Partial Type and Effect Inference for Rcc/Java is NP-Complete
Williams College Technical Note 04-01

Cormac Flanagan
Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA 95064

Stephen N. Freund
Computer Science Department
Williams College
Williamstown, MA 01267

February 25, 2004

1 Introduction

This technical note shows that reconstructing valid type annotations for Rcc/Java programs [FF00] is NP-complete. In particular we show that the problem of finding a satisfying assignment for a Boolean formula reduces in polynomial time to inferring the remaining annotations for a partially-annotated program.

2 Type Inference for Rcc/Java

We begin by reviewing the type inference problem for Rcc/Java. The Rcc/Java type system defines the notion of *well-typed programs*. Well-type programs contain a number of type annotations that document, for example, what locks protect which variables, and what locks are used to instantiate various parameterized classes. For example, the syntax "`int f guarded_by l`" declares a field `f` that is protected by lock `l`. That is, the lock `l` must be held before accessing `f`. Similarly, in the context of a parameterized class declaration

```
class C<ghost a> { ...  int f guarded_by a; ...  }
```

the type "`C<l>`" refers to an instantiation of the parameterized class `C` in which the lock `l` protects the field `f`. As demonstrated below, "`requires`" clauses can be added to method declarations to indicates that certain locks must be held upon method entry. A syntax-directed type checking algorithm, based on the Rcc/Java type system, can verify that a program is well-typed in polynomial time.

Removing some type annotations from a well-type program yields a *partially-annotated program*. In particular, we use the lock variables α, β, γ to refer to specific, but unknown, locks in scope. Thus, a partially-annotated program can contain field declarations such as "`int f guarded_by` α" and types such as "`C<`α`>`", as well as requires clauses containing lock variables.

The *type inference problem* for Rcc/Java is, given such a partially-annotated program P, to infer an appropriate substitution for lock variables that yields a well-typed program.

3 Translation

We start with a 3-SAT instance D with l variables $v_1, \ldots, v_l$ and n clauses:

$$\begin{array}{c} d_{1,1} \ \vee \ d_{1,2} \ \vee \ d_{1,3} \\ d_{2,1} \ \vee \ d_{2,2} \ \vee \ d_{2,3} \\ \vdots \\ d_{n,1} \ \vee \ d_{n,2} \ \vee \ d_{n,3} \end{array}$$

where each $d_{j,k}$ is either v_i or $\neg v_i$ for some i. We assume no clause refers to any variable v_i more than once.

Given such a 3-SAT instance D we generate a corresponding partially-annotated program P such that P is typeable if and only if D is satisfiable. Moreover, given an annotated version P' of P, we can generate a corresponding satisfying assignment for D.

The program P contains n classes, one for each clause, plus two additional classes `F` and `S`. The class `F` is a trivial class that takes a single ghost parameter.

```
class F ⟨ghost X⟩ { }
```

In addition, for each clause j, we define a class `C`j in P. We use the lock variables β_j as place-holders for unknown locks.

```
class Cj ⟨ghost a1, a2, a3⟩ {
    int y guarded_by βi;
    int z1 guarded_by a1;
    int z2 guarded_by a2;
    int z3 guarded_by a3;
    F⟨a1⟩ w1 guarded_by this;
    F⟨a2⟩ w2 guarded_by this;
    F⟨a3⟩ w3 guarded_by this;
}
```

Finally, P also contains the following class `S`. In this definition, we use $p_{j,k}$ to refer to `p`i if $d_{j,k} = v_i$, and use $p_{j,k}$ to refer to `n`i if $d_{j,k} = \neg v_i$.

```
class S  {
    final Object pi, ni;                                     // for i ∈ 1..l

    final Cj ⟨αj,1, αj,2, αj,3⟩ cj;                          // for j ∈ 1..n

    void f_j_k() requires pi, ni { cj.zk = 0; }              // for j ∈ 1..n, k ∈ 1..3, i ∈ 1..l
                                                             // such that dj,k ∈ {vi, ¬vi}

    void eq_j_k_j′_k′() requires cj, cj′ { cj.wk = cj′.wk′; }  // for j, j′ ∈ 1..n, k, k′ ∈ 1..3
                                                             // such that j ≠ j′ and dj,k and dj′,k′
                                                             // refer to the same variable

    void clause_j() requires pj,1, pj,2, pj,3 { cj.y = 0; }  // for j ∈ 1..n
}
```

In essence, the "`f`" functions enforce that the lock $\alpha_{j,k}$ corresponding to term $d_{j,k}$ is set to either `p`i or `n`i. It will be `p`i if v_i is true in the truth assignment we construct and `n`i if v_i is false. The "`eq`" functions ensure that all type variables corresponding to terms referring to the same variable are equal, i.e. the type variables for terms referring to v_i are either all `p`i or all `n`i. The "`clause`" functions ensure that at least one term in each clause of D is satisfied.

4 Correctness of the Translation

Suppose we can infer valid locks for the resulting program. We construct the satisfying assignment for the variables in the 3-SAT problem as follows. For all $i \in 1..l$,

$$v_i = \begin{cases} \textbf{true} & \text{if there exist } j,k \text{ such that } d_{j,k} \text{ is } v_i \text{ and } \beta_j \text{ is } \texttt{a}k \\ \textbf{false} & \text{otherwise} \end{cases}$$

Theorem 1 *This truth assignment satisfies D.*

Proof Consider any clause j. It must be that $\beta_j \in \{\mathtt{a1}, \mathtt{a2}, \mathtt{a3}\}$ because only locks external to class $\mathtt{C}j$ are held when $\mathtt{c}j\mathtt{.y}$ is accessed in the method $\mathtt{clause\_}j$. Suppose that $\beta_j = \mathtt{a}k$. There are two cases to consider:

1. Suppose $d_{j,k}$ is v_i. Then $v_i = \textbf{true}$ and clause j is satisfied.
2. Suppose $d_{j,k}$ is $\neg v_i$.
   (a) If $v_i = \textbf{false}$, then clause j is satisfied.
   (b) Suppose $v_i = \textbf{true}$. Then there exists j', k' such that

$$\beta_j = \mathtt{a}k \ \wedge \ d_{j,k} = \neg v_i$$
$$\beta'_j = \mathtt{a}k' \ \wedge \ d_{j',k'} = v_i$$

These two lines imply that the following two methods appear in $\mathtt{S}$:

```
void f_j_k() requires pi, ni { cj.zk = 0; }
void f_j'_k'() requires pi, ni { cj'.zk = 0; }
```

Since no other locks besides $\mathtt{p}i$ and $\mathtt{n}i$ are held when $\mathtt{c}j\mathtt{.z}k$ and $\mathtt{c}j'\mathtt{.z}k'$ are accessed, we have that

$$\alpha_{j,k} \in \{\mathtt{p}i, \mathtt{n}i\}$$
$$\alpha_{j',k'} \in \{\mathtt{p}i, \mathtt{n}i\}$$

Also, the type of $\mathtt{c}j\mathtt{.z}k$ is $\mathtt{F}\langle\alpha_{j,k}\rangle$, and the the type of $\mathtt{c}j'\mathtt{.z}k'$ is $\mathtt{F}\langle\alpha_{j',k'}\rangle$. Since $d_{j,k}$ and $d_{j',k'}$ refer to the same variable, we have the following method definition:

```
void eq_j_k_j'_k'() requires cj, cj'  { cj.wk = cj'.wk'; }
```

From this, it must be that $\mathtt{F}\langle\alpha_{j,k}\rangle$ is the same type as $\mathtt{F}\langle\alpha_{j',k'}\rangle$. Therefore, $\alpha_{j,k} = \alpha_{j',k'}$.

Since the field $\mathtt{y}$ in class $\mathtt{C}j$ is guarded by β_j and $\beta_j = \mathtt{a}k$, the lock $\alpha_{j,k}$ must be held when $\mathtt{c}j\mathtt{.y}$ is accessed in method $\mathtt{clause\_}j$. Since $d_{j,k} = \neg v_i$, then $\mathtt{n}i$ is in the $\mathtt{requires}$ set for $\mathtt{clause\_}j$. However, $\mathtt{p}i$ is not in the $\mathtt{requires}$ set since we assume no clause contains both a variable and its negation. Therefore, $\alpha_{j,k} = \mathtt{n}i$.

Similarly, the lock $\alpha_{j',k'}$ must be held when $\mathtt{c}j'\mathtt{.y}$ is accessed in method $\mathtt{clause\_}j'$. Since $d_{j',k'} = v_i$, then $\mathtt{p}i$ is in the $\mathtt{requires}$ set for $\mathtt{clause\_}j'$. However, $\mathtt{n}i$ is not in the $\mathtt{requires}$ set since we assume no clause contains both a variable and its negation. Therefore, $\alpha_{j',k'} = \mathtt{p}i$.

But, $\alpha_{j,k} = \alpha_{j',k'}$, and we have a contradiction. Thus, the case where $d_{j,k}$ is $\neg v_i$ and $v_i = \textbf{true}$ cannot happen.

□

Consider any satisfying truth assignment for D. We may construct a substitution from this assignment that shows P is typeable as follows. For all $j \in 1..n$,

$$\beta_i = \mathtt{a}k \text{ such that } d_{j,k} \text{ is } \textbf{true} \text{ under the satisfying assignment.}$$

Such a k must exist for each j. For all j, k,

$$\alpha_{j,k} = \begin{cases} \mathtt{p}i & \text{if } d_{j,k} \text{ refers to } v_i \text{ and } v_i = \textbf{true} \\ \mathtt{n}i & \text{if } d_{j,k} \text{ refers to } v_i \text{ and } v_i = \textbf{false} \end{cases}$$

Theorem 2 *This substitution shows that P is typeable.*

Proof We consider the typing requirements for each method in `S` and demonstrate that all `guarded_by` and type equality requirements are satisfied.

First, consider each method of the form

```
void f_j_k() requires pi, ni { cj.zk = 0; }
```

where $d_{j,k} \in \{v_i, \neg v_i\}$. The lock $\alpha_{j,k}$ guards the field `cj.zk` and must be held for this method to be well-typed. If $v_i =$ **true**, then $\alpha_{j,k} =$ `pi`, and the field access is race free since `pi` is in the `requires` clause. Similarly, if $v_i =$ **false**, then $\alpha_{j,k} =$ `ni`, and the field access is race free.

Next, consider each method

```
void eq_j_k_j'_k'() requires cj, cj' { cj.wk = cj'.wk'; }
```

where $d_{j,k}$ and $d_{j',k'}$ refer to the same variable v_i. In this case, the typing requirements are that the locks guarding the fields `cj.wk` and `cj'.wk'` are held, and that the types of `cj.wk` and `cj'.wk'` are equal. The locks are clearly held since they are required. The type of `cj.wk` is $\mathtt{F}\langle\alpha_{j,k}\rangle$ and the type of `cj'.wk'` is $\mathtt{F}\langle\alpha_{j',k'}\rangle$. Given the construction of the substitution above, $\alpha_{j,k}$ and $\alpha_{j',k'}$ will either both be `pi` or both be `ni`.

Finally, consider each method

```
void clause_j() requires p_{j,1},p_{j,2},p_{j,3} { cj.y = 0; }
```

The lock β_j guarding `cj.y` is `ak`, for some k such that $d_{j,k}$ is true under the truth assignment. Since `cj` is an instantiation for `Cj`, with lock argument $\alpha_{j,k}$ for `ak`, the lock $\alpha_{j,k}$ must be held when `cj.y` is accessed. There are two cases:

1. $\alpha_{j,k} =$ `pi`: Then $d_{j,k}$ refers to v_i and $v_i =$ **true**. Since $d_{j,k}$ must be true, $d_{j,k} = v_i$. Thus, $p_{j,k} =$ `pi`, and $\alpha_{j,k}$ is held inside `clause_j`.

2. $\alpha_{j,k} =$ `ni`: Then $d_{j,k}$ refers to v_i and $v_i =$ **false**. Since $d_{j,k}$ must be true, $d_{j,k} = \neg v_i$. Thus, $p_{j,k} =$ `ni`, and $\alpha_{j,k}$ is held inside `clause_j`.

□

A consequence of these two theorems is that any type inference scheme for Rcc/Java will have potentially exponential worst-case behavior. We believe this result holds for Parameterized Race-Free Java [BR01], and similar systems, as well.

5 Example

Consider the SAT problem:

$$v_1 \vee \neg v_2 \vee v_3$$
$$\neg v_1 \vee v_2 \vee v_3$$

This SAT problem reduces to the following type inference problem:

```
class F ⟨ghost X⟩ { }
class C1 ⟨ghost a1, a2, a3⟩ {
    int y guarded_by β1;
    int z1 guarded_by a1;
    int z2 guarded_by a2;
    int z3 guarded_by a3;
    F⟨a1⟩ w1 guarded_by this;
    F⟨a2⟩ w2 guarded_by this;
    F⟨a3⟩ w3 guarded_by this;
}
class C2 ⟨ghost a1, a2, a3⟩ {
    int y guarded_by β2;
    int z1 guarded_by a1;
    int z2 guarded_by a2;
    int z3 guarded_by a3;
    F⟨a1⟩ w1 guarded_by this;
    F⟨a2⟩ w2 guarded_by this;
    F⟨a3⟩ w3 guarded_by this;
}
```

```
class S  {
    final Object p1, p2, p3, n1, n2, n3;
    final C1 ⟨α1,1, α1,2, α1,3⟩ c1;
    final C2 ⟨α2,1, α2,2, α2,3⟩ c2;

    void f_1_1() requires p1, n1 { c1.z1 = 0; }
    void f_1_2() requires p2, n2 { c1.z2 = 0; }
    void f_1_3() requires p3, n3 { c1.z3 = 0; }

    void f_2_1() requires p1, n1 { c2.z1 = 0; }
    void f_2_2() requires p2, n2 { c2.z2 = 0; }
    void f_2_3() requires p3, n3 { c2.z3 = 0; }

    void eq_1_1_2_1() requires c1, c2 { c1.w1 = c2.w1; }
    void eq_1_2_2_2() requires c1, c2 { c1.w2 = c2.w2; }
    void eq_1_3_2_3() requires c1, c2 { c1.w3 = c2.w3; }

    void clause_1() requires p1, n2, p3 { c1.y = 0; }
    void clause_2() requires n1, p2, p3 { c2.y = 0; }
}
```

The following substitution for lock variables converts the above partially-typed program into a well-typed program. (Other substitutions also exist.)

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\beta_1 & = & \texttt{a1} & \alpha_{1,1} & = & \texttt{p1} & \alpha_{2,1} & = & \texttt{p1} \\
\beta_2 & = & \texttt{a3} & \alpha_{1,2} & = & \texttt{n2} & \alpha_{2,2} & = & \texttt{n2} \\
 & & & \alpha_{1,3} & = & \texttt{p3} & \alpha_{2,3} & = & \texttt{p3}
\end{array}
$$

From this substitution we generate the truth assignment where $v_1 = \textbf{true}$, $v_2 = \textbf{false}$, and $v_3 = \textbf{true}$. This truth assignment satisfies the original 3-SAT problem.

It is also straight-forward to construct substitutions for the type variables that make the program well-typed for any satisfying assignment of the 3-SAT problem.

References

[BR01] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 56–69, 2001.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2000.