Interactive Decompilation

José Manuel Rios Fonseca

Faculty of Engineering of the University of Porto

13 December 2006

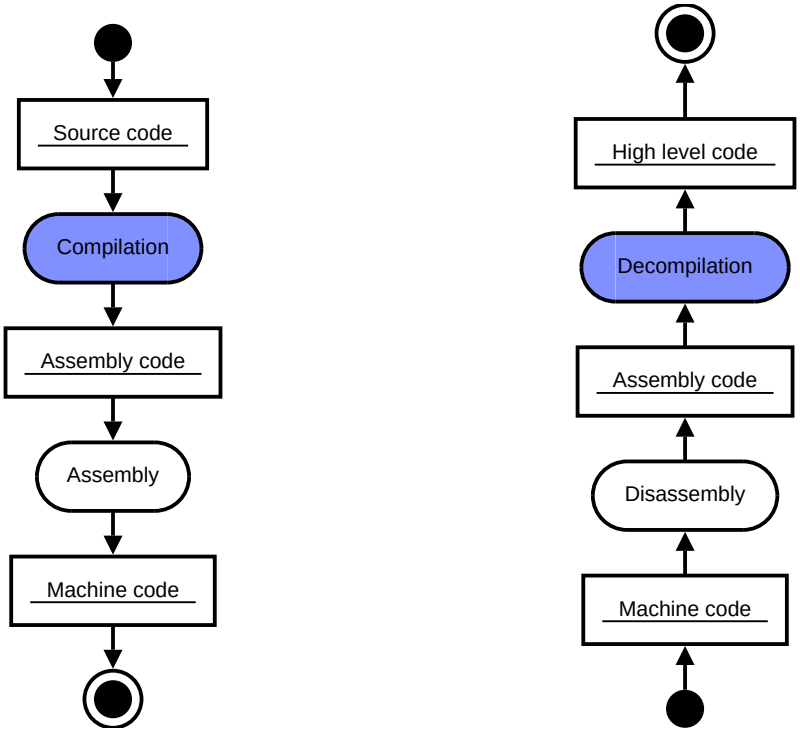
Dissertation prepared under the supervision of
Dr. Ademar Manuel Teixeira de Aguiar and of
Dr. João Alexandre Baptista Vieira Saraiva

Presentation Outline

Motivation for Reverse Engineering

- Software development is a fast paced technology field.
- Reverse engineering techniques can be used to:
  - port software into new programming languages or hardware architectures;
  - maintain software from a disappeared vendor;
  - attest the violation of patents or business secrets;
  - detect malicious code.

Compilation *vs.* Decompilation



Decompilation Feasibility

Fully automated decompilation

Is not always possible because:

- there is an ambiguous correspondence between high-level language statements and the respective machine code instructions;
- much of the original information is discarded during the compilation process;
- the distinction between data and code in an executable is often blurred.

Human intervention

Human action can be employed to:

- disambiguate code semantics,
- organize code,
- and improve readability.

Proposed Strategy

1. Define a set of transformations of low-level (near Assembly) code that aims at improving its structure, readability, semantics without changing its behavior (i.e., refactorings).
2. Developed an interactive decompilation tool that assists the user in the task of reverse engineering Assembly code, by automating the application of the above mentioned transformations.

Refactoring and Decompilation

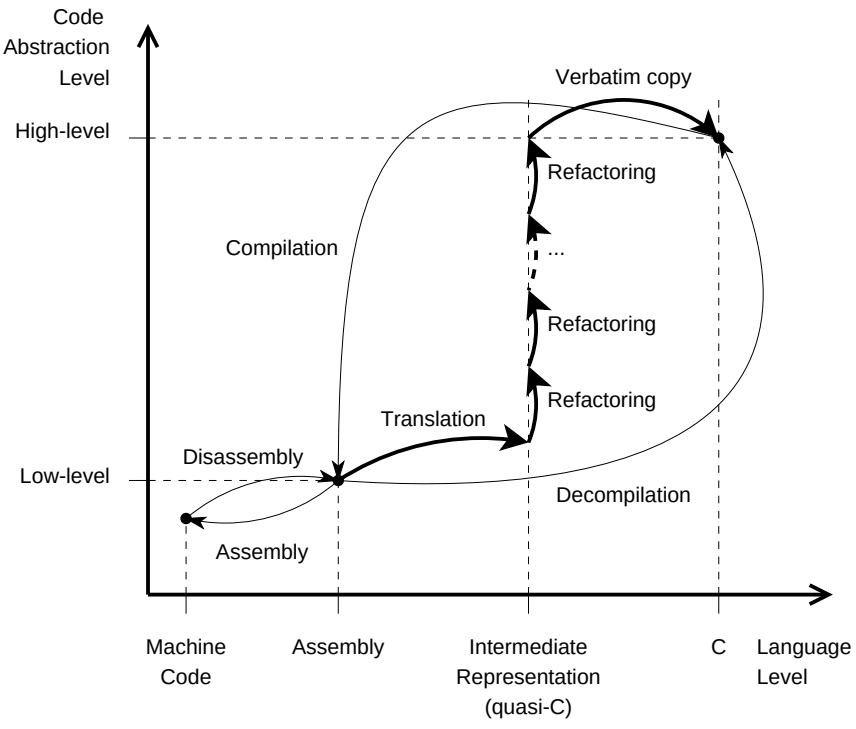
Refactoring Definition

A refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Refactoring *vs.* decompiling

- The decompilation of a program has both the same understanding and maintenance simplification aims and the same behavior-preserving property as does a refactoring.
- Thus the decompilation of a program could be carried out as the composition of basic refactorings.

Decompilation as a Sequence of Refactorings



Refactoring Categorization

Category	Rationale	Intent
Function prototyping	Information about the function bodies, arguments, and local variables is not properly retained by the Assembly code.	Lift the bodies, prototypes, and frames of functions.
Organizing data	During compilation all the data flow is mapped to accesses from/to the processor registers, stack, and global memory.	Transpose that data flow in terms of local and global variables.
Structuring control flow	High-level language control structures are translated into jumps and conditional jumps on Assembly language.	Recover the high-level control structure that match the jumps control-flow graph.

Category	Name
Function prototyping	Extract Function Set Function Return Add Function Argument
Organizing data	Extract Local Variable Inline Temp Split Temporary Variable Replace Magic Number with Symbolic Constant Replace Data Values with Record Replace Type Dead Code Elimination Rename Symbol Simplify Expression
Structuring control flow	Structure *If* Statement Structure *If-Else* Statement Structure *Do-While* Statement Structure Infinite Loop Structure *Continue* Statement Structure *Break* Statement Structure *While* Statement Inline *Return* Statement Consolidate Boolean *And* Expression

Original C code *vs.* generated Assembly

```
int factorial(int n)
{
    int f;
    f = 1;
    while(n)
        f *= n--;
    return f;
}
```

$\Longrightarrow$

```
        .text
.globl factorial
factorial:
        testl   %eax, %eax
        jne     .L2
        movl    $1, %edx
        jmp     .L4
.L2:
        movl    $1, %edx
.L5:
        imull   %eax, %edx
        decl    %eax
        jne     .L5
.L4:
        movl    %edx, %eax
        ret
```


Assembly code *vs.* transliterated IR

```
    .text
.globl factorial
factorial:                  ⇒   factorial:
    testl   %eax, %eax      ⇒       tmp1 = eax & eax;
                                    cf = 0;
                                    of = 0;
                                    zf = tmp1 == 0;
                                    nf = tmp1 >> 31 & 1;
    jne .L2                 ⇒       if(!zf)
                                        goto .L2;
    movl    $1, %edx        ⇒       edx = 1;
    jmp .L4                 ⇒       goto .L4;
.L2:                        ⇒   .L2:
    movl    $1, %edx        ⇒       edx = 1;
.L5:                        ⇒   .L5:
    imull   %eax, %edx      ⇒       tmp2 = (long) edx * (long) eax;
                                    edx = edx * eax;
                                    cf = (tmp2 >> 32 & 0xffffffffL) == 0
                                 || (tmp2 >> 32 & 0xffffffffL) == 0xffff
                                ffffL;
                                    of = (tmp2 >> 32 & 0xffffffffL) == 0
                                 || (tmp2 >> 32 & 0xffffffffL) == 0xffff
                                ffffL;
    decl    %eax            ⇒       tmp3 = eax;
```

Extract Function

You have a set of code fragments that constitutes an individual function.
Turn the fragments into a function.

```
factorial:
    tmp1 = eax & eax;
    cf = 0;
    of = 0;
    zf = tmp1 == 0;
    nf = tmp1 >> 31 & 1;

    if(!zf)
        goto .L5;
.L4:
    eax = edx;
    return;
```

⇒

```
void factorial()
{
    tmp1 = eax & eax;
    cf = 0;
    of = 0;
    zf = tmp1 == 0;
    nf = tmp1 >> 31 & 1;
    ⋮
    if(!zf)
        goto .L5;
.L4:
    eax = edx;
    return;
}
```


Set Function Return

A register or the stack is used to pass the function return value.
Define the function return type with the appropriate type, making explicit that such stack position or register is the return value.

```
void factorial()
{
    tmp1 = eax & eax;
    cf = 0;
    of = 0;
    zf = tmp1 == 0;

    zf = eax == 0;
    if(!zf)
        goto .L5;
.L4:
    eax = edx;
    return;
}
```

⇒

```
int factorial()
{
    tmp1 = eax & eax;
    cf = 0;
    of = 0;
    zf = tmp1 == 0;
    ⋮
    zf = eax == 0;
    if(!zf)
        goto .L5;
.L4:
    eax = edx;
    return eax;
}
```

Add Function Argument

The stack or a register is used to pass an argument to a function.
Define a new function argument with the appropriate type, making explicit that such stack position or register is used to hold the argument.

```
int factorial()
{
    tmp1 = eax & eax;
    cf = 0;
    of = 0;
    zf = tmp1 == 0;
    nf = tmp1 >> 31 & 1;
    if(!zf)
        goto .L2;
    edx = 1;
    goto .L4;
.L2:
    edx = 1;
```

⇒

```
int factorial(int eax)
{
    tmp1 = eax & eax;
    cf = 0;
    of = 0;
    zf = tmp1 == 0;
    nf = tmp1 >> 31 & 1;
    if(!zf)
        goto .L2;
    edx = 1;
    goto .L4;
.L2:
    edx = 1;
```

⋮

Dead Code Elimination

You have several variable assignments, whose value is not used.
Remove those variable assignments.

```
int factorial(int eax)
{
    tmp1 = eax & eax;
    cf = 0;
    of = 0;
    zf = tmp1 == 0;
    nf = tmp1 >> 31 & 1;
    if(!zf)
        goto .L2;
    edx = 1;
    goto .L4;
.L2:
    edx = 1;
.L5:
    tmp2 = (long) edx * (long) eax;
    edx = edx * eax;
    cf = (tmp2 >> 32 & 0xffffffffL) == 0
 || (tmp2 >> 32 & 0xffffffffL) == 0xffff
ffffL;
    of = (tmp2 >> 32 & 0xffffffffL) == 0
 || (tmp2 >> 32 & 0xffffffffL) == 0xffff
```

⇒ ⇒ ⇒ ⇒ ⇒ ⇒ ⇒ ⇒

```
int factorial(int eax)
{
    tmp1 = eax & eax;

    zf = tmp1 == 0;

    if(!zf)
        goto .L2;
    edx = 1;
    goto .L4;
.L2:
    edx = 1;
.L5:

    edx = edx * eax;
```

Structure If-Else Statement

You have a conditional jump to two sets of consecutive statements.
Make each set of statements a clause of the conditional statement.

```
int factorial(int eax)
{
    tmp1 = eax & eax;
    zf = tmp1 == 0;
    if(!zf)
        goto .L2;
    edx = 1;
    goto .L4;
.L2:
    edx = 1;
.L5:
    edx = edx * eax;
    eax = eax - 1;
    zf = eax == 0;
    if(!zf)
        goto .L5;
.L4:


    eax = edx;
    return eax;
```

⇒

```
int factorial(int eax)
{
    tmp1 = eax & eax;
    zf = tmp1 == 0;
    if(!zf)
    {


        edx = 1;
    .L5:
        edx = edx * eax;
        eax = eax - 1;
        zf = eax == 0;
        if(!zf)
            goto .L5;
    }
    else
        edx = 1;
    eax = edx;
    return eax;
```


Structure Do-While Statement

You have a conditional jump to a previous label.
Make the intermediate statements the body of a do-while loop.

```
int factorial(int eax)
{
    tmp1 = eax & eax;
    zf = tmp1 == 0;
    if(!zf)
    {
        edx = 1;
    .L5:


        edx = edx * eax;
        eax = eax - 1;
        zf = eax == 0;


        if(!zf)
            goto .L5;
    }
    else
        edx = 1;
    eax = edx;
    return eax;
}
```

⇒

```
int factorial(int eax)
{
    tmp1 = eax & eax;
    zf = tmp1 == 0;
    if(!zf)
    {
        edx = 1;
        do
        {
            edx = edx * eax;
            eax = eax - 1;
            zf = eax == 0;
        }
        while(!zf);

    }
    else
        edx = 1;
    eax = edx;
    return eax;
}
```

Inline Temp

You have a temporary variable that is assigned and used just once or a few times.
Replace all references to that temporary value with the actual expression.

```
int factorial(int eax)
{
    tmp1 = eax & eax;                    ⇒
    zf = tmp1 == 0;                      ⇒
    if(!zf)                              ⇒
    {
        edx = 1;
        do
        {
            edx = edx * eax;
            eax = eax - 1;
            zf = eax == 0;               ⇒
        }
        while(!zf);                      ⇒
    }
    else
        edx = 1;
    eax = edx;                           ⇒
    return eax;                          ⇒
}
```

```
int factorial(int eax)
{

    if(!((eax & eax) == 0))
    {
        edx = 1;
        do
        {
            edx = edx * eax;
            eax = eax - 1;

        }
        while(!(eax == 0));
    }
    else
        edx = 1;

    return edx;
}
```


Simplify Expression

You have a mathematical expression with unnecessary complexities.
Simplify that expression.

```
int factorial(int eax)
{
    if(!((eax & eax) == 0))              ⇒
    {
        edx = 1;
        do
        {
            edx = edx * eax;
            eax = eax - 1;
        }
        while(!(eax == 0));              ⇒
    }
    else
        edx = 1;
    return edx;
}
```

```
int factorial(int eax)
{
    if(eax != 0)
    {
        edx = 1;
        do
        {
            edx = edx * eax;
            eax = eax - 1;
        }
        while(eax != 0);
    }
    else
        edx = 1;
    return edx;
}
```

Rename Symbol

You have a symbol with a meaningless machine generated name.
Rename that symbol into some meaningful.

```
int factorial(int eax)                    ⇒  int factorial(int n)
{                                         ⇒  {
    if(eax != 0)                          ⇒      if(n != 0)
    {                                     ⇒      {
        edx = 1;                          ⇒          f = 1;
        do                                ⇒          do
        {                                 ⇒          {
            edx = edx * eax;              ⇒              f = f * n;
            eax = eax - 1;                ⇒              n = n - 1;
        }                                 ⇒          }
        while(eax != 0);                  ⇒          while(n != 0);
    }                                     ⇒      }
    else                                  ⇒      else
        edx = 1;                          ⇒          f = 1;
    return edx;                           ⇒      return f;
}                                         ⇒  }
```


Original code *vs.* Final

```
int factorial(int n)
{
    int f;
    f = 1;
    while(n)
        f *= n--;
    return f;
}
```

⟺

```
int factorial(int n)
{
    if(n != 0)
    {
        f = 1;
        do
        {
            f = f * n;
            n = n - 1;
        }
        while(n != 0);
    }
    else
        f = 1;
    return f;
}
```

The IDC Tool

The IDC tool is an interactive decompiler, where the user starts with an almost literal translation of Assembly code in C language, which he progressively decompiles by the successive application of low-level refactorings, ultimately leading to high-level C code.

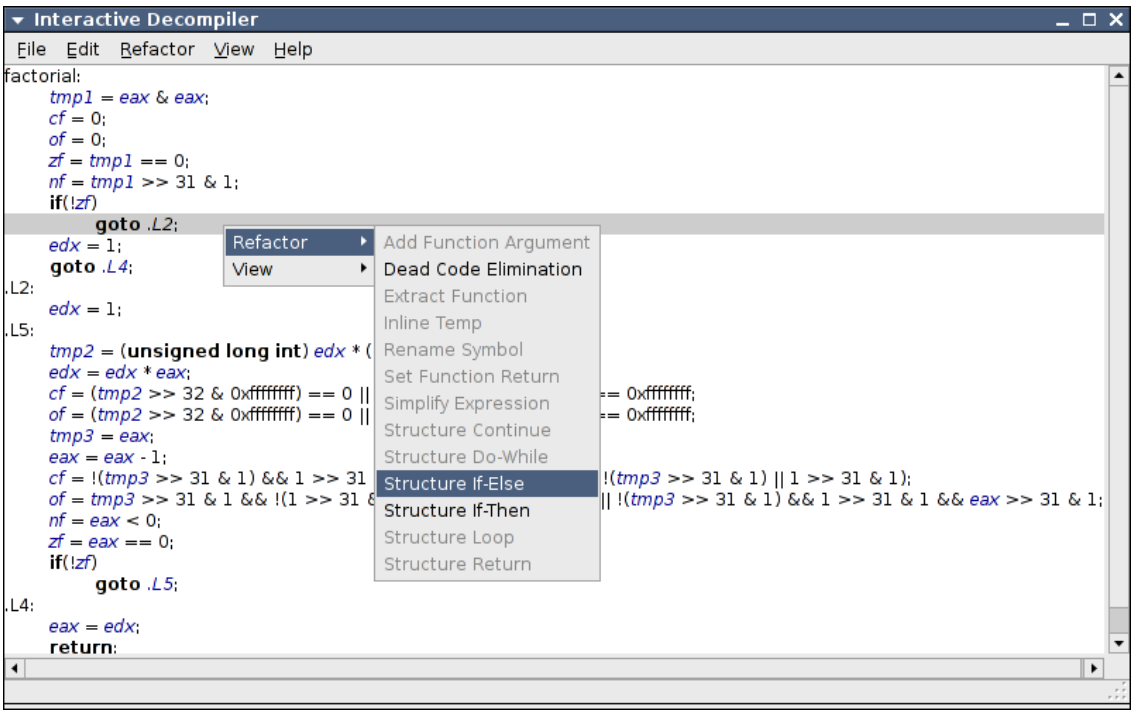
Features

- Import Intel IA32 Assembly code, in the AT&T syntax.
- Visualize and export *quasi*-C language code.
- Provides a context-sensitive refactoring browser to the low-level refactorings listed in the catalog.
- Visualize and manipulate the CFG and the AST of the program.

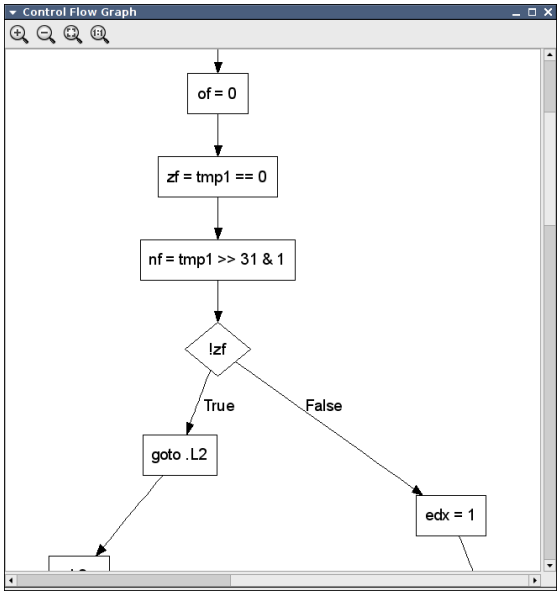
Availability

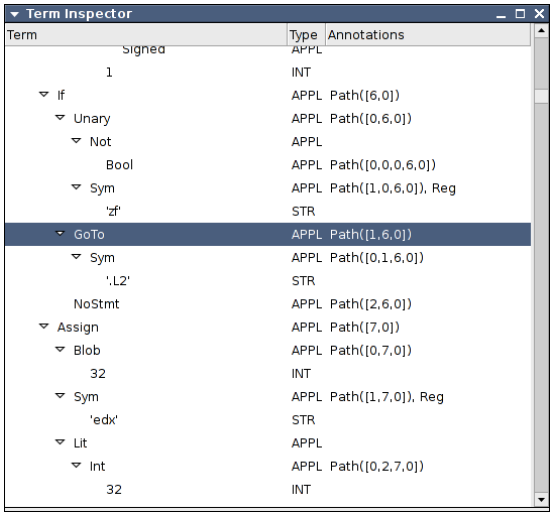
Source code, installation instructions, and examples are available from `http://paginas.fe.up.pt/~mei04010/idc/`.

Main View



Control Flow Graph and Term Inspector Views

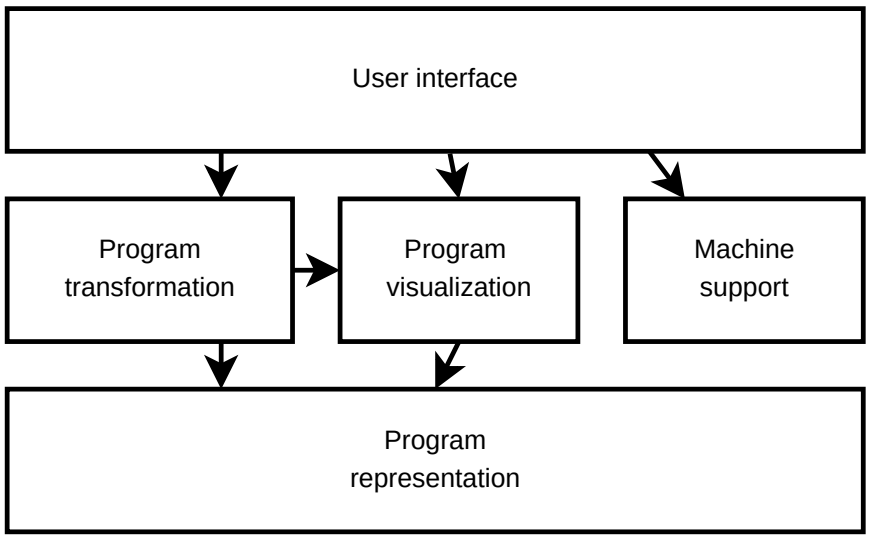




Application Example

Play Movie

Architecture of the IDC Tool



Intermediate Representation Data-type

The IR is the *Abstract Syntax Tree* (AST) encapsulated in an ATerm (*Annotated Term*).

ATerm representation

An ATerm can be:

- an *integer* literal, such as 1 and −28;
- a *real* literal, such as 1.414 and 1E+10;
- a *string* literal, such as "x" and "Hello World!";
- a *list* of zero or more ATerms, such as [1, 0.2, "a"] and [ ];
- or a function *application*, such as Plus(Var("x"), Int(1)), and True;
- and optionally followed by one or more *annotations* ATerms, such as, Mult(1,4){Type(Int)}, or Sym("x"){Line(14),Col(5)}.

Intermediate Representation Schema

$$
\begin{array}{rcl}
\textit{stmt} & = & \textbf{Asm}(\textit{string}\ \text{opcode}, \textit{expr}^*\ \text{operands}) \\
& | & \textbf{Assign}(\textit{type}, \textit{optExpr}\ \text{dest}, \textit{expr}\ \text{src}) \\
& | & \textbf{Label}(\textit{string}\ \text{name}) \\
& | & \textbf{GoTo}(\textit{expr}\ \text{addr}) \\
& | & \textbf{Break} \\
& | & \textbf{Continue} \\
& | & \textbf{Block}(\textit{stmt}^*) \\
& | & \textbf{If}(\textit{expr}\ \text{cond}, \textit{stmt}, \textit{stmt}) \\
& | & \textbf{While}(\textit{expr}\ \text{cond}, \textit{stmt}) \\
& | & \textbf{DoWhile}(\textit{expr}\ \text{cond}, \textit{stmt}) \\
& | & \textbf{Ret}(\textit{type}, \textit{optExpr}\ \text{value}) \\
& | & \textbf{Var}(\textit{type}, \textit{string}\ \text{name}, \textit{optExpr}\ \text{value}) \\
& | & \textbf{Function}(\textit{type}, \textit{string}\ \text{name}, \textit{arg}^*, \textit{stmt}^*\ \text{body}) \\
& | & \textbf{NoStmt}
\end{array}
$$

$\vdots$

Program Transformation

- Program decompilation and program refactoring are particular cases of program transformation.
- An object-oriented framework, inspired on the Stratego language, that allows to create complex term transformations from simple blocks was developed.
- A parser for a program transformation language similar to Stratego was implemented, to create transformations with less typing.

Assembly Loading and Translation Process

Input Assembly code

```
    .text
.globl main
main:
    movl    $1, %eax
    ret
```

⇓

Low-level IR

```
Module([
  Label("main"),
  Asm("movl", [Sym("eax"){Reg}, Lit(Int(32, Signed), 1)]),
  Asm("ret",[])
])
```

⇒ Pretty-print

```
main:
  asm("movl", eax, 1);
  asm("ret");
```

⇓

Translated IR

```
Module([
  Label("main"),
  Assign(Blob(32), Sym("eax"){Reg}, Lit(Int(32, Signed), 1)),
  Ret(Void,NoExpr)
])
```

⇒ Pretty-print

```
main:
  eax = 1;
  return;
```


Pointing Problem Resolution via Tree Path Annotation

Initial IR

```
If(
  Sym("x"),
  Assign(
    Int(32, Signed),
    Sym("x"),
    Lit(Int(32, Signed),0)
  ),
  NoStmt
)
```

⇒ Path annotated IR

```
If(
  Sym("x"){Path([0])},
  Assign(
    Int(32, Signed){Path([0,1])},
    Sym("x"){Path([1,1])},
    Lit(Int(32, Signed), 0){Path([2,1])}
  ){Path([1])},
  NoStmt{Path([2])}
){Path([])}
```

⇓

Path annotated Box representation

```
T("path", [],
  V([
    H([ T("type", "keyword", "if"), "(", T("path", [0], "x"), ")" ]),
    I(
      T("path", [1],
        H([ T("path", [1,1], "x"), " ", "=", " ", T("path", [2,1], "0"), ";" ])
      )
    )
  ])
)
```

⇐ Click sensitive UI

```
if( x )
    x = 0 ;
```

Conclusions

- Bringing together human interaction and refactoring has the potential to make decompilation a more useful and effective process.
- A catalog of refactorings for low-level code was defined, where each refactoring helps making the code incrementally more intelligible.
- An interactive decompilation tool employing this concept was developed.
- As side product of this work, a Python version of the ATerm library was developed, as well as program transformation system inspired on the Stratego language.

Directions for Future Work

- Implement the remaining refactorings.
- Annotate the IR with its *Static Single Assignment* representation.
- Visualize the *Program Dependency Graph* and *program slices*.
- Make the interactive tool a generic refactoring browser.
- Target the C++ language instead of plain C.
- More versatile undo mechanism.

Thank you