Compiler 2

1. Block: OO-Sprachen am Beispiel von Bantam

TECHNISCHE UNIVERSITÄT DARMSTADT

TECHNISCHE UNIVERSITÄT DARMSTADT

Organisatorisches

Material: Grundlage der Vorlesung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Jeweils auszugsweise
  Engineering a Compiler, 2. Auflage
  von Keith Cooper und Linda Torczon, MKP/Elsevier 2011

  Advanced Compiler Design and Implementation
  von Steven Muchnick, Morgan-Kaufman 1997

- Ausgewählte wissenschaftliche Veröffentlichungen (“papers”)
  - Werden auf Web-Seite im TU Netz bereitgestellt

Praktikum Optimierende Compiler

Begleitend zur Vorlesung Compiler 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wird in Dreiergruppen durchgeführt
  - ...die sich in der Regel selbst zusammenfinden
  - Erste Abgabe konfiguriert Gruppe
- Besuch zum Verständnis der Theorie dringend empfohlen
- Beginn 2. oder 3. Vorlesungswoche

Praktikum Optimierende Compiler

Begleitend zur Vorlesung Compiler 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wird in Dreiergruppen durchgeführt
  - ...die sich in der Regel selbst zusammenfinden
  - Erste Abgabe konfiguriert Gruppe
- Besuch zum Verständnis der Theorie dringend empfohlen
- Beginn 2. oder 3. Vorlesungswoche

Diesmal besonders spannend (für alle Beteiligten), da neuer moderner Basis-Compiler verwendet wird

Bantam Java

TECHNISCHE UNIVERSITÄT DARMSTADT

- Eingabe: Bantam, eine Untermenge von Java
  - Eigene Web-Seite `http://www.bantamjava.com/`
- Ausgabe: MIPS Assembler
- Veranschaulicht Implementierungskonzepte von OO-Sprachen

Bantam Java

TECHNISCHE UNIVERSITÄT DARMSTADT

- Eingabe: Bantam, eine Untermenge von Java
  - Eigene Web-Seite `http://www.bantamjava.com/`
- Ausgabe: MIPS Assembler
- Veranschaulicht Implementierungskonzepte von OO-Sprachen

- Material auf ESA Web-Seite
  - Auszug aus den Quellen: Vollständige IR(s)
  - Umfassendes Handbuch (ca. 100 Seiten)
  - Vollständiges JavaDoc einer Referenzimplementierung
  - Modifizierter MIPS-Simulator SPIM
    - Neuer System Call 18: Aktuelle Zeit
    - Zählt ausgeführte Instruktionen
    - Quellen und Binaries (Linux, Windows-Cygwin) auf ESA Web Seite

Bantam Java

TECHNISCHE UNIVERSITÄT DARMSTADT

- Eingabe: Bantam, eine Untermenge von Java
  - Eigene Web-Seite `http://www.bantamjava.com/`
- Ausgabe: MIPS Assembler
- Veranschaulicht Implementierungskonzepte von OO-Sprachen

- Material auf ESA Web-Seite
  - Auszug aus den Quellen: Vollständige IR(s)
  - Umfassendes Handbuch (ca. 100 Seiten)
  - Vollständiges JavaDoc einer Referenzimplementierung
  - Modifizierter MIPS-Simulator SPIM
    - Neuer System Call 18: Aktuelle Zeit
    - Zählt ausgeführte Instruktionen
    - Quellen und Binaries (Linux, Windows-Cygwin) auf ESA Web Seite

- Ist aber Premiere für alle Beteiligten
  ➥Viel Diskussion und gemeinsame Experimente im Forum auf `www.d120.de`

Aufgaben im Praktikum

TECHNISCHE UNIVERSITÄT DARMSTADT

- Erweitern des Referenz-Compilers um zusätzliche Funktionen
- Programmanalyse
- Optimierung
- Visualisierung

Bantam

Programmiersprache Bantam

TECHNISCHE UNIVERSITÄT DARMSTADT

- Java-artige objektorientierte Sprache
- Stark typisierte Sprache
  - Statische und dynamische Typprüfung

Programmiersprache Bantam

TECHNISCHE UNIVERSITÄT DARMSTADT

- Java-artige objektorientierte Sprache
- Stark typisierte Sprache
  - Statische und dynamische Typprüfung

- Eingebaute primitive Typen
  - `int` und `boolean`

Programmiersprache Bantam

TECHNISCHE UNIVERSITÄT DARMSTADT

- Java-artige objektorientierte Sprache
- Stark typisierte Sprache
  - Statische und dynamische Typprüfung
- Eingebaute primitive Typen
  - `int` und `boolean`
- Eingebaute Klassen
  - `Object`, `String`, `TextIO`, `Sys`

Programmiersprache Bantam

TECHNISCHE UNIVERSITÄT DARMSTADT

- Java-artige objektorientierte Sprache
- Stark typisierte Sprache
  - Statische und dynamische Typprüfung
- Eingebaute primitive Typen
  - `int` und `boolean`
- Eingebaute Klassen
  - `Object`, `String`, `TextIO`, `Sys`
- Programmausführung beginnt in
  - Klasse `Main`, Methode `main()`
  - Kurz: `Main.main()`

Bantam Extended

TECHNISCHE UNIVERSITÄT DARMSTADT

- Obermenge der einfachen Version Bantam Base, zusätzlich
  - Arrays
  - `break`
  - `++`, `--`

Bantam Extended

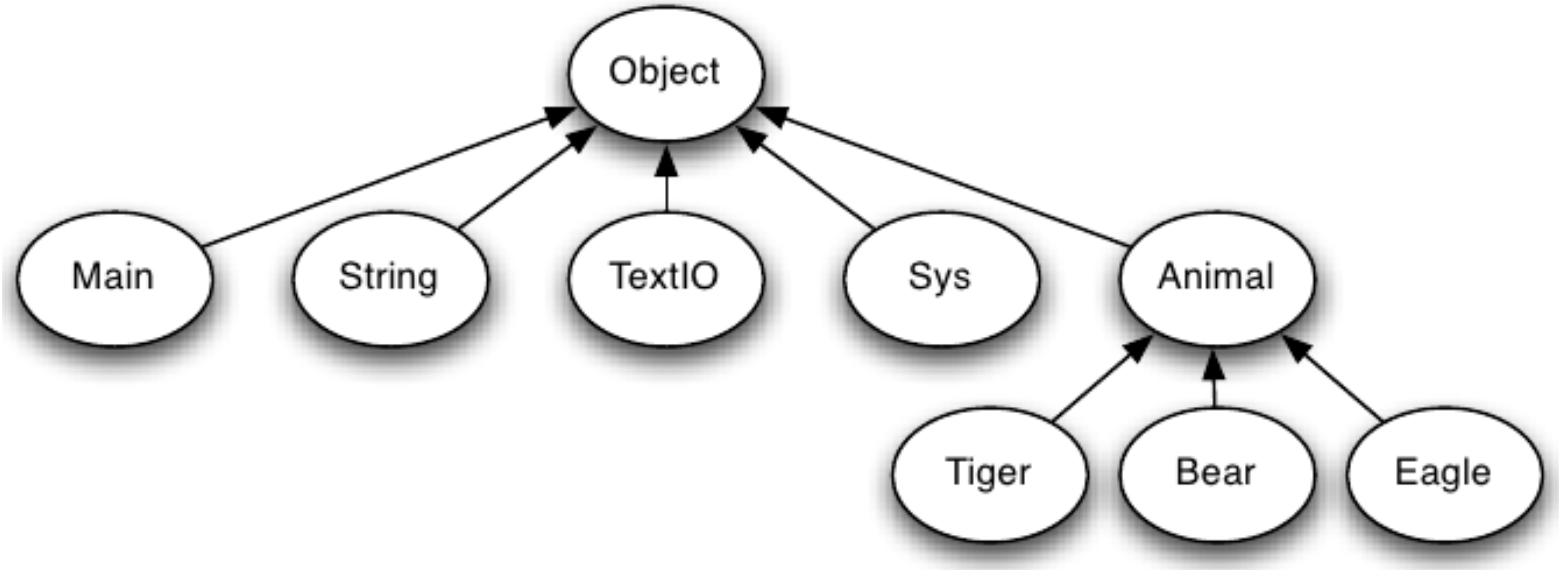
TECHNISCHE UNIVERSITÄT DARMSTADT

- Obermenge der einfachen Version Bantam Base, zusätzlich
  - Arrays
  - `break`
  - `++`, `--`
- Bei uns: Untermenge von Bantam Extended
  - Arrays
  - Kein `break`, `++`, `--`,
    - Sonst zu komplizierte Optimierung
  - Garbage Collection deaktiviert
    - Würde Instruktionszählung zu stark verfälschen

Vererbung in Bantam

TECHNISCHE UNIVERSITÄT DARMSTADT

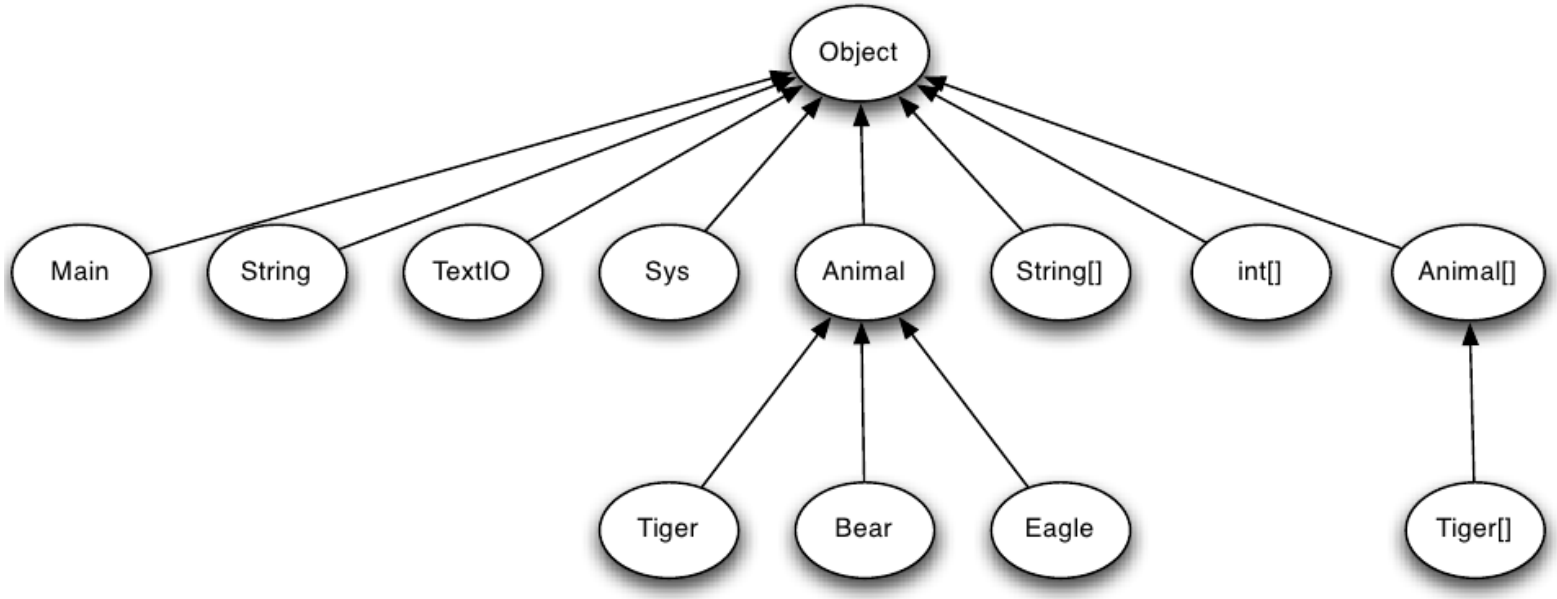
Einfache Vererbung, keine Interfaces



Sonderfall Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

Vererbungshierarchie der Basistypen wird auf verwendete Arrays angewandt



Array-Typen und ihre Vererbung werden automatisch angelegt

- Falls `Object[]` benutzt worden wäre ...
- ... hätte `Animal[]` von `Object[]` geerbt

Klassendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class <name> [ extends <parent> ] {
<members>
}
```

- Bezeichner beachten Groß/Kleinschreibung
- Keine Modifikatoren für Klassen
  - Kein `public`, `package`, `abstract`, `static`, `implements`
- Falls nicht anders angegeben, erben Klassen von `Object`

Bestandteile von Klassen: Attribute

Fields

TECHNISCHE UNIVERSITÄT DARMSTADT

```
<type> <name> [ = <expression> ] ;
```

- Definieren Typ und Namen der Attribute
- Optional: Ausdruck zur Initialisierung mit Startwert
  - Wenn fehlt: Initialisierung je nach Typ auf `0`, `false`, `null`

Beispiel: Attribute

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Animal {
  int      numLegs = 4;
  boolean canFly;              // false
  Animal  mate;                // null, rekursiver Typ
  int      strength = (numLegs * 10) + 50;
           // darf vorher definierte Attribute benutzen
  Object  obj = new Object();  // OK
  Animal  prey = new Animal(); // Illegal
}
```

- Initialisierungsausdrücke werden nur einmal bei Objekterzeugung ausgewertet
  - In Superklassen vor Subklassen
- Instanziierung von Objekten der eigenen Klasse oder von Subklassen illegal
  - Endloser Rekursiver Aufruf der Initialisierung
  - Führt bei Ausführung zum Abbruch wegen Speichermangel

Zugriff auf Attribute

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zugriff auf Attribute nur über `this` und `super`
  - Nicht über eine Objektreferenz, also nicht `mate.canFly = true`
- Keine Zugriffsmodifikatoren wie `public`, `static`, `final`
- Alle Attribute sind implizit `protected`
  - Zugreifbar aus Klasse selbst und ihren Unterklassen

Bestandteile von Klassen: Methoden

Methods

TECHNISCHE UNIVERSITÄT DARMSTADT

```
<type> <name> ( [ <parameters> ] ) {
   <statements>
   return [  <expression> ];
}
```

- Alle Methoden sind implizit `public`
- Kein Überladen von Methodennamen
  - Alle Methodennamen in einer Klasse müssen eindeutig sein
- Alle Methoden müssen als letzte Anweisung ein `return` haben
  - Auch solche, die den Rückgabetyp `void` haben!
  - Es darf nur genau ein `return` in jeder Methode geben
  - Härtere Anforderung als im “normalen” Bantam Java
  - Grund: Vereinfacht Optimierung (nur ein Ausgang aus Methode)

Beispiel: Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Animal {
  <member definitions>
  void fight (int amount, boolean isWinner) {
    if (isWinner)
      strength = strength + amount * 5;
    else
      strength = 0;
    return; // zwingend erforderlich
  }
}
```

Blöcke und Lokale Deklarationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Main {
    int x0 = 0;
    void main () {
        int a = x0;
        {b1
            int b = x0;
            int x1 = 1;
            int c = x1;
            {b2
                // this would be illegal:
                // int x2 = 2;
                int d = x1;
            }b2
        }b1
        {b3
            int e = x0;
            int x3 = 3;
            int f = x2;
        }3
    }
    return;
}
```

- Geltungsbereiche werden in {... } eingeschlossen
- Lokale Variablendeklarationen müssen immer einen Initialisierungsausdruck haben
- Lokale Variablen können Attribute gleichen Namens überlagern
  - Dann Zugriff auf überlagertes Attribut explizit über `this` möglich
- Lokale Variablen können sich nicht gegenseitig überlagern

Bedingte Ausführung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Main {
  void main () {
    ...
    if (animal.getStrength() > 100) {
      animal.fight(100, true);
      strength = animal.getStrength();
    }
    ...
  }
}
```

- Wie in Java, `else`-Zweig ist optional
- `else` bindet immer an innerstes noch offenes `if`
- Aber keine `switch`/`case`-Anweisung

`while`-Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Main {
  void main () {
    ...
    while (animal.getStrength() < 100)
      animal.fight(100, true);
    ...
  }
}
```

- Wie in Java, aber kein `break`/`continue`

`for`-Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( [ < initialization > ] ; [ <predicate> ] ; [ <update> ] )
  <statement>
```

- Ähnlich zu Java, aber kein `break`/`continue`
- Aber keine Deklaration neuer lokaler Variablen in `for`-Anweisung

```
// falsch!
for ( int i = 0; i < 10; i = i+1)
  sum = sum + i;
```

Anweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bestehen aus Untermenge von Ausdrücken
- Zuweisungen, Methodenaufrufe, instanziieren neuer Objekte mit `new`

```
x = y + 2;
rect.draw(COLOR_GREEN);
new Rectangle(20,30);
```

Ausdrücke: Zuweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

name = *expression*
name [*index*] = *expression*
objref . *name* = *expression*
objref . *name* [*index*] = *expression*

- Flexibilität der linken Seite eingeschränkt gegenüber Java
- Maximal eine Objektreferenz, nicht `shape.rgbcolor.red = 255`
  - Gültige Objektreferenzen sind nur `this` und `super`
- Wert des gesamten Ausdrucks ist *expression* auf rechter Seite
- Typprüfung
  - Bei primitiven Typen: Müssen identisch sein auf LHS und RHS
  - Bei Instanzen von Klassen: Klassen müssen kompatibel sein
    - Identisch
    - oder LHS Superklasse von RHS

Ausdrücke: Methodenaufruf

dynamic dispatch

TECHNISCHE UNIVERSITÄT DARMSTADT

methname (*actparameters*)
objref . *methname* (*actparameters*)

- Wenn keine Objektreferenz angegeben wird `this` angenommen
  - Auch möglich: `super`, um Suche nach Methode in Superklasse zu beginnen
  - Falls Objektreferenz `null`, Exception `_null_pointer_error` auslösen
- *methname* muß in Klasse/Superklasse der Objektreferenz definiert sein
- *actparameters* müssen zu formalen Parametern der Methode passen
- Parameterübergabe
  - Von rechts nach links
  - Als *call-by-value*
- Polymorphismus wird unterstützt
  - Laufzeittyp von *objref* bestimmt, welche Implementierung von *methname* ausgeführt wird

Ausdrücke: Anlegen neuer Objekte mit `new`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Animal {
  <member definitions>
  // Simuliert einen Konstruktor
  Animal init(int l, boolean f) {
    numLegs = l;
    canFly = f;
    return this;
  }
}
class Main {
  void main() {
    Animal animal = (new Animal()).init(4, false);
    return;
  }
}
```

- Anlegen immer mit `new` *Klassenname* `()`
  - Beachte: Keine Parameter für `new`, da keine Konstruktoren existieren
- Konstruktorverhalten manuell nachbilden durch Init-Methode
  - Wird aber nicht automatisch aufgerufen!

Ausdrücke: Anlegen von Arrays mit `new`

TECHNISCHE UNIVERSITÄT DARMSTADT

`new` *basetype* `[` *size* `]`

- Legt neues Array für *size* Elemente des Typs *basetype* an
- Initialisierung je nach Typ auf `0`, `false`, `null`
- Objekte müssen einzeln neu angelegt und ins Array eingetragen werden
- Einschränkungen in Bantam
  - Nur eindimensionale Arrays, keine Arrays-von-Arrays
  - Wenn Garbage Collection aktiviert ist, max 1500 Elemente je Array

Beispiel

```
int[] numArray = new int[10];
```

Ausdrücke: Typkonvertierung

TECHNISCHE UNIVERSITÄT DARMSTADT

(*targetclass*) (*expression*)

- Beachte: *expression* muß immer geklammert sein
- *targetclass* muß typkompatibel zu dynamischem Typ von *expression* sein
  - Identisch
  - Oder Superklasse
- Beachte: Nur Objektreferenzen können typkonvertiert werden
  - Keine primitiven Typen wie `boolean` oder `int`

Beispiel: Typkonvertierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Upcasts, funktionieren immer
Animal animal = new Tiger();
Animal animal = (Animal)(new Tiger());

// Downcast, geht nur wenn animal tatsächlich ein Tiger oder eine
// Subklasse davon ist, oder den Wert null hat
Tiger tiger = (Tiger)(animal);

// Gilt auch für Arrays: OK, falls alle Elemente von animals
// Tiger oder Subklassen davon sind
Tiger[] tigers = (Tiger[])(animals);
```

Beispiel: Typkonvertierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Upcasts, funktionieren immer
Animal animal = new Tiger();
Animal animal = (Animal)(new Tiger());

// Downcast, geht nur wenn animal tatsächlich ein Tiger oder eine
// Subklasse davon ist, oder den Wert null hat
Tiger tiger = (Tiger)(animal);

// Gilt auch für Arrays: OK, falls alle Elemente von animals
// Tiger oder Subklassen davon sind
Tiger[] tigers = (Tiger[])(animals);
```

➥Dynamische Typprüfung erforderlich!

Löst Exception `_class_cast_error` bei inkompatiblen Typen aus

Details: Typkonvertierung und Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Animal[] animals = (Animal[])( tigers );
animal[0] = new Bear();

// aber OK
animal[0] = (Bear) null;
```

- `animals` speichert Tiger
- Auch nach Typkonvertierung zu `Animals[]`
- Zuweisung von Bär $\neq$ `null` an Array fehlerhaft
- Löst Exception `_array_store_error` aus
- `null` ist aber zu allen Klassen typkompatibel

Ausdrücke: Prüfe dynamischen Typ von Objekt

instanceof

TECHNISCHE UNIVERSITÄT DARMSTADT

expression `instanceof` *targetclass*

- *expression* muß Objektreferenz liefern
  - Keinen primitiven Typ
- Liefert `true` falls dynamische Klasse der Objektreferenz
  - ...identisch zu *targetclass* ist, oder
  - ...eine Subklasse von *targetclass* ist
- Liefert `false` auch, wenn Objektreferenz = `null`

Beispiele: `instanceof`

TECHNISCHE UNIVERSITÄT DARMSTADT

Skalare Typen

```
Animal animal = getNextAnimal();
if (animal instanceof Tiger)
  numTigers = numTigers + 1;
else if (animal instanceof Bear)
  numBear = numBear + 1;
else if (animal instanceof Eagle)
  numEagles = numEagles + 1;
```

Array Typen

```
if (animals instanceof Tiger[]) {
  Tiger[] tigers = (Tiger[])( animals);
   ...
}
```

Ausdrücke: Operatoren

Im wesentlichen wie in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int x = 0;
int y = 1;
int z = 2;
x = y + z; // x is set to 3
x = y - z; // x is set to -1
x = y * z; // x is set to 2
x = y / z; // x is set to 0
x = y % z; // x is set to 1
x = - y;   // x is set to -1

boolean b1 = false;
boolean b2 = false;
boolean b3 = true;
b1 = b2 && b3; // b1 is set to false
b1 = b2 || b3; // b1 is set to true
b1 = !b2;      // b1 is set to true
```

```
boolean b = false;
int i1 = 0;
int i2 = 1;
int i3 = 0;
Object o1 = new Object();
Object o2 = new Object();
Object o3 = o1;
b = i1 == i2; // b is set to false
b = i1 != i2; // b is set to true
b = o1 == o2; // b is set to false
b = o1 == o3; // b is set to true
b = i1 < i2;  // b is set to true
b = i1 >= i3; // b is set to true
```

- Keine Kurzschreibweisen `+=`, `&&=`, `++`, `--` etc.

Ausdrücke: Konstanten und Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT

Konstanten

- `int`: Nur in Dezimaldarstellung, $0 \ldots 2147483647$ ($2^{31} - 1$)
- `boolean`: `true`, `false`
- `String`: Nur ASCII Zeichen, übliche Steuerzeichen `\n`, `\t`, etc.

Ausdrücke: Konstanten und Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT

Konstanten

- `int`: Nur in Dezimaldarstellung, $0 \ldots 2147483647$ ($2^{31} - 1$)
- `boolean`: `true`, `false`
- `String`: Nur ASCII Zeichen, übliche Steuerzeichen `\n`, `\t`, etc.

Variablen

- Lokale Variablen: Über Namen
- Attribute dieser Klasse: Namen, optional mit vorangestelltem `this`
- Attribute von Superklassen: Namen mit vorangestelltem `super`
- Größe eines Arrays: Array Name gefolgt von `.length`
- Ein Array-Element: Array Name gefolgt von `[` *index* `]`
- Beachte: Keine Zugriffe auf Attribute über sonstige Objektreferenzen

Eingebaute Klassen

Implementiert als MIPS-Assembler

TECHNISCHE UNIVERSITÄT DARMSTADT

Class	Method signature	Method description
Object	Object clone()	copy an object
	boolean equals(Object s)	test if objects are equal (*i.e.*, alias)
	String toString()	return string representation of object
Sys	void exit(int status)	exit program with specified status
	int time() *	return UTC time
	int random() *	return random int
String	int length()	return string length
	boolean equals(Object s)	test if strings are equivalent
	String toString()	return itself
	String substring(int beginIndex, int endIndex)	return substring between the indices
	String concat(String s)	return concatenated string
TextIO	void readFile(String filename)	set to read from specified file
	void writeFile(String filename)	set to write to specified file
	void readStdin()	set to read from standard input
	void writeStdout()	set to write to standard output
	void writeStderr()	set to write to standard error
	String getString()	read next string
	int getInt()	read next int
	void putString(String s)	write specified string
	void putInt(int i)	writes specified int

Beispiel: Benutzung der I/O-Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Main {
  TextIO io = new TextIO();
  String output = "";

  void error() {
    io.writeStderr();
    io.putString("Bad␣input;␣exiting\n");
    (new Sys()).exit(1);
    return;
  }

  String getNextLine() {
    String s = io.getString();
    if (s == null || s.length() < 2)
      error();
    return s.substring(1, s.length());
  }
```

```
  void main() {
    String s = "";
    io.readStdin();
    int n = io.getInt();
    if (n < 1)
      error();

    io.readFile("input.txt");
    int i = 0;
    for (i = 0; i < n && !s.equals("quit"); i++) {
      s = getNextLine();
      output = output.concat(s).concat("\n");
    }

    io.writeStdout();
    io.putString(output.toString());

    io.writeFile("output.txt");
    io.putString(output);
    return;
  }
}
```

Benutzung des Systems

Bantam Compiler

TECHNISCHE UNIVERSITÄT DARMSTADT

- `bin/bantamc`: Entwicklungscompiler
- `bin/bantamc-jj-ref`: Referenzimplementierung
- Einfachster Fall: Ausprobieren von Beispielprogrammen in `tests/`
  `../bin/bantamc-jj-ref HelloWorld.btm`
- Erzeugt MIPS-Assembler in Datei `out.s`
  - Kann mit `-o dateiname` in andere Datei geschrieben werden
- Compiler kann auch mehrere Dateien zusammen als ein Programm übersetzen
  - ...ging in Triangle ja nicht

Ausführen mit SPIM

TECHNISCHE UNIVERSITÄT DARMSTADT

- Mehr Informationen zu SPIM `http://spimsimulator.sourceforge.net/`
- Wichtig: SPIM nicht direkt aufrufen
  - Braucht einen Verweis auf zu verwendende Initialisierungsdatei
  - `exceptions.s`: Stellt Laufzeitumgebung bereitgestellt
  - Wichtig: Bantam-eigene `exceptions.s` verwenden
  - Enthält z.B. Implementierung eingebauter Klassen und Garbage Collection
  - Gegebenenfalls im Wrapper-Skript Pfadnamen an eigene Installation anpassen!

Ausführen mit SPIM

TECHNISCHE UNIVERSITÄT DARMSTADT

- Mehr Informationen zu SPIM `http://spimsimulator.sourceforge.net/`
- Wichtig: SPIM nicht direkt aufrufen
  - Braucht einen Verweis auf zu verwendende Initialisierungsdatei
  - `exceptions.s`: Stellt Laufzeitumgebung bereitgestellt
  - Wichtig: Bantam-eigene `exceptions.s` verwenden
  - Enthält z.B. Implementierung eingebauter Klassen und Garbage Collection
  - Gegebenenfalls im Wrapper-Skript Pfadnamen an eigene Installation anpassen!

Ausführen des Programmes in `out.s`

```
spim.sh -file out.s
```

- Liefert am Ende Ausführungsprofil
- Zählt Ausführungshäufigkeit der verschiedenen Instruktionen
  - Enthält auch Instruktionen aus Laufzeitumgebung

TECHNISCHE UNIVERSITÄT DARMSTADT

Zielarchitektur und Laufzeitsystem

MIPS-32 als Zielarchitektur

TECHNISCHE UNIVERSITÄT DARMSTADT

- RISC-Prozessor: Nur wenige einfache Befehle
  - Die aber (hoffentlich) schnell ausgeführt werden
- Load-Store-Architektur: Speicherzugriffe nur mit dedizierten Befehlen
  - Z.B. keine Speicherzugriffe in arithmetischen Instruktionen
- 32 Register, jeweils 32 Bit breit
  - In der Regel allgemein verwendbar
  - Keine Trennung zwischen Daten- und Adressregistern

MIPS-32 Architektur

Register und Verwendungskonventionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Registername	Üblicher Verwendungszweck
$zero	Konstante 0, kann nur gelesen werden
$at	Temporäre Variable für Pseudoinstruktionen (z.B. `blt`)
$v0...$v1	Ergebnis von Berechnungen oder Funktionen
$a0...$a3	Aufrufparameter für Routinen
$t0...$t9	Temporäre Variablen
$s0...$s7	Gesicherte Variablen
$k0...$k1	Temporäre Variablen für Betriebssystem
$gp	Zeiger auf globale Variablen im Speicher
$sp	Stapelzeiger
$fp	Zeiger auf aktuellen Stack-Frame
$ra	Rücksprungadresse aus Routine

Konventionen im Standard Bantam-Compiler

TECHNISCHE UNIVERSITÄT DARMSTADT

- $s0: In der Regel `this`-Zeiger auf aktuelle Instanz
- $v0: Für Zwischenergebnisse von Rechnungen, Rückgabewert von Methoden
- $a0: Empfängerobjekt bei Methodenaufruf
- $a1: Zeilennummer dieses Methodenaufrufs
- $a2: Quelldateiname dieses Methodenaufrufs
- $t0. . . $t3: Hilfsregister
- Stack: Adressiert über $sp, wächst von oben nach unten
- Parameterübergabe für Methoden via Stack
  - Von links nach rechts mit call-by-value
- Standard-Compiler hat keine intelligente Registerallokation
  - Register werden bestenfalls innerhalb eines Sprachkonstruktes verwendet
  - In der Regel Datenhaltung in lokalen Variablen im Stack-Frame
- Keine Verwaltung von globalen Variablen via $gp

Dynamic Dispatch

Polymorphe Methodenaufrufe

dynamic dispatch

TECHNISCHE UNIVERSITÄT DARMSTADT

Statische Bindung möglich

```
Rect rect = (new Rect()).init(0,0,100,200);

rect.draw();
```

Polymorphe Methodenaufrufe

dynamic dispatch

TECHNISCHE UNIVERSITÄT DARMSTADT

Statische Bindung möglich

```
Rect rect = (new Rect()).init(0,0,100,200);

rect.draw();
```

Dynamische Bindung erforderlich

```
Rect rect = (new Rect()).init(0,0,100,200);

Shape shape = rect;

shape.draw();
```

Dynamische Bindung aus Compiler-Sicht

TECHNISCHE UNIVERSITÄT DARMSTADT

- An unterschiedliche Stellen springen
  - Potentiell beliebig viele
  - Indirekter Sprung: `jral $t0`

Dynamische Bindung aus Compiler-Sicht

TECHNISCHE UNIVERSITÄT DARMSTADT

- An unterschiedliche Stellen springen
  - Potentiell beliebig viele
  - Indirekter Sprung: `jral $t0`
- An welche von diesen Stellen springen?
  - Methode heißt immer `draw()`
  - Aber unterschiedliche Implementierungen in verschiedenen Klassen
  - Auswahl der Implementierung anhand dynamischer Klasse der Instanz

Dynamische Bindung aus Compiler-Sicht

TECHNISCHE UNIVERSITÄT DARMSTADT

- An unterschiedliche Stellen springen
  - Potentiell beliebig viele
  - Indirekter Sprung: `jral $t0`
- An welche von diesen Stellen springen?
  - Methode heißt immer `draw()`
  - Aber unterschiedliche Implementierungen in verschiedenen Klassen
  - Auswahl der Implementierung anhand dynamischer Klasse der Instanz
- Idee
  1. Dynamische Klasse von jeder Instanz merken
  2. Je Klasse für jede Methode Startadresse der Routine merken
  3. Damit zur Laufzeit passende Routine aufrufen

Datenstrukturen für Dynamische Bindung

Zunächst vereinfacht: Ohne Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

- In jeder Instanz Typinformationen mitführen
  - Einfachster Fall: Alle Typen (Klassen) durchnumerieren
  - Type-ID je Instanz speichern

Datenstrukturen für Dynamische Bindung

Zunächst vereinfacht: Ohne Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

- In jeder Instanz Typinformationen mitführen
  - Einfachster Fall: Alle Typen (Klassen) durchnumerieren
  - Type-ID je Instanz speichern

- Alle Methoden einer Klasse durchnumerieren
  - Method-ID
- Tabelle ordnet jeder Method-ID die Startadresse der Routine zu
  - Oft genannt *virtual method table*, *vtable*

Datenstrukturen für Dynamische Bindung

Zunächst vereinfacht: Ohne Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

- In jeder Instanz Typinformationen mitführen
  - Einfachster Fall: Alle Typen (Klassen) durchnumerieren
  - Type-ID je Instanz speichern

- Alle Methoden einer Klasse durchnumerieren
  - Method-ID
- Tabelle ordnet jeder Method-ID die Startadresse der Routine zu
  - Oft genannt *virtual method table*, *vtable*

Beispiel: Vtable von Bantam Object-Klasse

```
Object_dispatch_table:
  .word   Object.clone      # Method-ID 0
  .word   Object.equals     # Method-ID 1
  .word   Object.toString   # Method-ID 2
```

Implementierung: 1. Versuch

TECHNISCHE UNIVERSITÄT DARMSTADT

- Type-ID in jeder Instanz
- Tabelle enthält zu jeder Type-ID die Vtable
- Dann Nachschlagen mit Method-ID in Vtable um Routine zu bestimmen

Beispiel für 1. Implementierungsversuch

`Object foo = new Object(); foo.toString()`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Object_dispatch_table:
  .word   Object.clone          # Method−ID 0
  .word   Object.equals         # Method−ID 1
  .word   Object.toString       # Method−ID 2
...
class_to_vtables:
  .word   Object_dispatch_table # Type−ID 0
  .word   TextIO_dispatch_table # Type−ID 1
  .word   Sys_dispatch_table    # Type−ID 2
...

Main.main: # Beispiel: Rufe Object.toString auf
...
  lw      $t0   0($s0)          # s0: this, Type−ID am Anfang
  sll     $t0   $t0     2       # Byte offset bestimmen
  li      $t1   class_to_vtables
  add     $t1   $t1    $t0
  lw      $t0   0($t1)          # Vtable von Object in $t0
  lw      $t1   8($t0)          # Routine Object.toString in $t1
  jalr    $t1                   # indirekter Aufruf
...
```

Implementierung: 2. Versuch

TECHNISCHE UNIVERSITÄT DARMSTADT

Diskussion 1. Versuch

- Funktioniert
- Potentiell aber sehr langsam für jeden Methodenaufruf
- Insbesondere drei Speicherzugriffe je Aufruf

Implementierung: 2. Versuch

TECHNISCHE UNIVERSITÄT DARMSTADT

Diskussion 1. Versuch

- Funktioniert
- Potentiell aber sehr langsam für jeden Methodenaufruf
- Insbesondere drei Speicherzugriffe je Aufruf

Verbesserung

- Trage Vtable-Adresse direkt in Instanzen ein
- Kostet 1 Wort mehr Speicher je Instanz ...
  - Unabhängig von Anzahl der Methoden
- ...läuft aber deutlich schneller

Beispiel für 2. Implementierungsversuch

`Object foo = new Object(); foo.toString()`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Object_dispatch_table:
  .word   Object.clone        # Method−ID 0
  .word   Object.equals       # Method−ID 1
  .word   Object.toString     # Method−ID 2
...
Main.main: # Beispiel: Rufe Object.toString auf
...
  lw      $t0   4($s0)        # s0: this, Vtable−Adresse an 2. Stelle
  lw      $t1   8($t0)        # Routine Object.toString in $t1
  jalr    $t1                 # indirekter Aufruf
...
```

Beispiel für 2. Implementierungsversuch

`Object foo = new Object(); foo.toString()`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Object_dispatch_table:
  .word   Object.clone        # Method−ID 0
  .word   Object.equals       # Method−ID 1
  .word   Object.toString     # Method−ID 2
...
Main.main: # Beispiel: Rufe Object.toString auf
...
  lw      $t0   4($s0)        # s0: this, Vtable−Adresse an 2. Stelle
  lw      $t1   8($t0)        # Routine Object.toString in $t1
  jalr    $t1                 # indirekter Aufruf
...
```

Nur noch zwei Speicherzugriffe je Methodenaufruf.
Noch weiter verbesserbar?

Implementierung: 3. Versuch

TECHNISCHE UNIVERSITÄT DARMSTADT

Idee: Vtable direkt in Instanzen speichern

```
# Beispiel: Instanz im Speicher
# this zeigt auf den Anfang
  .word   0                    # 0. Wort: Type−ID
  .word   Object.clone         # 1. Wort: Method clone
  .word   Object.equals        # 2. Wort: Method equals
  .word   Object.toString      # 3. Wort: Method toString
...
Main.main: # Beispiel: Rufe Object.toString auf
...
  lw      $t0    12($s0)       # s0: this, Vtable beginnt im 1. Wort
  jalr    $t0                  # indirekter Aufruf
...
```

Diskussion

TECHNISCHE UNIVERSITÄT DARMSTADT

3. Versuch
- Theoretisch noch schneller
  - Nur noch einen Speicherzugriff pro Methodenaufruf
- Braucht potentiell deutlich mehr Speicher
  - Kompletter Vtable je Instanz
  - Extrem teuer bei vielen Instanzen und vielen Methoden
- Ausblick: Problematisch auch bei Vererbung
  - Ändert Speicher-Layout der nachfolgenden Attribute

Diskussion

TECHNISCHE UNIVERSITÄT DARMSTADT

3. Versuch
- Theoretisch noch schneller
  - Nur noch einen Speicherzugriff pro Methodenaufruf
- Braucht potentiell deutlich mehr Speicher
  - Kompletter Vtable je Instanz
  - Extrem teuer bei vielen Instanzen und vielen Methoden
- Ausblick: Problematisch auch bei Vererbung
  - Ändert Speicher-Layout der nachfolgenden Attribute

➥Weit verbreitet: 2. Implementierung
- Gemeinsamer Vtable für alle Instanzen einer Klasse
- Jede Instanz hat eigenen Verweis auf Vtable ihrer Klasse
- Auch in Bantam verwendet

Gezielte Optimierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Ziel: Ersetze dynamische durch statische Bindung

- Analysiere Programm
- Erkenne Fälle, in denen Klasse statisch exakt bestimmbar ist
  - Superklasse reicht nicht!
- Ersetze dann indirekten Aufruf durch direkten Aufruf

Gezielte Optimierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Ziel: Ersetze dynamische durch statische Bindung

- Analysiere Programm
- Erkenne Fälle, in denen Klasse statisch exakt bestimmbar ist
  - Superklasse reicht nicht!
- Ersetze dann indirekten Aufruf durch direkten Aufruf

```
Object foo = new Object();

foo.toString();      // hier Typ genau bestimmbar
```

```
Main.main: # Beispiel: Rufe Object.toString auf
 ...
  jal      Object.toString                    # indirekter Aufruf
 ...
```

Erweiterung auf Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswirkungen von Vererbung

- Erweitere Funktionalität relativ zur Oberklasse
  - Füge neue Methoden hinzu
- Verändere Funktion relativ zur Oberklasse
  - Durch Überschreiben
  - Methode der Oberklasse muß aufrufbar bleiben (mit `super`)
- Nicht möglich: Entfernen von Funktionalität
- In Bantam: Nur einfache Vererbung

Erweiterung auf Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswirkungen von Vererbung

- Erweitere Funktionalität relativ zur Oberklasse
  - Füge neue Methoden hinzu
- Verändere Funktion relativ zur Oberklasse
  - Durch Überschreiben
  - Methode der Oberklasse muß aufrufbar bleiben (mit `super`)
- Nicht möglich: Entfernen von Funktionalität
- In Bantam: Nur einfache Vererbung

➥Idee: Realisiere in Vtable der Unterklasse

Anpassen der Vtables bei Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Keine Änderungen relativ zur Oberklasse

Anpassen der Vtables bei Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Keine Änderungen relativ zur Oberklasse
   - Vtable(Unter) ist Kopie von Vtable(Ober)

Anpassen der Vtables bei Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Keine Änderungen relativ zur Oberklasse
   - Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse

Anpassen der Vtables bei Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Keine Änderungen relativ zur Oberklasse
   - Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse
   - Füge entsprechende Methoden am Ende von Vtable(Unter) an

Anpassen der Vtables bei Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Keine Änderungen relativ zur Oberklasse
   - Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse
   - Füge entsprechende Methoden am Ende von Vtable(Unter) an
   - Alle Method-IDs der Oberklasse haben auch in Unterklasse Bestand

Anpassen der Vtables bei Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Keine Änderungen relativ zur Oberklasse
   - Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse
   - Füge entsprechende Methoden am Ende von Vtable(Unter) an
   - Alle Method-IDs der Oberklasse haben auch in Unterklasse Bestand
3. Ändere Methoden relativ zur Oberklasse

Anpassen der Vtables bei Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Keine Änderungen relativ zur Oberklasse
   - Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse
   - Füge entsprechende Methoden am Ende von Vtable(Unter) an
   - Alle Method-IDs der Oberklasse haben auch in Unterklasse Bestand
3. Ändere Methoden relativ zur Oberklasse
   - Trage neue Routinen unter alter Method-ID in Vtable(Unter) ein

Anpassen der Vtables bei Vererbung

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Keine Änderungen relativ zur Oberklasse
   - Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse
   - Füge entsprechende Methoden am Ende von Vtable(Unter) an
   - Alle Method-IDs der Oberklasse haben auch in Unterklasse Bestand
3. Ändere Methoden relativ zur Oberklasse
   - Trage neue Routinen unter alter Method-ID in Vtable(Unter) ein
   - Alte Implementierungen stehen noch in Vtable(Ober) zur Verfügung

Beispiel: 1. Keine Änderungen zur Oberklasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Bantam

```
class X {
  int i;
}
```

Vtable von X

```
X_dispatch_table:
  .word Object.clone     # Method-ID 0
  .word Object.equals    # Method-ID 1
  .word Object.toString  # Method-ID 2
```

Beispiel: 2. Zusätzliche Methode in Unterklasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Bantam

```
class X {
  int i;
  void setI(int i) {
    this.i = i;
    return;
  }
}
```

Vtable von X

```
X_dispatch_table:
  .word Object.clone     # Method-ID 0
  .word Object.equals    # Method-ID 1
  .word Object.toString  # Method-ID 2
  .word X.setI           # Method-ID 3
```

Beispiel: 2. Zusätzliche Methode in Unterklasse

Geerbte Methoden bleiben unverändert

TECHNISCHE UNIVERSITÄT DARMSTADT

Bantam

```
X x = new X();
Object obj = x;

obj.toString();    // ruft Method-ID 2 von Object auf
                   // ist auch Method-ID 2 von X
```

MIPS-Implementierung

```
X_dispatch_table:
  .word Object.clone    # Method-ID 0
  .word Object.equals   # Method-ID 1
  .word Object.toString # Method-ID 2
  .word X.setI          # Method-ID 3
...
Main.main: # Beispiel: Rufe Object.toString von x auf
...
  lw    $t0   4($s0)     # s0: this = x, Vtable-Adresse an 2. Stelle
  lw    $t1   8($t0)     # Routine Object.toString in $t1
  jalr  $t1              # indirekter Aufruf
...
```

Beispiel: 3. Veränderte Methode in Unterklasse

TECHNISCHE UNIVERSITÄT DARMSTADT

Bantam

```
class X {
  String toString() {
    return "foo".concat(super.toString());
  }
}
```

Vtable von X

```
X_dispatch_table:
  .word Object.clone     # Method-ID 0
  .word Object.equals    # Method-ID 1
  .word X.toString       # Method-ID 2
  .word X.setI           # Method-ID 3
```

Beispiel: 3. Veränderte Methode in Unterklasse

Geerbte Methode wird überschrieben

TECHNISCHE UNIVERSITÄT DARMSTADT

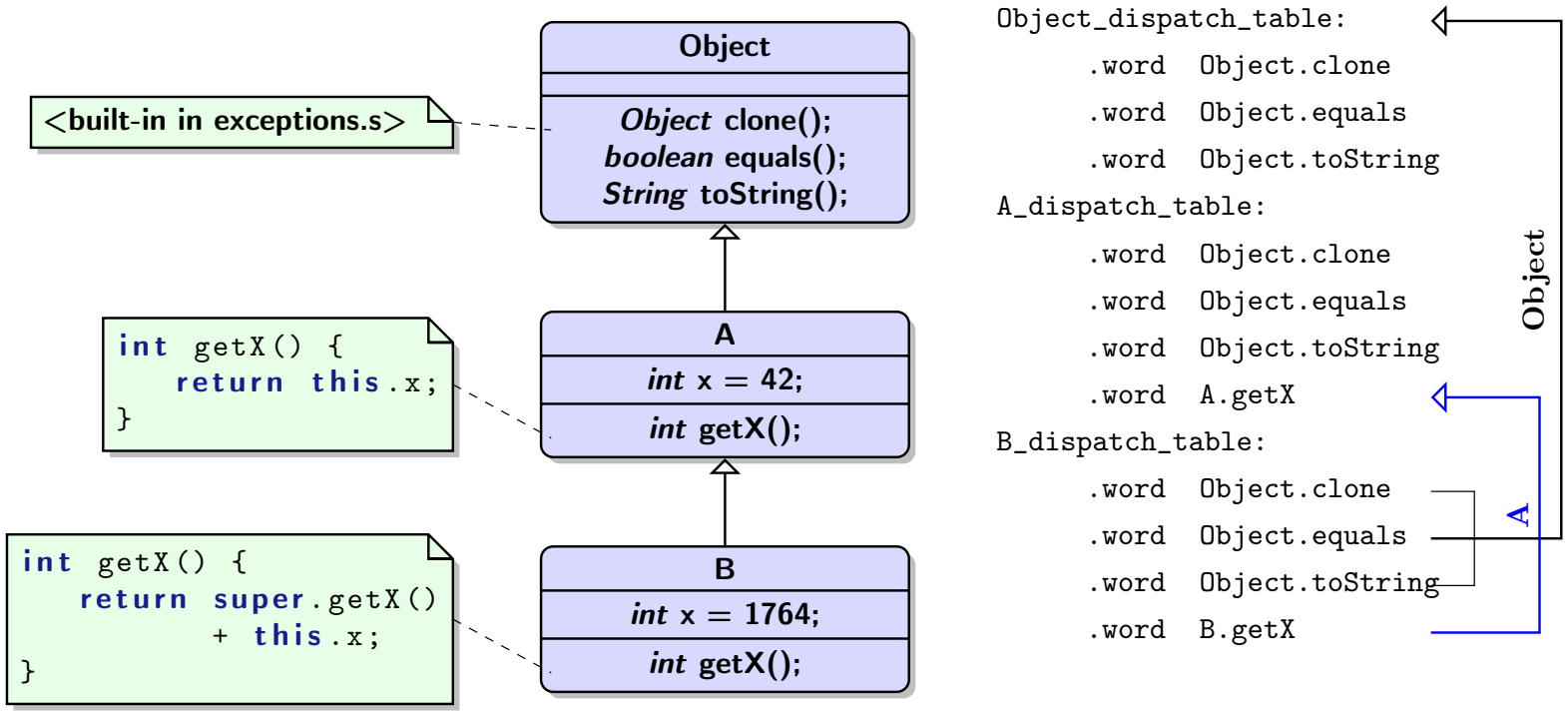
```
X x = new X();
Object obj = x;

obj.toString();    // ruft Method−ID 2 von X via obj auf
```

```
X_dispatch_table:
  .word Object.clone     # Method−ID 0
  .word Object.equals    # Method−ID 1
  .word X.toString       # Method−ID 2
  .word X.setI           # Method−ID 3
Main.main: # Beispiel: Rufe X.toString von obj auf
 ...
  lw     $t0   4($s0)                # s0: this = obj, Vtable−Adresse an 2. Stelle
  lw     $t1   8($t0)                # Routine X.toString in $t1
  jalr   $t1                         # indirekter Aufruf
 ...
X.toString: # Aufruf von super.toString() in X.toString()
 ...
  la     $t0   Object_dispatch_table # expliziter Zugriff auf Oberklasse
  lw     $t1   8($t0)                # Routine Object.toString in $t1
  jalr   $t1                         # indirekter Aufruf
```

Beispiel: Vererbung mit zwei Unterklassen

TECHNISCHE UNIVERSITÄT DARMSTADT



Dynamische Prüfungen beim Methodenaufruf

...bisher weggelassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Empfängerobjekt muß ungleich `null` sein

```
Object_dispatch_table:
  .word Object.clone     # Method−ID 0
  .word Object.equals    # Method−ID 1
  .word Object.toString  # Method−ID 2
Main.main: # Beispiel: Rufe Object.toString von obj auf
 ...
  lw      $a0   $s0                # s0: this = obj
  bne     $a0   $zero  label45     # this != null
  jal     _null_pointer_error      # definiert in Bantam exceptions.s
                                   # kehrt nicht mehr zurück (Programmende)
label45:
  lw      $t0   4($s0)             # s0: this = obj, Vtable−Adresse an 2. Stelle
  lw      $t1   8($t0)             # Routine X.toString in $t1
  jalr    $t1                      # indirekter Aufruf
 ...
```

Erzeugen von Objekten

Erzeugen von neuen Objekten

TECHNISCHE UNIVERSITÄT DARMSTADT

- Jede neue Instanz benötigt Speicher
  - Kann vom Betriebssystem mit `syscall` angefordert werden
- Muß aber richtig initialisiert werden
  - Type-ID und Zeiger auf Vtable von Klasse
  - Attribute
    - Standardwerte `0, false, null`
    - Eigene (explizite) Initialisierung
    - Initialisiere auch Attribute der Oberklasse(n) korrekt

Erzeugen von neuen Objekten

TECHNISCHE UNIVERSITÄT DARMSTADT

- Jede neue Instanz benötigt Speicher
  - Kann vom Betriebssystem mit `syscall` angefordert werden
- Muß aber richtig initialisiert werden
  - Type-ID und Zeiger auf Vtable von Klasse
  - Attribute
    - Standardwerte `0, false, null`
    - Eigene (explizite) Initialisierung
    - Initialisiere auch Attribute der Oberklasse(n) korrekt

➥Idee: Lege Vorlagen von Objekten an und kopiere diese bei Instanziierung

Objekterzeugung mit Vorlagen

templates

TECHNISCHE UNIVERSITÄT DARMSTADT

- Templates enthalten Type-ID, Vtable und Standardwerte für alle Atribute
  - In Bantam: *Klasse*`_template`
- Neue Instanzen werden mittels `Object.clone` von Vorlage erzeugt
- Dann Aufruf einer Hilfsroutine für explizite Initialisierung
  - In Bantam: *Klasse*`_init`

Beispiel: Objekterzeugung 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Bantam

```
class X {
  int i;
  int j = 42;
  void setI(int i) { ... }
  String toString() { ... }
}
```

MIPS-Implementierung: Vorlage

```
X_template:
  .word 2                  # Type−ID
  .word 20                 # Gesamtgröße der Template in Bytes
  .word X_dispatch_table   # Verweis auf Vtable der Klasse X
  .word 0                  # Attribut i, Standardwert
  .word 0                  # Attribut j, Standardwert
```

Beispiel: Objekterzeugung 2

TECHNISCHE UNIVERSITÄT DARMSTADT

MIPS-Implementierung: Initialisierung

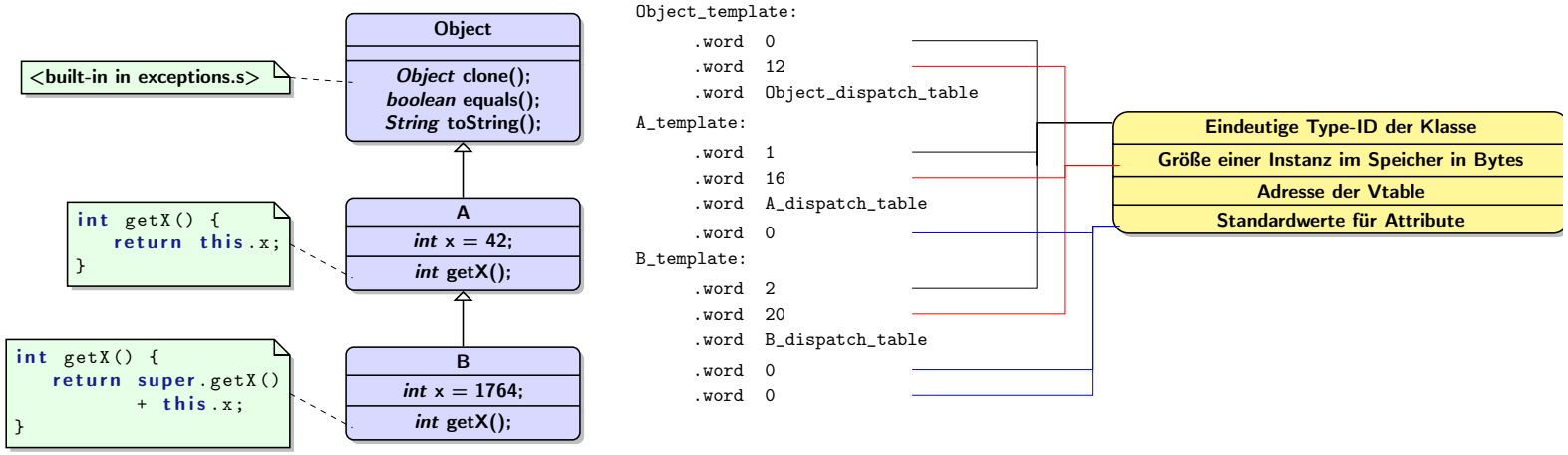
```
X_init :
...
  jal  Object_init          # rufe Init der Oberklasse auf
  li   $v0 42
  sw   $v0 16($s0)          # initialisiere Attribut j auf 42
...
```

MIPS-Implementierung: Realisierung von `new X()`

```
Main.main:
...
  la     $a0 X_template # Zielobjekt (this) ist X_template
  jal    Object.clone   # Kopie erzeugen, Zeiger auf Kopie zurück in $v0
  move   $a0 $v0        # Zielobjekt (this) ist neue Instanz von X
  jal    X_init         # Explizite Initialisierung der Attribute von neuer Instanz
...
```

Beispiel: Objektvorlagen mit zwei Unterklassen

TECHNISCHE UNIVERSITÄT DARMSTADT



Sonderfall String-Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

- Werden als Singletons realisiert
  - Für jede im Programm vorkommende Zeichenkette existiert genau ein String-Objekt
- Die String-Vorlagen enthalten bereits die einzelnen Zeichen des Strings
  - Müssten sonst einzeln initialisiert werden
- Haben variable Größe
  - `"Foo"` (3 Zeichen) vs. `"Blahfasel"` (9 Zeichen)
  - ASCII NUL (Wert 0x00) wird als Marker für String-Ende verwendet
  - String-Objekte müssen aber auf Wortgrenzen enden
    - Sonst schlägt Zugriff auf nachfolgende Daten mit `lw` fehlerhaft
    - Auffüllen auf Vielfache von vier Bytes

Beispiel: Objektvorlagen für String-Literale

Vorlage für `"foo"`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String_const_5:
  .word   11                    # Type−ID von Klasse String
  .word   20                    # Gesamtgröße in Bytes
  .word   String_dispatch_table # Vtable für Klasse String
  .word   3                     # String−Länge (ohne NUL)
  .ascii  "foo"                 # Zeichen
  .byte   0                     # Endemarker NUL
  .align  2                     # Fülle auf Vielfache von 2^2 auf
```

Beispiel: Objektvorlagen für String-Literale

Vorlage für `"foo"`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String_const_5:
  .word   11                    # Type−ID von Klasse String
  .word   20                    # Gesamtgröße in Bytes
  .word   String_dispatch_table # Vtable für Klasse String
  .word   3                     # String−Länge (ohne NUL)
  .ascii  "foo"                 # Zeichen
  .byte   0                     # Endemarker NUL
  .align  2                     # Fülle auf Vielfache von 2^2 auf
```

- Sonderbehandlung nur für String-Literale
  - Dynamisch erzeugte String-Objekte verwenden Vorlage `String_template`
- Bantam Compiler legt automatisch String-Literale an, z.B. für
  - Namen jeder Klasse
  - Dateinamen der Bantam-Quelldateien
  - ➥werden für Exception-Fehlermeldungen verwendet

TECHNISCHE UNIVERSITÄT DARMSTADT

Methodenprotokoll

Protokoll für Methodenaufruf

TECHNISCHE UNIVERSITÄT DARMSTADT

- Übergabe von Parametern
- Rückgabe von Ergebnis
- Aufräumen auf Stack
- Ausnahmebehandlung (gab es in Triangle nicht!)

Übergabe von Parametern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Parameter werden auf Stack übergeben
  - Von links nach rechts
  - Aufgerufene Methode räumt Stack auf
- `$a0`: `this`-Zeiger von Empfängerobjekt

Übergabe von Parametern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Parameter werden auf Stack übergeben
  - Von links nach rechts
  - Aufgerufene Methode räumt Stack auf
- `$a0`: `this`-Zeiger von Empfängerobjekt

- Für Ausnahmebehandlung bei ungültigem Empfängerobjekt
  - `$a1`: Zeilennummer dieses Aufrufs in Quelldatei
  - `$a2`: Zeigt auf Vorlage von String-Literal mit Quelldateiname
  - Werden verwendet zur Ausgabe einer Fehlermeldung in `_null_pointer_error`
    - Siehe Bantam-spezifisches `exceptions.s`

Beispiel: Methodenprotokoll

Bantam

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class T {
  ...
  int bar(int a, int b) {        // Aufgerufene Methode
    return a*b;
  }
}
...
class Main {
  void main() {
    int temp;
    T t = new T();
    temp = t.bar(42, 23);     // Relevante Aufrufstelle
     ...
  }
}
```

Beispiel: Methodenprotokoll beim Aufrufer

MIPS-Implementierung

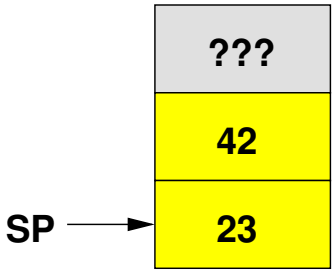
TECHNISCHE UNIVERSITÄT DARMSTADT

```
Main.main:
  ...
  li    $v0 42                # \
  sw    $v0 −4($sp)           #  | Push 42
  sub   $sp $sp 4             # /
  li    $v0 23                # \
  sw    $v0 −4($sp)           #  | Push 23
  sub   $sp $sp 4             # /
  li    $a1 85                # Zeile des Aufrufes in Quelldatei
  la    $a2 String_const_3    # String−Literal für Dateiname "TestInherit.btm"
  lw    $a0 12($fp)           # Hole Zeiger auf Instanz aus lokaler Variable t
  bne   $a0 $zero label34     # Gültiges Empfängerobjekt?
  jal   _null_pointer_error   # nein, −> Null Pointer Exception: Fehlermeldung
label34:
  lw    $t0 8($a0)            # Hole Vtable von Klasse T aus Instanz t
  lw    $t0 20($t0)           # Hole Startadresse von Routine bar
  jalr  $t0                   # indirekter Sprung
  sw    $v0 0($fp)            # speichere Rückgabewert in lokale Variable temp
  ...
```

Stackorganisation innerhalb einer Methode

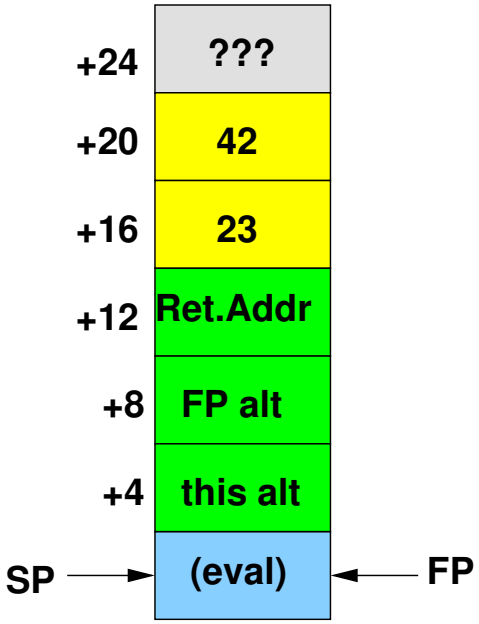
TECHNISCHE UNIVERSITÄT DARMSTADT

Vor Aufruf von bar()



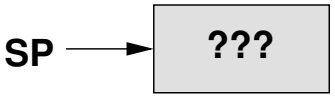
$a0: Empfängerobjekt t
$a1: Zeile Aufruf
$a2: Dateiname Aufruf

Vor Body von bar()



$s0: this
0($fp): lokale Variable für Evaluation

Am Ende von bar()



$v0: Rückgabewert (hier 966)

Beispiel: Methodenprotokoll beim Aufgerufenen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
T.bar:
  add  $sp $sp −16     # Platz für 3 Worte Verwaltungsdaten und eine lokale Variable
  sw   $ra 12($sp)     # Sichere Rücksprungadresse auf Stack
  sw   $fp 8($sp)      # Sichere alten Framepointer auf Stack (dynamic link)
  sw   $s0 4($sp)      # sichere this−Zeiger des Aufrufers auf Stack
  move $fp $sp         # aktueller FP zeigt nun auf erste lokale Variable
  move $s0 $a0         # this−Zeiger dieser Instanz wurde in $a0 übergeben
  lw   $v0 20($fp)     # Hole 1. formalen Parameter
  sw   $v0 0($fp)      # Lege ab als Zwischenergebnis der aktuellen Rechnung
  lw   $v0 16($fp)     # Hole 2. formalen Parameter
  lw   $t0 0($fp)      # Hole bisheriges Zwischenergebnis der aktuellen Rechnung
  mul  $v0 $t0 $v0     # Multipliziere 2. Parameter und Zwischenergebnis
  lw   $s0 4($sp)      # stelle this−Zeiger des Aufrufers wieder her
  lw   $fp 8($sp)      # stelle Framepointer des Aufrufers wieder her
  lw   $ra 12($sp)     # Hole Rücksprungadresse
  add  $sp $sp 24      # gib aktuellen Frame _und_ aktuelle Parameter wieder frei
  jr   $ra             # Rücksprung zum Aufrufer
```

Methodenprotokoll: Prolog und Epilog

Anfang und Ende von Methoden haben immer gleiches Muster

TECHNISCHE UNIVERSITÄT DARMSTADT

Prolog

```
AnyClass.anyMethod: # #Locals ist Anzahl der lokalen Variablen
  add  $sp $sp N      # N = (3 + #Locals) * 4 Bytes
  sw   $ra R($sp)     # R = (2 + #Locals) * 4 Bytes
  sw   $fp F($sp)     # F = (1 + #Locals) * 4 Bytes
  sw   $s0 T($sp)     # T = (0 + #Locals) * 4 Bytes
  move $fp $sp        # aktueller SP zeigt nun auf neuen Framepointer
  move $s0 $a0        # this−Zeiger dieser Instanz wurde in $a0 übergeben
 ...
```

Methodenprotokoll: Prolog und Epilog

Anfang und Ende von Methoden haben immer gleiches Muster

TECHNISCHE UNIVERSITÄT DARMSTADT

Prolog

```
AnyClass.anyMethod: # #Locals ist Anzahl der lokalen Variablen
  add  $sp $sp N     # N = (3 + #Locals) * 4 Bytes
  sw   $ra R($sp)    # R = (2 + #Locals) * 4 Bytes
  sw   $fp F($sp)    # F = (1 + #Locals) * 4 Bytes
  sw   $s0 T($sp)    # T = (0 + #Locals) * 4 Bytes
  move $fp $sp       # aktueller SP zeigt nun auf neuen Framepointer
  move $s0 $a0       # this-Zeiger dieser Instanz wurde in $a0 übergeben
 ...
```

Epilog

```
  ...                # erwarte Rückgabewert in $v0
  lw   $s0 T($sp)    # stelle this-Zeiger des Aufrufers wieder her
  lw   $fp F($sp)    # stelle Framepointer des Aufrufers wieder her
  lw   $ra R($sp)    # Hole Rücksprungadresse
  add  $sp $sp M     # M = N + #Params * 4 Bytes
  jr   $ra           # Rücksprung zum Aufrufer
```

Dynamische Typprüfung

Dynamische Typprüfung in Bantam

TECHNISCHE UNIVERSITÄT DARMSTADT

- Statische Typprüfung in Bantam nicht ausreichend
  - Down-Cast bei Typkonvertierung
  - `instanceof` Operatoren
  - Zuweisung an Array-Element
- Gute Nachricht
  - Objekte kennen ihre Klasse zur Laufzeit durch Type-ID
    - 1. Feld eines Objektes im Speicher

Dynamische Typprüfung in Bantam

TECHNISCHE UNIVERSITÄT DARMSTADT

- Statische Typprüfung in Bantam nicht ausreichend
  - Down-Cast bei Typkonvertierung
  - `instanceof` Operatoren
  - Zuweisung an Array-Element
- Gute Nachricht
  - Objekte kennen ihre Klasse zur Laufzeit durch Type-ID
    - 1. Feld eines Objektes im Speicher

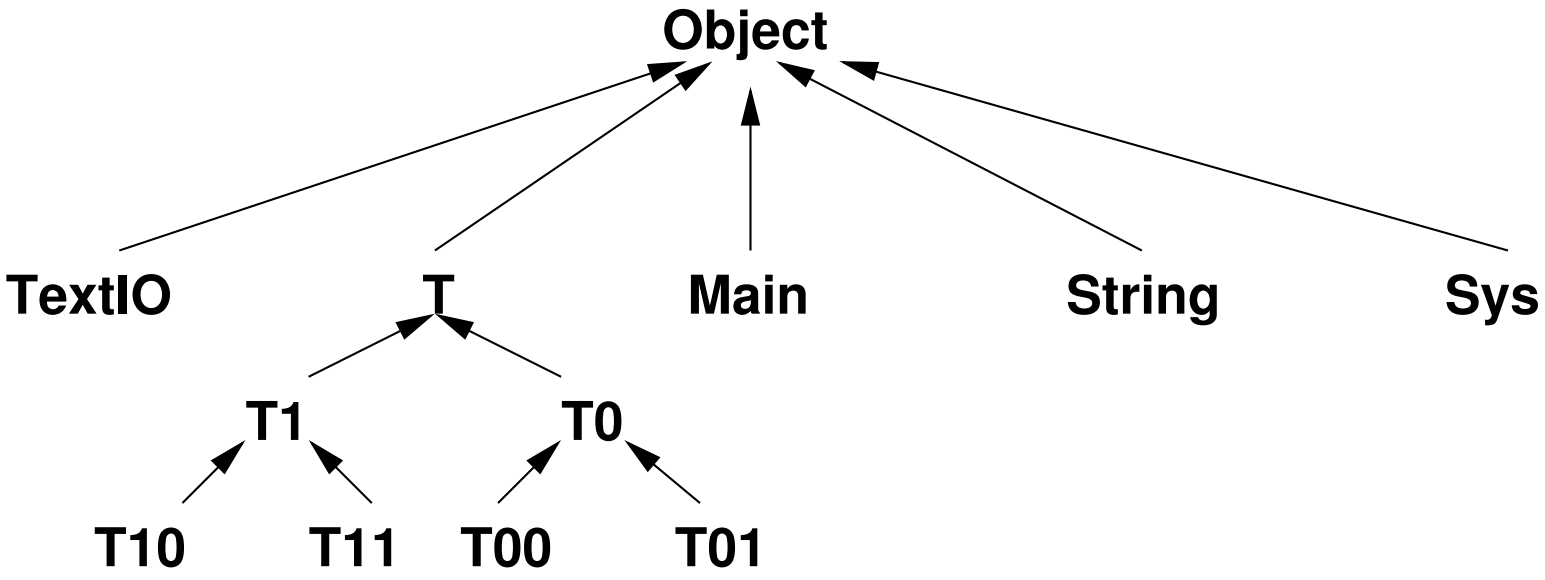
➥Gesucht: Verfahren, um

schnell zu bestimmen, ob Type-ID t_1 kompatibel zu Type-ID t_2 ist?

Kompatibel: $t_1 = t_2$ oder t_1 ist Unterklasse von t_2
Auch genannt: "t_1 ist konform zu t_2"

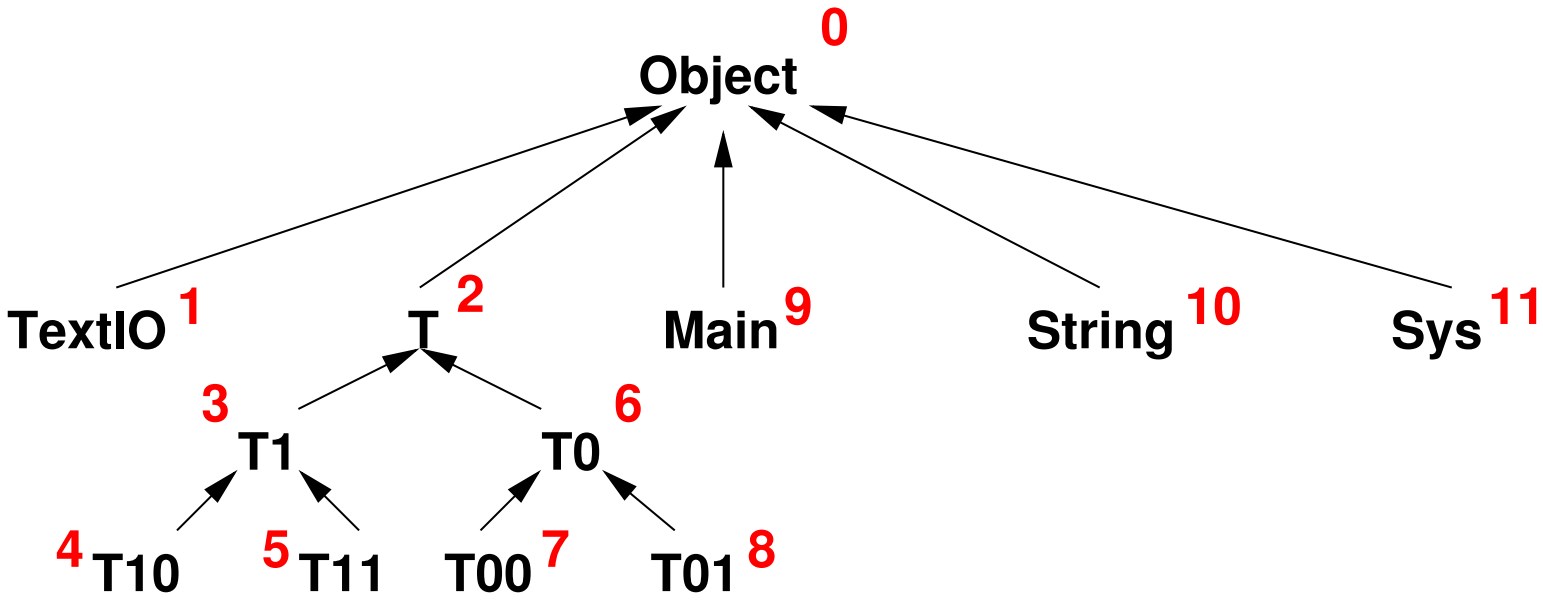
Idee: Geschickte Zuordnung von Klassen an Type-IDs

Beispiel



Idee: Geschickte Zuordnung von Klassen an Type-IDs

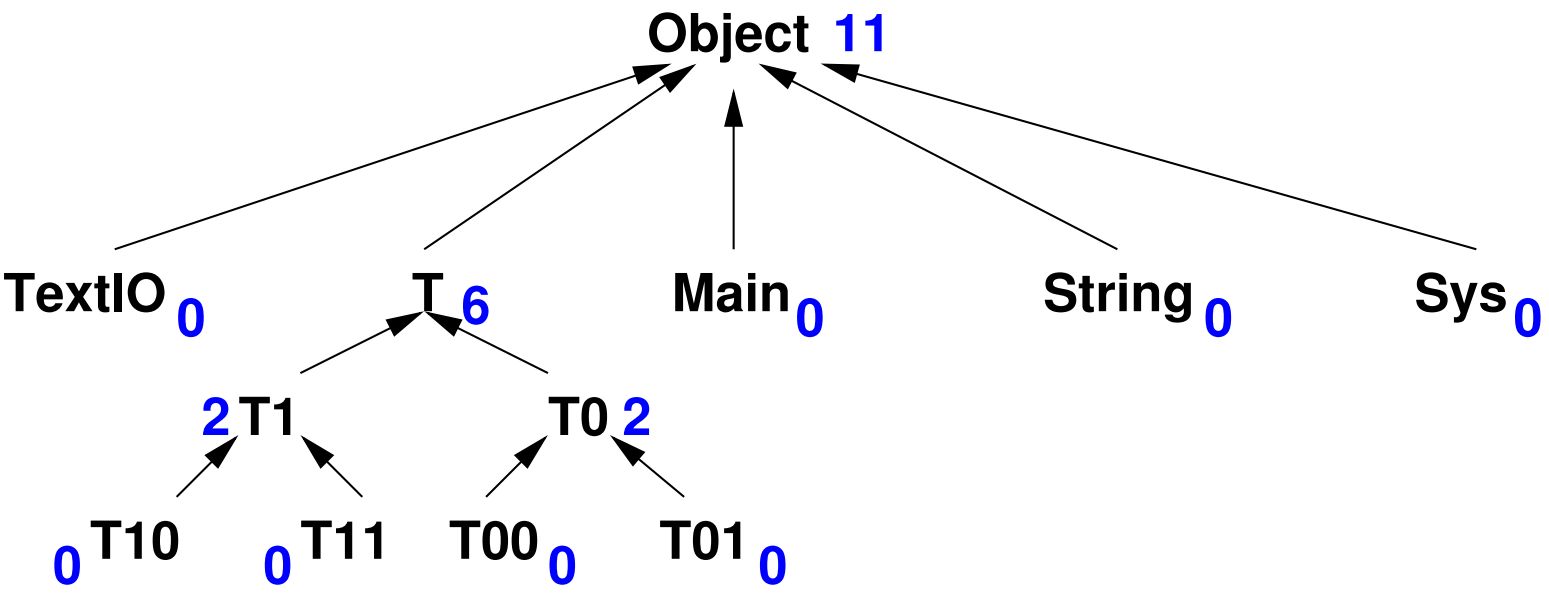
Pre-Order DFS



typeid: Numeriert in Pre–Order DFS

Idee: Geschickte Zuordnung von Klassen an Type-IDs

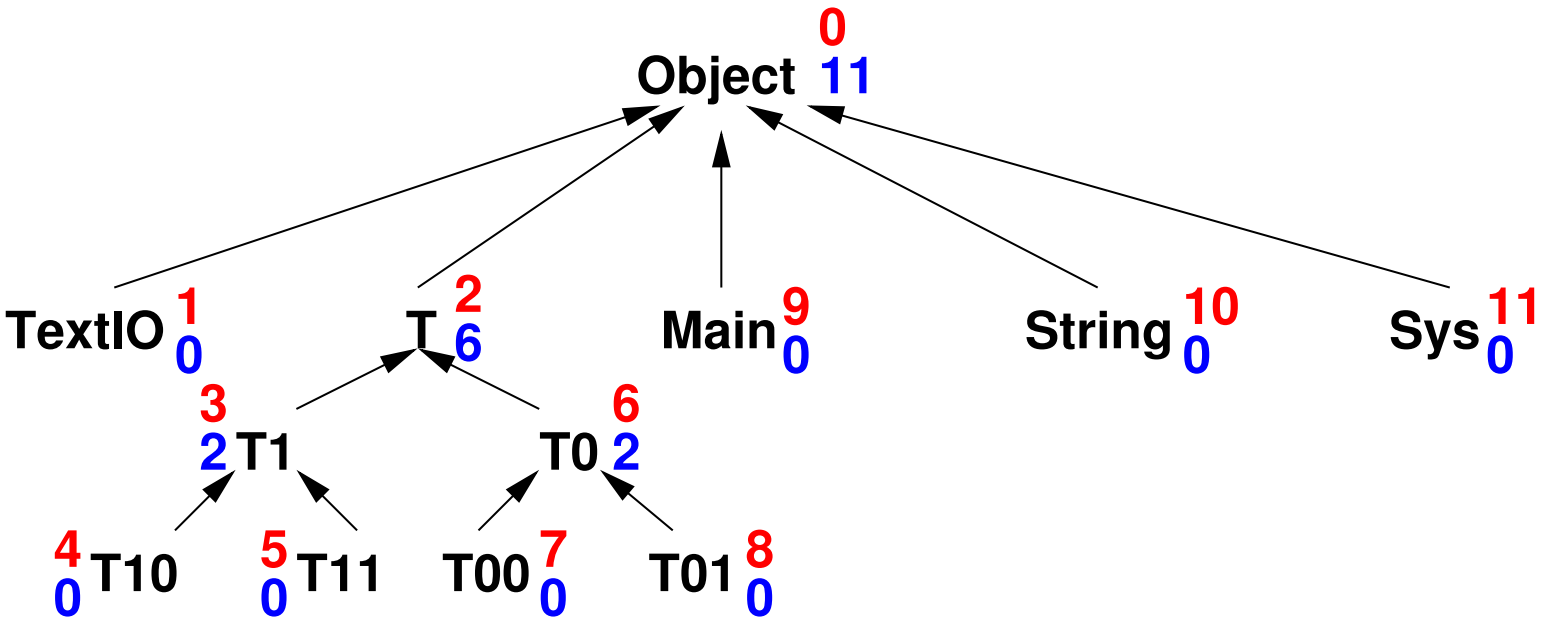
Größe der Unterbäume



k: Anzahl von Elementen im Unterbaum

Idee: Geschickte Zuordnung von Klassen an Type-IDs

Kombination



typeid: Numeriert in Pre–Order DFS

k: Anzahl von Elementen im Unterbaum

Diskussion dieses Schemas

TECHNISCHE UNIVERSITÄT DARMSTADT

Beobachtung: Für alle transitiven Unterklassen $u_i, 1 \leq i \leq k$ einer Oberklasse O gilt

$$\text{typeid}(O) \leq \text{typeid}(u_i) \leq \text{typeid}(O) + k$$

Diskussion dieses Schemas

TECHNISCHE UNIVERSITÄT DARMSTADT

Beobachtung: Für alle transitiven Unterklassen $u_i, 1 \leq i \leq k$ einer Oberklasse O gilt

$$\text{typeid}(O) \leq \text{typeid}(u_i) \leq \text{typeid}(O) + k$$

Damit formulierbar
"Klasse K_1 ist konform zu Klasse K_2"

$$\Leftrightarrow \text{typeid}(K_2) \leq \text{typeid}(K_1) \leq \text{typeid}(K_2) + k$$

Diskussion dieses Schemas

TECHNISCHE UNIVERSITÄT DARMSTADT

Beobachtung: Für alle transitiven Unterklassen $u_i, 1 \leq i \leq k$ einer Oberklasse O gilt

$$\text{typeid}(O) \leq \text{typeid}(u_i) \leq \text{typeid}(O) + k$$

Damit formulierbar
"Klasse K_1 ist konform zu Klasse K_2"

$$\Leftrightarrow \text{typeid}(K_2) \leq \text{typeid}(K_1) \leq \text{typeid}(K_2) + k$$

➥Nutzen für dynamische Typprüfungen

Typumwandlung mit Down-Cast

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class T { // Type−ID 2, k=1
  void foo() { (new TextIO()).putString("blah"); return; }
}

class T0 extends T { // Type−ID 3, k = 0
  void bar() { (new TextIO()).putString("fasel"); return; }
}

class Main {
  void main() {
    T0 t0 = new T0();
    Object obj = t0;
    ((T) (obj)).foo(); // Typkonvertierung mit Down−Cast
    t0 = (T0) (null); // ist immer legal
    return;
  }
}
```

Beobachtung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
((T) (obj)).foo();
t0 = (T0) (null);
```

- Zielklasse K_2 der Konvertierung ist bekannt (hier `T`)
- Klasse K_1 der Instanz `obj` ist unbekannt
- `null` ist zu jedem Typ konform
- Wenn K_1 nicht zu K_2 konform, Class Cast Exception auslösen

Dynamische Typprüfung für Down-Cast

MIPS-Assembler

TECHNISCHE UNIVERSITÄT DARMSTADT

```
  lw   $v0 4($fp)          # obj ist in 4($fp)
  beq  $v0 $zero label11   # falls t == null, Down Cast immer OK
  lw   $t0 0($v0)          # hole typeid(obj) nach $t0
  li   $t1 3               # hole typeid(T)+k = 2+1
  bgt  $t0 $t1 label10     # wenn typeid(obj) > typeid(T)+k, Fehler
  li   $t1 2               # hole typeid(T) = 2
  blt  $t0 $t1 label10     # wenn typeid(obj) < typeid(T), Fehler
  b    label11             # alles soweit OK, weitermachen
label10:                   # HIER: Class Cast Exception auslösen ----------
  li   $a1 13              # Zeilennummer dieses Down Casts
  la   $a2 String_const_3  # Dateiname dieses Down Casts
  jal  _class_cast_error   # Fehlermeldung ausgeben und Programm beenden
label11:                   # HIER: Down-Cast OK --------------------
  sw   $v0 8($fp)          # Speicher Zwischenergebnis von Down-Cast
  li   $a1 13              # Zeile des Aufrufes von foo()
  la   $a2 String_const_3  # Dateiname des Aufrufes von foo()
  lw   $a0 8($fp)          # Empfängerobjekt (this) ist Ergebnis von Down-Cast
  bne  $a0 $zero label8    # Empfängerobjekt gültig?
  jal  _null_pointer_error # Nein, Null Pointer Exception
label8:                    # HIER: Aufruf von foo() ausführen -------------
  lw   $t0 8($a0)          # Vtable von Instanz holen
  lw   $t0 12($t0)         # Methode foo() hat Method-ID 3, offset=12 Bytes
  jalr $t0                 # foo() anspringen
```

Typabfrage mit `instanceof`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
obj instanceof T
```

- Vergleichbar zur Typprüfung bei Down-Cast
- Kann aber keine Exception auslösen
- Liefert entweder `true` oder `false`
  - `false` wenn `obj == null` ist

Dynamische Typprüfung für `instanceof`

MIPS-Assembler

TECHNISCHE UNIVERSITÄT DARMSTADT

```
...
  lw    $v0 8($fp)          # lade obj nach $v0
  beq   $v0 $zero label14   # falls obj == null, instanceof -> false
  lw    $t0 0($v0)          # hole typeid(obj)
  li    $t1 3               # hole typeid(T)+k = 2+1
  bgt   $t0 $t1 label14     # wenn typeid(obj) > typeid(T)+k -> false
  li    $t1 2               # hole typeid(T) = 2
  blt   $t0 $t1 label14     # wenn typeid(obj) < typeid(T) -> false
  b     label13             # obj ist != null und konform zu T -> true
label14:                    # HIER: gebe Ergebnis false zurück ----------
  li    $v0 0               # setze Wert für false in Ergebnisregister
  b     label15             # zum Ende der Berechnung
label13:                    # HIER: gebe Ergebnis true zurück ----------
  li    $v0 -1              # setze Wert für true in Ergebnisregister
label15:                    # Ende der Berechnung von instanceof
...                         # Boolesches Ergebnis in Ergebnisregister $v0
```

Zuweisung von Array-Elementen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Animal { ... }                    // Type−ID 3

class Tiger extends Animal { ... } // Type−ID 5

class Bear extends Animal { ... }   // Type−ID 4

// implizit deklariert, werden automatisch erzeugt
// class Animal[] extends Object {} // Type−ID 11
// class Tiger[]  extends Animal[] {} // Type−ID 13
// Array Type−IDs sind Type−IDs der Basistypen
// + Anzahl der nicht−Array−Klassen (hier =8)

class Main {
  void main() {
    Animal[] animals = new Tiger[10];
    animals[0] = new Bear(); // dynamische Typprüfung erforderlich
    return;
  }
}
```

Zuweisung von Array-Elementen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Animal[] animals = new Tiger[10];
animals[0] = new Bear();
```

- Ähnlich wie Typprüfung für Down-Cast
- Aber: Hier K_1 (Typ von RHS) und K_2 (Basistyp des Arrays) variabel

Zuweisung von Array-Elementen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Animal[] animals = new Tiger[10];
animals[0] = new Bear();
```

- Ähnlich wie Typprüfung für Down-Cast
- Aber: Hier K_1 (Typ von RHS) und K_2 (Basistyp des Arrays) variabel
- Vorgehen
  - Zur Laufzeit Typen von LHS und RHS bestimmen
  - Zur Laufzeit Intervall der zu K_2 konformen Type-IDs bestimmen
  - Benötigt zusätzliche Datenstruktur
    - Zu jeder Klasse Anzahl der transitiven Unterklassen

Dynamische Typprüfung für Zuweisung an Array Element

Infrastruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

```
subclass_num_table:
  .word 17 # Unterklassen von Object (Type−ID 0)
  .word  0 # Unterklassen von String (Type−ID 1)
  .word  0 # Unterklassen von Sys (Type−ID 2)
  .word  2 # Unterklassen von Animal (Type−ID 3)
  .word  0 # Unterklassen von Bear (Type−ID 4)
  .word  0 # Unterklassen von Tiger (Type−ID 5)
  .word  0 # Unterklassen von Main (Type−ID 6)
  .word  0 # Unterklassen von TextIO (Type−ID 7)
```

- Bildet Type-IDs auf Anzahl der transitiven Unterklassen ab
- Wird nur benötigt für Basistypen, nicht für die automatisch angelegten Array-Typen

Dynamische Typprüfung für Zuweisung an Array Element

TECHNISCHE UNIVERSITÄT DARMSTADT

MIPS-Assembler

```
...
  lw   $t2 0($t0)              # hole typeid(LHS[]), hier Tiger[], nach $t0. Type-ID 13
  sub  $t2 $t2 8               # berechne typeid(LHS)=Tiger=Type-ID 5, den Basistyp von Tiger[]
  lw   $t1 4($fp)              # hole RHS nach $t1 (neue Bear-Instanz)
  beq  $t1 $zero label8        # wenn null -> Zuweisung immer OK
  lw   $t1 0($t1)              # hole typeid(RHS) = Bear, Type-ID 4
  blt  $t1 $t2 label9          # wenn typeid (RHS) < typeid(LHS), nicht konform -> Fehler
  mul  $t2 $t2 4               # Berechne Offset in subclass_num_table als Type-ID*4 Bytes
  la   $t3 subclass_num_table  # Adressberechnung für Array-Zugriff
  add  $t3 $t2 $t3             # --- " ---
  lw   $t3 0($t3)              # Lese Anzahl transitiver Unterklassen zu typeid(LHS), hier 0
  lw   $t2 0($t0)              # hole typeid(LHS[]), hier Tiger[], nach $t0. Type-ID 13
  sub  $t2 $t2 8               # berechne typeid(LHS)=Tiger=Type-ID 5, den Basistyp von Tiger[]
  add  $t2 $t2 $t3             # Bestimme Obergrenze der konformen Type-IDs in $t3, hier Type-ID 5
  bgt  $t1 $t2 label9          # wenn typeid(RHS) > konforme Obergrenze -> Fehler
  b    label8                  # sonst Typprüfung der Zuweisung OK
label9:                        # HIER: Inkompatible Typen, Fehlermeldung ausgeben ---------
  lw   $t0 0($t0)              # Für Fehlermeldung: $t0 = typeid(LHS[]), $t1 = typeid(RHS)
  jal  _array_store_error      # Fehlermeldung und Programmende
label8:                        # HIER: Zuweisung ist legal ----------------------------
...
```

Hilfsinfrastruktur

Bilde Type-IDs auf Namen ab

Für Fehlermeldungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class_name_table:
  .word String_const_1  # "Object"
  .word String_const_2  # "String"
  .word String_const_3  # "Sys"
  .word String_const_5  # "Animal"
  .word String_const_6  # "Bear"
  .word String_const_7  # "Tiger"
  .word String_const_8  # "Main"
  .word String_const_11 # "TextIO"
  .word 0               # nicht benutzt, wäre Object[]
  .word 0               # nicht benutzt, wäre String[]
  .word 0               # nicht benutzt, wäre Sys[]
  .word String_const_9  # "Animal[]"
  .word 0               # nicht benutzt, wäre Bear[]
  .word String_const_10 # "Tiger[]"
  .word 0               # nicht benutzt, wäre Main[]
  .word 0               # nicht benutzt, wäre TextIO[]
  .word 0               # nicht benutzt, wäre int[] (Sonderfall!)
  .word 0               # nicht benutzt, wäre boolean[] (Sonderfall!)
```

- Für skalare und Array-Klassen
  - Enthält Einträge für alle potentiellen Array-Klassen
  - Bleiben leer, wenn Array-Klassen nicht benutzt wurden

Organisation des Bantam-Compilers

Allgemeine Schnittstelle im Middle-End

TECHNISCHE UNIVERSITÄT DARMSTADT

Bekommt als Eingabedaten

- Start Basisblock des CFGs
- Name der Methode
- Ob die Operation mit Debug-Ausgaben (*verbose*) erfolgen soll

Allgemeine Schnittstelle im Middle-End

TECHNISCHE UNIVERSITÄT DARMSTADT

Bekommt als Eingabedaten

- Start Basisblock des CFGs
- Name der Methode
- Ob die Operation mit Debug-Ausgaben (*verbose*) erfolgen soll

Realisiert in Form des Interfaces `opt.Optimization`

- Wird automatisch auf alle Methoden in allen Klassen angewandt
- Einschließlich der internen `_init` Methoden
- Fehlerbehandlung durch Auslösen von `Exception`

Allgemeine Schnittstelle im Middle-End

TECHNISCHE UNIVERSITÄT DARMSTADT

Bekommt als Eingabedaten

- Start Basisblock des CFGs
- Name der Methode
- Ob die Operation mit Debug-Ausgaben (*verbose*) erfolgen soll

Realisiert in Form des Interfaces `opt.Optimization`

- Wird automatisch auf alle Methoden in allen Klassen angewandt
- Einschließlich der internen `_init` Methoden
- Fehlerbehandlung durch Auslösen von `Exception`

JavaDoc zur ESA-Version des Compilers liegt auf Web-Seite als
`bantam-api-min.zip`

Allgemeine Schnittstelle im Middle-End

Parameterübergabe

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Optimization {
  /**
   * Sets the current root of a CFG.
   * @param block BasicBlock
   */
  public void setEntryBlock(final BasicBlock block);

  /**
   * Sets the current CFGs method name.
   * @param name Name
   */
  public void setMethodName(final String name);

  /**
   * Sets verbose mode.
   * @param verbose on/off.
   */
  public void setVerbose(final boolean verbose);

  /**
   * Returns true, if in verbose mode.
   * @return verbose?
   */
  public boolean isVerbose();
```

Allgemeine Schnittstelle im Middle-End

Aktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/**
 * Defines a generic interface for Optimizations:
 * Two phase operation, analysis and transformation.
 */
public interface Optimization {

  ...

  /**
   * Perform analysis for optimization.
   */
  public void analyze();

  /**
   * Perform actual optimization on CFG.
   * @return new start basic block of CFG
   */
  public BasicBlock transform();
}
```

Anwenden auf CFG

Beispiel aus `example_driver`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import opt.Optimization;
...
// Vorher Lexing, Parsing, AST−Erzeugung, Semantische Analyse und CFG−Erzeugung


// Erzeuge Instanz von neuem Optimierungspass
// muß Interface Optimization implementieren
Optimization myOpt = new MyNewOptimizationPass(MyNewOptimizationPass.DOEVERYTHING, "logfile.txt");

// Weise bestehenden Optimierungsverwalter an, den neuen Pass auf
// auf alle Methoden aller Klassen anzuwenden
optimizer.optimize(myOpt);

// Nun Code−Erzeugung aus CFG
...
```

Konvertieren von CFG nach SSA-CFG

Bereits im Framework realisiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Ebenfalls als Optimierungspass realisiert

```
import opt.ssa.CFG2SSA;
...
// Vorher normale CFG−Erzeugung

// Optimierungspass für SSA−Konvertierung anlegen und konfigurieren
CFG2SSA cfg2ssa = new CFG2SSA();
cfg2ssa.setVerbose(false);

// Pass auf alle CFGs aller Methoden aller Klassen anwenden
optimizer.optimize(cfg2ssa);

// Ab jetzt CFGs aller Methoden aller Klassen in SSA−CFG Form
...
```

Konvertieren von SSA-CFG nach CFG

Bereits im Framework realisiert

TECHNISCHE UNIVERSITÄT DARMSTADT

Ebenfalls als Optimierungspass realisiert

```
import opt.ssa.PhiRemover;
...
// Bisher SSA−CFG Form

// Optimierungspass für SSA−Konvertierung anlegen und konfigurieren
PhiRemover phiremover = new PhiRemover();
phiremover.setVerbose(false);

// Pass auf alle CFGs aller Methoden aller Klassen anwenden
optimizer.optimize(phiremover);

// Ab jetzt CFGs aller Methoden aller Klassen in CFG Form
...
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Zwischendarstellung für Optimierung

Kontrollflußgraph

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beschrieben durch `cfg.BasicBlock`
  - Doku in Abschnitt des Bantam Extended Lab Manuals (p. 90ff)

Kontrollflußgraph

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beschrieben durch `cfg.BasicBlock`
  - Doku in Abschnitt des Bantam Extended Lab Manuals (p. 90ff)
- Kanten
  - Nicht explizit modelliert, aber maximal zwei Ausgangskanten
  - Dargestellt als Mengen von Vorgänger/Nachfolger-Blöcken
  - Beispiel `int getNumOutEdges()`, `BasicBlock getNthOutEdge(int n)`

Kontrollflußgraph

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beschrieben durch `cfg.BasicBlock`
  - Doku in Abschnitt des Bantam Extended Lab Manuals (p. 90ff)
- Kanten
  - Nicht explizit modelliert, aber maximal zwei Ausgangskanten
  - Dargestellt als Mengen von Vorgänger/Nachfolger-Blöcken
  - Beispiel `int getNumOutEdges()`, `BasicBlock getNthOutEdge(int n)`

- Hilfsfunktionen
  - Konsistenzprüfung: `void check()`
  - Ausgabe eines Blockes: `void print()`
  - Des ganzen CFGs: `void printAll()` auf Startblock

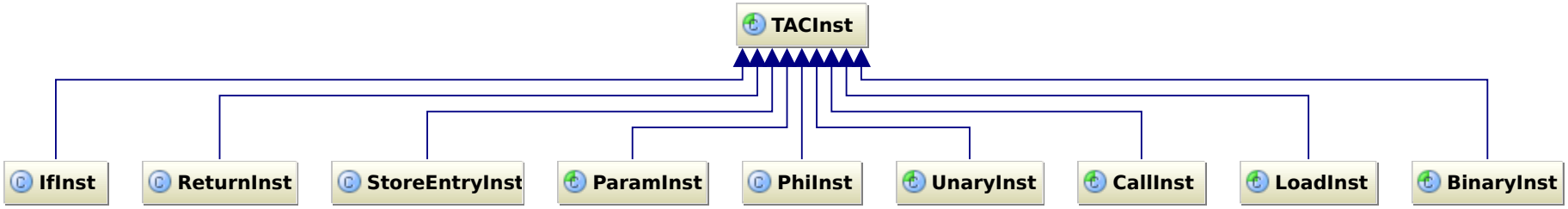
Kontrollflußgraph

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beschrieben durch `cfg.BasicBlock`
  - Doku in Abschnitt des Bantam Extended Lab Manuals (p. 90ff)
- Kanten
  - Nicht explizit modelliert, aber maximal zwei Ausgangskanten
  - Dargestellt als Mengen von Vorgänger/Nachfolger-Blöcken
  - Beispiel `int getNumOutEdges()`, `BasicBlock getNthOutEdge(int n)`
- Hilfsfunktionen
  - Konsistenzprüfung: `void check()`
  - Ausgabe eines Blockes: `void print()`
  - Des ganzen CFGs: `void printAll()` auf Startblock
- Anweisungen
  - Dargestellt als Listen von Drei-Address-Instruktionen (TAC)
  - `List<TACInst> getInstructions()`, `int getNumInsn()`

Drei-Address-Code

three-address code (TAC)



Schon vorbereitet für Benutzung durch Visitor

```
Object accept(cfg.TACInstVisitor visitor, Object o)
```

Operanden in TAC-Anweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Variablen, noch keine Register

Operanden in TAC-Anweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Variablen, noch keine Register
- Alle identifiziert durch Strings
  - An sich unschön, vereinfacht aber Bantam-Compiler
  - Kaum Lookups in Symboltabellen, direkt den String prüfen

Operanden in TAC-Anweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Variablen, noch keine Register
- Alle identifiziert durch Strings
  - An sich unschön, vereinfacht aber Bantam-Compiler
  - Kaum Lookups in Symboltabellen, direkt den String prüfen
- Namenskonventionen in Lab Manual Abschnitt 6.2 (p. 91ff)
- Suffixe beschreiben Geltungsbereiche
  - `@l`: Lokale Variable (z.B. `foo@f`
  - `@p`: Formaler Parameter (z.B. `bar@p`
  - `@f_`*Klassenname*: Attribut von *Klassenname* (z.B. `color@f_Shape`)

Operanden in TAC-Anweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Variablen, noch keine Register
- Alle identifiziert durch Strings
  - An sich unschön, vereinfacht aber Bantam-Compiler
  - Kaum Lookups in Symboltabellen, direkt den String prüfen
- Namenskonventionen in Lab Manual Abschnitt 6.2 (p. 91ff)
- Suffixe beschreiben Geltungsbereiche
  - `@l`: Lokale Variable (z.B. `foo@f`
  - `@p`: Formaler Parameter (z.B. `bar@p`
  - `@f_`*Klassenname*: Attribut von *Klassenname* (z.B. `color@f_Shape`)
- Temporäre Variablen `@t`*zahl* (z.B. `@t42`)
- This `this`
- Kein Super, direkt auf Attribut oder Methode verweisen

Operanden in TAC-Anweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Variablen, noch keine Register
- Alle identifiziert durch Strings
  - An sich unschön, vereinfacht aber Bantam-Compiler
  - Kaum Lookups in Symboltabellen, direkt den String prüfen
- Namenskonventionen in Lab Manual Abschnitt 6.2 (p. 91ff)
- Suffixe beschreiben Geltungsbereiche
  - `@l`: Lokale Variable (z.B. `foo@f`
  - `@p`: Formaler Parameter (z.B. `bar@p`
  - `@f_`*Klassenname*: Attribut von *Klassenname* (z.B. `color@f_Shape`)
- Temporäre Variablen `@t`*zahl* (z.B. `@t42`)
- This `this`
- Kein Super, direkt auf Attribut oder Methode verweisen
- Literale
  - Integer, `true`, `false`, `null`, `"string"`

Bedingte Ausführung: IfInst

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bedingung: `EQ, GE, GT, LE, LT, NE`
  - `getType()`, `setType()`
- Linker/Rechter Operand des Vergleichs
  - `String getLeftSource()`, `String getRightSource()`
  - `setLeftSource(String l)`, `setRightSource()`
- Sprungziele: Basisblocks
  - `BasicBlock getFalseTarg()`, `BasicBlock getTrueTarg()`
  - `setFalseTarg(BasicBlock falseTarg)`, `setTrueTarg(BasicBlock trueTarg)`

Bedingte Ausführung: IfInst

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bedingung: `EQ, GE, GT, LE, LT, NE`
  - `getType()`, `setType()`
- Linker/Rechter Operand des Vergleichs
  - `String getLeftSource()`, `String getRightSource()`
  - `setLeftSource(String l)`, `setRightSource()`
- Sprungziele: Basisblocks
  - `BasicBlock getFalseTarg()`, `BasicBlock getTrueTarg()`
  - `setFalseTarg(BasicBlock falseTarg)`, `setTrueTarg(BasicBlock trueTarg)`

Beispiel:

```
if (@t13 != null) goto bb18;
goto bb17;
```

Unterprogrammaufruf: CallInst

`DirCallInst`, `IndirCallInst`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Anzahl Parameter
  - `setNumParam(int i)`, `int getNumParam()`
- Optional: Zielvariable für Rückgabewert (`null` wenn keine)
  - Als String gemäß Namenskonventionen für TAC-Anweisungen
  - `String getDestVar()`, `setDestVar(String varname)`
- Sprungziel `setTarget(String target)`
  - Direkter Sprung `DirCallInst`: Routinenname gemäß MIPS-Konventionen
  - Indirekter Sprung `IndirCallInst`: Variablenname mit Sprungzieladresse

Unterprogrammaufruf: CallInst

`DirCallInst`, `IndirCallInst`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Anzahl Parameter
  - `setNumParam(int i)`, `int getNumParam()`
- Optional: Zielvariable für Rückgabewert (`null` wenn keine)
  - Als String gemäß Namenskonventionen für TAC-Anweisungen
  - `String getDestVar()`, `setDestVar(String varname)`
- Sprungziel `setTarget(String target)`
  - Direkter Sprung `DirCallInst`: Routinenname gemäß MIPS-Konventionen
  - Indirekter Sprung `IndirCallInst`: Variablenname mit Sprungzieladresse

Beispiel:

```
// direkter Sprung
dircall Object_init, 0;      // keine Parameter, kein Ergebnis

// indirekter Sprung
@t4 = Object_dispatch_table;
@t5 = @t4[2];
@t3 = indircall @t5, 1;      // ein Parameter, mit Ergebnis
```

Parameterübergabe: ParamInst

`StdParamInst`, `RefParamInst`, `ErrParamInst`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Quelle des übergebenen Wertes als String
  - `String getSource()`, `setSource(String valorvarname)`
- Zielobjekt mit `RefParamInst` (genau eines bei Methodenaufruf)
- Aktuelle Parameter mit `StdParamInst`
- Fehlerparameter für Exception-Handler mit `ErrParamInst`
  - Braucht auch noch Art des Fehlerparameters
  - `FILENAME`, `LINENUM`, `OBJECTID`, `TARGETID`, `ARRAYIDXID`, ...
- Als erstes Empfängerobjekt, dann aktuelle Parameter von links nach rechts

Beispiel: Parameterübergabe / Methodenaufruf

`tmp = t.bar(42, 23)`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
refparam t@l;  // Empfängerobjekt
stdparam 42;   // aktuelle Parameter
stdparam 23;
errparam filename, "TestInherit.btm";
errparam linenum, 78;
if (t@l != null) goto bb50;
goto bb49;

# basic block near source line 78
bb50:
@t41 = t@l[2];   // vtable von t
@t42 = @t41[5];  // Method-ID 5
tmp@l = indircall @t42, 3;
...

# basic block near source line 78
bb49:
dircall _null_pointer_error, 0;
goto bb50;
```

Rücksprung: ReturnInst

TECHNISCHE UNIVERSITÄT DARMSTADT

- Rückkehr von Unterprogramm
- Optional mit Rückgabewert
  - `String getSource()`, `setSource(String valorvarname)`
  - `null` wenn kein Rückgabewert

```
// int bar(int a, int b) { return a * b; }
T.bar:
# basic block near source line 12
bb20:
@t17 = a@p * b@p;
retn @t17;
```

Werte in Variablen laden: LoadInst

`LoadAddrInst`, `LoadConstInst`, `LoadEntryInst`, `LoadVarInst`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zielvariable: `String getDestination()`, `setDestination(String varname)`
- Lade von Variable `LoadVarInst`
  - Quellvariable als Name `String getSource()`, `setSource(String varname)`

Werte in Variablen laden: LoadInst

`LoadAddrInst`, `LoadConstInst`, `LoadEntryInst`, `LoadVarInst`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zielvariable: `String getDestination()`, `setDestination(String varname)`
- Lade von Variable `LoadVarInst`
  - Quellvariable als Name `String getSource()`, `setSource(String varname)`
- Lade Konstante `LoadConstInst`
  - Typ der Konstante: Integer, Boolean, String, `null`
  - Wert der Konstante `String getSource()`, `setSource(String literal)`

Werte in Variablen laden: LoadInst

`LoadAddrInst`, `LoadConstInst`, `LoadEntryInst`, `LoadVarInst`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zielvariable: `String getDestination()`, `setDestination(String varname)`
- Lade von Variable `LoadVarInst`
  - Quellvariable als Name `String getSource()`, `setSource(String varname)`
- Lade Konstante `LoadConstInst`
  - Typ der Konstante: Integer, Boolean, String, `null`
  - Wert der Konstante `String getSource()`, `setSource(String literal)`
- Lade Adresse, gegeben durch Label im Programm `LoadAddrInst`
  - Label `String getSource()`, `setSource(String label)`

Werte in Variablen laden: LoadInst

`LoadAddrInst`, `LoadConstInst`, `LoadEntryInst`, `LoadVarInst`

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zielvariable: `String getDestination()`, `setDestination(String varname)`
- Lade von Variable `LoadVarInst`
  - Quellvariable als Name `String getSource()`, `setSource(String varname)`
- Lade Konstante `LoadConstInst`
  - Typ der Konstante: Integer, Boolean, String, `null`
  - Wert der Konstante `String getSource()`, `setSource(String literal)`
- Lade Adresse, gegeben durch Label im Programm `LoadAddrInst`
  - Label `String getSource()`, `setSource(String label)`
- Lade Speicherwort `LoadEntryInst`
  - Basisadresse ist Variable oder String-Literal
    - `String getSource()`, `setSource(String varorstring)`
  - Offset ist Variable oder Integer-Literal > 0
    - `String getIndex()`, `setIndex(String idx)`

Beispiel: Laden von Werten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Load variable
canswim@f_Animal = canswim@p;
...

// Load Constant (integer)
numwheels@f_Car = 4;
...

// Load Address
@t11 = TextIO_template;
refparam @t11;
errparam filename, "TestInherit.btm";
errparam linenum, 9;
@t12 = dircall Object.clone, 1;
...

// Load Entry
@t41 = t@l[2];   // vtable von t
@t42 = @t41[5];  // Method-ID 5
```

Schreibe Speicherwort: StoreEntryInst

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ziel: Basisadresse (Variable, String-Literal) plus Offset (Variable, Integer-Literal$>$ 0)
  - Basisadresse `String getDestination()`, `setDestination(String varorstring)`
  - Offset `String getIndex()`, `setIndex(String varorint)`
- Quelle: Variable oder Literal (Bool, String, Int, `null`)

Beispiel:

```
// int[] a;
// a[i] = 42;
a@l[i@l] = 42;
```

Arithmetische und Logische Operatoren: UnaryInst und BinaryInst

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ziel für evaluierten Ausdruck: Variable
  - `String getDestination()`, `setDestination(String varname)`

Arithmetische und Logische Operatoren: UnaryInst und BinaryInst

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ziel für evaluierten Ausdruck: Variable
  - `String getDestination()`, `setDestination(String varname)`
- Unäre Operatoren `UnaryNegInst`, `UnaryNotInst`
  - Operand Neg `String getSource()`, `setSource(String varorint)`
  - Operand Not `String getSource()`, `setSource(String varorbool)`

Arithmetische und Logische Operatoren: UnaryInst und BinaryInst

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ziel für evaluierten Ausdruck: Variable
  - `String getDestination()`, `setDestination(String varname)`
- Unäre Operatoren `UnaryNegInst`, `UnaryNotInst`
  - Operand Neg `String getSource()`, `setSource(String varorint)`
  - Operand Not `String getSource()`, `setSource(String varorbool)`
- Binäre Operatoren
  - Operanden Arithmetische `BinaryAddInst`, `BinarySubInst`, ...
    - `String getLeftSource()`, `String getRightSource()`
    - `setLeftSource(String varorint)`, `setRightSource(String varorint)`
  - Operanden Logische `BinaryAndInst`, `BinaryOrInst`
    - `String getLeftSource()`, `String getRightSource()`
    - `setLeftSource(String varorbool)`, `setRightSource(String varorbool)`

Beispiel: Binäre und unäre Operatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@t0 = @t1 * @t2;
@t3 = @t0 * 4;
@t4 = 4 + @t3;

isflyingfish@l = canswim@f_Animal && canfly@f_Animal;
@t8 = isflyingfish@l && true;
```

Phi-Instruktion: PhiInst

TECHNISCHE UNIVERSITÄT DARMSTADT

- Realisiert Phi-Funktion in Join-Knoten der SSA-Form
- Ziel: `String getDestination()`, `setDestination(String varname)`
- Parameter: Einer für jede eingehende Kante, in gleicher Reihenfolge
  - `int getNumParams()`
  - `String getNthSource(final int n)`
  - `setNthSource(final int n, final String varname)`

Phi-Instruktion: PhiInst

TECHNISCHE UNIVERSITÄT DARMSTADT

- Realisiert Phi-Funktion in Join-Knoten der SSA-Form
- Ziel: `String getDestination()`, `setDestination(String varname)`
- Parameter: Einer für jede eingehende Kante, in gleicher Reihenfolge
  - `int getNumParams()`
  - `String getNthSource(final int n)`
  - `setNthSource(final int n, final String varname)`

Beispiel:

```
@t100003 = phi(@t100001, @t100002);
```

Experimente mit CFG und TAC

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zum Experimentieren kann der Bantam Referenz-Compiler benutzt werden
- Textuelle Ausgabe des CFG mit den TAC-Instruktionen durch

`bin/bantamc-jj-ref -opt 1 -so HelloWorld.btm`

```
Main.main:
# basic block near source line 2
bb7:
@t1 = TextIO_template; # load address instruction
refparam @t1;
errparam filename, "HelloWorld.btm";
errparam linenum, 3;
@t2 = dircall Object.clone, 1;
refparam @t2;
@t3 = dircall TextIO_init, 1;
refparam @t3;
stdparam "Hello, World!
";
errparam filename, "HelloWorld.btm";
...
```

Diskussion CFG und TAC

TECHNISCHE UNIVERSITÄT DARMSTADT

- Keine unbedingten Sprünge
  - Werden implizit durch Kanten zwischen Blöcken realisiert
  - Textausgabe des CFG “phantasiert” `gotos` als Pseudoinstruktionen dazu
- Wenn Block zwei ausgehende Kanten hat, muss er mit `IfInst` enden
- Blöcke ohne ausgehende Kanten müssen enden mit
  - `ReturnInst`
  - Aufruf einer Exception Handler-Routine
- Alle Blöcke mit Ausnahme des CFG-Startblockes müssen mindestens eine Eingangskante haben
- Alle TAC-Instruktionen sind auf Visitor vorbereitet.

Beispiel: CFG/TAC-Visitor

`LoadAddrInst` **und Visitor-Interface** `TACInstVisitor`

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class LoadAddrInst extends LoadInst {
  ...
  public Object accept(TACInstVisitor visitor, Object o) {
    return visitor.visitInst(this, o);
  }
}
```

```
public interface TACInstVisitor {
  abstract Object visitInst(BinaryAddInst inst, Object o);
  ...
  abstract Object visitInst(LoadAddrInst inst, Object o);
  ...
  abstract Object visitInst(UnaryNegInst inst, Object o);
}
```

Beispiel: CFG/TAC-Visitor

Code-Generierung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
...
  public Object visitInst (LoadAddrInst inst, Object o) {
    final String src = inst.getSource();
    final String dest = inst.getDestination();
    final String dreg =   MipsSupport.isRegister(dest)
                        ? dest
                        : assemblySupport.getResultReg();

    // la dreg src
    assemblySupport.genLoadAddr(dreg, src);

    return null;
  }
...
```