeiten auf
   - Modulname sowie der Feldname & Funktionstyp werden genannt

```
("dumb-math", "quadruple", (func_type 1))   // func #0
("dumb-math", "pi", (global_type i64 immutable))
```

   - Es ist Aufgabe des Browsers (WASM-Laufzeitumge.), die Abhängigkeiten aufzulösen und ebnet den Weg für dynamisches Linken
   - Ermöglicht die Einbindung anderer Module, aber auch die Nutzung externer Funktionen (z.B. JavaScript)

Anatomie eines WebAssembly Moduls

3. **Function** — Funktionsdeklarationen
   - Definiert den Index der verwendeten Funktionen, die durch das Codesegment implementiert werden.
   - Index beginnt unter Aufschlag der Anzahl der importierten Funkt.
   - Index (`func #N`) wird später verwendet um Funktion aufzurufen

```
1  (func_type 1)  // func #1
2  (func_type 1)  // func #2
3  (func_type 0)  // func #3
```

4. **Table** — Indirekte Funktionstabelle und andere Tabellen
   - Enthält eine oder mehrere Tabellen
   - Tabellen enthalten Elemente, auf die von WASM aus nicht direkt zugegriffen werden kann
     - Beispiele: JavaScript-Objekte und Dateideskriptoren
   - Hier wird eine Verbindung zwischen dem low-level (& nicht vertrauenswürdigen) linearen Speicher und den Hochsprachenfunktionen und -referenzen hergestellt

Anatomie eines WebAssembly Moduls

5. **Memory**
   - Zuweisung von Speicherplatz für das Modul
   - Es wird die initiale Größe & die maximale Größe angegeben
6. **Global** — Globale Deklarationen
   - Es werden alle globalen Variablen deklariert.
   - Vgl. `static` im Falle von C/C++

Anatomie eines WebAssembly Moduls

7. **Export** — exportierte Funktionen
   - Es werden alle extern zugänglichen Elemente des Module definiert

```
("half" (func 1))
```

   - In diesem Fall kann `half` in Kombination mit der function- und der type-Sektion kombiniert werden zu:

```
function half(arg0 :int64) :int64
```

   - Neben Funktionen können auch noch Tabellen, Speichersegmente und globale Variablen exportiert werden

Anatomie eines WebAssembly Moduls

8. **Start** — Startfunktion des Moduls
   - Definiert eine Funktion, die beim Laden initial ausgeführt wird
   - Diese Funktion kann als *main()*-Funktion oder nur zur Initialisierung der globalen Variablen und der zuvor definierten Speicherbereiche verwendet werden
9. **Element**
   - Zuständig für die Initialisierung der von außen importierten oder zuvor definierten Tabellen

Anatomie eines WebAssembly Moduls

10. **Code** — Codesegment
    - Enthält den Code aller Funktionen des Moduls
    - Funktion sind der Reihenfolge nach angegeben wie in der Function-Sektion angegeben
    - Beispielfunktion: `half`

```
1       get_local 0  // push parameter #0 on stack (our dividend)
2       i64.const 2  // push constant int64 "2" on stack (our divisor)
3       i64.div_u    // unsigned division; pushes result onto stack
4       end          // ends function, resulting in one i64 (top of stack)
```

    - WebAssembly definiert eine abstrakte Stack-Engine, die gezielt auf eine Zielarchitektur abgebildet werden kann.
    - Es gibt eine Vielzahl von Standardoperatoren (z.B. `div` & `add`) aber auch spezifische wie `eqz` um zu testen ob ein Operand `0` ist

Anatomie eines WebAssembly Moduls

11. **Data** — Datensegement
    - Wird verwendet um importierte oder lokal definierte Speicherbereiche zu belegen

```
(data_segment
  0                          // linear memory index
  (init_expr (i32.const 4))  // byte offset at which to place the data
  (data 0x2a 0x0 0x0 0x0))
```

    - Das Beispiel belegt im Speicherbereich mit dem Index 0 die Bytes [4–8] mit einem unsignierten i32 Wert (42)

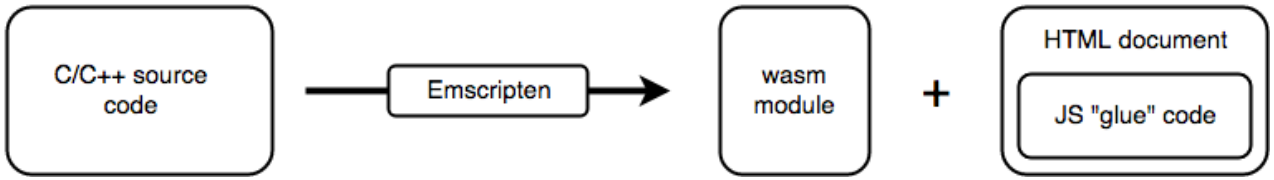
WebAssembly

Kernkonzepte für die Verbindung zwischen WASM & JavaScript

- **Module:** Stellt einen im Browser kompilierten binären Blob dar. Er ist zustandslos und kann zwischengespeichert werden.
- **Memory:** Stellt einen variablen `ArrayBuffer` dar, welcher durch die Basisinstruktionen von WebAssembly gelesen und geschrieben werden kann
- **Table:** Ein Array variabler Größe für Referenzen (z.B., um Funktionen zu verwalten). Zielsetzung ist Portabilität und Sicherheit
- **Instance:** Eine Instanz eines Moduls mit initialisiertem Zustand und aufgelösten Importen

Übersetzen und Ausführen

- Verschiedene Wege möglich, hier mittels Emscripten[2]



1. Emscripten reicht die C/C++ Dateien intern an clang+LLVM weiter
2. clang+LLVM generiert eine Zwischenrepräsentation welche Emscripten in eine `.wasm`-Binärdatei umsetzt
3. WebAssembly kann bisher nicht direkt auf das DOM zugreifen, sondern nur Funktionen in JavaScript aufrufen und dabei Integer- und Fließkommazahlen austauschen.
   - Es können nur über JavaScript Web APIs aufgerufen werden.
   - Emscripten erzeugt hierfür die nötigen Hilfsfunktionen

[2] `http://kripken.github.io/emscripten-site/index.html`

Übersetzen und Ausführen

- Emscripten stellt eine Reihe populäre C/C++ Bibliotheken bereit
  - Z.B. SDL, OpenGL, OpenAL und einen Teil der Posix-Schnittstelle
  - Teile dieser Bibliotheken müssen mit den Web APIs des Browsers verknüpft werden
- Hilfsfunktionen müssen auch zum Laden und Einbinden der Bibliotheken bereitgestellt werden
- Bspw. werden Ausgaben über `stdout` in ein `<textarea>` umgeleitet
- Im Falle von OpenGL wird entsprechend ein `<canvas>` genutzt

WebAssembly

Direkte Erstellung eigener WebAssembly-Module

- (Die Textform kann direkt in die Binärform umgesetzt werden)
- Damit ist es prinzipiell möglich, WebAssembly-Module direkt aus JavaScript zu generieren, zu kompilieren und auszuführen

Einfaches Beispielprogramm

- **hello.c** als Klassiker

```c
#include <stdio.h>

int main(int argc, char ** argv) {
  printf("Hello World\n");
}
```

- Verwendung der Emscripten-Werkzeugkette

```
emcc hello.c -o hello.html
```

- `-o hello.html` erstellt eine einfach zu verwendende Vorlage mit Hilfsfunktionen
- Als Ergebnis werden drei Ausgabedateien erstellt: `hello.wasm/.js/.html`

Aufruf einer WebAssembly-Funktion

- Um Funktionen aus JavaScript verfügbar zu machen müssen sie entsprechend markiert werden via `EMSCRIPTEN_KEEPALIVE`

```c
#include <stdio.h>
#include <emscripten/emscripten.h>
int main(int argc, char ** argv) {
    printf("Hello World\n");
}
#ifdef __cplusplus
extern "C" {
#endif
EMSCRIPTEN_KEEPALIVE void myFunction() {
  printf("MyFunction Called\n");
}
#ifdef __cplusplus
}
#endif
```

- Damit die Funktionen des Moduls zur Verfügung stehen, müssen sie mit `EXPORTED_FUNCTIONS` und `EXPORTED_RUNTIME_METHODS=ccall` übersetzt werden.

Aufruf einer WebAssembly-Funktion

- Nun muss noch die Funktion aus JavaScript aufgerufen werden – hier als Beispiel, wenn ein Button ausgelöst wird:

```
1  document.querySelector('.mybutton').addEventListener('click', function(){
2    alert('check console');
3    var result = Module.ccall('myFunction', // name of C function
4                              null, // return type
5                              null, // argument types
6                              null); // arguments
7  });
```

Laden eines WebAssembly-Modules

- Im Prinzip gibt es zwei Vorgehensweisen - entweder klassisch Laden und Übersetzen

```
fetch('simple.wasm').then(response =>
  response.arrayBuffer()
).then(bytes =>
  WebAssembly.instantiate(bytes, importObject)
).then(results => {
  results.instance.exports.exported_func();
});
```

- ...oder Verwendung der neueren Streaming-API

```
WebAssembly.instantiateStreaming(fetch('simple.wasm'), importObject)
.then(obj => obj.instance.exports.exported_func());
```

- Der Rückgabewert von `WebAssembly.instantiate()` liefert

```
{
  module : Module // The newly compiled WebAssembly.Module object,
  instance : Instance // A new WebAssembly.Instance of the module object
}
```

Verwendung von Speicher in WebAssembly

- Jedes Modulinstanz hat seinen *eigenen* Speicherbereich, dessen Adressierung bei der Adresse 0 beginnt
- Zielsetzung ist eine bessere Isolation der Modulinstanzen
- Aus Sicht von JavaScript kann der Speicher einer Modulinstanz als `ArrayBuffer` variabler Größe betrachtet werden
- Man kann den Speicher für WebAssembly auch in JavaScript erzeugen via `WebAssembly.Memory()`
  - Zwei Parameter können übergeben werden: initiale & maxi. Größe
  - Es wird dabei in Vielfachen von 64KB allokiert
  - Beispiel:

```
1  var memory = new WebAssembly.Memory({initial:10, maximum:100});
```

Verwendung von Speicher in WebAssembly

- Auf den Speicher einer Modulinstanz kann man mittels einfacher getter/setter-Funktionen zugreifen

```
1  // Set 42 ...
2  new Uint32Array(memory.buffer)[0] = 42;
3  // and get it from ...
4  new Uint32Array(memory.buffer)[0]
```

- Der allokierte Speicher kann bis zur maximalen Kapazität erhöht werden – darüber hinaus wird eine `WebAssembly.RangeError` Exception ausgelöst

```
1  memory.grow(1);
```

Verwendung von Speicher in WebAssembly

- Im Prinzip kann Speicher in beiden Welten erzeugt und jeweils mit der anderen Seite ausgetauscht werden
- Beispiel in JavaScript:

```
<script>
var memory = new WebAssembly.Memory({initial:10, maximum:100});
WebAssembly.instantiateStreaming(fetch('memory.wasm'), { js: { mem: memory } })
.then(obj => {
  var i32 = new Uint32Array(memory.buffer);
  for (var i = 0; i < 10; i++) {
    i32[i] = i;
  }
  var sum = obj.instance.exports.accumulate(0, 10);
  console.log(sum);
});
</script>
```

Verwendung von Speicher in WebAssembly

- Warum ist es sinnvoll den Speicher im Prinzip von beiden Seiten aus bereitstellen zu können?
  - Man kann wie im Beispiel den Speicher in JavaScript initialisieren während das Modul noch geladen und kompiliert wird
  - Man kann Speicher auch in verschiedene Modulinstanzen importieren (z.B. für dynamisches Linken)

Verwendung von Tabellen

- Tabellen sind aus Sicherheitsgründen der einzige Mechanismus um Funktionen aufzurufen
- Ähnlich zur Manipulation von Speicher können auch Tabellen in JavaScript ausgelesen, verändert oder vergrößert werden
- Auch diese Funktionalität wird für das dynamische Linken benötigt

Zusammenfassung

- WebAssembly ist ein portables, plattformunabhängiges Bytecodeformat speziell für den Einsatz im Web.
  - Es wird derzeit von allen gängigen Browsern unterstützt.
  - Es ermöglicht den Einsatz von neuen Programmiersprachen im Web und die Nutzung von existierendem Code
  - Durch Ahead-Of-Time Übersetzung und die Struktur des Formats kann WebAssembly schnell geladen und ausgeführt werden
- Trotz breiter Verfügbarkeit steckt das Format noch im Anfang
  - Bspw. Multithreading?
- Weitere Anwendungszwecke ausserhalb des Webs sind auf dem Vormarsch: Edge Computing, Cloud und IoT

Referenzen

- `https://developer.mozilla.org/en-US/docs/WebAssembly`
- `https://webassembly.github.io/spec/`

Literatur

[1] Andreas Haas et al. **“Bringing the Web Up to Speed with WebAssembly”.** In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, S. 185–200.