Comparing Type-Based and Proof-Directed Decompilation

Alan Mycroft
Computer Laboratory, University of Cambridge
William Gates Building, J J Thomson Avenue, Cambridge CB3 0FD, UK
am@cl.cam.ac.uk

Atsushi Ohori
School of Information Science, Japan Advanced Institute of Science and Technology
Tatsunokuchi, Ishikawa 923-1292, Japan
ohori@jaist.ac.jp

Shin-ya Katsumata
Laboratory for Foundations of Computer Science, University of Edinburgh
James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, UK
sxk@dcs.ed.ac.uk

Abstract

In the past couple of years interest in decompilation has widened from its initial concentration on reconstruction of control flow into well-founded-in-theory methods to reconstruct type information. Mycroft described Type-Based Decompilation and Katsumata and Ohori described Proof-Directed Decompilation. This note summarises the two approaches and identifies their commonality, strengths and weaknesses; it concludes by suggesting how they may be integrated.

1. Introduction

In the past couple of years interest in decompilation has widened from its initial concentration on reconstruction of control flow (Cifuentes [2] is a very readable treatment) into well-founded-in-theory methods to reconstruct type information. Mycroft [7] described Type-Based Decompilation and Katsumata and Ohori [4] described Proof-Directed Decompilation. This note summarises the two approaches and makes some preliminary steps towards comparing and contrasting them.

The two approaches may at first appear disparate, but interesting similarities and differences emerge; we attempt to classify differences as to whether or not they are due to the different frameworks used.

Decompilation is part of the general problem of reconstructing higher-level information from previously written code. While this paper keeps its attention on recovering such information from assembly-level code, analogous techniques are applicable to reconstructing higher-level structures (e.g. object-oriented ones) from lower-level legacy code. Two examples are: (*i*) in telephony applications extracting intellectual property encoded as narrowband switching code (written in e.g. CHILL) for re-use in broadband telephony; and (*ii*) reconstructing COBOL-encoded commercial programs to enable them to interface to Internet browser servers.

1.1. Relation to other work

Both Type-Based Decompilation [7] and Proof-Directed Decompilation [4] were attempts to look at the problem of decompilation in a new manner and do not have a great deal of related work. However, the problem of retrospectively adding types to untyped programs has appeared in other communities, notably those of Lisp and Prolog. Post-hoc type inference for Prolog, in particular, has received significant attention. A good bibliographic survey can be found in Lu's papers [5, 6] on type analysis for logic programs. A particular distinction in this community is between *descriptive types* and *prescriptive types*. The former represents the situation when type declarations are reconstructed from an untyped program and the latter the situation when datatype declarations are known but the type of the executable parts of a program are left to be inferred. Because of the differences in starting point, Mycroft's work, based on untyped machine code can be seen as reconstructing descriptive types whereas the Katsumata-Ohori system based on JVM code can be seen as prescriptive.

1.2. The type-based approach

Mycroft's [7] approach to program reconstruction from machine code can be summarised as follows:

- start with classical untyped register-based machine code (hence type information has to be reconstructed to give descriptive types);
- unshare identifications caused by register allocation etc. by converting machine code to Single Static Assignment (SSA) form [3];
- associate one or more types with each machine instruction, e.g. `ld.w r0,0(r1)` with `r0` : t and `r1` : $ptr(t)$ for any type 32-bit type t;
- use type inference to identify overall types for procedures; type inference failures (e.g. occurs-check) cause data-structures to be reconstructed; inference can then continue.

1.3. The proof-directed approach

The Katsumata and Ohori [4] approach to decompilation takes as its foundation Ohori's [8] work on the Logical Abstract Machine. This provided a compilation strategy based around transforming from natural deduction (i.e. typed λ-calculus under the Curry-Howard correspondence) to the so-called sequential sequent calculus (which is very close to machine code).

Katsumata and Ohori extended this idea by observing that the inverse direction gave a novel decompilation technique:

- start with Java Virtual Machine (JVM) code (hence prescriptive types are available at procedure boundaries—see Section 4.2 for some exceptions);
- a system of rules describes when a sequence of instructions ending in return can be typed; the proof creates a term in which the type-structure of the JVM stack is known at each program point;
- each basic block ending in return is thereby translated into a function;
- basic blocks not ending in return are mapped to functions ending in tailcalls to one (for single exit blocks) or more (for multiple-exit blocks) functions;
- while the functions themselves are side-effect-free (as they use the λ-calculus with integers, products, sums and functions), side-effects are modelled as in the JVM via primitive instructions for accessing and updating object state;
- proofs for the inference system correspond to programs.

For natural reasons the decompilation process described is oriented to JVM with its stack-based structure; however Ohori's work on the sequential sequent calculus [8] has both stack-machine and register-machine instances.

1.4. Comparison framework

To aid comparison, we will use a register machine variant of the Katsumata-Ohori system instead of the stack machine form presented in [4]. Most of the formulation of this can be taken directly from Ohori [8]. This change has no real effect on the power of their system (and would be necessary if the JVM used a register file for temporary results instead of a local stack). However, it does make comparison with Mycroft's work easier (basically it obviates the worry about aliasing between local variables and expression temporaries on RISC-like implementations).

Hence the judgements on a basic block b are of the form $\Gamma \rhd b : \tau$ where Γ is a list of type assumptions $r_i : t_i$, at most one for each register r_i, and τ is the return type of the procedure when starting at block b. An example Katsumata-Ohori rule for an operator $\oplus$ of type $t_1 \times t_2 \to t_3$ would be

$$\frac{\Gamma, r_1 : t_1, r_2 : t_2, r_3 : t_3 \rhd C : \tau}{\Gamma, r_1 : t_1, r_2 : t_2 \rhd (r_3 := r_1 \oplus r_2); C : \tau}.$$

Axioms in the system correspond to `return` instructions:

$$\frac{}{\Gamma, r_1 : t_1 \rhd (\texttt{return}\, r_1) : t_1}.$$

The restriction in the above that r_3 differs from r_1 and r_2 can be avoided by introducing a logical weakening rule or by transformation to SSA form—we return to this point later to draw an interesting parallel between these ideas.

Similarly, while Mycroft decompiles a machine code function to a C function, and Katsumata and Ohori to a set of mutually recursive functions in an ML-like language, the techniques are nearer than immediately apparent. For example, Mycroft's C program concentrates on constructing larger expressions from C code. The C code initially consists of statements each representing one 3-address instruction and then as many temporary registers as possible are mapped into registers which are not manifest at the source code level. On the other hand the Katsumata-Ohori technique automatically synthesises large expressions from functional aspects of JVM code. When explicit updating operations are added to the JVM (e.g. `putfield`) these must be serialised carefully using `let` during the decompilation $[\![\cdot]\!]$ function whereas Mycroft's work serialises everything and then identifies expressions which can be made into a compound C expression (recall that ISO/ANSI C renders undefined expressions which rely on order of evaluation). This is achieved by finding SSA definitions which are only used once and whose def-use range does not cross a side effect.

Incidentally neither work addresses reconstruction of higher-level control structures (Cifuentes [2] deals with this topic).

2. Overall comparison

Decompilation is a process to reconstruct various high-level structures from a given low-level code. The quality and strength of a decompilation method can be assessed by analyzing how and what sort of structures are reconstructed.

In this general perspective, Mycroft approach and Katsumata-Ohori approach are complementary to each other in an important way, which reflects the two different requirements of decompilation. To decompile a code, one needs to recover:

(1) type information on the data structure the code manipulates, and

(2) the logical structure of the behaviour of the code.

The unification-based type inference approach of Mycroft is particularly suitable for (1). It generates disjunction of constraints for possible data structures, and then narrows the possibility by unification. In addition to simple term unification, the algorithm also handles multiple possibilities of representation due to untyped nature of the code language. This aspect is not addressed in Katsumata and Ohori's approach. They start with JVM code, where most type information of data objects is available. On the other hand, their approach provides a systematic and general treatment for (2). By representing a code as a proof in a logic that is equivalent to natural deduction, it can systematically transform the code to a lambda term. Due to its proof-theoretical nature, it is also suitable for establishing the semantic correctness of the decompilation.

Both of them are essential for decompilation and we believe that we can achieve a better formalism by combining the strength of the two. For this purpose, we need to know precise relationship between the two formalisms. Since the first aspect of recovering data type information is not addressed by Katsumata and Ohori's approach, in the following comparison, we focus reconstructing the control structure of the code. Based on the comparison, in Section 4, we comment on the possibility of developing a formalism integrating type-based approach and the proof-directed one, thus combining the strengths of the two approaches.

Limiting to reconstruction of control structures of a code, Katsumata and Ohori's formalism is stronger in that it support higher-order functions through their use of a proof system equivalent to the lambda calculus. This means that their approach can deal with code that dynamically manipulate code pointers such as `call r0(r1,r2)` for calling a code whose address is in a register `r0`, which is decompiled to a term with higher-order functions in an ML-like language.

However, when we restrict to code with only static jumps, then the two approaches achieve equivalent transformation. In fact, we can show the following. Given a basic block b terminated in `return` $\mathtt{r}_i$ then a judgment, J, of the form $\Gamma \rhd b : \tau$ can be proved if and only if $SSA(b)$ can be assigned a type. Moreover $[\![J]\!]$ (the Katsumata-Ohori decompilation) corresponds (and indeed significantly formalises) Mycroft's informal decompilation whereby each 3-address instruction is mapped into a corresponding C assignment, but variables which are only referenced once (and are not prevented by side-effects from being moved) have their definition substituted into their use.

This suggests that there should be some close connections underlying the two systems. Our main purpose is to reveal the connections.

3. SSA form and proof transformation

The major difference between the two approaches is that Mycroft uses SSA transformation while Katsumata and Ohori perform proof transformation. This section shows that, for a code with static jumps, these seemingly different formalisms are indeed equivalent.

Let us first review the basic concepts of SSA transformation, which is used in most of optimizing compilers of imperative languages.

There are roughly two aspects of SSA form. One is the restriction of register usages to "static single assignment". The other is the usage of the ϕ-function to represent control flow merge. Static single assignment restriction implies that the SSA transformation maps each user-variable (`x` say) into several intermediate variables, $\mathtt{x}_1, \ldots \mathtt{x}_k$, (essentially) one for each assignment to `x`. Uses of `x` are replaced by the definition of `x` which reaches the use. This is trivial in straight line code. The situation is more complicated at path merge when multiple definitions of `x` can reach a use. For example consider (assuming that the definition of $\mathtt{x}_1$ is extant on entry)

```
if (y) { x2 = x1+1; x3 = x2+y; }
  else x4 = x1-1;
z = x*2;
```

where both $\mathtt{x}_3$ and $\mathtt{x}_4$ reach the final use of `x`. In this case we introduce a ϕ-function (which selects from its arguments according to the path reaching the merge (left for left, right for right)):

```
if (y) { x₂ = x₁+1; x₃ = x₂+y; }
  else x₄ = x₁-1;
x₅ = φ(x₃,x₄); z = x₅*2;
```

In essence, the usefulness of static single assignment form in *compilation* is that it increases the number of user variables so as to permit a better packing to machine registers by register allocation by colouring. The key observation of Mycroft [7] is that separating disparate uses of a single register by various variables in the *decompilation* process is well achieved by using the same SSA mapping to undo register allocation.

We can now provide a fundamental justification of this technique as follows. When using a constraint system to reconstruct type information for decompilation, the constraint generated by a given instruction sequence should reflect the sequential nature of machine language. With regard to this point, the code $x := f(x); B$ generates the following constraint:

$$\exists z. P_f(x, z) \wedge P_B[z/x]$$

where $z \notin FV(P_B) \cup \{x\}$. Here $P_f(x, y)$ specifies the constraint on input x and output y to f and P_B is the constraint generated from B. To solve the above constraint, we first perform skolemization to shift all quantifiers to the outside. In this process all $\exists$-bound variables are chosen to be distinct. Then we notice that the constraint after skolemization is equivalent to that resulting from the SSA form of $x := f(x); B$. To see this suppose that x_{42} is the next variant of x to be used in the SSA transformation; then $SSA(B)$ will have x_{42} replacing all occurrences of x in B which are reached by the definition $x := f(x)$. Thus the generated constraint is

$$P_f(x, x_{42}) \wedge P_{SSA(B)}.$$

Hence we can see SSA as a form of skolemisation and conversely by taking SSA forms as the start point, we can generate a correct constraint in a simple and direct way.

The effect corresponding to SSA transformation, which is necessary for decompiling register machine code, is achieved in Katsumata and Ohori's proof transformation by representing each instruction as a logical rule of special kind that yields a proof system without explicit structural rules. Let us analyze this correspondence using the instruction `fst` which yields the first component of a pair as an example.

In Ohori's sequential sequent calculus, on which Katsumata and Ohori's system is based, this instruction is represented by the following logical rule.

$$\frac{\Gamma, r_1 : \tau_1 * \tau_2, r_2 : \tau_1 \rhd C : \tau}{\Gamma, r_1 : \tau_1 * \tau_2 \rhd (r_2 := \mathtt{fst}(r_1)); C : \tau}$$

This corresponds to a conjunction-left rule of the form:

$$\frac{\Gamma, \tau_1 \wedge \tau_2, \tau_1 \rhd \tau}{\Gamma, \tau_1 \wedge \tau_2 \rhd \tau}$$

Compared with the standard rule in the sequent calculus of the form,

$$\frac{\Gamma, \tau_1 \rhd \tau}{\Gamma, \tau_1 \wedge \tau_2 \rhd \tau}$$

the above rule keeps the primary formula $\tau_1 \wedge \tau_2$ in the premise. Using this form of rules systematically for all the logical connectives, one obtains a logic without structural rules, known as Kleene's G3 system. In this form of proof system, the number of assumptions used in a proof monotonically increases when we go up in the proof. For machine code, this property corresponds to the introduction of new register for every machine instruction. In the above example, r_1 holding the argument value is always retained and a new register r_2 for the result value is introduced. Reusing registers by colouring corresponds to reintroduction of the (logically redundant) weakening rule to G3-style system. The above analysis reveals that Mycroft's use of SSA corresponds to the use of G3-style rules in Katsumata and Ohori's system.

Let us turn to the second point of ϕ-function to represent control block merges. Mycroft's SSA translation has inserted ϕ-functions at control join nodes. His inference rule for type reconstruction treats all operands and result of a ϕ-function as being of the same type (this matters insofar that violations of this type constraint can be handled by inserting C cast operations around the operands to ϕ for decompiling code with enthusiastic use of casts).

Katsumata and Ohori use tailcall to represent branches. Re-assigning a new register at the merge point is represented by binding the formal parameter to the corresponding actual one with the requirement that both have the same type. A similar correspondence between ϕ-function and parameter passing in function call is noted by Appel [1]. One can observe further that ϕ-function in SSA formalism is a mechanism invented to reconcile the essentially functional nature of static single assignment requirement in a language with imperative jumps. Since Katsumata and Ohori's formalism uses higher-order functions to represent control structures, ϕ-functions are not needed to achieve the effect of SSA transformation.

In addition to the basic SSA transformation, Mycroft's contains a refinement for type-safe code (such as that which might be generated by Java). In his system, all variables appearing as result or operands of a given ϕ-function can be mapped into a single variable. This thereby selectively undoes the SSA transformation to recover variables which might have appeared at the user-level (and leaves disjoint user-variables which might have been mapped to the same register 'by accident' as it were). In Katsumata and Ohori's system based on function calls, this corresponds to suppressing the redundant copies between actual parameters and formal parameters by using the same registers.

From the above analysis, we have seen the equivalence of Mycroft's SSA transformation and Katsumata and Ohori's proof transformation for code with static jumps. The above comparison also reveals the reason of the extra generality of Katsumata and Ohori's for dynamic code pointers or first-class functions. A ϕ-function is indexed by static code blocks and therefore inherently static, while parameter passing does not have this restriction.

4. Further discussions

This section considers a possible way of integrating the two approaches, and discusses some issues on reconstructing types and SSA forms.

4.1. Integration of the two approaches

In the previous section we have noted that Katsumata and Ohori's proof transformation is more general in that it can deal with code pointers and higher-order functions. On the other hand, Mycroft approach provides a sophisticated mechanism for reconstruction of descriptive types, which is not handled in Katsumata and Ohori's system. A desirable next step would therefore be to develop a decompilation framework by integrating both of these features.

In Katsumata and Ohori's approach, each instruction corresponds to one logical rule, which directs the necessary proof transformation for decompilation. This reflects the view that JVM types are prescribed. This is largely true for JVM (there is an issue associated with punning between floating point data of type `double` and its two components and the issue of widening casts which generate no code). Mycroft's approach, however, takes the view that in ordinary machine code, each instruction corresponds to multiple different rules manipulating different data structures. To decompile untyped code, we must infer appropriate proof rule among multiple possibilities. One promising way is to use Mycroft type-based approach as a meta-level framework to construct appropriate proof transformation.

For each instruction Mycroft system generates the set of possible data types the instruction manipulates. It then generates the set of constraints imposed by control flow and register usage, and performs unification to solve the constraints. This approach is general enough to be applicable to reconstructing proof transformation. Since each data type is associated with a specific logical rule, the type-based approach can generate for each instruction the set of possible proof rules. It then generates constraints imposed by combination of proof rules, and perform unification to reconstruct a best possible proof tree, which can then be translated to lambda term by Katsumata and Ohori's approach.

4.2. Decompilation and JDK compiler

One problem which decompilation tools sometimes suffer from is that of being too strongly associated to a given compiler. Here we explore a few idiosyncrasies of the JDK compiler, in reality rather associated with the Java instruction set too.

One problem is that the JDK compiler uses a separate stack location for each user-variable. Although the Katsumata-Ohori approach can cope with re-use of local variable locations for variables of different types we see the naïve approach to compilation adopted by the JDK JVM compiler in the following example

```
class A {};
class B extends A {};
class C extends B {
   static void h(C z) {
      { B y = z; g(y); }
      { A x = z; f(x); }
   }
};
```

which results in JVM code of the form:

```
Method void h(C)
   0 aload_0
   1 astore_1
   2 aload_1
   3 invokestatic #6 <Method g(B)>
   6 aload_0
   7 astore_2
   8 aload_2
   9 invokestatic #5 <Method f(A)>
  12 return
```

Here three local storage locations are used for what amounts to a single value (clearly `y` and `x` could share a storage location here.)

When inheritance is used, a JVM instruction can be associated with multiple types. However, the presence of type information in JVM code produced by the JDK compiler is not total, consider the code

```
class A {};
class B extends A {};
class C extends B {
   static A f(C z) {
     B y = z;
     A x = y;
     return x;
   }
};
```

which yields

```
Method A f(C)
   0 aload_0
   1 astore_1
   2 aload_1
   3 astore_2
   4 aload_2
   5 areturn
```

Note that the widening casts have been effectively lost, although curiously the assignments have not! Hence the types of `x` and `y` are not derivable (note that the type of the declaration of `y` can be any of `A`, `B` or `C` and still give the same code.

Similar effects happen with

```
void f(B x, C y) {
  A z;
  z=x; g(z);
  z=y; h(z);
}
```

5. Conclusions

We conclude that Mycroft's system and the Katsumata-Ohori system produce very similar results in the areas in which they overlap. They have different strengths and weaknesses. Mycroft's model for decompiling untyped machine code corresponds to a descriptive, or implicit, type system (like some of those for typing Prolog [5, 6]) in which the system attempts to reconstruct data-type declarations, whereas Katsumata and Ohori's use of Java means that type reconstruction there can be limited to local variables as full (prescriptive) type information is available at procedure boundaries (however their framework would have no problem with parametric polymorphism).

On the other hand the formal basis of the Katsumata-Ohori system is much better founded in theory than Mycroft's, in that it can handle both stack and register machines without prior transformation to SSA form. Indeed, their system essentially un-aliases local stack locations shared by push (`load_i`), pop (`store_i`) and arithmetic temporaries in a very clean manner.

The unexpected parallel between SSA transformation and logical systems is a bonus.

Acknowledgments

The authors were respectively supported partly by UK EPSRC grant GR/N64256, Japanese Ministry of Education Grant-in-aid for Scientific Research No. 12680345 and a scholarship provided by the Laboratory for Foundations of Computer Science, University of Edinburgh.

References

[1] A. W. Appel. SSA is Functional Programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[2] C. Cifuentes. *Reverse Compilation Techniques*, PhD thesis, Queensland University of Technology, 1994. Available via `www.csee.uq.edu.au/~csmweb/dcc.html`

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. W. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451-490, 1991.

[4] S. Katsumata and A. Ohori. Proof-Directed De-Compilation of Low-Level Code. In *European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 352–366. Springer-Verlag, 2001.

[5] L. Lu. Type Analysis of Logic Programs in the Presence of Type Definitions. In *Proc. 1995 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 241–252. ACM press, 1995.

[6] L. Lu. A Polymorphic Type Analysis in Logic Programs by Abstract Interpretation. *Journal of Logic Programming*, 36(1):1–54, 1998.

[7] A. Mycroft. Type-Based Decompilation. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 208–223. Springer-Verlag, 1999.

[8] A. Ohori. The Logical Abstract Machine: a Curry-Howard Isomorphism for Machine Code. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722 of *Lecture Notes in Computer Science*, pages 300–318. Springer-Verlag, 1999.