PyPy

Mehr als eine Implementation von Python in Python

Henning Pridöhl

9. Januar 2014

1. Einführung
2. Interpreter
   - Kompilieren zu Bytecode
   - Interpretation – Eine einfache virtuelle Maschine
   - Interpreter und Object Spaces
3. Aufbau des Translation Frameworks
   - Restriktionen in RPython
   - Übersicht des "Kompiliervorganges"
   - Flow Graph
   - Annotator
   - RTyper
   - RTyped Flow Graph → C-Code
4. Garbage Collectoren
5. JIT im Detail
   - Wie funktioniert ein Tracing JIT
   - JIT in PyPy

1. Einführung
2. Interpreter
   - Kompilieren zu Bytecode
   - Interpretation – Eine einfache virtuelle Maschine
   - Interpreter und Object Spaces
3. Aufbau des Translation Frameworks
   - Restriktionen in RPython
   - Übersicht des "Kompiliervorganges"
   - Flow Graph
   - Annotator
   - RTyper
   - RTyped Flow Graph → C-Code
4. Garbage Collectoren
5. JIT im Detail
   - Wie funktioniert ein Tracing JIT
   - JIT in PyPy

Was ist PyPy?

- Eine Implementation von Python in (R)Python
- Ein Framework um Interpreter und virtuelle Maschinen insbesondere für dynamische Programmiersprachen zu schreiben.
- Eine Implementation von Prolog in (R)Python
- Eine (unvollständige) Implementation von Scheme in (R)Python

Python-Implementation (Interpreter)

- Implementiert bisher Python 2.7 und 3.2
- Just-In-Time-Compiler (x86, x86_64, ARM in Arbeit)
- Vernünftiger Garbage Collector – kein Reference Counting

Beispiel, bei dem man den anderen Garbage Collector merkt

```
open('dateiname', 'w').write('inhalt')
```

- Keine vollständige Kompatibilität mit CPython-Erweiterungen (cpyext . . .)

Translation Framework

- Generiert aus einen in RPython geschriebenen Interpreter eine Binary über C-Code als Zwischenrepräsentation. [1]
- Baut automatisch einen Garbage Collector ein (austauschbar zur Compile-Zeit!)
- Baut nahezu automatisch einen JIT-Compiler ein

[1] theoretisch auch andere Backends, die werden aber nicht mehr gepflegt

1 Einführung

2 Interpreter
- Kompilieren zu Bytecode
- Interpretation – Eine einfache virtuelle Maschine
- Interpreter und Object Spaces

3 Aufbau des Translation Frameworks
- Restriktionen in RPython
- Übersicht des “Kompiliervorganges”
- Flow Graph
- Annotator
- RTyper
- RTyped Flow Graph → C-Code

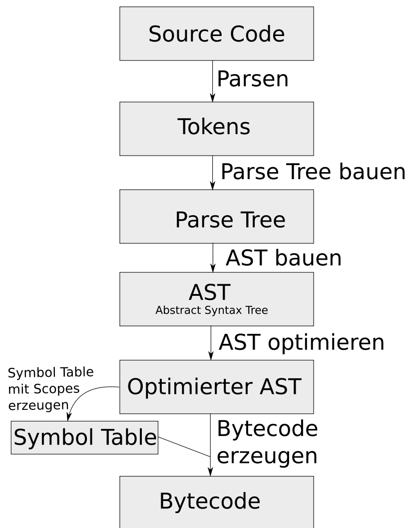
4 Garbage Collectoren

5 JIT im Detail
- Wie funktioniert ein Tracing JIT
- JIT in PyPy

Wie funktioniert eine VM-Implementation in PyPy

- Bytecode compiler: Quelltext → Bytecode
- Bytecode evaluator: Interpretiert den Bytecode
- Standard Object Space: Erstellt und manipuliert Objekte.

Kompilieren zu Bytecode



Einfache virtuelle Maschine

- Ein Akkumulator `A`, 256 Register
- 6 Anweisungen:
  - `JUMP_IF_A` *addr* – Springt zu Adresse, wenn der Akkumulator positiv ist.
  - `MOV_A_R` *n* – Speichert den Wert des Akkumulators in das *n*te Register.
  - `MOV_R_A` *n* – Speichert den Wert des *n*ten Registers in den Akkumulator
  - `ADD_R_TO_A` *n* – Addiert den Wert des *n*ten Registers auf den Akkumulator
  - `DECR_A` – Dekrementiert den Wert des Akkumulators um 1
  - `RETURN_A` – Gibt den Wert des Akkumulators zurück.

Beispiel aus:

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski and Armin Rigo

Tracing the meta-level: PyPy's tracing JIT compiler.

http://codespeak.net/pypy/extradoc/talk/icooolps2009/bolz-tracing-jit-final.pdf

Einfache virtuelle Maschine

```
def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        opcode = ord(bytecode[pc])
        pc += 1
        if opcode == JUMP_IF_A:
            target = ord(bytecode[pc])
            pc += 1
            if a:
                pc = target
        elif opcode == MOV_A_R:
            n = ord(bytecode[pc])
            pc += 1
            regs[n] = a
```

und weiter ...

Einfache virtuelle Maschine (fortges.)

```
16          elif opcode == MOV_R_A:
17              n = ord(bytecode[pc])
18              pc += 1
19              a = regs[n]
20          elif opcode == ADD_R_TO_A:
21              n = ord(bytecode[pc])
22              pc += 1
23              a += regs[n]
24          elif opcode == DECR_A:
25              a -= 1
26          elif opcode == RETURN_A:
27              return a
```

Einfacher Bytecode zum Quadrat berechnen

```
MOV_A_R     0 # i = a
MOV_A_R     1 # copy of 'a'
# 4:
MOV_R_A     0 # i--
DECR_A
MOV_A_R     0
MOV_R_A     2 # res += a
ADD_R_TO_A 1
MOV_A_R     2
MOV_R_A     0 # if i!=0: goto 4
JUMP_IF_A  4
MOV_R_A     2 # return res
RETURN_A
```

Object Spaces

Interpreter delegiert die eigentlichen Operationen an den Object Space. Dieser führt die an den Objekten durch.

Standard Object Space Implementiert die Operationen und Funktionen des Object Spaces mit der normalen Python-Semantik.

Flow Object Space Führt die Operationen nicht durch, sondern erstellt einen Flow Graph. ($\rightarrow$ Translation Framework)

Trace Object Space Zeigt Operationen an und leitet diese an einen anderen Object Space weiter (meist Standard Object Space).

Operationen im Object Space

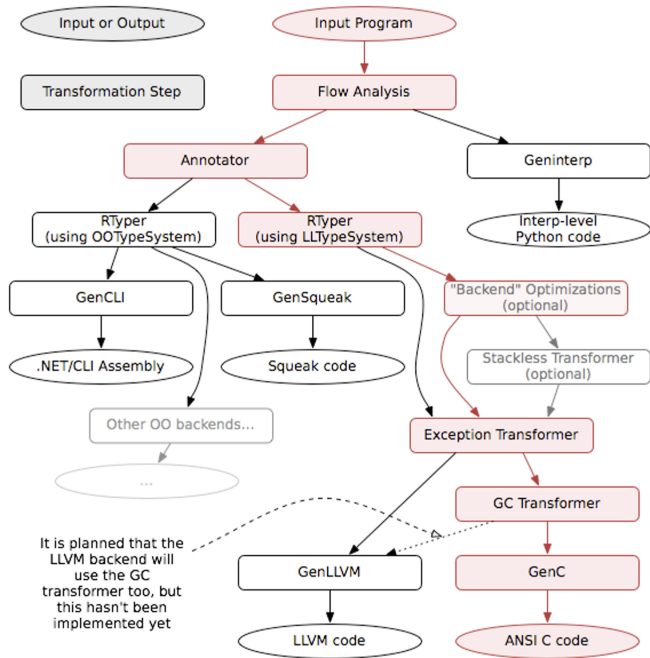
```
id, type, issubtype, iter, next, repr, str, len, hash,
getattr, setattr, delattr, getitem, setitem, delitem,
pos, neg, abs, invert, add, sub, mul, truediv,
floordiv, div, mod, divmod, pow, lshift, rshift, and_,
or_, xor, nonzero, hex, oct, int, float, long, ord,
lt, le, eq, ne, gt, ge, cmp, coerce, contains,
inplace_add, inplace_sub, inplace_mul,
inplace_truediv, inplace_floordiv, inplace_div,
inplace_mod, inplace_pow, inplace_lshift,
inplace_rshift, inplace_and, inplace_or, inplace_xor,
get, set, delete, userdel, call, index, is_,
isinstance, exception_match
```

1. Einführung
2. Interpreter
   - Kompilieren zu Bytecode
   - Interpretation – Eine einfache virtuelle Maschine
   - Interpreter und Object Spaces
3. Aufbau des Translation Frameworks
   - Restriktionen in RPython
   - Übersicht des “Kompiliervorganges”
   - Flow Graph
   - Annotator
   - RTyper
   - RTyped Flow Graph $\rightarrow$ C-Code
4. Garbage Collectoren
5. JIT im Detail
   - Wie funktioniert ein Tracing JIT
   - JIT in PyPy

RPython - Untermenge von Python mit Restriktionen

- Variablen können maximal einen Typ haben. `None` darf mit Listen/Dictionaries/Objekten gemischt werden, jedoch nicht mit `int` und `float`.
- Klassen und Funktionsdefinitionen dürfen nicht zur Laufzeit geändert werden oder neu hinzukommen. Attribute/Methoden von Klassen können zur Laufzeit nicht geändert werden.
- Globale Modulvariablen sind Konstanten, sie dürfen nicht zur Laufzeit geändert werden.
- Kein `yield` und somit auch keine Generators, Coroutinen
- for-Schleifen nur mit builtin types.
- ... noch paar kleine weitere

PyPy: Translation Process

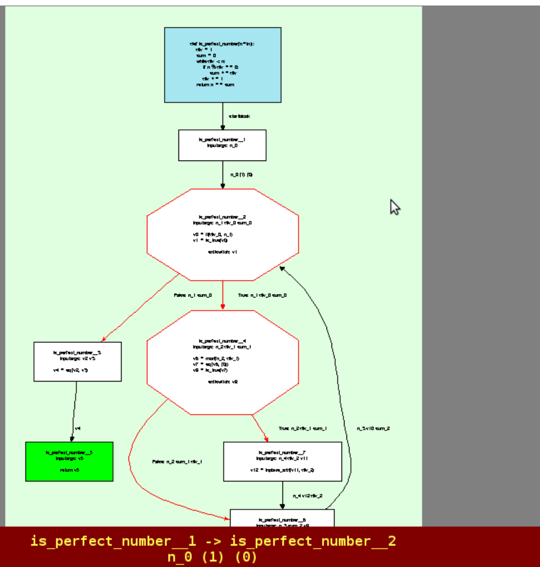

Input or Output
Transformation Step
Input Program
Flow Analysis
Annotator
Geninterp
Interp-level
Python code
RTyper
(using OOTypeSystem)
RTyper
(using LLTypeSystem)
GenCLI
GenSqueak
"Backend" Optimizations
(optional)
.NET/CLI Assembly
Squeak code
Stackless Transformer
(optional)
Other OO backends...
...
Exception Transformer
GC Transformer
It is planned that the
LLVM backend will
use the GC
transformer too, but
this hasn't been
implemented yet
GenLLVM
GenC
LLVM code
ANSI C code

Flow Graph

Pro Funktion wird genau ein Flow Graph generiert!

- **FunctionGraph**(startblock, returnblock, exceptblock)
- **Block**(inputargs, operations, exitswitch, exits)
- **Link**(prevblock, target, args, exitcase, last_exception, last_exc_value)
- **SpaceOperation**(opname, args, result)
- **Variable**(name)
- **Constant**(value, key)

Flowgraph

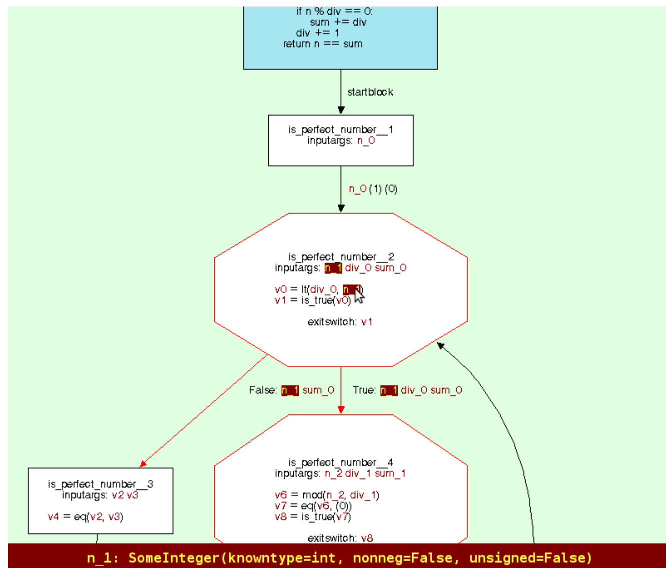


Annotator

Jeder Variable wird genau ein Typ zugeordnet.

- `SomeObject` – Basisklasse für alle Annotations
- `SomeInteger`
- `SomeString`, `SomeChar`
- `SomeTuple([s1, s2, ..., sn])` – Ein Tupel, der einzelne Typen enthält, z. B. `SomeTuple(SomeInteger(), SomeString())`
- `SomeList`
- `SomeDict`
- `SomeInstance`

Annotator



RTyper

- Verbindet annotierten Flow Graph mit dem Low-Level-Code-Generator (z. B. GenC)
- Übersetzt hohe Typen (`SomeInteger`, `SomeString` etc.) in Low-Level-Typen. Setzt Attribut `concretetype` der Variablen.
- Abstrakte Operationen werden in konkrete Operationen konvertiert. Möglicherweise werden Methoden hinzugefügt, für Operationen auf Low-Level-Typen.
- Low-Level-Typen: `Signed`, `Unsigned`, `Float`, `Char`, `Bool`, `Void`, `Struct`, `GcStruct`, `Array`, `GcArray`, `Ptr`, `FuncType`

Beispiel

`v3 = add(v1, v2)` → `v3 = int_add(v1, v2)`
`v1, v2, v3` wurden vom Annotator als SomeInteger annotiert.
`(v1|v2|v3).concretetype` wird vom RTyper auf Signed gesetzt

Backend Optimizations

- Function Inlining
- Malloc Removal
- (Stackless Transform)

Transformers

C-Code kann weder Exceptions noch hat es einen Garbage Collector. Diese werden durch Transformer eingefügt.

Exception Transformer
: Fügt Code fürs Exception-Handling ein. Spezielle Rückgabewerte für Exceptions etc.

GC Transformer
: Fügt Code für die Garbage Collection ein. Es sind mehrere Algorithmen implementiert, die man auswählen kann.

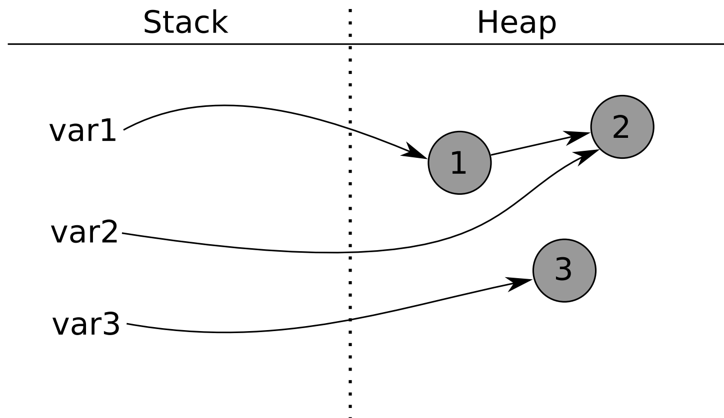
Garbage Collectoren

- Räumt nicht mehr benötigte Objekte auf dem Heap weg.
- Zeitpunkt des Aufräumens hängt vom Algorithmus ab.

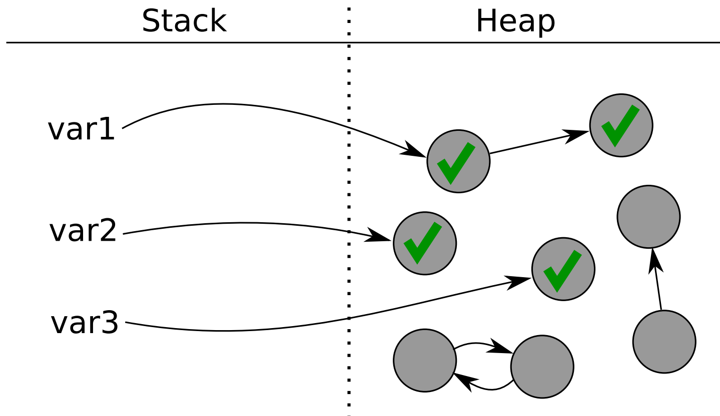
Verfügbare Algorithmen

- Reference Counting
- Mark and Sweep
- Semispace copying collector
- Generational GC (Spezialisierung des Semispace CGC, 2 Generationen)
- Hybrid GC (wie Generational GC jedoch mit externen Platz der per Mark and Sweep verwaltet wird.)
- Mark and Compact GC
- Incremental GC

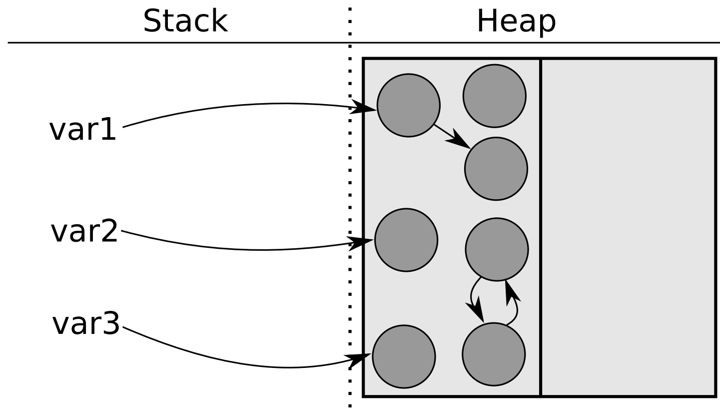
Reference Counting



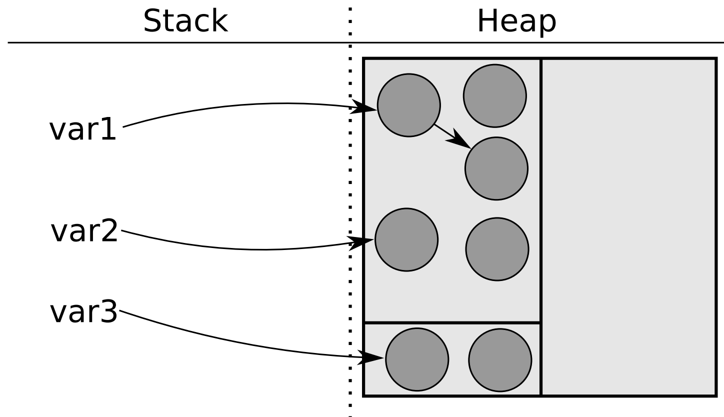
Mark and Sweep



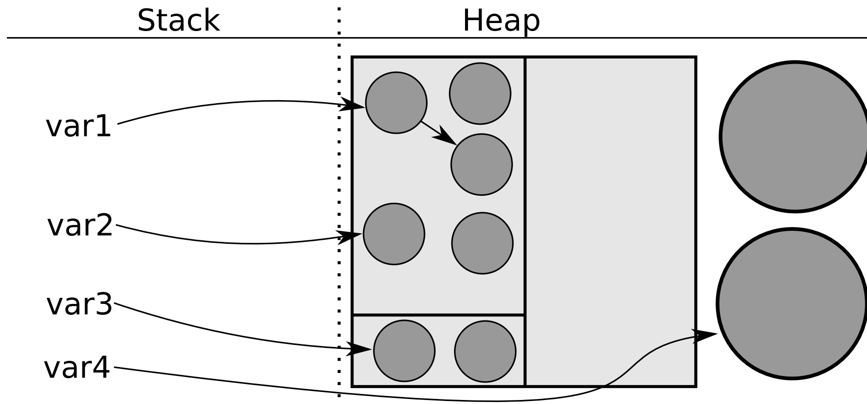
Semispace copying collector



Generational GC



Hybrid GC



1 Einführung

2 Interpreter
- Kompilieren zu Bytecode
- Interpretation – Eine einfache virtuelle Maschine
- Interpreter und Object Spaces

3 Aufbau des Translation Frameworks
- Restriktionen in RPython
- Übersicht des “Kompiliervorganges”
- Flow Graph
- Annotator
- RTyper
- RTyped Flow Graph → C-Code

4 Garbage Collectoren

5 JIT im Detail
- Wie funktioniert ein Tracing JIT
- JIT in PyPy

Ein Beispiel der Hot-Loop-Erkennung

```
1  def f(a, b):
2      if b % 46 == 41:
3          return a - b
4      else:
5          return a + b
6  def strange_sum(n):
7      result = 0
8      while n >= 0:
9          result = f(result, n)
10         n -= 1
11     return result
```

Ein Beispiel der Hot-Loop-Erkennung: Trace

```
1 loop_header(result0, n0)
2 i0 = int_mod(n0, Const(46))
3 i1 = int_eq(i0, Const(41))
4 guard_false(i1)
5 result1 = int_add(result0, n0)
6 n1 = int_sub(n0, Const(1))
7 i2 = int_ge(n1, Const(0))
8 guard_true(i2)
9 jump(result1, n1)
```

Virtuelle Maschine - Revisited: Trace von DECR_A

```
def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        opcode = ord(bytecode[pc])
        pc += 1
#       if opcode == JUMP_IF_A: [...]
        elif opcode == DECR_A:
            a -= 1
#       [...]
```

```
loop_start(a0, regs0, bytecode0, pc0)
opcode0 = strgetitem(bytecode0, pc0)
pc1 = int_add(pc0, Const(1))
guard_value(opcode0, Const(7))
a1 = int_sub(a0, Const(1))
jump(a1, regs0, bytecode0, pc1)
```

Hot Loops des Programms erkennen (Hints einbauen)

```
1  tlrjitdriver = JitDriver(greens = ['pc', 'bytecode'],
2                            reds   = ['a', 'regs'])
3  def interpret(bytecode, a):
4      regs = [0] * 256
5      pc = 0
6      while True:
7          tlrjitdriver.jit_merge_point(pc=pc,
8              bytecode=bytecode, a=a, regs=regs)
9          opcode = ord(bytecode[pc])
10         pc += 1
11         if opcode == JUMP_IF_A:
12             target = ord(bytecode[pc])
13             pc += 1
14             if a:
15                 if target < pc:
16                     tlrjitdriver.can_enter_jit(
17                         pc=target, bytecode=bytecode,
18                         a=a, regs=regs)
19                 pc = target
```

Bytecode zum Quadrat berechnen - Revisited

```
MOV_A_R    0 # i = a
MOV_A_R    1 # copy of 'a'
# 4:
MOV_R_A    0 # i--
DECR_A
MOV_A_R    0
MOV_R_A    2 # res += a
ADD_R_TO_A 1
MOV_A_R    2
MOV_R_A    0 # if i!=0: goto 4
JUMP_IF_A  4
MOV_R_A    2 # return res
RETURN_A
```

Trace der Quadratsumme

```
1  loop_start(a0, regs0, bytecode0, pc0)
2  # MOV_R_A 0
3  opcode0 = strgetitem(bytecode0, pc0)
4  pc1 = int_add(pc0, Const(1))
5  guard_value(opcode0, Const(2))
6  n1 = strgetitem(bytecode0, pc1)
7  pc2 = int_add(pc1, Const(1))
8  a1 = call(Const(<* fn list_getitem>), regs0, n1)
9  # DECR_A
10 opcode1 = strgetitem(bytecode0, pc2)
11 pc3 = int_add(pc2, Const(1))
12 guard_value(opcode1, Const(7))
13 a2 = int_sub(a1, Const(1))
```

...und noch mehr!

Trace der Quadratsumme (forts.)

```
14 | # MOV_A_R 0
15 | opcode1 = strgetitem(bytecode0, pc3)
16 | pc4 = int_add(pc3, Const(1))
17 | guard_value(opcode1, Const(1))
18 | n2 = strgetitem(bytecode0, pc4)
19 | pc5 = int_add(pc4, Const(1))
20 | call(Const(<* fn list_setitem>), regs0, n2, a2)
21 | # MOV_R_A 2
22 | opcode2 = strgetitem(bytecode0, pc5)
23 | pc6 = int_add(pc5, Const(1))
24 | guard_value(opcode2, Const(2))
25 | n3 = strgetitem(bytecode0, pc6)
26 | pc7 = int_add(pc6, Const(1))
27 | a3 = call(Const(<* fn list_getitem>), regs0, n3)
28 | # ADD_R_TO_A 1
29 | opcode3 = strgetitem(bytecode0, pc7)
30 | pc8 = int_add(pc7, Const(1))
31 | guard_value(opcode3, Const(5))
32 | n4 = strgetitem(bytecode0, pc8)
33 | pc9 = int_add(pc8, Const(1))
34 | i0 = call(Const(<* fn list_getitem>), regs0, n4)
35 | a4 = int_add(a3, i0)
36 | # MOV_A_R 2
37 | opcode4 = strgetitem(bytecode0, pc9)
38 | pc10 = int_add(pc9, Const(1))
39 | guard_value(opcode4, Const(1))
40 | n5 = strgetitem(bytecode0, pc10)
41 | pc11 = int_add(pc10, Const(1))
42 | call(Const(<* fn list_setitem>), regs0, n5, a4)
43 | # MOV_R_A 0
44 | opcode5 = strgetitem(bytecode0, pc11)
45 | pc12 = int_add(pc11, Const(1))
46 | guard_value(opcode5, Const(2))
```

...und noch mehr: 58 Zeilen gesamt!

Optimierung durch Constant Folding der „green-Variablen“

```
1  loop_start(a0, regs0)
2  # MOV_R_A 0
3  a1 = call(Const(<* fn list_getitem>), regs0, Const(0))
4  # DECR_A
5  a2 = int_sub(a1, Const(1))
6  # MOV_A_R 0
7  call(Const(<* fn list_setitem>), regs0, Const(0), a2)
8  # MOV_R_A 2
9  a3 = call(Const(<* fn list_getitem>), regs0, Const(2))
10 # ADD_R_TO_A 1
11 i0 = call(Const(<* fn list_getitem>), regs0, Const(1))
12 a4 = int_add(a3, i0)
13 # MOV_A_R 2
14 call(Const(<* fn list_setitem>), regs0, Const(2), a4)
15 # MOV_R_A 0
16 a5 = call(Const(<* fn list_getitem>), regs0, Const(0))
17 # JUMP_IF_A 4
18 i1 = int_is_true(a5)
19 guard_true(i1)
20 jump(a5, regs0)
```