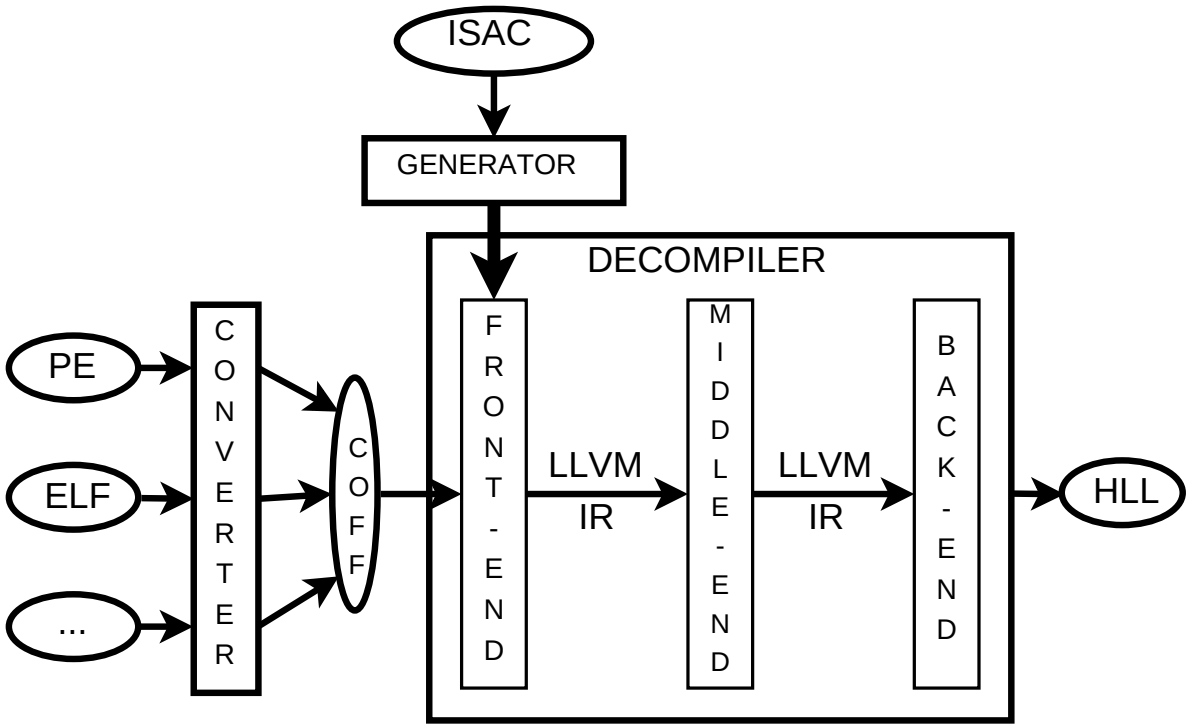
Reconstruction of Data Types for Decompilation

Peter Matula
xmatul01[at]fit.vutbr.cz
December 4, 2012

Brno University of Technology
Faculty of Information Technology

Retargetable decompiler

Platform-specific binary file ⇒ high-level language.



[L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, A. Meduna, 2011]

State of the art

[A. Mycroft, 1999]

Type-inference technique: builds a set of formulas based upon the type constraints, solving of these formulas to infer specific object types. Algorithm is often divergent.

```
%u2 = add i32 %u0, %u1
```

$$T_{u2} = ptr(\alpha), T_{u0} = ptr(\alpha), T_{u1} = int \vee$$
$$T_{u2} = ptr(\alpha), T_{u0} = int, T_{u1} = ptr(\alpha) \vee$$
$$T_{u2} = int, T_{u0} = int, T_{u1} = int$$

[J. Lee, T. Avgerinos, D. Brumley, 2011]

Improvement of the Mycrofts constraints solving, always convergent.

[E. N. Dolgova, A. V. Chernov, 2009]

Solving data flow equations in an iterative way, always convergent.

Objects

Registers, global/local local variables, function parameters and return values.

Lattice over the data types

$$\begin{aligned} core &\in \{integer, pointer, float\} \\ size &\in \{1, 8, 16, 32, 64\} \\ sign &\in \{signed, unsigned\} \\ T &= \langle \tau^{core}, \tau^{size}, \tau^{sign} \rangle \end{aligned}$$

Join function $\sqcup$

$$T_1 \sqcup T_2 = \langle \tau_1^{core} \sqcup^{core} \tau_2^{core}, \tau_1^{size} \sqcup^{size} \tau_2^{size}, \tau_1^{sign} \sqcup^{sign} \tau_2^{sign} \rangle$$
$$\sqcup^{core} = \sqcup^{size} = \sqcup^{sign} = \cap$$

Object type

Each occurrence of the object gets type: T_i
All types of one object: $T = \{T_1, \ldots, T_n\}$
Recovered type of the object is: $T = T_1 \sqcup \cdots \sqcup T_n$

Constraints

Sources of the type information

- Register constraints (unreliable) – affects *core*, *size*.
- Instruction constraints – affects *core*, *size*, *sign*.
- Flag constraints – affects *sign*.
- Environment constraints – affects *core*, *size*, *sign*.

First pass

Iterate through all instructions, create equations only for instructions that propagate the type information, initialize object types.

```
; add v0, v0, 1234
    %u0 = add i16 1234, 0
    %u1 = sext i16 %u0 to i32
    %u2 = load i32* @gpregs0
    %u3 = add i32 %u1,  %u2
    store i32 %u3, i32* @gpregs0
```

First pass

Iterate through all instructions, create equations only for instructions that propagate the type information, initialize object types.

```
; add v0, v0, 1234
    %u0 = add i16 1234, 0
    %u1 = sext i16 %u0 to i32
    %u2 = load i32* @gpregs0
    %u3 = add i32 %u1,  %u2
    store i32 %u3, i32* @gpregs0
```

Direct type derivation, no equation:

$$T_{u0} \leftarrow \langle \{int\}, \{16\}, \{signed, unsigned\} \rangle$$

First pass

Iterate through all instructions, create equations only for instructions that propagate the type information, initialize object types.

```
; add v0, v0, 1234
    %u0 = add i16 1234, 0
    %u1 = sext i16 %u0 to i32
    %u2 = load i32* @gpregs0
    %u3 = add i32 %u1,  %u2
    store i32 %u3, i32* @gpregs0
```

Direct type derivation, no equation, join type:

$$T_{u1} \leftarrow \langle \{int\}, \{32\}, \{signed\} \rangle$$
$$T_{u0} \leftarrow T_{u0} \sqcup \langle \{int\}, \{16\}, \{signed\} \rangle$$

First pass

Iterate through all instructions, create equations only for instructions that propagate the type information, initialize object types.

```
; add v0, v0, 1234
    %u0 = add i16 1234, 0
    %u1 = sext i16 %u0 to i32
    %u2 = load i32* @gpregs0
    %u3 = add i32 %u1,  %u2
    store i32 %u3, i32* @gpregs0
```

Type initialization, equation:

$$T_{u2} \leftarrow T_{gpregs0} \leftarrow \langle \{integer, pointer, float\}, \{32\}, \{signed, unsigned\} \rangle$$
$$T_{u2} \Leftrightarrow T_{gpregs0}$$

First pass

Iterate through all instructions, create equations only for instructions that propagate the type information, initialize object types.

```
; add v0, v0, 1234
    %u0 = add i16 1234, 0
    %u1 = sext i16 %u0 to i32
    %u2 = load i32* @gpregs0
    %u3 = add i32 %u1,  %u2
    store i32 %u3, i32* @gpregs0
```

Type initialization, equation:

$$T_{u1} \leftarrow T_{u2} \leftarrow T_{u3} \leftarrow \langle \{int, pointer\}, \{32\}, \{signed, unsigned\} \rangle$$
$$T_{u3} \Leftrightarrow T_{u1}\ ADD\ T_{u_2}$$

First pass

Iterate through all instructions, create equations only for instructions that propagate the type information, initialize object types.

```
; add v0, v0, 1234
    %u0 = add i16 1234, 0
    %u1 = sext i16 %u0 to i32
    %u2 = load i32* @gpregs0
    %u3 = add i32 %u1,  %u2
    store i32 %u3, i32* @gpregs0
```

Type initialization, equation:

$$T_{gpregs0} \leftarrow T_{u3} \leftarrow \langle \{integer, pointer, float\}, \{32\}, \{signed, unsigned\} \rangle$$
$$T_{gpregs0} \Leftrightarrow T_{u3}$$

Operands to destination propagation

Propagation of the type information from operands to destination.
Depends on the operation.

Example: $T_{u3} \Leftarrow T_{u1}\ ADD\ T_{u_2}$

- core: $\frac{int \in \tau_{u1}^{core} \wedge int \in \tau_{u2}^{core}}{int \in \tau_{u3}^{core}} \wedge \frac{ptr \in \tau_{u1}^{core} \wedge int \in \tau_{u2}^{core}}{ptr \in \tau_{u3}^{core}} \wedge \frac{int \in \tau_{u1}^{core} \wedge ptr \in \tau_{u2}^{core}}{ptr \in \tau_{u3}^{core}}$
- size: $\tau_{u3}^{size} \leftarrow \tau_{u1}^{size} \sqcap \tau_{u2}^{size}$ where $\sqcap$ is upper intersection.
- sign: $\frac{unsigned \in \tau_{u1}^{sign} \wedge unsigned \in \tau_{u2}^{sign}}{unsigned \in \tau_{u3}^{sign}} \wedge \frac{unsigned \in \tau_{u1}^{sign} \wedge signed \in \tau_{u2}^{sign}}{unsigned \in \tau_{u3}^{sign}} \wedge$

$$\frac{signed \in \tau_{u1}^{sign} \wedge unsigned \in \tau_{u2}^{sign}}{unsigned \in \tau_{u3}^{sign}} \wedge \frac{signed \in \tau_{u1}^{sign} \wedge signed \in \tau_{u2}^{sign}}{signed \in \tau_{u3}^{sign}} \wedge$$

Note: additive variant presented, subtractive implemented.

Destination to operands propagation

Propagation of the type information from destination to operands.
Complication: τ_{u3}^{core} may be a pointer, which operand is a pointer?

Proposed solution

Introduction of the alternative values for attributes of object types.
$\tau_p^{class}?[q]$ denotes that $class \in \{core, size, sign\}$ of object p contains alternative element, corresponding with the alternative element of object q.

Example: $T_{u3} \Rightarrow T_{u1}\ ADD\ T_{u2}$

- core: $\dfrac{ptr \in \tau_{u3}^{core}}{\tau_{u1}^{core} \leftarrow \tau_{u1}^{core} \cup pointer_{u1}^{u3}?[u2] \wedge \tau_{u2}^{core} \leftarrow \tau_{u2}^{core} \cup pointer_{u2}^{u3}?[u1]}$

At least one of the elements belongs to the resulting set after the join.
If both do not belong, they must be explicitly added to the both sets.

Unnecessarily complicated

Lazy rule application

Algorithm is iterating over the propagation equations over and over, create lazy rules that are triggered only when they can be applied.

Example: $T_{u3} \Rightarrow T_{u1}\ ADD\ T_{u_2}$

- core:

$$\frac{\{ptr\} = \tau_{u3}^{core} \ \wedge \ ptr \notin \tau_{u1}^{core}}{\tau_{u2}^{core} \leftarrow \{ptr\}} \vee \frac{\{ptr\} = \tau_{u3}^{core} \ \wedge \ ptr \notin \tau_{u2}^{core}}{\tau_{u1}^{core} \leftarrow \{ptr\}} \vee \frac{\{int\} = \tau_{u3}^{core}}{\tau_{u1}^{core} \leftarrow \{int\} \ \wedge \ \tau_{u2}^{core} \leftarrow \{int\}}$$

  otherwise do nothing at the moment.

- size: $\tau_{u1}^{size} \leftarrow \tau_{u1}^{size} \sqcap \tau_{u3}^{size}$ and $\tau_{u2}^{size} \leftarrow \tau_{u2}^{size} \sqcap \tau_{u3}^{size}$
- sign: same principle as for core.

Problem #1

LLVM IR temporary variables produced for every instruction have the same names, but they are not the same objects.
⇒ ?

```
; 8000 - mv v0, v5
    %u0 = load i32* @gpregs5
    store i32 %u0, i32* @gpregs0
...
; 8500 - lw v0, 4(sp)
    %u0 = load i32* %stack_var_4
    store i32 %u0, i32* @gpregs0

; 8510 - lw v1, 8(sp)
    %u0 = load i32* %stack_var_8
    store i32 %u0, i32* @gpregs1

; 8520 - add v0, v0, v1
    %u0 = load i32* @gpregs0
    %u1 = load i32* @gpregs1
    %u2 = add i32 %u0, %u1
    store i32 %u2, i32* @gpregs0
```

Problem #1

LLVM IR temporary variables produced for every instruction have the same names, but they are not the same objects.
⇒ Already solved in the IR.

```
; 8000 - mv v0, v5
    %u0_8000 = load i32* @gpregs5
    store i32 %u0_8000, i32* @gpregs0
...
; 8500 - lw v0, 4(sp)
    %u0_8500 = load i32* %stack_var_4
    store i32 %u0_8500, i32* @gpregs0

; 8510 - lw v1, 8(sp)
    %u0_8510 = load i32* %stack_var_8
    store i32 %u0_8510, i32* @gpregs1

; 8520 - add v0, v0, v1
    %u0_8520 = load i32* @gpregs0
    %u1_8520 = load i32* @gpregs1
    %u2_8520 = add i32 %u0_8520, %u1_8520
    store i32 %u2_8520, i32* @gpregs0
```

Problem #2

Some objects (registers, stack variables) may be used for several different variables (types) on different places in the program.
⇒ ?

```
; 8000 - mv v0, v5
    %u0_8000 = load i32* @gpregs5
    store i32 %u0_8000, i32* @gpregs0
...
; 8500 - lw v0, 4(sp)
    %u0_8500 = load i32* %stack_var_4
    store i32 %u0_8500, i32* @gpregs0

; 8510 - lw v1, 8(sp)
    %u0_8510 = load i32* %stack_var_8
    store i32 %u0_8510, i32* @gpregs1

; 8520 - add v0, v0, v1
    %u0_8520 = load i32* @gpregs0
    %u1_8520 = load i32* @gpregs1
    %u2_8520 = add i32 %u0_8520, %u1_8520
    store i32 %u2_8520, i32* @gpregs0
```

Problem #2

Some objects (registers, stack variables) may be used for several different variables (types) on different places in the program.
⇒ Distinguish every usage.

```
; 8000 - mv v0, v5
    %u0_8000 = load i32* @gpregs5_8000
    store i32 %u0_8000, i32* @gpregs0_8000
...
; 8500 - lw v0, 4(sp)
    %u0_8500 = load i32* %stack_var_4_8500
    store i32 %u0_8500, i32* @gpregs0_8500

; 8510 - lw v1, 8(sp)
    %u0_8510 = load i32* %stack_var_8_8510
    store i32 %u0_8510, i32* @gpregs1_8510

; 8520 - add v0, v0, v1
    %u0_8520 = load i32* @gpregs0_8520
    %u1_8520 = load i32* @gpregs1_8520
    %u2_8520 = add i32 %u0_8520, %u1_8520
    store i32 %u2_8520, i32* @gpregs0_8520
```

Problem #3

Solution of the problem #2 breaks type propagation between instructions – there are no ties between objects used in several instructions.
$\Rightarrow$?

```
; 8000 - mv v0, v5
    %u0_8000 = load i32* @gpregs5_8000
    store i32 %u0_8000, i32* @gpregs0_8000
...
; 8500 - lw v0, 4(sp)
    %u0_8500 = load i32* %stack_var_4_8500
    store i32 %u0_8500, i32* @gpregs0_8500

; 8510 - lw v1, 8(sp)
    %u0_8510 = load i32* %stack_var_8_8510
    store i32 %u0_8510, i32* @gpregs1_8510

; 8520 - add v0, v0, v1
    %u0_8520 = load i32* @gpregs0_8520
    %u1_8520 = load i32* @gpregs1_8520
    %u2_8520 = add i32 %u0_8520, %u1_8520
    store i32 %u2_8520, i32* @gpregs0_8520
```

Problem #3

Solution of the problem #2 breaks type propagation between instructions – there are no ties between objects used in several instructions.
⇒ Find and merge objects that have the same type.

```
; 8000 - mv v0, v5
    %u0_8000 = load i32* @gpregs5_8000
    store i32 %u0_8000, i32* @gpregs0_8000
...
; 8500 - lw v0, 4(sp)
    %u0_8500 = load i32* %stack_var_4_8500
    store i32 %u0_8500, i32* @gpregs0_8500

; 8510 - lw v1, 8(sp)
    %u0_8510 = load i32* %stack_var_8_8510
    store i32 %u0_8510, i32* @gpregs1_8510

; 8520 - add v0, v0, v1
    %u0_8520 = load i32* @gpregs0_8520
    %u1_8520 = load i32* @gpregs1_8520
    %u2_8520 = add i32 %u0_8520, %u1_8520
    store i32 %u2_8520, i32* @gpregs0_8520
```

$T_{@gpregs0\_8520} \Leftrightarrow T_{@gpregs0\_8500}$ $T_{@gpregs1\_8520} \Leftrightarrow T_{@gpregs1\_8510}$

Problem #3

Solution of the problem #2 breaks type propagation between instructions – there are no ties between objects used in several instructions.
⇒ Find and merge objects that have the same type.

How to find objects to merge?

1. Iterate over all instructions.
2. If instruction writes to the object (registers, stack variables) do nothing, object type may have been rewritten. First pass created equation saying that the source and destination types are the same.
3. If instruction reads from the objects, find last usage of the object and merge them – create an equation saying that their types are the same.

LLVM IR is a load/store "architecture".

Algorithm

```
1  firstPass(instructions, &equations, &objects);
2  mergeObjects(&equations, &objects);
3  objects.joinAll();
4
5  flag = true;
6  while (flag)
7  {
8      flag = false;
9
10     equations.opsToDstPropagation();
11     flag |= objects.joinAll();
12
13     equations.dstToOpsPropagation();
14     flag |= objects.joinAll();
15 }
```

Conclusion

- Presented algorithm should be able to perfectly reconstruct the base data types if the input program strictly conforms to the standard.
- Because algorithm operates on the finite lattices it is convergent.
- Algorithm is being implemented as a part of the retargetable decompiler developed by the Lissom team at FIT BUT.

Future work

[K. Troshina, Y. Derevenets, A. Chernov, 2010]

Reconstruction of the composite data types – array, structures, unions.

References

- A. Mycroft: *Type-Based Decompilation*, in 8th European Symp. on Programming Languages and Systems, Lect Notes Comput.Sci., 1999, vol. 1576, pp. 208-223.
- E. N. Dolgova, A. V. Chernov: *Automatic Reconstruction of Data Types in the Decompilation Problem*, Programming and Computing Software 35, 2009.
- K. Troshina, Y. Derevenets, A. Chernov: *Reconstruction of Composite Types for Decompilation*, SCAM, 2010.
- J. Lee, T. Avgerinos, D. Brumley: *TIE: Principled Reverse Engineering of Types in Binary Programs*. NDSS 2011
- L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, A. Meduna: *Design of a retargetable decompiler for a static platform-independent malware analysis*, In The 5th International Conference on Information Security and Assurance, Volume 200, pages 72-86, Springer Verlag, 2011