Overloading is NP-complete

A tutorial dedicated to Dexter Kozen

Jens Palsberg

UCLA, University of California, Los Angeles

Abstract. We show that overloading is NP-complete. This solves exercise 6.25 in the 1986 version of the Dragon book.

1 Introduction

Overloading is a form of polymorphism in which a name denotes multiple functions and "the context is used to decide which function is denoted by a particular instance of the name" [3]. Many programming languages support overloading. For example, in MATLAB [8] the name `mpower` is overloaded to denote both:

- a function of two arguments `a,b` where `a` is a square matrix and the exponent `b` is a scalar, and
- a function of two arguments `a,b` where `a` is a scalar and the exponent `b` is a square matrix.

When we call `mpower` in MATLAB, the arguments will be used to decide which function will be called. Both functions return a square matrix. Similarly, in Java we can program an interface with two methods that are both named `mpower`:

```
interface Math {
  SquareMatrix mpower(SquareMatrix a, Scalar b);
  SquareMatrix mpower(Scalar a, SquareMatrix b);
}
```

If a class implements `Math` and we call `mpower` on an object of that class, the arguments will be used to decide which method will be called.

We say that an implementation *resolves overloading* when it decides which overloaded function will be called.

How does a language implementation decide which function will be called? The easiest case is when different functions have different numbers of arguments. In that case, the number of arguments at the call site is sufficient to decide which function will be called. The harder case is when different functions have the same number of arguments, like in the `mpower` example above. In that case, the *types* of the arguments at the call site must be used to decide which function will be called.

Both MATLAB and Java has a notion of type associated with each value. A MATLAB value can be a square matrix or scalar, for example, and a Java value can be a SquareMatrix object or a Scalar object, for example. The type of a value determines which operations can be performed on that value. For example, we can perform `mpower` on a scalar and a square matrix, but not on two square matrices.

MATLAB and Java differ in *when* the type of a value is known to the implementation. MATLAB has a *dynamic* type system in which the types are known at run time but are unknown to the compiler, at least in some cases. So, in some cases, a MATLAB implementation must decide at run time which overloaded function to call:

"MATLAB determines which implementation to call at the time of evaluation in this case." [12]

Java has a *static* type system in which the type of every value is known to the compiler. Java requires that a Java implementation can resolve overloading by investigating the number and the types of the arguments passed into a method:

"When a method is invoked, the number of actual arguments (and any explicit type arguments) and the compile-time types of the arguments are used, at compile time, to determine the signature of the method that will be invoked This enables a Java implementation to decide at compile time which overloaded function to call." [7]

For dynamically typed languages like MATLAB, overloading may decrease run-time performance. This happens particularly when MATLAB has to examine the types of function arguments at run time to decide which overloaded function to call.

For statically typed languages like Java, overloading is essentially a syntactic convenience that frees programmers from inventing different names for functions with similar functionality. Instead, the responsibility to invent such different names is passed on to the compiler. The compiler uses the types of function arguments to do a source-to-source transformation that eliminates the use of overloading by giving different names to different functions. After such a transformation, compilation can proceed as usual.

In this paper we focus on the computational complexity of overloading resolution in statically typed languages. In Sections 2–7 we will explain in detail why overloading resolution is NP-complete for an important case: a λ-calculus with overloading. Our proof consists of three polynomial-time reductions:

$$\begin{aligned} \text{3SAT} \;&\leq\; \text{Monotone One-in-Three 3SAT} \;\leq\; \text{Overloading Resolution} \\ &\leq\; \text{Constraint Solving} \end{aligned}$$

and an easy proof that the Constraint Solving problem is in NP. Thus, all of the listed problems are NP-complete. In Section 8 we will discuss interactions of overloading and other language features.

Our proof is inspired by two lines of research. The first inspiration is hardware description languages that allow component overloading [17, 11]. The idea of component overloading is similar to function overloading: the context is used to decide which component is denoted by a particular instance of a name. Vachharajani et al [17], and Mathaikutty and Shukla [11] both sketched reductions that show that component overloading is NP-complete. Our reduction targets a λ-calculus with overloading and we give a detailed proof.

The second inspiration is a paper by Kozen et al. [10] who showed how to reduce a type-checking problem to a constraint solving problem.

In the 1986 version of the Dragon book [1], Exercise 6.25 is to show that overloading is NP-complete:

****6.25** The resolution of overloading becomes more difficult if identifier declarations are optional. More precisely, suppose that declarations can be used to overload identifiers representing function symbols, but that all occurrences of an undeclared identifier have the same type. Show that the problem of determining if an expression in this language has a valid type is NP-complete. This problem arises during type checking in the experimental language Hope (Burstall, MacQueen, and Sannella [1980]). [1, p.384]

However, the exercise is difficult, we think, and the literature contains no detailed, formal proof, hence this tutorial. We formalize the exercise as a problem about a λ-calculus, as we explain next.

2 Example Language

Our example language is a λ-calculus with overloading:

$$(\textit{Expression}) \quad e ::= x \mid \lambda x.e \mid e\ e$$

We use x to range over identifiers. Overloaded functions all come from the initial environment: each free variable of an expression refers to an overloaded function. We assume that in every expression, the bound variables are distinct and different from the free variables.

Each overloaded function has a restricted form of *intersection type* [5, 9]: each intersection type is an intersection of simple types. When a program uses one of the overloaded functions from the initial environment, the function gets one of the simple types in the intersection. In contrast, every programmer-defined function has a simple type.

Our interpretation of Exercise 6.25 in the 1986 version of the Dragon book is that the initial environment provides identifiers that are declared with intersection types, while each programmer-defined function leaves an identifier undeclared, that is, without a type annotation, and required to have a simple type.

Let us now define the type system formally. We use c to range over a finite set of at least two base types. The types are:

$$(\textit{Type}) \quad s, t ::= c \mid t \to t$$
$$(\textit{Intersection Type}) \quad u ::= u \wedge u \mid t$$

If $u = (\ldots \wedge t \wedge \ldots)$, then we write $u \leq t$.

A type environments is a finite mapping from identifiers to types of the form u. We use A to range over type environments.

A type judgment is of the form $A \vdash e : t$. The type rules are:

$$A \vdash x : t \qquad \text{if } A(x) \leq t \tag{1}$$

$$\frac{A[x : s] \vdash e : t}{A \vdash \lambda x.e : s \to t} \tag{2}$$

$$\frac{A \vdash e_1 : s \to t \qquad A \vdash e_2 : s}{A \vdash e_1\ e_2 : t} \tag{3}$$

We say that an expression e is typable with a type environment A if and only if there exists t such that $A \vdash e : t$ is derivable.

We define the resolution of overloading decision problem as follows:

Overloading Resolution
Instance: An expression e and a type environment A.
Problem: Is e typable with A?

Lemma 1. *Suppose a derivation of $A \vdash e_0 : t$ contains the judgment $A' \vdash e' : t'$. If x is a free variable of e_0, then $A(x) = A'(x)$.*

Proof. We proceed by induction on e_0. We have three cases.

- If $e_0 \equiv x$, then e' is also x, so we have $A = A'$, hence $A(x) = A'(x)$.

– If $e_0 \equiv \lambda y.e$, then from the induction hypothesis and type rule (2), we have (1) $(A[y : s])(x) = A'(x)$. From (1) and that bound variables are different from the free variables, we have $A(x) = (A[y : s])(x) = A'(x)$, as desired.
– If $e_0 \equiv e_1 e_2$, then e' occurs in either e_1 or e_2. Let us do a case analysis of the two cases. Suppose e' occurs in e_1. From the induction hypothesis and type rule (3), we have $A(x) = A'(x)$, as desired. The other case where e' occurs in e_2 is similar, and also here we immediately get $A(x) = A'(x)$, as desired.

□

3 Constraints

We define

$$(\mathit{Term}) \quad r ::= v \mid c \mid r \to r$$

A constraint system over a set of type variables V is a finite collection of constraints of the forms:

$$u \leq r$$
$$r = r'$$

where u is an intersection type as defined above, and each variable that occurs in r or r' is a member of V.

We use φ to range over finite mappings from type variables to types of the form t.

We define:

$$\varphi(c) = c$$
$$\varphi(r_1 \to r_2) = \varphi(r_1) \to \varphi(r_2)$$

A constraint system C has solution φ if and only if

– for each constraint $u \leq r$ in C, we have $u \leq \varphi(r)$,
– for each constraint $r = r'$ in C, we have $\varphi(r) = \varphi(r')$, and

We say that a constraint system C is satisfiable if C has a solution.

Constraint Solving
Instance: A constraint system C.
Problem: Is C satisfiable?

Theorem 1. *Constraint Solving is in NP.*

Proof. Let C be a constraint system. For each constraint $u \leq r$ in C, where $u = \bigwedge_{i=1}^{n} t_i$, guess t_{i_0} and replace constraint $u \leq r$ with the constraint $t_{i_0} = r$. The resulting constraint system can be solved with first-order unification which is doable in polynomial time [13]. □

Let us define a transformation S on constraint systems. The idea of S is to remove a constraint without changing whether the constraint system is satisfiable.

We define the transformation S:

$$S(C, v = r) = \begin{cases} (C \setminus \{\ v = r\ \})[v := r] & \text{if } (v = r) \in C \text{ and } v \text{ doesn't occur in } r \\ C & \text{otherwise} \end{cases}$$

Intuitively, $S(C, v = r)$ removes the constraint $v = r$ from C and then replaces all occurrences of v by r in the resulting constraint system.

Lemma 2. *C is satisfiable if and only if $S(C, v = r)$ is satisfiable.*

Proof. We have two cases.

If $C = S(C, v = r)$, then the lemma is immediate.

If C contains the constraint $v = r$, where v doesn't occur in r, then we consider the two directions of the lemma.

In the forwards direction, we have immediately that if C has solution φ, then also $S(C, v = r)$ has solution φ.

In the backwards direction, suppose $S(C, v = r)$ has solution φ. We now have two cases.

In the first case, suppose v doesn't occur in $C \setminus \{\ v = r\ \}$. Let $\{\ v_1, \ldots, v_m\ \}$ be the set of type variables that occur in r but don't occur in $C \setminus \{\ v = r\ \}$. Let c be a base type. Define:

$$\psi = \varphi,\ (v \mapsto (\varphi(r))[v_1 := c, \ldots, v_m := c]),\ (v_1 \mapsto c),\ \ldots,\ (v_m \mapsto c)$$

We have immediately that ψ solves all constraints in $C \setminus \{\ v = r\ \}$. For the constraint $v = r$, we have that $\psi(v) = (\varphi(r))[v_1 := c, \ldots, v_m := c] = \psi(r)$ so ψ also solves $v = r$.

In the second case, suppose v does occur in $C \setminus \{\ v = r\ \}$. Notice that since v occurs in $C \setminus \{\ v = r\ \}$, we have that r occurs in $(C \setminus \{\ v = r\ \})[v := r]$. Define:

$$\psi = \varphi,\ (v \mapsto \varphi(r))$$

We have immediately that ψ solves all constraints in $C \setminus \{\ v = r\ \}$. For the constraint $v = r$, we have that $\psi(v) = \varphi(r) = \psi(r)$, so ψ also solves $v = r$. □

4 From Overloading to Constraints

We will now show a reduction of the overloading resolution problem to a constraint solving problem. The reduction is useful both for showing that overloading resolution is in NP and that it is NP-hard.

For a expression e_0 and a type environment A, we define the set V_{e_0} of type variables:

$$\begin{aligned} V_{e_0} = &\{\ v_x \mid x \text{ is an occurrence of a free variable in } e_0\ \} \\ &\cup \{\ v_x \mid \lambda x.e \text{ is an occurrence of a subexpression in } e_0\ \} \\ &\cup \{\ v_{\lambda x.e} \mid \lambda x.e \text{ is an occurrence of a subexpression in } e_0\ \} \\ &\cup \{\ v_{e_1 e_2} \mid e_1 e_2 \text{ is an occurrence of a subexpression in } e_0\ \} \end{aligned}$$

From e_0 and A, generate these type constraints over V_{e_0}:

- For each occurrence of a free variable x in e_0, the constraint $A(x) \leq v_x$.
- For each occurrence of $\lambda x.e$ in e_0, the constraint $v_{\lambda x.e} = v_x \to v_e$.
- For each occurrence of $e_1 e_2$ in e_0, the constraint $v_{e_1} = v_{e_2} \to v_{e_1 e_2}$.

We use $C_{e_0,A}$ to denote the constraint system generated from e_0 and A.

Theorem 2. *An expression e_0 is typable with type environment A if and only if $C_{e_0,A}$ is satisfiable.*

Proof. For the forwards direction, assume that e_0 is typable with type environment A. In other words, there exists t_0 such that $A \vdash e_0 : t_0$ is derivable.

We will define a function $\varphi : V_{e_0} \to \mathit{Type}$. In the derivation of $A \vdash e_0 : t$, each occurrence of a subexpression e' of e_0 occurs exactly once in a type judgment of the form $A' \vdash e' : t'$. If the occurrence of e' is a free variable of e_0 or of one of the forms $\lambda x.e$ and $e_1 e_2$, then define $\varphi(v_{e'}) = t'$. Additionally, each occurrence of a subexpression $\lambda x.e'$ of e_0 occurs exactly once in a type judgment of the form $A' \vdash \lambda x.e' : s' \to t'$; define $\varphi(v_x) = s'$.

We need to show that $C_{e_0,A}$ has solution φ. Let us do a case analysis on the members of $C_{e_0,A}$.

- For an occurrence of a free variable x in e_0, we have the constraint $A(x) \leq v_x$. From the type rule (1) and Lemma 1, we have that there exists a type t such that $A(x) \leq t$ and $\varphi(v_x) = t$. We conclude that φ solves the constraint.
- For an occurrence of $\lambda x.e$ in e_0, we have the constraint $v_{\lambda x.e} = v_x \to v_e$. From the type rule (2), we have that there exist types s, t such that $\varphi(v_{\lambda x.e}) = s \to t$ and $\varphi(v_x) = s$ and $\varphi(v_e) = t$. We conclude that φ solves the constraint.
- For an occurrence of $e_1 e_2$ in e_0, we have the constraint $v_{e_1} = v_{e_2} \to v_{e_1 e_2}$. From the type rule (3), we have that there exist types s, t such that $\varphi(v_{e_1}) = s \to t$ and $\varphi(v_{e_2}) = s$ and $\varphi(v_{e_1 e_2}) = t$. We conclude that φ solves the constraint.

This concludes the proof of the forwards direction.

For the backwards direction, assume that $C_{e_0,A}$ is satisfiable. Let φ a solution of $C_{e_0,A}$. For each occurrence of a subexpression e' of e_0, let $x_1, \ldots, x_n$ be the bound variables of the λ-abstractions that enclose e' in e_0. Define

$$A_{e'} = A, (x_1 : \varphi(v_{x_1})), \ldots, (x_n : \varphi(v_{x_n}))$$

Notice that $A = A_{e_0}$. We will prove that for each occurrence of a subexpression e' of e_0, we have

$$A_{e'} \vdash e' : \varphi(v_{e'})$$

We proceed by induction on e'. We have four cases.

- For an occurrence of a free variable x in e_0, we have (1) $A(x) \leq \varphi(v_x)$. Notice that (2) $A(x) = A_x(x)$. From (1), (2), we have $A_x(x) = A(x) \leq \varphi(v_x)$ so we can use type rule (1) to conclude $A_x \vdash x : \varphi(v_x)$, as desired.
- For an occurrence of a bound variable x in e_0, we have from type rule (1) that $A_x \vdash x : \varphi(v_x)$, as desired.
- For an occurrence of $\lambda x.e$ in e_0, we have (1) $\varphi(v_{\lambda x.e}) = \varphi(v_x) \to \varphi(v_e)$. From the induction hypothesis used on e, we have (2) $A_e \vdash e : \varphi(v_e)$. Next notice that (3) $A_e = A_{\lambda x.e}, (x : \varphi(v_x))$. From (2), (3), and type rule (2), we conclude (4) $A_{\lambda x.e} \vdash \lambda x.e : \varphi(v_x) \to \varphi(v_e)$. From (1) and (4), we conclude $A_{\lambda x.e} \vdash \lambda x.e : \varphi(v_{\lambda x.e})$, as desired.
- For an occurrence of $e_1 e_2$ in e_0, we have (1) $\varphi(v_{e_1}) = \varphi(v_{e_2}) \to \varphi(v_{e_1 e_2})$. From the induction hypothesis used on e_1 and e_2, we have (2) $A_{e_1} \vdash e_1 : \varphi(v_{e_1})$ and $A_{e_2} \vdash e_2 : \varphi(v_{e_2})$. Next notice that (3) $A_{e_1} = A_{e_2} = A_{e_1 e_2}$. From (2) and (3) we conclude (4) $A_{e_1 e_2} \vdash e_1 : \varphi(v_{e_1})$ and $A_{e_1 e_2} \vdash e_2 : \varphi(v_{e_2})$. From type rule (3) and from (1) and (4), we derive $A_{e_1 e_2} \vdash e_1 e_2 : \varphi(v_{e_1 e_2})$, as desired.

This concludes the proof of the backwards direction. □

5 Monotone One-in-Three 3SAT

The Monotone One-in-Three 3SAT problem is an excellent fit for proving that overloading resolution is NP-hard. Schaefer proved that Monotone One-in-Three 3SAT is NP-complete [14]. Indeed, he proved a more general result with a proof that is a bit complicated. We will give a straightforward proof that Monotone One-in-Three 3SAT is NP-complete.

We will use 0,1 to denote the Boolean values *false*, *true*, respectively.

Let R be a three-place Boolean relation which is true if and only if exactly one of its three arguments is true. Thus, $R(1,0,0) = R(0,1,0) = R(0,0,1) = 1$, while $R(1,1,1) = R(1,1,0) = R(1,0,1) = R(0,1,1) = R(0,0,0) = 0$.

We use φ to range over mappings from variables to Boolean values.

We say that a mapping φ *satisfies* a formula if, after we replace each variable x with $\varphi(x)$, the formula evaluates to 1. We also say that a formula is *satisfiable* if there exists a mapping φ that satisfies the formula.

We define the function T:

$$T(z_1, z_2, z_3) = R(z_1, a_1, a_4) \wedge R(z_2, a_2, a_4) \wedge R(a_1, a_2, a_5) \wedge \\ R(a_3, a_4, a_6) \wedge R(z_3, a_3, f) \wedge R(t, f, f)$$

where in each application of T we have that $a_1, a_2, a_3, a_4, a_5, a_6, t, f$ are fresh and distinct variables.

We say that φ' extends φ if and only if $dom(\varphi') \supseteq dom(\varphi) \wedge \forall x \in dom(\varphi') \cap dom(\varphi) : \varphi'(x) = \varphi(x)$.

Lemma 3. *Let φ be an assignment of the variables z_1, z_2, z_3 to Boolean values. We have that φ satisfies $(z_1 \vee z_2 \vee z_3)$ if and only if φ can be extended to φ' that satisfies $T(z_1, z_2, z_3)$.*

Proof. In the forwards direction, let us extend each of the seven mappings that satisfy $(z_1 \vee z_2 \vee z_3)$ to mappings that also satisfy $T(z_1, z_2, z_3)$:

Mapping	z_1	z_2	z_3	a_1	a_2	a_3	a_4	a_5	a_6	t	f
φ'_1	0	0	1	0	0	0	1	1	0	1	0
φ'_2	0	1	0	1	0	1	0	0	0	1	0
φ'_3	0	1	1	1	0	0	0	0	1	1	0
φ'_4	1	0	0	0	1	1	0	0	0	1	0
φ'_5	1	0	1	0	1	0	0	0	1	1	0
φ'_6	1	1	0	0	0	1	0	1	0	1	0
φ'_7	1	1	1	0	0	0	0	1	1	1	0

In the backwards direction, consider the only mapping 1) $\varphi = [z_1 \mapsto 0, z_2 \mapsto 0, z_3 \mapsto 0]$ that doesn't satisfy $(z_1 \vee z_2 \vee z_3)$. Suppose we can extend φ to φ' that satisfies $T(z_1, z_2, z_3)$. From $R(t, f, f)$ we have that $\varphi'(t) = 1$ and 2) $\varphi'(f) = 0$. From $R(z_3, a_3, f)$, (1), and (2), we have 3) $\varphi'(a_3) = 1$. From $R(a_3, a_4, a_6)$ and (3), we have 4) $\varphi'(a_4) = 0$. From $R(z_1, a_1, a_4) \wedge R(z_2, a_2, a_4)$ and (1) and (4), we have 5) $\varphi'(a_1) = \varphi'(a_2) = 1$. However, (5) implies that φ' doesn't satisfy $R(a_1, a_2, a_5)$, a contradiction. □

Let us now define the Monotone One-in-Three 3SAT problem.

Monotone One-in-Three 3SAT
Instance: A formula $\bigwedge_{i=1}^{n} R(x_{i1}, x_{i2}, x_{i3})$, where each x_{ij} is a variable.
Problem: Is the formula satisfiable?

Theorem 3. *Monotone One-in-Three 3SAT is NP-complete.*

Proof. Monotone One-in-Three 3SAT is in NP because we can guess a mapping φ and then check in polynomial time where φ satisfies the formula.

To prove that Monotone One-in-Three 3SAT is NP-hard we will show a reduction from 3SAT. An instance of 3SAT is a formula of the form $\mathcal{F} = \bigwedge_{i=1}^{n}(l_{i1} \vee l_{i2} \vee l_{i3})$, where each l_{ij} is either a variable x_i or the negation of a variable $\overline{x}_i$. The 3SAT problem is whether the formula is satisfiable. 3SAT is NP-complete [6].

Let $x_1, \ldots, x_m$ be the variables used in $\mathcal{F}$. We will define a formula $\mathcal{H}$ over the variables $x_1, \ldots, x_m, y_1, \ldots y_m, t, f$, where $y_1, \ldots y_m, t, f$ are all distinct and different from $x_1, \ldots, x_m$. We will use the helper mapping π:

$$\pi(x_i) = x_i$$
$$\pi(\overline{x}_i) = y_i$$

Now we define $\mathcal{H}$:

$$\mathcal{H} = [\ \bigwedge_{i=1}^{n} T(\pi(l_{i1}), \pi(l_{i2}), \pi(l_{i3}))\ ] \ \wedge\ [\ \bigwedge_{j=1}^{m} R(x_j, y_j, f)\ ] \ \wedge\ R(t, f, f)$$

Notice that $\mathcal{H}$ is an instance of Monotone One-in-Three 3SAT. Notice also that, for each $j \in 1..m$, the clauses $R(x_j, y_j, f_1)$ and $R(t, f, f)$ force any assignment that satisfies $\mathcal{H}$ to map x_j to 1 and y_j to 0, or map x_j to 0 and y_j to 1. The reason is that $R(t, f, f)$ forces the assignment to map f to 0, and so the assignment must map exactly one of x_j and y_j to 1. So, y_j plays the role of $\overline{x}_j$.

Suppose φ satisfies $\mathcal{F}$. From Lemma 3 we have that we can extend φ to φ' such that φ' satisfies $T(\pi(l_{i1}), \pi(l_{i2}), \pi(l_{i3}))$. From the observation above about y_j, t, f we have that we can easily extend φ' to φ'' such that φ'' satisfies $\mathcal{H}$.

Conversely, suppose φ satisfies $\mathcal{H}$. From Lemma 3 we have that φ satisfies $\mathcal{F}$. □

6 From Monotone One-in-Three 3SAT to Overloading

Let F, T be two base types. Define

$$u_0 = (T \to F \to F \to T) \wedge (F \to T \to F \to T) \wedge (F \to F \to T \to T)$$

For an instance of Monotone One-in-Three 3SAT

$$\mathcal{H} = \bigwedge_{i=1}^{n} R(x_{i1}, x_{i2}, x_{i3})$$

that uses the variables $y_1, \ldots, y_m$. Thus, each x_{ij} is one of $y_1, \ldots, y_m$. We define the type environment:

$$A = \{\ (f_1 \mapsto u_0), \ldots,\ (f_n \mapsto u_0)\ \}$$

and we define the λ-expression:

$$e_0 = \lambda g.\lambda y_1. \ldots \lambda y_m.\ g\ (f_1 x_{11} x_{12} x_{13})\ \ldots\ (f_n x_{n1} x_{n2} x_{n3})$$

Theorem 4. *$\mathcal{H}$ is satisfiable if and only if e_0 is typable with a type environment A.*

Proof. From Theorem 2 we have that e_0 is typable with a type environment A if and only if $C_{e_0,A}$ is satisfiable. So all we need to prove is that:

$$\mathcal{H} \text{ is satisfiable if and only if } C_{e_0,A} \text{ is satisfiable.} \tag{4}$$

Let us calculate $C_{e_0,A}$:

$$v_{\lambda g....} = v_g \to v_{\lambda y_1....} \tag{5}$$
$$v_{\lambda y_1....} = v_{y_1} \to v_{\lambda y_2....} \tag{6}$$
$$\dots$$
$$v_{\lambda y_m....} = v_{y_m} \to v_{g\ (f_1 x_{11} x_{12} x_{13})\ \dots\ (f_n x_{n1} x_{n2} x_{n3})} \tag{7}$$
$$v_{g\ (f_1 x_{11} x_{12} x_{13})\ \dots\ (f_{n-1} x_{(n-1)1} x_{(n-1)2} x_{(n-1)3})} = v_{f_n x_{n1} x_{n2} x_{n3}} \to v_{g\ (f_1 x_{11} x_{12} x_{13})\ \dots\ (f_n x_{n1} x_{n2} x_{n3})} \tag{8}$$
$$\dots$$
$$v_{g\ (f_1 x_{11} x_{12} x_{13})} = v_{f_2 x_{21} x_{22} x_{23}} \to v_{g\ (f_1 x_{11} x_{12} x_{13})\ (f_2 x_{21} x_{22} x_{23})} \tag{9}$$
$$v_g = v_{f_1 x_{11} x_{12} x_{13}} \to v_{g\ (f_1 x_{11} x_{12} x_{13})} \tag{10}$$
$$v_{f_i x_{i1} x_{i2}} = v_{x_{i3}} \to v_{f_i x_{i1} x_{i2} x_{i3}} \quad i \in 1..n \tag{11}$$
$$v_{f_i x_{i1}} = v_{x_{i2}} \to v_{f_i x_{i1} x_{i2}} \quad i \in 1..n \tag{12}$$
$$v_{f_i} = v_{x_{i1}} \to v_{f_i x_{i1}} \quad i \in 1..n \tag{13}$$
$$u_0 \leq v_{f_i} \quad i \in 1..n \tag{14}$$

Notice that for each constraint in the lines (5)–(13), the left-hand side doesn't occur on the right-hand side. So, we can use the S transformation $(m+4n)$ times on those constraints to produce the following constraint system that we call C':

$$u_0 \leq v_{x_{i1}} \to v_{x_{i2}} \to v_{x_{i3}} \to v_{f_i x_{i1} x_{i2} x_{i3}} \quad i \in 1..n$$

From Lemma 2 we have that each of the $(m+4n)$ applications of S preserves satisfiability so we have:

$$C_{e_0,A} \text{ is satisfiable if and only if } C' \text{ is satisfiable} \tag{15}$$

We can combine (4) and (15) and get that we must prove:

$$\mathcal{H} \text{ is satisfiable if and only if } C' \text{ is satisfiable} \tag{16}$$

In the forwards direction, suppose $\mathcal{H}$ has solution φ. We will use the helper function δ:

$$\delta(1) = T$$
$$\delta(0) = F$$

Define:

$$\psi(v_{x_{ij}}) = \delta(\varphi(x_{ij})) \quad i \in 1..n,\ j \in 1..3$$
$$\psi(v_{f_i x_{i1} x_{i2} x_{i3}}) = T$$

From that $\mathcal{H}$ has solution φ, we have that for each $i \in 1..n$, exactly one of $\varphi(x_{ij}), j \in 1..3$ is 1, while the other two are 0. So for each $i \in 1..n$, exactly one of $\delta(\varphi(x_{ij})), j \in 1..3$ is T, while the other two are F. We conclude that ψ solves C'.

In the backwards direction, suppose C' has solution ψ. We will use δ^{-1} to denote the inverse of δ. Define:

$$\varphi(x_{ij}) = \delta^{-1}(\psi(v_{x_{ij}})), \quad i \in 1..n,\ j \in 1..3$$

From C' and the definition of u_0, we have that for each $i \in 1..n$, exactly one of $\psi(v_{x_{ij}}), j \in 1..3$, is T, while the other two are F. So for each $i \in 1..n$, exactly one of $\delta^{-1}(\psi(v_{x_{ij}})), j \in 1..3$, is 1, while the other two are 0. We conclude that φ solves $\mathcal{H}$. □

7 Putting it All Together

Theorem 5. *Overloading Resolution and Constraint Solving are both NP-complete.*

Proof. From Theorem 3 we have that (1) Monotone One-in-Three 3SAT is NP-complete. From (1) and Theorem 4, we have that (2) Overloading Resolution is NP-hard. From (2) and Theorem 2, we have that (3) Constraint Solving is NP-hard. From Theorem 1 we have that (4) Constraint Solving is in NP. From (3) and (4) we have that (5) Constraint Solving is NP-complete. From (5) and Theorem 2 we have that (6) Overloading Resolution is in NP. From (2) and (6) we have that Overloading Resolution is NP-complete.

8 Interactions of Overloading and other Language Features

While a compiler can resolve overloading for all practical statically typed languages of which we are aware, the complexity of the resolution algorithm varies from language to language. The complexity varies because overloading may be restricted in various ways and because overloading may interact with other language constructs. Let us consider some of the possibilities.

8.1 Type annotations versus type inference

Our example language relies on type inference to assign a type to every bound variable. The essence of our *overloading is NP-complete* theorem is that such type inference must be done by exhaustive search. We have implemented such a search algorithm for a subset of MATLAB and we found that it works well in practice.

In contrast to our λ-calculus, Java requires every formal parameter to be annotated with a type. This changes the complexity of the overloading problem from NP-complete to polynomial time. To see this, let us first take a look at what our reduction from Monotone One-in-Three 3SAT to Overloading might look like if we target Java instead of λ-calculus. For example, we might map the formula:

$$R(x_1, x_2, x_3)$$

to this Java program:

```
class B { }
class T implements B { }
class F implements B { }

public class Test {
  T f(T a, F b, F c) { return new T(); }
  T f(F a, T b, F c) { return new T(); }
  T f(F a, F b, T c) { return new T(); }

  T run() {
    B x1,x2,x3;
    return f(x1,x2,x3);
  }
}
```

The idea of the Java program is as follows. Class `B` is a common superclass of two classes `T` and `F` that represent the Boolean values. In class `Test`, the three versions of the overloaded method `f` mimic the intersection type of f in the λ-calculus program. In the `run` method, we don't know what we need to store in the variables `x1,x2,x3` to satisfy the formula so we declare them to be of type `B`. The expression `f(x1,x2,x3)` is the same kind of call that we used in the λ-term.

The Java program doesn't type check! The problem lies with the type annotation `B` for the variables `x1,x2,x3`. The type annotation prevents the expression `f(x1,x2,x3)` from type checking because none of the declared `f` methods take an argument of type `B`. If only we could omit the annotation `B`, then one could imagine that type inference could assign a type `T` or `F` to each of `x1,x2,x3`. However, Java doesn't support such type inference so this style of reduction to Java doesn't work.

Let us now explain why overloading resolution in Java can be done in polynomial time. We will do an informal proof by induction on the structure of expressions. In the base case, we have expressions such as variables and constants whose nonoverloaded types are known to the compiler from type annotations or the language specification. Java allows overloading only of methods so in the key induction step, we can consider a call of an overloaded method. From the induction hypothesis we have that the compiler knows a nonoverloaded type for each argument expression. Recall the quote from Section 1 that says that Java resolves overloading by using the number and compile-time types of the arguments. Thus the compiler can now either declare the call type correct or give a type error. This concludes the informal proof for Java.

Notice that the proof relies on type annotations for variables, which is exactly what ruined the Java example above.

8.2 Overloading and subtyping

Java has all of overloading, type annotations, and subtyping. Java's notion of subtyping presents no new problems for overloading resolution. The reason is that Java's type system enables the compiler to know a static type of every expression, even in the presence of subtyping. Thus, the informal induction proof in the previous subsection works for Java in this case too. We conclude that also in the presence of subtyping can we resolve overloading in Java in polynomial time.

Castagna, Ghelli, and Longo [4] studied a typed λ-calculus in which the programmer can define overloaded functions. This goes beyond the λ-calculus in Section 2 where all overloaded functions all come from the initial environment. As far as we know, the problem of devising a type checker or a type inference algorithm for their calculus remains an open problem.

8.3 Overloading and Hindley-Milner polymorphism

Let us consider the notion of polymorphism known as Hindley-Milner polymorphism that can be found in such languages as ML and Haskell. If we combine overloading with Hindley-Milner polymorphism, the result is an undecidable type system [15, 20, 16]. The undecidability result has led researchers to look for restrictions of overloading. The idea is that Hindley-Milner polymorphism together with restricted overloading may be decidable. Volpano showed that if we in a system with Hindley-Milner polymorphism make a Haskell-style restriction on overloading, the resulting type system is NP-hard [18, 19]. Camarao, Figueiredo, and Vasconcellos studied another restriction on overloading and reported on experiments with a type checking algorithm [2].

9 Conclusion

We have given a detailed proof of why overloading resolution is NP-complete for a typed λ-calculus. We hope that the proof techniques will be helpful to researchers who want to prove similar results for other languages. We also hope that our paper can help clarify which overloading problems are NP-complete and which problems have higher complexity due to interactions with other language features.

Exercise 1: Design a variant of our calculus that captures the essence of overloading in Java. Prove that type checking can be done in polynomial time.

Exercise 2: Consider a variant of our calculus in which every bound variable is annotated with a simple type, using the notation $\lambda x : t.e$. What is the complexity of type checking?

Exercise 3: Consider a variant of our calculus in which we disallow $\lambda x.e$. What is the complexity of type checking?

Acknowledgments. We thank Matt Brown, Jakob Rehof, and Alexander Sherstov for helpful comments and discussions.

References

1. Alfred V. Aho, Ravi I. Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.
2. Carlos Camarao and Lucilia Figueiredo, and Cristiano Vasconcellos. Constraint-set satisfiability for overloading. In *Proceedings of PPDP*, 2004.
3. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
4. Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995.
5. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and lambda-calculus semantics. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 535–560. Academic Press, 1980.
6. Michael R. Garey and David S. Johnson. *Computers and Intractability.* Freeman, 1979.
7. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification.* Third edition, 2005. http://java.sun.com/docs/books/jls/third_edition/html/classes.html#8.4.9.
8. Desmond J. Higham and Nicholas J. Higham. *MATLAB Guide.* SIAM, second edition, 2005.
9. J. Roger Hindley. Types with intersection: An introduction. *Formal Aspects of Computing*, 4:470–486, 1991.
10. Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994. Preliminary version in Proceedings of FOCS'92.
11. Deepak A. Mathaikutty and Sandeep K. Shukla. *Metamodeling-Driven IP Reuse for SoC Integration and Microprocessor Design.* Artech House, 2009.
12. MathWorks. Product documentation, matlab compiler. http://www.mathworks.com/help/techdoc/matlab_prog/f2-38133.html, 2011.
13. M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.
14. Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC'95)*, pages 216–226, 1978.
15. Geoffrey Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping.* PhD thesis, Cornell University, 1991.
16. Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.
17. Manish Vachharajani, Neil Vachharajani, Sharad Malik, and David I. August. Facilitating reuse in hardware models with enhanced type inference. In *Proceedings of IEEE/ACM/IFIP Second International Conference on Hardware/Software Codesign and System Synthesis*, 1994.
18. Dennis Volpano. Haskell-style overloading is NP-hard. In *Proceedings of ICCL'94, Fifth IEEE International Conference on Computer Languages*, pages 88–95, May 1994.
19. Dennis Volpano. Lower bounds on type checking overloading. *Information Processing Letters*, 57:9–14, 1996.
20. Dennis Volpano and Geoffrey Smith. On the complexity of ML typability with overloading. In *Proceedings of FPCA'93, Sixth ACM Conference on Functional Programming Languages and Computer Architecture*, pages 15–28, 1991.