Compiling RoopI++ to HSSA

Lukas Gail, Uwe Meyer, Tristan Schönhals

THM University of Applied Sciences, Gießen, Germany

Motivation

Why Roopl++ ?

- OO is prevalent in today‘s software development
- Challenging due to
  - Aliasing
  - Dynamic dispatch
  - Reversible (!) heap allocation and deallocation
- Starting point for the development of a larger reversible OO-language

Why HSSA ?

- Provides compact reversible intermediate representation
- Enables retargetable compilation to multiple backends
- Allows reusable IR-level optimizations: constant folding, dead-code elimination, …
- Hybrid features such as discard may support better runtime error handling

Background – Roopl++

- Roopl++ is an object-oriented reversible language introduced by M. Cservenka in (2018, Copenhagen)
- Extension of Roopl with **dynamic memory management and arrays**
- **Class-based**: inheritance, method overriding, subtype polymorphism, dynamic dispatch
- Supports dynamic object creation/deletion, arrays, and multiple references via copy / uncopy
- Control flow follows the Janus style: reversible conditionals and loops with entry/exit assertions
- A compiler/backend to Pisa exists in Cservenka's implementation

Example (Linked List):

```
class Cell
  Cell next
  int data

  method constructor(int value)
    data ^= value
  …
class LinkedList
  Cell head …
```

```
class Program
  LinkedList linkedList
  int sumResult
  int listLength

  method main()
    new LinkedList linkedList
      …
```

Cservenka, M.H.: Design and implementation of dynamic memory management in a reversible object-oriented programming language. arXiv preprint arXiv:1804.05097 (2018)

Roopl++ Restrictions

- Roopl++ allows **multiple references** to the same object to support complex data structures
- Copied references must be removed using uncopy before the object can be deleted
- Objects and arrays must be zero-cleared before deletion
- Argument **aliasing** is restricted: the same variable/object must not be passed through multiple parameters
- Some aliasing situations cannot be fully detected, e.g. through arrays or copied references

➢ Violations may lead to runtime errors or undefined behavior in Roopl++

```
new Node a
copy Node a b
delete Node a
```

b is a dangling pointer

```
call c::add(a, a)
```

Error: arguments passed to method must be unique.

```
copy Node a b
call c::f(a,b)
```

Aliasing conflict – cannot always be detected at compile-time!

Background - HSSA

- HSSA (Hybrid Single Static Assignment) is a reversible intermediate language developed by L. Gail in Giessen
- Conceptually **based on RSSA**, but extends the reversible core with **explicit irreversible operations**
- discard primitive can deliberately erase information, making non-reversible behaviour explicit
- **Optimizations**, i.e. constant propagation/folding and dead-code elimination
- Designed as **retargetable**

Example (integer increment):

```
rel inc:
  n, 0 := begin <-
     t := add 1 =: n
          -> end =: t, 0

rel main:
  (), 0 := begin <-
      n := inc =: 3
        -> end =: n, 0
```

Gail, L., Meyer, U.: Connecting reversible and classical computing through hybrid SSA. In: Reversible Computation: 16th International Conference, RC 2024, Toruń, Poland.

Compiling Roopl++ to HSSA – Classes (1)

```
class S
 int f1
 method a1()
  skip

 method a2(int x)
  skip

class T inherits S
 int f2
 method a2(int x)
  f1+=1
…
T obj
obj::a2(y)
```

Memory Layout: One memory slot per field + one for vtable:

Slot 0: vtable
Slot 1: field f1
Slot 2: field f2

Method **_new** is invoked when an object of this class is created:

```
rel _T._new:
   _m, 0       :=        begin <-
  _m.0, _anon  := mmem.allocate 3      =:         _m
  _vtable      :=    dup _T._vtable    =:         ()
  _m.1, 0      := mmem.readwrite _anon =: _m.0, _vtable
                  -> end          =: (_m.1, _anon), 0
```

Code for **_delete** is not explicitly generated.
Instead, the above is applied in reverse.

```
def generateDelete(typ: Types.ArrayType | Types.Class,
                   name: Translatable.VariableReference): Unit = {
  relation.blockBuilder.adds(invert(generateNewCode(typ, name)))
}
```

Compiling RoopI++ to HSSA – Classes (2)

```
class S
 int f1
 method a1()
  skip

 method a2(int x)
  skip

class T inherits S
 int f2
 method a2(int x)
  f1+=x
…
T obj
obj::a2(y)
```

+ One relation for every method

```
rel _T.a2 _this, x:
   // _m is the memory, _this is the reference to callee object
   _m, 0                  :=        begin <-

   _m.0, _this_ref        :=  mmem.read _this   =: _m
   f1                     :=   add _this_ref    =: 1    // addr. of f1
   f2                     :=   add _this_ref    =: 2    // addr. of f2

   _m.1, _t.0             :=    mmem.read x     =: _m.0
   _m.2, _assignee_val    := mmem.readwrite f1  =: _m.1, ()
   _assignee_val.0        :=      add _t.0      =: _assignee_val
   _m.3, ()               := mmem.readwrite f1  =: _m.2, _assignee_val.0
   _m.4                   :=    ~mmem.read x    =: _m.3, _t.0

   2                      :=   ~add _this_ref   =: f2    // finalizer
   1                      :=   ~add _this_ref   =: f1    // finalizer

   _m.5                   :=  ~mmem.read _this =: _m.4, _this_ref
                                      -> end     =: _m.5, 0
```

Compiling Roopl++ to HSSA - vTable

- The vTable (or dispatch table) is commonly used in compilers to resolve method overriding

```
class S
  int f1
  method a1()
    skip

  method a2(int x)
    skip

class T inherits S
  int f2
  method a2(int x)
    f1+=x
…
T obj
obj::a2(y)
```

S.vTable

```
0 -> _S._a2
1 -> _S._a1
```

T.vTable

```
0 -> _T._a2
1 -> _S._a1
```

Every vTable is a HSSA relation, that takes the method‘s index as input.
At runtime:
- vTable is read from slot 0 of the object
- vTable relation is invoked with method index 0 (returns `_T._a2`)
- `_T._a2` relation is invoked with argument y

Compiling RoopI++ to HSSA - vTable

T.vTable

```
0 -> _T._a2
1 -> _S._a1
```

implemented by

```
rel _T._vtable _method:
  (), 0  := begin <-
  i      :=  dup _method =: ()
                -> L0,L2 =: (), i
  (), 0  := L0 <-
  res    :=  dup _T.a2    =: ()
                -> L1 =: res, 0
  (), 0  := L2 <-
  res    :=  dup _S.a1    =: ()
                -> L3 =: res, 0
  res, i := L1,L3 <-
  ()     := ~dup _method =: i
                -> end =: res, 0
```

Every vTable is a relation taking a method-index parameter.
At runtime:

- vTable is read from slot 0 of the object
- vTable is applied to method index 0 (returns `_T._a2`)
- `_T._a2` relation is invoked with argument y

Compiling Roopl++ - Memory Primitives

`mmem.new`

- Once creates a new, dynamic memory object with a fixed size
- Example: `_m := mmem.new =: ()`

`mmem.allocate<size>`

- allocates a number of consecutive, unallocated memory slots on a memory object and returns the address of the first slot.

`mmem.readwrite`

- updates one slot in managed memory **(_m)** at a given **address**: it returns the **previous** cell value and writes a **new** value into that slot.
- It is the **reversible** alternative to a plain write: the old contents are **not destroyed** but passed out as data, so the step stays **invertible** (self-inverse up to carrying the right values).

Demo

https://github.com/Leridon/hssa/

Error Handling

Two classes of runtime errors

- Some errors arise naturally as reversibility violations in HSSA
- Example: wrong fi / loop exit condition.
- HSSA finalization/pattern matching fails automatically when the backward branch information does not match. No special compiler code needed.

- Others need explicit runtime checks generated by the compiler
- Examples: array bounds, dangling pointers, deletion type mismatch, aliasing/type-variance cases.
- Fully reversible error handling would require treating exceptions and call stacks as explicit values.

Improvements with HSSA (future work)

- Roopl++ treats many of these cases as undefined behaviour
- Use HSSA‘s discard to raise better runtime errors and discard the call stack
- Source maps + generated runtime checks for better error messages