En implementation av Featherweight Javas typregler

FABIAN CARLSTRÖM

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2010

En implementation av Featherweight Javas typregler

F A B I A N C A R L S T R Ö M

Examensarbete i datalogi om 15 högskolepoäng
vid Programmet för datateknik
Kungliga Tekniska Högskolan år 2010
Handledare på CSC var Mads Dam
Examinator var Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/carlstrom_fabian_K10064.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Sammanfattning

Den här rapporten presenterar Featherweight Javas syntax och typregler. Featherweight Java är en liten modell av programmeringsspråket Java som har skapats för att detaljstudera vissa egenskaper hos Java. Efter presentationen av språket beskrivs en implementation av Featherweight Javas typregler som författaren har gjort. Implementationen demonstreras med ett exempel. Slutsatsen är att typreglerna är relativt rättframma att implementera även om vissa egenheter kvarstår i den implementation som har gjorts.

Abstract

Featherweight Java is a minimal language designed to study some aspects of the Java programming language in detail. The report begins with a presentation of Featherweight Java's syntax and typing rules. It then proceeds to a discussion of an implementation that the author has made of Featherweight Java's typing rules. The implementation is demonstrated with an example. It is concluded that the process of implementing the typing rules is rather straightforward, though some imperfections remain in the implementation that was made for the purpose of this report.

Innehåll

1 Introduktion

Syftet med denna rapport är att presentera en implementation som berör delar av språket *Featherweight Java*. För att läsaren ska kunna tillgodogöra sig informationen ges först en presentation av de relevanta delarna av språket: syntaxen och typreglerna. Det motiveras varför det här är ett intressant ämne att studera. Läsaren förväntas ha kännedom om Java och den terminologi som används inom språket. I rapporten kommer den svenska Javaterminologi som föreslås i [2] att användas.

Presentationen av Featherweight Java som görs här kommer inte att täcka en tillräckligt stor del av språket för att kunna redogöra för beviset av typsundhet (som spelade en viktig roll i framtagningen av språket). Istället går rapporten noggrant igenom de delar av språket som implementationen berör. Den intresserade läsaren bör därefter kunna tillgodogöra sig materialet i [1] där språket från början presenterades. Rapporten kommer alltså att ägnas åt att presentera *hur* syntaxen och typreglerna fungerar medan svaret på frågan *varför* de ser ut som de gör utelämnas. Den skeptiske läsaren kan dock hitta den fullständiga motiveringen i referensmaterialet.

2 Bakgrund

Språket Featherweight Java (hädanefter refererat till som *FJ*) skapades för att ha en liten modell av Java att studera. Skaparna menar att en liten modell är en bra utgångspunkt för att t.ex. studera vissa egenskaper hos ett språk i detalj. FJ utformades med huvudsyftet att göra beviset för typsundhet så koncist som möjligt, samtidigt som de väsentliga delarna av motsvarande argument för det fullständiga Javaspråket bibehölls [1]. Typsundhet innebär i sammanhanget att ett vältypat program – ett program som godkänns enligt de givna typreglerna – inte fastnar under körningen (med vissa undantag i FJs fall). Genom att låta ett givet FJ-program genomgå en typkontroll ska vi alltså kunna garantera att programmet inte fastnar.

Det visar sig att FJ även är en bra utgångspunkt för att studera utvidgningar av Java. Upphovsmännen till FJ demonstrerar detta genom att utvidga språket till något som de kallar för *Featherweight Generic Java*, tänkt att modellera *Generic Java*[1] (förkortat GJ). Trots att modellen utelämnar viktiga aspekter av GJ ledde den till att en bugg i GJs kompilator upptäcktes och reparerades [1].

Att studera en liten modell har även pedagogiska poänger. Det är lättare att ta vara på idéerna bakom designen av ett mindre språk. Det fullständiga Javaspråket är större och det blir då oundvikligen mer information som måste läsas in och tas hänsyn till.

3 Featherweight Java

FJ är, som namnet antyder, ett litet språk. T.ex. har språket endast fem typer av uttryck: klassinstansiering (med nyckelordet `new`), metodanrop, fältåtkomst, typkonverte-

[1] Tidigare en utvidgning av Java som la till stöd för så kallad *generisk programmering*, men ingår sedan version 5.0 i Java.

ring (eng. *cast*) och variabler. Exempel på saker som saknas i FJ är tilldelning, gränssnitt (eng. *interface*), överlagring, primitiva typer (`int`, `float`, etc.), `null`-pekare, abstrakta metoddeklarationer, överskuggning av superklassens fält i subklassen (men överskuggning av metoder är tillåtet), åtkomstmodifierare (`public`, `private`, etc.) och särfall (eng. *exceptions*).

Att FJ saknar tilldelning gör modellen mer lätthanterlig. Språket har inga sidoeffekter över huvud taget. Fälten i ett objekt initialiseras i konstruktorn och därefter ändras de aldrig. Skaparna av FJ menar att de flesta av svårigheterna kring typning i Java är oberoende av tilldelning [1] och denna modell är därför fortfarande intressant att studera.

Låt oss börja presentationen av FJ med ett exempel på några klassdeklarationer och en diskussion kring några aspekter hos språket utifrån dem. Därefter presenteras den formella syntaxen och språkets typregler.

3.1 Översiktsbeskrivning

Ett program i FJ består av en samling klassdeklarationer samt ett uttryck som ska evalueras. Implementationen berör endast klassdeklarationerna och det är därför dem vi kommer att intressera oss för. Se figur 1 för några exempel på klassdeklarationer i FJ. Läsaren som känner till Java upplever det nog som bekant.

```
class Trivial extends Object {
    Trivial() { super(); }
}
class Par extends Object {
    Object x;
    Object y;
    Par(Object x, Object y) {
        super();
        this.x=x;
        this.y=y;
    }
    Par bytx(Object nyttx) {
        return new Par(nyttx, this.y);
    }
}
class NyttPar extends Par {
    Object z;
    NyttPar(Object x, Object y, Object z) {
        super(x,y);
        this.z=z;
    }
}
```

Figur 1: Exempel på några klassdeklarationer i FJ.

FJ skiljer sig från Java bl.a. genom att ha en striktare syntax. En klassdeklaration (d.v.s. `class C extends D {...}`) innehåller alltid superklassen, även om den är `Object`. Vidare skrivs konstruktorn alltid ut, även om den är trivial som i klassen `Trivial` i figur 1. Mottagaren för en fältåtkomst skrivs alltid ut, även när mottagaren är `this`, se t.ex. uttrycket i metoden `bytx` i det givna exemplet.

Konstruktorn ser alltid likadan ut. Den tar en parameter för varje fält i klassen och parametern måste ha samma namn som motsvarande fält. `super` anropas alltid, med de parametrar som motsvarar fälten som har ärvts från superklassen. De fält som inte tillhör superklassen initieras därefter i samma ordning som de deklarerades. `Object` behandlas som en särskild klass som saknar fält och metoder, därför anropas `super` utan parametrar i klasser som ärver från `Object`. Konstruktorn är den enda platsen i ett FJ-program där `super` eller = förekommer.

Efter konstruktorn kommer metoderna. Metodhuvudet antar den vanliga formen och består av returtyp, metodens namn samt argumenten till metoden. Metodkroppen, däremot, består alltid av `return` följt av ett uttryck, eftersom FJ inte har några sidoeffekter. Det finns som tidigare nämnt endast fem typer av uttryck i FJ: klassinstansiering, metodanrop, fältåtkomst, variabler och typkonvertering.

Ett exempel på en variabel är `nyttx` i metoden `bytx` i klassen `Pair` i exemplet i figur 1. I samma metod är `this.y` ett exempel på ett uttryck som är en fältåtkomst. För fler exempel, låt oss betrakta uttrycket

```
new Par(new Object(), new Trivial()).x.
```

Här är `new Trivial()` ett exempel på en klassinstansiering, medan *[uttryck]*`.x` är ytterliggare ett exempel på en fältåtkomst. För exempel på resterande typer av uttryck, d.v.s. metodanrop och typkonvertering, betrakta

```
new Par(new Object(), new Object()).bytx((Object) new Trivial()).
```

I det här uttrycket är *[uttryck]*`.bytx(`*[annat uttryck]*`)` ett exempel på ett metodanrop och `(Object) new Trivial()` är ett exempel på en typkonvertering.

3.2 Syntax

Den formella syntaxen presenteras i figur 2. Bokstäverna som förekommer i figuren används på följande vis: `B`, `C`, `D` och `E` betecknar klassnamn, `f` och `g` fältnamn, `m` metodnamn, `x` variabler, `d` och `e` uttryck, `L` klassdeklarationer, `K` konstruktordeklarationer och `M` metoddeklarationer.

När bokstaven skrivs med ett streck ovanför innebär det en godtyckligt lång följd (den kan vara tom) av den typen av uttryck, t.ex. $\overline{\texttt{M}}$ är följden $\texttt{M}_1,\ldots,\texttt{M}_n$ av metoddeklarationer. En klassdeklaration `L` innehåller alltså noll eller fler metoddeklarationer efter konstruktorn `K`. Notationen generaliseras till par på det vis att $\overline{\texttt{C}}\ \overline{\texttt{f}}$ innebär $\texttt{C}_1\ \texttt{f}_1,\ldots,\texttt{C}_n\ \texttt{f}_n$ medan $\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}$ (notera semikolonet) innebär $\texttt{C}_1\ \texttt{f}_1\texttt{;}\ldots\texttt{;}\texttt{C}_n\ \texttt{f}_n\texttt{;}$ och $\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}$ är det förkortade skrivsättet för $\texttt{this.f}_1\texttt{=f}_1\texttt{;}\ldots\texttt{;this.f}_n\texttt{=f}_n\texttt{;}$. Följderna antas sakna upprepningar av namn.

Syntax:

```
L ::= class C extends C { C̄ f̄; K M̄ }
K ::= C(C̄ f̄) { super(f̄); this.f̄=f̄; }
M ::= C m(C̄ x̄) { return e; }
e ::= x | e.f | e.m(ē) | new C(ē) | (C)e
```

Subtypning:

$$\texttt{C} <: \texttt{C} \qquad \frac{\texttt{C} <: \texttt{D} \qquad \texttt{D} <: \texttt{E}}{\texttt{C} <: \texttt{E}} \qquad \frac{\texttt{class C extends D \{...\}}}{\texttt{C} <: \texttt{D}}$$

Uppslagning av fält:

$$fields(\texttt{Object}) = \bullet$$

$$\frac{\texttt{class C extends D \{ } \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\texttt{ \}} \qquad fields(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}}}{fields(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}}}$$

Uppslagning av metodtyp:

$$\frac{\texttt{class C extends D \{ } \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\texttt{ \}} \qquad \texttt{B m(}\overline{\texttt{B}}\ \overline{\texttt{x}}\texttt{) \{ return e; \}} \in \overline{\texttt{M}}}{mtype(\texttt{m}, \texttt{C}) = \overline{\texttt{B}} \rightarrow \texttt{B}}$$

$$\frac{\texttt{class C extends D \{ } \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\texttt{ \}} \qquad \texttt{m} \notin \overline{\texttt{M}}}{mtype(\texttt{m}, \texttt{C}) = mtype(\texttt{m}, \texttt{D})}$$

Figur 2: Syntax, subtypning och hjälpfunktioner.

Att `C` är en subtyp till `D` skrivs som `C <: D`. Hur `<:` är definierad står beskrivet under rubriken subtypning i figur 2. Notationen

$$\frac{\texttt{C} <: \texttt{D} \qquad \texttt{D} <: \texttt{E}}{\texttt{C} <: \texttt{E}}$$

utläses som ”om `C <: D` och `D <: E`, så `C <: E`”. Skrivsättet förekommer på samma sätt i resterande figurer: det som står under strecket följer av det som står ovanför. Lägg märke till att `<:` är reflexiv och transitiv. Det antas även att `<:` är antisymmetrisk, d.v.s. att det inte finns några cykler i arven mellan klasserna.

För uppslagning av fält används funktionen *fields* som definieras i figur 2. Funktionen tar en klass och returnerar en följd med klassens fält, inklusive de fält som har ärvts från superklassen. Notera ordningen som fälten returneras i, superklassens fält kommer först. Symbolen $\bullet$ betecknar den tomma följden (`Object` har inga fält).

Vidare används funktionen *mtype*(`m`,`C`) för att slå upp typen av metod `m` i klass `C`. Metoden returnerar ett par $\overline{\texttt{B}} \rightarrow \texttt{B}$ där $\overline{\texttt{B}}$ är argumentens typer och `B` är returtypen. Om metoden med namn `m` återfinns bland metoderna i klass `C` returneras typerna som de är

deklarerade där. Om metod `m` inte finns deklarerad bland metoderna i klass `C`, $\texttt{m} \notin \overline{\texttt{M}}$, anropar funktionen sig själv med superklassen till `C` som det andra argumentet. Från detta syns det att *mtype* är definierad på ett sätt som möjliggör överskuggning av metoder. Typreglerna kommer att visa att överskuggning endast är möjligt då metoden i subklassen har samma typ ($\overline{\texttt{B}} \rightarrow \texttt{B}$) som metoden i superklassen. Eftersom `Object` saknar metoder är *mtype*(`m`,`Object`) inte definierad.

3.3 Typning

FJs typregler presenteras i figur 3. En miljö Γ är en ändlig avbildning från variabler till typer och skrivs som $\overline{\texttt{x}}\texttt{:}\overline{\texttt{C}}$. Miljön kan t.ex. ges av argumenten till en metod (tillsammans med den särskilda variabeln `this`, se T-Method). Beteckningen $\Gamma \vdash \texttt{e} : \texttt{C}$ ska läsas som "i miljön Γ har uttrycket `e` typen `C`". På samma sätt som ovan används förkortningen $\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{C}}$ för följden $\Gamma \vdash \texttt{e}_1 : \texttt{C}_1, \ldots, \Gamma \vdash \texttt{e}_n : \texttt{C}_n$. Vidare är $\overline{\texttt{C}} <: \overline{\texttt{D}}$ det förkortade skrivsättet för $\texttt{C}_1 <: \texttt{D}_1, \ldots, \texttt{C}_n <: \texttt{D}_n$. Lägg märke till hur det finns en typregel för varje typ av uttryck, med undantaget att det finns tre typregler om uttrycket är en typkonvertering.

Låt oss först betrakta T-Var. Den säger att i miljön Γ har variabeln `x` den typ som variabeln `x` har i miljön. Titta till exempel på metoden `bytx` i klassen `Par` i figur 1. Uttrycket i metodkroppen evalueras i miljön med variablerna `nyttx` (från argumenten till metoden) och `this` (som implicit följer med enligt regeln T-Method som vi strax kommer till) med typerna `Object` respektive `Par`. I den miljön kommer alltså variabeln `nyttx` ha typen `Object`.

T-Field säger att om det i miljön Γ finns ett uttryck $\texttt{e}_0$ av typ $\texttt{C}_0$ och $\texttt{C}_0$ har fälten $\overline{\texttt{C}}\ \overline{\texttt{f}}$ så har fältåtkomsten $\texttt{e}_0\texttt{.f}_i$ typen $\texttt{C}_i$. Om vi återigen ser till exemplet i figur 1 så har fältåtkomsten `this.y` i metoden `bytx` typen `Object`.

Regeln T-Invk tillämpas på metodanrop så låt oss betrakta uttrycket $\texttt{e}_0\texttt{.m(}\overline{\texttt{e}}\texttt{)}$. Om uttrycket $\texttt{e}_0$ har typen $\texttt{C}_0$ i miljön Γ, typen av metod `m` i klass $\texttt{C}_0$ är $\overline{\texttt{D}} \rightarrow \texttt{C}$ (d.v.s. argumenten har typ $\overline{\texttt{D}}$ och metoden har returtyp `C`), uttrycken $\overline{\texttt{e}}$ har typ $\overline{\texttt{C}}$ i miljön Γ och $\overline{\texttt{C}} <: \overline{\texttt{D}}$ så har uttrycket $\texttt{e}_0\texttt{.m(}\overline{\texttt{e}}\texttt{)}$ typ `C` i miljön Γ. Resterande typregler följer på samma sätt som ovan.

En regel som sticker ut från mängden är T-SCast (S för *stupid*, dum). Den har introducerats för att typsundhetsbeviset ska vara korrekt. Beviset kommer inte att tas upp i den här rapporten och diskussionen om denna regel utelämnas därför. Anledningen till att det står *dum varning* i regeln är för att den antas gälla med just det särskilda antagandet *dum varning*. Javas kompilator förkastar uttryck som skulle godkännas enligt denna regel.

För typning av metoder, se T-Method. Miljön ges av argumenten till metoden tillsammans med variabeln `this` (alltså är det inte tillåtet att döpa ett argument till en metod till `this`). Regeln säger att *om* metoden överskuggar en metod i någon superklass måste båda metoderna ha samma typ på argumenten och samma returtyp. Det är inte tillåtet för en metod med typerna $\overline{\texttt{C}} \rightarrow \texttt{C}$ att överskugga en metod med typerna $\overline{\texttt{D}} \rightarrow \texttt{D}$ (där $\overline{\texttt{C}} \neq \overline{\texttt{D}}$ och/eller $\texttt{C} \neq \texttt{D}$). Om metoden uppfyller kraven ges utlåtandet `M OK IN C`, om `M` är den metoddeklaration som förekommer i klassen `C`.

Typning av uttryck:

$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}_\mathtt{i} \ : \ \mathtt{C}_\mathtt{i}} \qquad \text{(T-Field)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \ : \ \mathtt{C}} \qquad \text{(T-Invk)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) \ : \ \mathtt{C}} \qquad \text{(T-New)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \ : \ \mathtt{C}} \qquad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{D} \qquad \mathtt{C} <: \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \ : \ \mathtt{C}} \qquad \text{(T-DCast)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \qquad \textit{dum varning}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \ : \ \mathtt{C}} \qquad \text{(T-SCast)}$$

Typning av metoder:

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \\ \mathtt{class\ C\ extends\ D\ \{...\}} \\ \text{om } \mathit{mtype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{D}} \to \mathtt{D}_0, \text{ så är } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ och } \mathtt{C}_0 = \mathtt{D}_0\end{array}}{\mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \mathtt{\{\ return\ e_0;\ \}\ OK\ I\ C}} \qquad \text{(T-Method)}$$

Typning av klasser:

$$\frac{\mathtt{K\ =\ C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{C}}\ \overline{\mathtt{f}})\ \mathtt{\{\ super}(\overline{\mathtt{g}})\mathtt{;\ this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;\ \}} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ I\ C}}{\mathtt{class\ C\ extends\ D\ \{}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\ \mathtt{\}\ OK}} \qquad \text{(T-Class)}$$

Figur 3: Typregler.

T-Class anger när en klass är korrekt typad. Det är denna regel som bestämmer klassens form. Konstruktorn måste ha argumenten i rätt ordning (argumenten som tillhör superklassens fält kommer först och därefter kommer argumenten som tillhör klassen

som deklareras) och `super` ska anropas med de argument vars fält tillhör superklassen. Därefter måste resterande fält initieras. Till sist måste alla metoder i klassen godkännas enligt T-Method.

4 Implementation

Implementationen av typreglerna har genomförts i programmeringsspråket Haskell. Programmet tar en fil som argument, analyserar den efter FJs givna grammatik och meddelar resultatet av typkontrollen. Om allt går bra och klassdeklarationerna i den givna filen godkänns säger programmet `[den givna filen] passed the type check.` Om klassdeklarationerna inte godkänns ges istället ett felmeddelande som med varierande grad av precision delger vad det är för fel som har uppstått. Detta kommer att exemplifieras nedan.

För att analysera den givna filen används biblioteket Parsec[2]. FJs grammatik har försökts översättas på bästa sätt, men vissa egenheter har uppstått. Parsningen av uttryck är gjord på ett sätt som gör att `m(`$\overline{\texttt{e}}$`)`, ensamt förekommande, ses som ett uttryck. Det kommer dock senare att förkastas av typkontrollen. Ytterliggare en brist i implementationen är att den inte kollar efter cirkulära klassdeklarationer. Huvudsyftet med att implementera typreglerna var att öka förståelsen och fokus har därför inte legat på att släta ut alla små detaljer. Effektivitet har inte heller eftersträvats, implementationen är naiv och ingen hänsyn till tidskomplexitet har tagits. Datastrukturerna har gjorts så enkla som möjligt.

Implementationen följer inte alltid den teoretiska beskrivningen av FJ, men där avvikelser görs har målet alltid varit att avvikelsen ska ge samma resultat. Ett exempel på detta är att innan alla klasser som har lästs in skickas till de typkontrollerande funktionerna läggs klassen `Object` (som saknar fält och metoder) till. Detta är för att minska antalet specialfall i funktionerna. Ett annat exempel är att det egentligen är typreglerna som gör att `this` inte är tillåtet som variabelnamn, men i implementationen behandlas `this` (och `Object`) redan i parsningen som nyckelord, och därför kommer klassdeklarationer där dessa ord förekommer på otillåtna platser (t.ex. som klassnamn eller argument i en metod) förkastas.

För att kunna använda implementationen behövs *Glasgow Haskell Compiler* (GHC)[3]. En typkontroll av en given fil med klassdeklarationer genomförs sedan om man i katalogen där källkoden finns kör kommandot

```
runhaskell FJ.hs exempel.java,
```

om `exempel.java` är sökvägen till filen som ska kontrolleras. Om klassdeklarationerna i `exempel.java` godkänns meddelar programmet, som beskrivet ovan,

```
exempel.java passed the type check.
```

[2] http://legacy.cs.uu.nl/daan/parsec.html
[3] http://www.haskell.org/ghc/

4.1 Exempel

Se figur 4 för ett exempel på några klassdeklarationer som godkänns av typkontrollen.

```
class X extends Object {
    Y a;
    X(Y a) {
        super();
        this.a = a;
    }
    Y exempelmetod() {
        return new Y(this.a, new Object());
    }
}
class Y extends X {
    Object b;
    Y(Y a, Object b) {
        super(a);
        this.b = b;
    }
    Y exempelmetod() {
        return new X(new Y((Y) new Object(), new Object())).a;
    }
}
```

Figur 4: Exempel på klassdeklarationer som godkänns av typkontrollen.

Här är ett exempel på att det är tillåtet för en superklass att ha fält som beror på en subklass (notera fältet `Y a` i klassen `X`). Det förekommer även exempel på överskuggning, se metoden `exempelmetod` i klass `Y`. I uttrycket i `exempelmetod` i klass `Y` förekommer det en typkonvertering av uttrycket `new Object()` till typen `Y`. Enligt T-DCast har uttrycket `(Y) new Object()` typen `Y` och det är därför godkänt som första parametern till `new Y(...)`. Om uttrycket evalueras kommer dock ett fel att uppstå. Det här är ett av undantagen för FJs typsundhetsbevis: ett program kan fastna vid en misslyckad typkonvertering neråt, även om det har godkänts av typkontrollen [1]. Om typkonverteringen tas bort blir resultatet av typkontrollen:

```
Type error in method exempelmetod. T-New failed.
```

Felmeddelandet lämnar en del att önska men ger ändå en fingervisning om var felet har uppstått. För att ge fler exempel på felmeddelanden, låt oss ändra på konstruktorn i klass `Y` och anropa `super` utan parametrar. Resultatet av typkontrollen blir då:

```
The parameters to super in class Y are incorrect. T-Class failed.
```

Till sist, om returtypen för `exempelmetod` i klass `Y` ändras från `Y` till `X`, blir resultatet av typkontrollen:

```
Method exempelmetod in class Y does not match the types of its namesake
                in some superclass. T-Method failed.
```

5 Diskussion

Det har varit intressant att se hur en så förenklad modell av ett språk kan komma till praktisk nytta. Istället för att studera det fullständiga språket med allt som hör därtill är det alltså möjligt att, i vissa sammanhang, studera en grov förenkling men ändå komma fram till relevanta resultat. Den här rapporten behandlar förvisso inte några sådana resultat, de praktiska följderna är något som har framgått vid studier av referensmaterialet. Att genomföra implementationen har dock bidragit till en ökad förståelse och en bättre känsla för vad de olika reglerna innebär samt hur Featherweight Java är uppbyggt. Således har min uppskattning för vad språket kan åstadkomma ökat. Jag hoppas att läsaren i sin tur använder implementationen och därigenom får en ökad förståelse för FJ.

Något som blev särskilt tydligt under implementeringen var hur typreglerna för de olika sorternas uttryck var utformade och användes. Detta upptäcktes efter det att datastrukturen för uttryck hade valts. Det var därefter väldigt rättframt att implementera de olika typreglerna. Det svåra med uttrycken var att implementera parsningen. Som beskrivet i avsnitt 4 kvarstår vissa egenheter här, men de berör inte typreglerna och har därför ansetts mindre viktiga i sammanhanget. Huvudsyftet med arbetet var att implementera typreglerna. Förhoppningsvis har läsaren blivit inspirerad att utforska FJ vidare.

Referenser

[1] Igarashi, Pierce, Wadler. *Featherweight Java: A Minimal Core Calculus for Java and GJ.* ACM Transactions on Programming Languages and Systems 2001.

[2] Nilsson. *Svensk Javaterminologi, version 1.1.* http://www.csc.kth.se/~snilsson/publications/Javaterminologi/. 2010-04-30.

Bilagor

A Källkod

Källkoden som hittas nedan samt det exempel som presenterades i avsnitt 4.1 återfinns på http://www.csc.kth.se/~fabianc/FJ/. Eventuellt har koden på hemsidan uppdaterats sen dess att detta dokument publicerades.

A.1 FJ.hs

```
1  -- An attempt to implement the typing rules of Featherweight Java as
       specified in
2  --
3  -- [1] Igarashi, Pierce, Wadler. Featherweight Java: A Minimal Core
       Calculus for Java and GJ. ACM Transactions on Programming Languages and
        Systems 2001.
4  --     http://www.cis.upenn.edu/~bcpierce/papers/fj-toplas.pdf
5
6  module FJ (tcheckClassTable, runTypeCheck) where
7
8  -- {{{ Import statements
9
10 import System (getArgs)
11 import Data.List
12 import Data.Maybe
13 import Text.ParserCombinators.Parsec
14 import Text.ParserCombinators.Parsec.Language
15 import qualified Text.ParserCombinators.Parsec.Token as P
16 import qualified Text.ParserCombinators.Parsec.Expr as E
17
18 -- }}}
19
20 -- {{{ Data structures and types
21
22 data Class =
23     Class { cname       :: Name
24           , csuperClass :: Name
25           , cfields     :: [Field]
26           , cconstructor :: Constructor
27           , cmethods    :: [Method]
28           }
29     deriving (Eq, Show)
30
31 data Field =
32     Field { ftype :: Type
33           , fname :: Name
34           }
35     deriving (Eq, Show)
36
37 data Method =
```

```
38      Method { mtype :: Type -- Not to be confused with mtype in [1].
39             , mname :: Name
40             , margs :: [Argument]
41             , mexpr :: Expr
42             }
43      deriving (Eq, Show)
44
45  data Constructor =
46      Constructor { kname          :: Name
47                  , kargs          :: [Argument]
48                  , ksuperParam    :: [Parameter]
49                  , kassignments   :: [Assignment]
50                  }
51      deriving (Eq, Show)
52
53  data Expr =
54      ExprVar    { ename  :: Name }
55    | ExprField  { eexpr  :: Expr
56                 , ename  :: Name
57                 }
58    | ExprMethod { eexpr  :: Expr
59                 , ename  :: Name
60                 , eexprs :: [Expr]
61                 }
62    | ExprNew    { etype  :: Type
63                 , eexprs :: [Expr]
64                 }
65    | ExprCast   { etype  :: Type
66                 , eexpr  :: Expr
67                 }
68      deriving (Eq, Show)
69
70  type Name         = String
71  type Type         = Name
72  type Argument     = Field
73  type Parameter    = Name
74  type Assignment   = (Name, Name)
75  type ClassTable   = [Class]
76
77  -- }}}
78
79  -- {{{ Auxiliary functions
80
81  -- Checks if a class with the given name exist in the given class table.
82  classExists :: ClassTable -> Name -> Bool
83  classExists ct clname = any (\c -> clname == cname c) ct
84
85  -- Find and return the superclass of a given class.
86  superClassOf :: ClassTable -> Class -> Class
87  superClassOf ct c
88      | superClass == Nothing = error "The superclass does not exist in the
            given class table."
89      | otherwise             = fromJust superClass
90      where
```

```
91          superClass = find (\x -> cname x == csuperClass c) ct
92
93  -- Check if c1 <: c2 in the given class table.
94  isSubClass :: ClassTable -> Class -> Class -> Bool
95  isSubClass ct c1 c2
96      | c1 == c2              = True
97      -- If c1 is Object the alternatives have been exhausted and c1 </: c2.
98      | cname c1 == "Object" = False
99      | otherwise             = isSubClass ct (superClassOf ct c1) c2
100
101 -- Find and return the class with the given name (if it exists in the
        class table).
102 getClass :: ClassTable -> Name -> Class
103 getClass ct cn
104     | c /= Nothing = fromJust c
105     | otherwise    = error $ "No class with name " ++ cn ++ " exists in the
            given class table."
106     where
107         c = find (\x -> cname x == cn) ct
108
109 -- Field lookup
110 -- Returns a list of all the fields in the given class and its superclasses
        .
111 fields :: ClassTable -> Class -> [Field]
112 fields ct c
113     | cname c == "Object" = [] -- As specified in [1], Object does not
            contain any fields.
114     -- Note the importance of the order of the fields returned below.
115     | otherwise           = fields ct (superClassOf ct c) ++ cfields c
116
117 -- Method type lookup
118 -- mtype(m,C) as found in [1].
119 methtype :: ClassTable -> Class -> Name -> ([Type], Type)
120 methtype = mcombine mtype ftype
121
122 -- Method body lookup
123 -- mbody(m,C) as found in [1].
124 -- Not actually used for the typing rules though.
125 mbody :: ClassTable -> Class -> Name -> ([Name], Expr)
126 mbody = mcombine mexpr fname
127
128 -- Since methtype and mbody are so similar, mcombine was written to combine
        them.
129 mcombine :: (Method -> a) -> (Argument -> b) -> ClassTable -> Class -> Name
        -> ([b], a)
130 mcombine mfunc afunc ct c m
131     -- If this point is ever reached something is wrong with the given code
            .
132     | cname c == "Object" = error "Object does not contain any methods."
133     -- If the method name given is in the class return what is needed from
            there,
134     -- otherwise search for the method in the superclass.
135     | method /= Nothing   = (map afunc (margs theMethod), mfunc theMethod)
136     | otherwise           = mcombine mfunc afunc ct (superClassOf ct c) m
```

```
137     where
138         -- Try to find the method with the given name in the class.
139         method    = find (\x -> m == mname x) (cmethods c)
140         -- If it was found it is given by theMethod.
141         theMethod = fromJust method
142
143 -- }}}
144
145 -- {{{ Some things needed from Parsec.Token
146
147 lexer :: P.TokenParser ()
148 lexer = P.makeTokenParser fjDef
149
150 whiteSpace = P.whiteSpace lexer
151 lexeme     = P.lexeme lexer
152 symbol     = P.symbol lexer
153 parens     = P.parens lexer
154 semi       = P.semi lexer
155 identifier = P.identifier lexer
156 reserved   = P.reserved lexer
157 braces     = P.braces lexer
158 reservedOp = P.reservedOp lexer
159
160 -- }}}
161
162 -- {{{ Parser related
163
164 -- Use Parsec.Language:s LanguageDef
165 fjDef :: LanguageDef st
166 fjDef = LanguageDef
167         { commentStart = "/*" -- While comments are not actually part of
                Featherweight Java,
168         , commentEnd   = "*/" -- when Parsec offers the convenience, why
                not use it?
169         , commentLine  = "//"
170         , nestedComments = False
171         , identStart   = letter
172         , identLetter  = alphaNum <|> char '_'
173         , opStart      = oneOf "."
174         , opLetter     = oneOf "."
175         , reservedNames = ["class", "extends", "super", "return"
176                           , "new", "Object", "this"
177                           ]
178         , reservedOpNames = ["."]
179         , caseSensitive = True
180         }
181
182 -- Class head:
183 -- class C extends D { <class body> }
184 classHeadParser =
185     do { reserved "class"
186        ; className <- identifier -- className may not be Object.
187        ; reserved "extends"
188        ; classSuperClass <- typeParser -- But the superclass may be Object.
```

```
189         ; (classFields, classConstructor, classMethods) <- braces
                classBodyParser
190         ; return $ Class className classSuperClass classFields
                classConstructor classMethods
191         }
192     <?> "class head"
193
194 -- Class body:
195 -- C* f*; <constructor> M*
196 -- (Note that * denotes zero or more occurrences.)
197 classBodyParser =
198     do { classFields <- many (try fieldParser) <|> return []
199         ; classConstructor <- constructorParser
200         ; classMethods <- many methodParser
201         ; return (classFields, classConstructor, classMethods)
202         }
203     <?> "class body"
204
205 -- Field:
206 -- C f;
207 fieldParser =
208     do { fieldType <- typeParser
209         ; fieldName <- identifier
210         ; semi
211         ; return $ Field fieldType fieldName
212         }
213     <?> "field"
214
215 -- Type:
216 -- C
217 -- May be either Object or some identifier.
218 typeParser = identifier
219           <|> do { reserved "Object"; return "Object" }
220           <?> "type"
221
222 -- Constructor:
223 -- C(D* g*, C* f*) { <body> }
224 constructorParser =
225     do { constructorName <- identifier
226         ; constructorArgs <- parens argumentParser
227         ; (superParams, constructorAssignments) <- braces
                constructorBodyParser
228         ; return $ Constructor constructorName constructorArgs superParams
                constructorAssignments
229         }
230     <?> "constructor"
231
232 -- Constructor body:
233 -- super(g*);
234 -- this.f* = f*;
235 constructorBodyParser =
236     do { reserved "super"
237         ; superParams <- parens $ sepBy identifier $ symbol ","
238         ; semi
```

```
239         ; constructorAssignments <- many assignmentParser
240         ; return (superParams, constructorAssignments)
241         }
242
243 -- Assignment:
244 -- this.f* = f*;
245 --
246 -- It would be possible to check if classField == constructorArgument here,
247 -- but it is instead done later for consistency.
248 assignmentParser =
249     do { reserved "this"
250         ; symbol "."
251         ; classField <- identifier
252         ; symbol "="
253         ; constructorArgument <- identifier
254         ; semi
255         ; return (classField, constructorArgument)
256         }
257     <?> "assignment"
258
259 -- Method:
260 -- C m(C* x*) { return e; }
261 methodParser =
262     do { methodType <- typeParser;
263         ; methodName <- identifier;
264         ; methodArgs <- parens argumentParser
265         ; expression <- braces (do { reserved "return"
266                                    ; expr <- exprParser
267                                    ; semi
268                                    ; return expr
269                                    })
270         ; return $ Method methodType methodName methodArgs expression
271         }
272     <?> "method"
273
274 -- Argument:
275 -- C* f*
276 argumentParser = sepBy p $ symbol ","
277     where
278         p = do { argumentType <- typeParser
279                ; argumentName <- identifier
280                ; return $ Field argumentType argumentName -- type Argument
                       = Field
281                }
282             <?> "argument"
283
284 -- Expression:
285 -- x | e.f | e.m(e*) | new C(e*) | (C)e
286 --
287 -- The idea is to parse this using Parsec's buildExpressionParser
288 -- with "." as a binary operator. We can then distinguish the expression
289 -- to the right of "." and initialize the appropriate type of Expr.
290 exprParser = E.buildExpressionParser table term
291     <?> "expression"
```

```
292
293 table :: E.OperatorTable Char () Expr
294 table = [[E.Infix (reservedOp "." >> return makeExpr) E.AssocLeft]]
295     where
296         makeExpr e1 (ExprMethod _ e2name e2exprs) = ExprMethod e1 e2name
                e2exprs
297         makeExpr e1 (ExprVar e2name) = ExprField e1 e2name
298         makeExpr _ _ = error "Something is terribly wrong with some
                expression."
299
300 -- Begin with the distinguished "new ..." or "(C)" and proceed with the
        options
301 -- that begin with an identifier.
302 term  = exprNewParser
303     <|> try exprCastParser -- try is used since (whatever) can also be (
            expression).
304     <|> parens exprParser
305     <|> try exprMethodParser -- try is used since it is possible that the
            identifier is
306                              -- not followed by (e*), i.e. it should be a
                                    variable/field.
307     <|> exprOtherParser
308     <?> "expression"
309
310 -- Parse "new C(e*)"
311 exprNewParser =
312     do { reserved "new"
313        ; exprType <- typeParser
314        ; exprs <- parens $ sepBy exprParser $ symbol ","
315        ; return $ ExprNew exprType exprs
316        }
317
318 -- Parse "(C) e"
319 exprCastParser =
320     do { castType <- parens typeParser
321        ; expr <- exprParser
322        ; return $ ExprCast castType expr
323        }
324
325 -- Parse "m(e*)"
326 -- Known problem: this makes m(e*) valid syntax even if it does not occur
327 -- to the right of an expression.
328 exprMethodParser =
329     do { method <- identifier
330        ; exprs <- parens $ sepBy exprParser $ symbol ","
331        ; return $ ExprMethod (ExprVar "_") method exprs
332        }
333
334 -- Parse an identifier and make sense of it later (if it should be ExprVar
        or ExprField).
335 exprOtherParser =
336     do { raw <- identifier <|> do { reserved "this"; return "this" }
337        ; return $ ExprVar raw
338        }
```

```
339
340
341 -- Parse and return the whole class table.
342 classTableParser =
343     do { whiteSpace
344        ; ct <- many classHeadParser
345        ; eof
346        ; return ct
347        }
348
349 -- }}}
350
351 -- {{{ Type checking functions
352
353 -- Check the whole class table that consists of the parsed class table plus
354 -- the special class Object.
355 tcheckClassTable :: ClassTable -> Bool
356 tcheckClassTable ct = tcheckClassTable1 ct'
357     where
358         ct'         = object:ct
359         object      = Class "Object" "" [] (Constructor "Object" [] [] [])
                []
360
361 -- Check the whole class table.
362 tcheckClassTable1 :: ClassTable -> Bool
363 tcheckClassTable1 ct
364     | not classesAreUnique = error "The classes are not unique."
365     | not classTableOK     = error "Some class failed the type check."
366     | otherwise            = True
367     where
368         classNames = map cname ct
369
370         -- Check that all the classes in the class table are unique.
371         classesAreUnique = nub classNames == classNames
372
373         -- Check that all the classes in the class table are OK.
374         classTableOK     = all ((==) True) $ map (tcheckClass ct) ct
375
376         -- Would also like to check that there are no circular class
377         -- dependencies here.
378
379
380 -- Check that the class is OK.
381 tcheckClass :: ClassTable -> Class -> Bool
382 tcheckClass ct c = classHeadOK && fieldsOK && constructorOK && methodsOK
383     where
384         classHeadOK   = tcheckClassHead   ct c
385         fieldsOK      = tcheckFields      ct c
386         constructorOK = tcheckConstructor ct c
387         methodsOK     = tcheckMethods     ct c
388
389 -- Check that the superclass exists
390 tcheckClassHead :: ClassTable -> Class -> Bool
391 tcheckClassHead ct c =
```

```
392     if cname c == "Object"
393         then True -- We know Object is OK
394     -- Check that the superclass exists (and is not the same as the class
395     -- itself) in the given class table. That the classes in the class
            table
396     -- are unique was already checked in tcheckClassTable1 which is why we
            can
397     -- allow ourselves to have "x /= c" below.
398         else not $ null [x | x <- ct, x /= c, cname x == csuperClass c]
399
400 -- Check that the fields are given a unique name and that their type exists
        .
401 tcheckFields :: ClassTable -> Class -> Bool
402 tcheckFields ct c = fieldsAreUnique && fieldsOK
403     where
404         -- Check that all the types of the fields exist in the class table.
405         fieldsOK =
406             if all ((==) True) $ map (classExists ct . ftype) (cfields c)
407                 then True
408                 else error $ "Some field in class " ++ (cname c) ++
409                              " is of a type which does not exist. T-Class
                                    failed."
410
411         -- Check that all the fields have unique names all through the
                superclasses.
412         fieldsAreUnique =
413             if nubBy (\x -> \y -> fname x == fname y) (fields ct c) ==
                    fields ct c
414                 then True
415                 else error $ "The class " ++ (cname c) ++ " contains fields
                        with duplicate names \
416                                 \(possibly a duplicate of a field in a
                                        superclass). T-Class failed."
417
418 -- Part of (T-Class). (Everything but M* OK IN C.)
419 tcheckConstructor :: ClassTable -> Class -> Bool
420 tcheckConstructor ct c = argumentsOK && assignmentsOK && superParametersOK
421     where
422         -- For easy access to the constructor.
423         cons = cconstructor c
424
425         -- Partition the arguments by if they are part of the current class
                or part of some superclass.
426         (classFields, superClassFields) = partition (\x -> x `elem` (
                cfields c)) (kargs cons)
427
428         -- Check that the assignments are OK in the sense that the fields
                that are assigned to
429         -- have the same name as the argument to the constructor and the
                same name as the fields
430         -- in the class.
431         assignmentsOK =
432             if map fname classFields == map fst (kassignments cons) &&
```

```
433                 all ((==) True) (map (\(x,y) -> x == y) (kassignments cons)
                        )
434                 then True
435                 else error $ "Something wrong with the assignments in the
                        constructor of class "
436                                 ++ (cname c) ++ ". T-Class failed."
437
438         -- Check that all the types in the arguments exist and that the
                names of the parameters
439         -- correspond to the names in the class and its superclasses.
440         argumentsOK =
441             if kargs cons == fields ct c
442                 then True
443                 else error $ "The arguments to the constructor in class "
                        ++ (cname c) ++
444                                 " are incorrect. T-Class failed."
445
446         -- Check that the parameters to the super constructor correspond to
                the fields that
447         -- exist in the superclasses.
448         superParametersOK =
449             if map fname superClassFields == ksuperParam cons
450                 then True
451                 else error $ "The parameters to super in class " ++ (cname
                        c) ++ " are incorrect. \
452                                 \T-Class failed."
453
454 -- (T-Method)
455 tcheckMethods :: ClassTable -> Class -> Bool
456 tcheckMethods ct c = methodsAreUnique && argsOK && mtypeOK && mexprsOK
457     where
458         -- For easy access to the methods.
459         methods = cmethods c
460
461         -- Run the method head and expression checkers on all methods in
                the class.
462         mtypeOK = all ((==) True) $ map mheadOK methods
463         mexprsOK = all ((==) True) $ map expressionOK methods
464
465         -- Check that the methods have unique names.
466         methodsAreUnique = if nub methodNames == methodNames
467                               then True
468                               else error $ "Class " ++ (cname c) ++ "
                                      contains duplicate methods. \
469                                       \T-Class failed."
470             where
471                 methodNames = map mname methods
472
473         -- Check that the arguments have unique names.
474         argsOK = if map nub methodArgNames == methodArgNames
475                     then True
476                     else error $ "Methods are not allowed to have several
                            arguments with the same \
```

```
477                                     \name (error in class " ++ (cname c) ++ "
                                            ). T-Method failed."
478             where
479                 methodArgNames = map (map fname . margs) methods
480
481         -- Check that the method head is OK, i.e. the "if mtype ..."-part
                of T-Method.
482         -- The implication [P -> Q] in T-Method is rewritten as [(not P) or
                Q].
483         mheadOK m =
484             if (sclassTypeSig == Nothing) || fromJust sclassTypeSig == (map
                    ftype (margs m), mtype m)
485                 then True
486                 else error $ "Method " ++ (mname m) ++ " in class " ++ (
                        cname c) ++ " does not match \
487                                 \the types of its namesake in some superclass.
                                        T-Method failed."
488             where
489                 sclassTypeSig = typeSig (superClassOf ct c)
490                 -- Find the type signature of the method in the superclass,
                        if it exists.
491                 typeSig c'
492                     | cname c' == "Object" = Nothing
493                     | mfound /= Nothing    = Just (map ftype (margs result)
                            , mtype result)
494                     | otherwise            = typeSig (superClassOf ct c')
495                     where
496                         mfound = find (\x -> mname m == mname x) (cmethods
                                c')
497                         result = fromJust mfound
498
499         -- Check the expression part of T-Method, i.e. if the expression
                has type E0
500         -- and the method is supposed to return type C0, check that E0 <:
                C0.
501         expressionOK (Method mt mn ma me) =
502             if isSubClass ct (getClass ct (exprType ct m' me)) (getClass ct
                    mt)
503                 then True
504                 else error $ "Expression in " ++ mn ++ " in class " ++ (
                        cname c) ++
505                                 " is not a subtype of " ++ mt ++ ". T-Method
                                        failed."
506             where
507                 -- Prepend "this" to the method arguments, as specified by
                        the typing rules.
508                 m' = Method mt mn (this : ma) me
509                 this = Field (cname c) "this"
510
511 -- Expression typing:
512 exprType :: ClassTable -> Method -> Expr -> Type
513 -- (T-Var)
514 exprType ct m (ExprVar var) =
515     if foundVar /= Nothing
```

```
516         then ftype $ fromJust foundVar
517         else error $ "Variable " ++ var ++ " does not exist in the
                environment of method "
518                     ++ (mname m) ++ ". T-Var failed."
519     where
520         foundVar = find (\x -> fname x == var) $ margs m
521
522 -- (T-Field)
523 exprType ct m (ExprField e0expr e0name) =
524     if foundField /= Nothing
525         then ftype $ fromJust foundField
526         else error $ "Type error in method " ++ (mname m) ++ ". T-Field
                failed."
527     where
528         foundField = find (\x -> fname x == e0name) $ fields ct (getClass
                ct (exprType ct m e0expr))
529 -- (T-Invk)
530 exprType ct m (ExprMethod e0expr e0name e0exprs) =
531     if subTypesOK
532         then retType
533         else error $ "Type error in method " ++ (mname m) ++ ". T-Invk
                failed."
534     where
535         (argTypes, retType) = methtype ct (getClass ct (exprType ct m
                e0expr)) e0name
536         exprTypes = map (exprType ct m) e0exprs
537         subTypesOK = length exprTypes == length argTypes &&
538                      all ((==) True) (zipWith (isSubClass ct) (map (
                         getClass ct) exprTypes) (map (getClass ct)
                         argTypes))
539
540 -- (T-New)
541 exprType ct m (ExprNew newType e0exprs) =
542     if subTypesOK
543         then newType
544         else error $ "Type error in method " ++ (mname m) ++ ". T-New
                failed."
545     where
546         classFieldTypes = map ftype $ fields ct $ getClass ct newType
547         exprTypes = map (exprType ct m) e0exprs
548         subTypesOK = length exprTypes == length classFieldTypes &&
549                      all ((==) True) (zipWith (isSubClass ct) (map (
                         getClass ct) exprTypes) (map (getClass ct)
                         classFieldTypes))
550
551 -- (T-*Cast)
552 -- While all the casts should return castType as the type of the expression
        ,
553 -- the code below is just to show how it is possible to distinguish between
        them.
554 -- One might want to inform when a stupid cast takes place, for example.
555 -- Otherwise, the function could just take the form:
556 -- exprType ct m (ExprCast castType _) = castType
557 exprType ct m (ExprCast castType e0expr)
```

```
558     | isSubClass ct class1 class2 = castType -- T-UCast
559     | isSubClass ct class2 class1 = castType -- T-DCast
560     | otherwise                   = castType -- T-SCast
561     where
562         class1 = getClass ct (exprType ct m e0expr)
563         class2 = getClass ct castType
564
565 -- }}}
566
567 -- {{{ The rest
568
569 runTypeCheck :: SourceName -> IO ()
570 runTypeCheck inputFile =
571     do { result <- parseFromFile classTableParser inputFile
572        ; case result of
573              Left err -> print err
574              Right res ->
575                  do putStrLn $ if typesOK
576                                then inputFile ++ " passed the type
                                        check."
577                                else inputFile ++ " did NOT pass the
                                        type check."
578                        where
579                            typesOK = tcheckClassTable res
580
581        }
582
583 main :: IO ()
584 main = do { args <- getArgs
585           ; if (length args) > 0
586                then runTypeCheck $ head args
587                else putStrLn "Usage: runhaskell FJ.hs <Featherweight Java
                        file>"
588           }
589
590 -- }}}
```