En prototypimplementation av Featherweight Java i Prolog

JOHAN ÖHLIN

KTH Datavetenskap
och kommunikation

Examensarbete
Stockholm, Sverige 2010

En prototypimplementation av Featherweight Java i Prolog

J O H A N Ö H L I N

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/ohlin_johan_K10031.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Sammanfattning

Med en abstrakt beskrivning av ett programmeringsspråk kan man bevisa att det fungerar i teorin. Featherweight Java är en abstrakt beskrivning av Java med några borttagna egenskaper.

Rapporten beskriver Featherweight Java och visar exempel på hur inferensreglerna används. Dessutom har reglerna implementerats i logikprogrammeringsspråket Prolog. Det kräver bland annat en representation av den programsyntax som Featherweight Java använder sig av för Prolog.

Abstract

A Prototype Implementation of Featherweight Java in Prolog

With an abstract description of a programming language, one can show that it works in theory. Featherweight Java is just that, an abstract description of Java but with some properties removed.

The report describes Featherweight Java and shows examples of how the inference rules are used. Furthermore, the inference rules have been implemented in the logic programming language Prolog. That requires, amongst other things, a representation of the program syntax Featherweight Java uses for Prolog.

Innehåll

Kapitel 1

Introduktion

Denna rapport är en del av ett kandidatexamensarbete för kursen DD143X "Examensarbete inom datalogi, grundnivå" på KTH, vårterminen 2010. Examensarbetet består av två delar varav denna rapport representerar den ena på 6 hp (högskolepoäng). Den andra delen är ett programmeringsprojekt i en större grupp för skapa en produkt åt industrin på 9 hp. Förutom att klara denna kurs krävs totalt 180 hp för att ta en kandidatexamen.

Läsare av rapporten kan behöva kunskaper i Java för att förstå en del av innehållet. Dessutom är det bra om man har har stött på Prolog för att förstå koden, men de flesta med goda kunskaper om datorer och framförallt programmering ska utan problem kunna läsa denna rapport.

1.1 Bakgrund

En abstrakt specifikation av ett programmeringsspråk kan användas för att visa hur språket fungerar och bevisa att det fungerar så som man har tänkt sig att det ska göra. Featherweight Java är abstrakt beskrivning av delar ur programmeringsspråket Java. Konceptet är att allting är objekt.

Abstrakta beskrivningarna är även bra för att kontrollera att en ny implementation eller ny version av språket kan utvecklas. Det kan göras genom att utöka den abstrakta specifikationen först, och sedan bevisa att det man tänkt sig göra med språket fungerar innan det implementeras.

1.2 Mål

Målet är att undersöka om den abstrakta specifikation av programmeringsspråket Java, just Featherweight Java, kan användas för att skapa en prototypimplementation i logikprogrammeringsspråket Prolog. Att programmet fungerar kommer att visas med att använda Prologs `trace` i kombination med en motivering.

Kapitel 2

Featherweight Java

Den huvudsakliga referensen för detta kapitel är dokumentet av A. Igarashi, B.C. Pierce och P. Wadler där de definierar Featherweight Java (förkortat FJ), referens [1], och därför kommer bara övriga referenser att anges. Notera att delen om Generic Java (GJ) ligger utanför denna rapports omfång.

2.1 Grunderna

Featherweight Java är en abstrakt beskrivning av kärnan i Java men saknar grundläggande egenskaper såsom vanliga fälttilldelningar, gränssnitt, metodöverlagring, meddelanden till `super()`, inre klasser, `null`-pekare, bastyper (såsom `int` och `bool`), abstrakta metoddeklarationer, `private` och `public`, variabelskuggning samt undantagshantering. Det innehåller däremot rekursiva klassdefinitioner, objektskapande, fältåtkomst, metodanrop, metodöverskuggning, metodrekursion med `this`, subtypning och typkonverteringar. Dessutom har alla fält, för enkelhetens skull, implicit `final`.

Featherweight Java har en liknande relation till Java (där allting är objekt) som λ-kalkyl har till funktionella programspråk såsom Haskell (där allting är en funktion). Ett exempel för ett program i FJ finns här nedanför, och är samma som i FJ-dokumentet:

```
class A extends Object {
    A() { super(); }
}
class B extends Object {
    B() { super(); }
}
class Pair extends Object {
    Object fst;
    Object snd;
    Pair(Object fst, Object snd) {
        super(); this.fst=fst; this.snd=snd;
```

```
        }
        Pair setfst(Object newfst) {
            return new Pair(newfst, this.snd);
        }
    }
```

Varje klass måste ärva ifrån någon klass, och någon klass i programmet måste ärva ifrån `Object`-klassen. Dessutom har måste alla klasser ha en konstruktor, även de mest triviala klasserna (`A` och `B` i exemplet). Det finns fem stycken uttryck och tre stycken räkneregler i FJ. Uttrycken är objektkonstruktorer, metodanrop, fältåtkomst, variabler och typkonvertering, och räknereglerna är för fältåtkomst, metodanrop och typkonvertering. Reglerna för dessa beskrivs med Featherweight Java.

2.2 Syntax och skrivregler

`A`, `B`, `C`, `D` och `E` representerar klasser, `f` och `g` representerar fältnamn, `m` metodnamn och `x` en variabel. `K` är en konstruktordeklaration, $\overline{\texttt{M}}$ är en mängd metoddeklarationer. En klass kan då abstrakt definieras som `class A extends B {`$\overline{\texttt{C}}$ $\overline{\texttt{f}}$`; K `$\overline{\texttt{M}}$`}`.

En överstruken bokstav denoterar en mängd. $\overline{\texttt{M}}$ betecknar en mängd metoder, där en metod `m` beskrivs `C m (`$\overline{\texttt{C}}$ $\overline{\texttt{f}}$`){return e;}`. I metoden `m` har uttrycket `e` returtypen `C` och tar $\overline{\texttt{C}}$ $\overline{\texttt{f}}$ som argument. $\overline{\texttt{C}}$ $\overline{\texttt{f}}$ är ett skrivsätt för mängden av par $\texttt{C}_1$ $\texttt{f}_1$, $\texttt{C}_2$ $\texttt{f}_2$, ..., $\texttt{C}_n$ $\texttt{f}_n$ i vilken fältnamnet $\texttt{f}_i$ är av klassen $\texttt{C}_i$.

Γ denoterar hela programmet, miljön, som avbildas från variabler till typer. $\Gamma \vdash \texttt{x} : \Gamma(\texttt{x})$ är sant om variabeln eller uttrycket `x` har typen $\Gamma(\texttt{x})$ i Γ. $\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{C}}$ är en annan notation för $\Gamma \vdash \texttt{e}_1 : \texttt{C}_1$, $\Gamma \vdash \texttt{e}_2 : \texttt{C}_2$, ..., $\Gamma \vdash \texttt{e}_n : \texttt{C}_n$.

För att fråga om en klass `C` är en subklass till `D` används `C <: D`, där $\overline{\texttt{C}}$ `<:` $\overline{\texttt{D}}$ är skrivsätt för $\texttt{C}_1$ `<:` $\texttt{D}_1$, $\texttt{C}_2$ `<:` $\texttt{D}_2$, ..., $\texttt{C}_n$ `<:` $\texttt{D}_n$. `C` $\not<:$ `D` denoterar att `D` inte är superklass till `C`. Dessutom räknas en klass som en superklass till sig själv, det vill säga `C <: C`.

En konstruktor `K` definieras som `C (`$\overline{\texttt{D}}$ $\overline{\texttt{g}}$`; `$\overline{\texttt{C}}$ $\overline{\texttt{f}}$`) {super(`$\overline{\texttt{g}}$`); this.`$\overline{\texttt{f}}$`=`$\overline{\texttt{f}}$`;}` i klassen `C`. Argumenten till konstruktorn kan delas upp i två delar, $\overline{\texttt{D}}$ $\overline{\texttt{g}}$ och $\overline{\texttt{C}}$ $\overline{\texttt{f}}$, där $\overline{\texttt{g}}$ används som argument till superklassens konstruktor och $\overline{\texttt{C}}$ $\overline{\texttt{f}}$ används för att tilldela klassendeklarationerna fältvärden.

CT är en klasstabell som avbildar klassnamn `C` till klassdeklaration `L`. Ett program kan definieras som ett par (CT, `e`), där `e` är ett inmatningsuttryck. I CT finns alla klasser förutom `Object`. Det finns heller inga cykliska subtypsrelationer i klasstabellen, det vill säga att inga två klasser `C` $\neq$ `D` som ger `C <: D` och `D <: C`.

Eftersom `Object` är en klass som inte finns definierad i Featherweight Java så måste undantag göras för just `Object`. Den finns, som nämnt, inte i CT, och metoderna för att göra fält- respektive metoddeklarationer har specialfall för just `Object` som returnerar en tom mängd, som denoteras med $\bullet$.

`m` $\in \overline{\texttt{M}}$ betyder att metoden `m` finns bland metoderna i mängden $\overline{\texttt{M}}$, och `m` $\notin \overline{\texttt{M}}$ att `m` inte finns i mängden. Fältet av en klass, skrivet *fields*(`C`), är en parsekvens av $\overline{\texttt{C}}$ $\overline{\texttt{f}}$ av varje fält i klassen `C`, samt av dess superklasser.

Metodtypen för metoden `m` i klassen `C` skrivs *mtype*(`m, C`) och är reducering $\overline{\texttt{A}} \rightarrow \texttt{B}$ där alla $\texttt{A} \in \overline{\texttt{A}}$ är argumenttyperna och `B` är returtypen i metoden. *mbody*(`m, C`) ger innehållet av `m` som en sekvens av parametrar och ett uttryck, $\overline{\texttt{x}}.\texttt{e}$. Både *mbody*(`m, Object`) och *mtype*(`m, Object`) är odefinierade eftersom `Object` inte har några metoder i Featherweight Java.

I Featherweight Java finns det tre typer av typomvandlingar; *upcast*, *downcast* och *stupid cast*. En *upcast* är en konvertering från `C` till `D` när `C <: D` och *downcast* samma omvandling men med `D <: C`. En *stupid cast* är en otillåten typomvandling då $\texttt{C} \not<: \texttt{D}$ och $\texttt{D} \not<: \texttt{C}$ med $\texttt{C} \neq \texttt{D}$.

För att kontrollera om metoder och klassdeklarationer är korrekta finns `M OK IN C` och `L OK` som är vad man kan förvänta sig att de ska vara. `M OK IN C` säger att metoden `M` är okej i klassen `C`, och motsvarande för `L OK` att klassdeklarationen `L` är okej.

Till sist finns vad som kallas för reduktion och symboliseras med $e \rightarrow e'$ som betyder att e reduceras till e'. Reglerna för reduktion ges i sektion 2.3.4. Notationen för att ersätta $\texttt{x}_1$ med $\texttt{d}_1$, $\texttt{x}_2$ med $\texttt{d}_2$, ..., $\texttt{x}_n$ med $\texttt{d}_n$ samt `y` med `e` är $[\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{e/y}]\texttt{e}_0$ i uttrycket $\texttt{e}_0$.

2.3 Regler

Här ges inferensreglerna för Featherweight Java och en kort beskrivning av dem.

2.3.1 Definitioner

Definitioner av en klass `L`, en konstruktor `K`, en metod `M` och ett uttryck `e`:

$$\texttt{L ::= class C extends D \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K}\ \overline{\texttt{M}}\texttt{\}}$$

$$\texttt{K ::= C (}\overline{\texttt{D}}\ \overline{\texttt{g}}\texttt{; }\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{)\{super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{\}}$$

$$\texttt{M ::= C m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{)\{return e;\}}$$

$$\texttt{e ::= x | e.f | e.m(}\overline{\texttt{e}}\texttt{) | new C(}\overline{\texttt{e}}\texttt{) | (C)e}$$

Ett uttryck är antingen en variabel, ett fält från en uttryck, en mängd uttryck som används i ett ursprungligt uttrycks metodanrop, en ny klassdeklaration eller en typomvandling.

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

En klass `C` räknas som en subklass till `E` om det finns ett `D` så att `C <: D` och `D <: E`.

2.3.2 Uppslag

Fält

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

Regeln säger att fältet av en klass består av egna fältet samt fältet av sin superklass. Med notationen från exemplet blir detta $\mathit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}}$ om klassdeklarationen ser ut som angivet och om fältet av `D` är $\overline{\texttt{D}}\ \overline{\texttt{g}}$. Detta ger naturligt att fältet av `Object` är en tom sekvens, alltså att den inte finns.

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

Metodtyp

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

mtype returnerar en reduktion av klasser, eftersom metoden i den angivna klassen tar $\overline{\texttt{B}}$ som argument och returnerar `B`. Om `m` inte finns i mängden av klassens metoder så måste den finnas bland superklassens metoder, alltså:

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

Notera att reduktionen i regeln är inte av samma typ som de angivna i del 2.3.4.

Metodinnehåll

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

Denna är lik regeln för att få metodens alla uttryck. Den har också lika naturligt fall för om den ärvda klassen är den som har metoden.

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

2.3.3 Skrivning

Uttryck

T-VAR:

$$\Gamma \vdash x : \Gamma(x)$$

T-VAR förklarar att angivna miljön Γ innehåller av ett uttryck eller en variabel av en specifik typ. Man förklara på flera olika sätt som nämns här under.

T-Field:

$$\frac{\Gamma \vdash \mathtt{e}_0 \mathtt{:}\ \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i\ \mathtt{:}\ \mathtt{C}_i}$$

Ett fält $\mathtt{f}_i$ av typen $\mathtt{C}_i$ i ett uttryck $\mathtt{e}_i$ kan bytas ut mot annat fält i den mängd av fält som finns för uttryckets klass $\mathtt{C}_0$.

T-Invk:

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

Den mängd uttrycken som metoden `m` tar som argument måste alla finnas i den angivna miljön samt att att de typerna måste vara subklasser till de argument som `m` tar i $\mathtt{C}_0$.

T-New:

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

Miljön innehåller en ny deklaration av en typ med en mängd uttryck om mängden också finns i miljön med några typer. De typerna måste vara subklasser till de typer som fälten är av i den nya klassen.

T-UCast:

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

För att göra en uppåttypomvandling (*upcast*) gäller är att variabeln redan finns i miljön med av en annan typ och att den klass man vi konvertera till är av en subtyp till föregående klassen.

T-DCast:

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

Motsatsen till uppåttypomvandling, alltså en nedåttypomvandling (*downcast*). Eftersom `C <: C` alltid är sant så måste antingen T-UCast eller T-DCast innehåller kravet om $\mathtt{C} \neq \mathtt{D}$ för att omvandling inte ska kunna vara både omvandling

uppåt och nedåt, och då har författarna har valt att sätta olikheten i T-DCast.

T-SCast:

$$\frac{\Gamma \vdash \texttt{e}_0 \ \texttt{:}\ \texttt{D} \qquad \texttt{D} \not<: \texttt{C} \qquad \texttt{C} \not<: \texttt{D} \qquad \textit{stupid warning}}{\Gamma \vdash \texttt{(C)e}_0 \ \texttt{:}\ \texttt{C}}$$

En dum typomvandling sker då varken `C` $\not<:$ `D` eller `D` $\not<:$ `C` är giltiga. Dessutom ges en "dum" varning.

Metoder

T-Method:

$$\frac{\begin{array}{c}\overline{\texttt{x}}\ \texttt{:}\ \overline{\texttt{C}},\ \texttt{this}\ \texttt{:}\ \texttt{C} \vdash \texttt{e}_0\ \texttt{:}\ \texttt{E}_0 \\ \texttt{E}_0 <: \texttt{C}_0 \qquad \texttt{C} <: \texttt{D} \qquad \text{if } \textit{mtype}(\texttt{m},\texttt{D}) = \overline{\texttt{D}} \to \texttt{D}, \text{ then } \overline{\texttt{C}} = \overline{\texttt{D}} \text{ and } \texttt{C}_0 = \texttt{D}_0\end{array}}{\texttt{C}_0\texttt{m}\ (\overline{\texttt{C}}\ \overline{\texttt{x}})\{\ \texttt{return e}_0\texttt{;}\ \}\ \texttt{OK IN C}}$$

Den angivna metoden är "OK" om miljön som består av mängd uttryck och variabler $\overline{\texttt{x}}$ av typerna $\overline{\texttt{C}}$ och `this` av typen `C`. Miljön innehåller $\texttt{e}_0$ av $\texttt{E}_0$ där $\texttt{E}_0$ är en subklass till $\texttt{C}_0$ och `C` är subklass till `D`. Om det är så att $\textit{mtype}(\texttt{m},\texttt{D}) = \overline{\texttt{D}} \to \texttt{D}$ så måste $\overline{\texttt{D}}$ vara $\overline{\texttt{C}}$ och $\texttt{D}_0$ är $\texttt{C}_0$.

Klasser

T-Class:

$$\frac{\texttt{K} = \texttt{C}(\overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{g}})\texttt{;}\ \texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}\} \qquad \textit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \qquad \overline{\texttt{M}}\ \texttt{OK IN C}}{\texttt{class C extends D }\{\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}\ \texttt{K}\ \overline{\texttt{M}}\}\ \texttt{OK}}$$

Klassen är "OK" om konstruktordeklarationen består av de fält som inte är angivna i klassen och att de metoder som angivits är "OK" i klassen.

2.3.4 Reduktionsregler

Uträkning

$$\frac{\textit{fields}(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{f}}}{\texttt{(new C}(\overline{\texttt{e}})\texttt{).f}_i \to \texttt{e}_i}$$

Om fältet av en klass innehåller ett uttryck kan det uttrycket i en ny deklarationen reduceras till något element i den nya deklarationen.

$$\frac{\textit{mbody}(\texttt{m},\texttt{C}) = \overline{\texttt{x}}\texttt{.e}_0}{\texttt{(new C}(\overline{\texttt{e}})\texttt{).m}(\overline{\texttt{d}}) \to [\overline{\texttt{d}}/\overline{\texttt{x}}\texttt{, new C}(\overline{\texttt{e}})\texttt{/this}]\texttt{e}_0}$$

En metod reduceras till uttrycket $\texttt{e}_0$, men med $\overline{\texttt{d}}$ ersatta med $\overline{\texttt{x}}$ och `new C(`$\overline{\texttt{e}}$`)` med `this`.

$$\frac{\texttt{C <: D}}{\texttt{(D)(new C(}\overline{\texttt{e}}\texttt{))} \rightarrow \texttt{new C(}\overline{\texttt{e}}\texttt{)}}$$

Om `C` är en subklass till `D` blir typomvandlingen onödig.

Kongruens

Alla dessa nedanstående regler liknar varandra och är enkla att förstå utan vidare förklaringar.

$$\frac{\texttt{e}_0 \rightarrow \texttt{e}_0'}{\texttt{e}_0\texttt{.f} \rightarrow \texttt{e}_0'\texttt{.f}}$$

$$\frac{\texttt{e}_0 \rightarrow \texttt{e}_0'}{\texttt{e}_0\texttt{.m(}\overline{\texttt{e}}\texttt{)} \rightarrow \texttt{e}_0'\texttt{.m(}\overline{\texttt{e}}\texttt{)}}$$

$$\frac{\texttt{e}_i \rightarrow \texttt{e}_i'}{\texttt{e}_0\texttt{.m(...,e}_i\texttt{,...)} \rightarrow \texttt{e}_0\texttt{.m(...,e}_i'\texttt{,...)}}$$

$$\frac{\texttt{e}_i \rightarrow \texttt{e}_i'}{\texttt{new C(...,e}_i\texttt{,...)} \rightarrow \texttt{new C(...,e}_i'\texttt{,...)}}$$

$$\frac{\texttt{e}_0 \rightarrow \texttt{e}_0'}{\texttt{(C)e}_0 \rightarrow \texttt{(C)e}_0'}$$

2.4 Exempel

Låt oss ta ett exempel med en modifierad variant av den exempelkoden som gavs tidigare och kontrollerar den med inferensreglerna.

```
class A extends B {
    A() { super(); }
}
class B extends Object {
```

```
    B() { super(); }
}
class Pair extends Object {
    A fst;
    B snd;
    Pair(A fst, B snd) {
        super(); this.fst=fst; this.snd=snd;
    }
    Pair setfst(B newfst) {
        return new Pair(new Pair((A)newfst, this.snd).fst, this.snd);
    }
}
```

Kalla hela programmet för `P` i vilken ett antal uttryck $\overline{\texttt{e}}$ av typerna $\overline{\texttt{C}}$ finns, det vill säga $\texttt{P} \vdash \overline{\texttt{e}} : \overline{\texttt{C}}$. Låt oss börja med att kontrollera att klasserna och metoderna i dem är okej.

$$\text{Ex1}\ \frac{\texttt{A OK} \quad \texttt{B OK} \quad \texttt{Pair OK}}{\overline{\texttt{L}}\ \texttt{OK}}$$

Varken `A` och `B` innehåller några metoder och behöver inga argument och är således okej. Nästa steg visar `Pair OK`:

$$\text{Ex2}\ \frac{\texttt{Pair(A fst, B snd)\{super(); this.fst=fst; this.snd=snd;\}} \quad \mathit{fields}(\texttt{Object}) = \bullet \quad \texttt{setfst OK IN Pair}}{\texttt{Pair OK}}$$

Inga av konstruktorns argument vidarebefordras till superklassen och därför ger $fields$-funktionen ingenting. Den enda metoden i `Pair` är `setfst` som vi ska visa vara okej i klassen:

$$\text{Ex3}\ \frac{\{\texttt{newfst}, \texttt{this}\} : \{\texttt{B}, \texttt{Pair}\} \vdash \texttt{new Pair(new Pair((A)newfst, this.snd).fst, this.snd) : Pair} \quad \texttt{Pair <: Pair} \quad \texttt{class Pair extends Object \{...\}} \quad \text{if } \mathit{mtype}(\texttt{setfst}, \texttt{Object}) = \overline{\texttt{D}} \to \texttt{D}}{\texttt{setfst OK IN Pair}}$$

Eftersom `Object` inte finns definierad i Featherweight Java finns inga metoder i den, alltså finns inga $\overline{\texttt{D}} \to \texttt{D}$. `Pair <: Pair` gäller och därmed får vi visa uttrycksregeln:

$$\text{Ex4}\quad \frac{\begin{array}{c}fields(\texttt{Pair}) = \{\texttt{A fst; B snd}\} \\ \{\texttt{newfst}, \texttt{this}\} : \{\texttt{B}, \texttt{Pair}\} \vdash \overline{\texttt{e}} \ : \ \overline{\texttt{C}} \qquad \overline{\texttt{C}} \ \texttt{<:} \ \overline{\texttt{D}}\end{array}}{\{\texttt{newfst}, \texttt{this}\} : \{\texttt{B}, \texttt{Pair}\} \vdash \texttt{new Pair}(\overline{\texttt{e}}) \ : \ \texttt{Pair}}$$

Fälten ges av:

$$\text{Ex5}\quad \frac{\texttt{class Pair extends Object \{A fst, B snd; K } \overline{\texttt{M}}\texttt{\}} \qquad fields(\texttt{Object}) = \bullet}{fields(\texttt{Pair}) = \{\texttt{A fst; B snd}\}}$$

Det ytterligare uttrycket ges av följdande steg, där vi förkortar $\{\texttt{newfst}, \texttt{this}\} : \{\texttt{B}, \texttt{Pair}\}$ till Γ.

$$\text{Ex6}\quad \frac{\Gamma \vdash \texttt{new Pair((A)newfst, this.snd).fst : X} \qquad \Gamma \vdash \texttt{this.snd : Y}}{\Gamma \vdash \texttt{new Pair((A)newfst, this.snd).fst, this.snd : \{X, Y\}}}$$

Vi inte ännu vet vad `Y` och `X` är men de borde vara `A` respektive `B`, eftersom metoden `setfst` tar just de två argumenttyperna. Vi går vidare, och börjar med den kortare som ger oss `Y`:

$$\text{Ex7}\quad \frac{\Gamma \vdash \texttt{this : Pair} \qquad fields(\texttt{Pair}) = \{\texttt{A fst; B snd}\}}{\Gamma \vdash \texttt{this.snd : B}}$$

`Y` ska alltså vara `B` i Ex6. Låt oss gå vidare med den andra för att få `X`:

$$\text{Ex8}\quad \frac{\begin{array}{c}\Gamma \vdash \texttt{new Pair((A)newfst, this.snd) : Pair} \\ fields(\texttt{Pair}) = \{\texttt{A fst; B snd}\}\end{array}}{\Gamma \vdash \texttt{new Pair((A)newfst, this.snd).fst : X}}$$

Vi går vidare med uttrycket:

$$\text{Ex9}\quad \frac{\begin{array}{c}fields(\texttt{Pair}) = \{\texttt{A fst; B snd}\} \\ \Gamma \vdash \texttt{(A)newfst : A} \qquad \Gamma \vdash \texttt{this.snd : B} \qquad \texttt{A <: A} \qquad \texttt{B <: B}\end{array}}{\Gamma \vdash \texttt{new Pair((A)newfst, this.snd) : Pair}}$$

I Ex9 vet vi att $\Gamma \vdash \texttt{this.snd : B}$ är sant ifrån Ex7, så vi går vidare med typomvandlingen av `newfst`:

$$\text{Ex10}\ \frac{\Gamma \vdash \texttt{newfst : B} \qquad \texttt{A <: B} \qquad \texttt{A} \neq \texttt{B}}{\Gamma \vdash \texttt{(A)newfst : A}}$$

Nu kan vi gå tillbaka till Ex4:

$$\text{Ex11}\ \frac{\begin{array}{c} \mathit{fields}(\texttt{Pair}) = \{\texttt{A fst; B snd}\} \\ \{\texttt{newfst}, \texttt{this}\} : \{\texttt{B}, \texttt{Pair}\} \vdash \{\texttt{new Pair((A)newfst, this.snd).fst, this.snd : \{A, B\}}\} \\ \texttt{A <: A} \qquad \texttt{B <: B} \end{array}}{\{\texttt{newfst}, \texttt{this}\} : \{\texttt{B}, \texttt{Pair}\} \vdash \texttt{new Pair}(\overline{\texttt{e}})\ \texttt{: Pair}}$$

Programmet är alltså helt okej, eftersom vi kan visa att alla klasser är okej.

Kapitel 3

Implementation

3.1 Tillvägagång

Prolog är ett programmeringsspråk som bygger på logik, där Prolog står just för ***Pro****gramming in* ***log****ic*. I Prolog specificerar man den fakta man har och skapar sedan en fråga i from av Hornklausuler, som de facto är fakta eftersom allt som inte är definierat inte är sant.[2] Här är ett exempel på ett litet Prologprogram:

```
pappa(anders, bertil).
pappa(david, anders).
mamma(cecilia, bertil).
morforalder(A, B) :- mamma(A, C), mamma(C, B).
morforalder(A, B) :- pappa(A, C), mamma(C, B).
farforalder(A, B) :- mamma(A, C), pappa(C, B).
farforalder(A, B) :- pappa(A, C), pappa(C, B).
```

Namnen `anders`, `bertil`, `cecilia` och `david` kallas i Prolog för objekt[2] och `farforalder`, `mamma`, `morforalder` och `farforalder` kallas för evalueringsuttryck[2]. På Prologvis skulle `farforalder` skrivas `farforalder/2` för att visa att den tar två argument.

Alla rader i programmet är fakta. Det sant att `mamma(cecilia, bertil)` medan `mamma(cecilia, david)` inte är sant. Det är också sant att `farforalder(A, B)` om `mamma(A, C)` och `pappa(C, B)` eller om `pappa(A, C)` och `pappa(C, B)` . Detta kan representeras i Hornklausuler som

$$\exists a\ \exists b\ \exists c\ (Fab \vdash (Pac \wedge Pcb) \vee (Mac \wedge Pcb))$$

Nedan är körning av Prologprogrammet i interpretatorn:

```
?- mamma(X,Y).
X = cecilia,
Y = bertil.
```

```
?- morforalder(X, bertil).
false.

?- farforalder(david, Barnbarn).
Barnbarn = bertil.
```

Tanken är att översätta reglerna från Featherweight Java till frågor i Prolog. Då krävs det att man representerar notationen på ett nytt sätt.

3.2 Omskrivning

Av definitionerna i del 2.3.1 vet vi att en klass har ett namn, en superklass, ett antal deklarationer, en konstruktor samt ett antal metoder. Det kan representeras i Prolog som

`class(Namn, Superklassnamn, Deklarationer, Konstruktor, Metoder)`

`Namn` och `Superklassnamn` är självbeskrivande och `Deklarationer` är en lista av par på formatet `(Klass, Fältnamn)`. `Konstruktor` är klassens konstruktor och är definierad som

`constructor(Namn, Argument, Innehåll)`

där `Namn` måste vara samma namn som klassen den tillhör. `Argument` är en lista med argumenten som den tar och `Innehåll` är en lista med `super`-anropet och `super`-tilldelningarna. Formatet på `Innehåll` kan vara en lista av par som `[(super, [`g_1`, ..., `g_m`]), [(dot(this, `f_1`), `f_1`), ..., (dot(this, `f_n`), `f_n`)]`. Varje g_i finns i superklassens deklarationer och varje f_j måste finnas bland klassens egna deklarationer. `(super, [`g_1`, ..., `g_m`])` står för `super(`$\overline{g}$`)`, och `[(dot(this, `f_1`),`f_1`), ..., (dot(`f_n`), `f_n`)]` för `this.`$\overline{f}$`=`$\overline{f}$.

För att klassen ska bli komplett behövs en representation av metoderna i klassen.

`method(Namn, Returtyp, Argument, Retur)`

`Namn` är en metodnamnet, `Returtyp` är namnet på den klassen som uttrycket `Retur` väntas ge. `Argument` är en lista med argumenten som par `(Klass, Fältnamn)`.

En klass måste ha ett namn, en superklass och en konstruktor, men inte några deklarationer eller några metoder. För enkelhetens skull får de två listorna vara tomma vid fall att inga deklarationer eller inga metoder finns.

Utöver dessa representationer behövs representation för punkter, som i `e.m(`$\overline{e}$`)`, samt för reduktion, som i `e` $\rightarrow$ `e`$'$. Då behövs representation för `m(`$\overline{e}$`)` som därför inte behöver vara knepigare än `(m, [`e_1`, ..., `e_n`])`, och eftersom punkt i Prolog används i slutet av fakta får punkt istället representeras som `dot(e, f)`. Det gäller även `this.x`. `e.m(`$\overline{e}$`)` representeras då som `dot(e, (m, [`e_1`, ..., `e_n`]))`.

Reduktionen görs lika enkelt till `reduction(e, e′)`. För typomvandling kan man kan tillämpa samma princip, så `(C)e` skrivs som `cast(C, e)`.

Själva klasserna kommer att finnas i klasstabeller som fakta så att de enkelt går att komma åt ifrån Prologimplementationen. Varje klass skrivs som `class(...)` mellan `classtable(...)`, och man får ut en klass ur klasstabellen med `classtable(Klass)`. Dessutom kan man få ut en specifik klass med namnet `'Namn'` genom att lägga till kravet `Klass = class('Namn', _, _, _)` där `_` i Prolog markerar att värdet inte spelar någon roll[2].

Kapitel 4

Resultat

4.1 Koden

Koden har använt den notation för klasser, konstruktorer, metoder, etc. angivet tidigare i rapporten och har så gott det går samma notationer. Eftersom endast versaler används för variabler så har gemenerna oftast bytts ut mot just versaler. Överstrukna bokstäver har representerats med ordet 'makron' efter det namnet på det streck som skrivs över vissa vokaler, exempelvis som ā och ō, i en del språk och i transkriptioner.[3][4]

Två hjälpfrågor har skapats för att förenkla programmet. Den ena är `issuper/2` som tar två atomer och kollar om den första är subklass till den andra. Den kan även ta två listor och kollar då rekursivt parvis kontrollerar samma sak.

Andra hjälpfunktionen är `helper/2` som kan dela upp en lista av par som `[(A, B), (C, D) ...]`) till två listor med paren separerade som `[A, C, ...], [B, D, ...]`. Den funktionen används både för att skapa en lista av par och för att dela upp listan.

4.2 Programöversättning

Det lilla Featherweight Java-programmet som används exemplet i del 2.4 omskrivet till det Prologformat som valts blir:

```
% Klass A
classtable(class('A','B',
                 [],
                 constructor('A',[],[(super, [])]),
                 [])).
% Klass B
classtable(class('B','Object',
                 [],
                 constructor('B',[],[(super, [])]),
```

```
                [])).
% Klass Pair
classtable(class('Pair', 'Object',
                [ ('A', 'fst'),
                  ('B', 'snd') ],
                constructor('Pair',
                   [ ('A', 'fst'),
                     ('B', 'snd') ],
                   [ (super, []),
                     (dot(this,'fst'), 'fst'),
                     (dot(this,'snd'), 'snd') ]),
                [ method('setfst', 'Pair',
                    [ ('B', 'newfst') ],
                      new('Pair',
                       [ dot( new( 'Pair',
                           [ cast('A', 'newfst'),
                             dot(this,'snd') ] ),
                           'fst'),
                           dot(this,'snd') ] ))
                ])).
```

4.3 Kodexekvering

Nedan visas ovanstående program exekverat som en fråga om vilka klasser som finns i klasstabellen och som är "OK" enligt T-Class. Programmet visar att alla tre klasser, `A`, `B` och `Pair`, är OK. Den sista raden står `false.` som uttrycker att inga fler `X` som satisfierar uttrycket kan hittas.

```
?- classtable(X), isok(X).
X = class('A', 'B', [], constructor('A', [], [ (super, [])]), []) ;
X = class('B', 'Object', [], constructor('B', [], [ (super, [])]), []) ;
X = class('Pair', 'Object', [ ('A', fst), ('B', snd)], constructor('Pair', [
('A', fst), ('B', snd)], [ (super, []), (dot(this, fst), fst), (dot(this, snd),
snd)]), [method(setfst, 'Pair', [ ('B', newfst)], new('Pair', [dot(new('Pair',
[cast('A', newfst), dot(..., ...)]), fst), dot(this, snd)]))]) ;
false.
```

I Prolog laddas program med kommandot `consult/1`[2]. Om implementationen ligger i en fil `Featherweightjava.pl` skild från klasslistan så laddas all nödvändig kod in i interpretatorn med:

```
?- consult(Featherweightjava), consult(klasstabell).
```

4.3.1 `trace`

`trace/0` i Prolog är kommando som gör att man får se alla steg som görs av programmet i hur den letar upp svaren[2]. I en exekvering av ovanstående program i SWI Prolog med `trace` kan vi se alla steg som gjordes i exemplet från del 2.4. I bilaga B finns hela sökningen. *Det föreslås att läsaren skriver ut bilaga B för sig eller river ut de sidorna ut sitt exemplar för att följa nedanstående förklaring.* Mening är att detta ska visa att implementationen följer reglerna, och eftersom det är samma program som användes i exemplet i sektion 2.4 så ska denna `trace` använda samma regler som i exemplet.

På rad 1 kallas `classtable/1` för att hitta ett värde på `X` och finner då `A` först eftersom den är överst i filen med klasserna. Värdet på `X` används i `isok/1` på rad 5, som börjar med att be om fälten för `B` eftersom det är namnet på superklassen till `A`. Det gör den, som synes på följande rader 6-12, genom att hämta `B` ur klasstabellen och hämta fälten från `Object`, som bara är en tom lista. Sedan sätts båda klassernas fält ihop med `flatten/2`. Efter tillplattningen anropas `helper/3` som bara kommer ge tomma listor och eftersom `A` inte har några metoder kommer `isok/2` vara sann och ge att `A` är en okej klass. På rad 18 ges ett förslag på `X` som är sant men vi vill se alla resultat och ber således interpretatorn att fortsätta leta.

Eftersom det första förslaget på `X` nekades får interpretatorn gå och hämta nästa värde ur `classtable/1`, vilket blir klassen `B`. Samma saker som för att kontrollera om klassen `A` är okej sker, men med undantag för att `B` bara har superklassen `Object`. *fields* hämtar då den tomma listan av `Object`s fält som sätts ihop med den tomma listan av fält som `B` har. Dessutom har `B`, precis som `A`, inga metoder. På rad 29 föreslås klassen `B` som en lösning, men vi ber interpretatorn att fortsätta leta.

Det nästa `X`-värdet som ges av `classtable(X)` blir klassen `Pair` som börjar på liknande sätt som för klassen `B` men har skillnaden att metoderna undersöks. Listan av metoderna börjar på rad 37 och på nästföljande rad kontrolleras första och enda elementet i listan, metoden `setfst`. `mtype/3` försöker hämta typen från `Object` till en början och misslyckas eftersom `Object` inte är definierad. Men i nästa försök på `mtype/3` på rad 48 så är det meningen att den inte ska lyckas och kan därför gå vidare till `helper/3` i `isok/2`.

Efter att ha delat upp typerna och fältnamnen, lagt till `this` av typen `Pair` och sedan satt ihop dem igen kommer vi till `expression/3` på rad 66. Eftersom vi ytters har `new(...)` kommer vi till T-New. Först hämtas fälten av klassen man skapar på raderna 67-74. `expression/3` anropas med innehållet i `new(...)` på rad 81 som nu är en lista, där alla element måste undersökas. I implementationen kontrolleras de med rekursiva steg för listor.

På rad 85 hamnar vi hos T-Field eftersom vi matchas ihop med `dot(...)`. Anledningen till varför man inte hamnar i T-Invk är för att den inte går att matcha med på formatet `dot(E0, (M, Emakron))`. Där emot kan något som ska till T-Invk komma till T-Field, men eftersom `(M, Emakron)` inte kan finnas i `Fmakron` kommer den att misslyckas och hamna i T-Invk ändå.

Det första T-Invk kommer att göra är att gå anropa `expression/3` med ut-

trycket som fältet ska hämtas ifrån. Där hamnar vi direkt på rad 90 efter att misslyckas med att hitta `new(...)` i Γ direkt och hamnar i T-New. Det görs på samma vis som tidigare anrop och listan som är andra argumentet i `new(...)`.

Precis som tidigare kontrolleras först om man kan finna första elementet direkt i listan i Γ, vilket vi inte kommer göra utan skickas vidare till den första av typomvandlingsfunktionerna, som är T-UCast. Den kollar först vilken typ man vill omvandla ifrån, och där får vi på rad 115 att det är `B`. Då kollar vi om `A`, som vill omvandla till, är superklass till `B` för i så fall är vi i `T-UCast`. Men eftersom `issuper/2` inte hittar någon klass där `A` är superklass till `B` så misslyckas den på rad 119.

Då får man gå bakåt och göra om `expression/3` med samma värden och hamnar i T-DCast. Där hämtar vi åter typen av `newfst` och kollar nu om `B` är superklass till `A` och att `A` inte är `B`. Det satisfieras och på rad 147 anropas andra delen av listan som är argument till `new('Pair', ...)`, som är `dot(this, snd)`. Den anropas på samma vis genom att kolla vad `this` har för typ i Γ och får på rad 153 att det är `Pair`. Nu går vi bakåt i de rekursiva steg vi påbörjade i `expression/3` tidigare.

Nu kan man fortsätta på T-New med och kolla om de klasser vi fått ifrån `fields/2` är superklasser till klasserna vi fått ifrån `expression/3` på rad 177. Det visar de sig vara och vi går ytterligare ett steg bakåt till den `dot(new(...), fst` som påbörjades. Här hämtas fälten av `Pair` på rad 193 och kollar om dess `fst` av typen `A` finns i Γ, och lyckas igen.

Interpretatorn fortsätter nu på tidigare `new(...)`-uttryck som också ska kolla om `dot(this, snd)` finns i Γ. Den gör på samma sätt som tidigare och blir klar på rad 241. Som vid tidigare T-New kontrolleras att klasserna från `fields/2` är superklasser av de värden returnerade ifrån `expression/3`. Vi lyckas med det och kan gå tillbaka till `isok/2` som inte har något mer att göra, så vi hamnar tillbaka i `isok/1` där vi konstaterar att det värde på `X` som föreslagits är okej.

Som goda Prologprogrammerare vill vi se om vi kan få andra förslag på rad 266. Interpretatorn försöker då göra om `issuper('Pair', 'Pair')` men misslyckas med att hitta en annan lösning, och misslyckas därmed helt med att hitta en till klass som är okej.

4.3.2 `trace`-analys

Jämför vi detta resultat med exemplet i del 2.4 ser vi att för `Pair OK` görs samma steg, men däremot inte i samma ordning eftersom implementationen går ner på djupet först. Men först hämtas fälten från superklassen `Object` som är en tom lista, och efter det kontrolleras `setfst`.

`setfst OK IN Pair` går rekursivt in i alla uttrycksregler i ordningen T-New på rad 66, T-Field på rad 85, T-New på rad 90, T-DCast på rad 109, T-Field på rad 147 och T-Field på rad 216. Varken fler eller färre används i exemplet.

4.4 Mer data

Tabell 4.1. Tabell över tester som gjorts för bevisa implementationens duglighet

Klasslistbeskrivning	**Resultat**	**Förklaring**
Fem tomma klasser där alla har varandra som superklasser, utom en som har `Object` som superklass.	Alla klasser är OK.	
Fem tomma klasser där alla har varandra som superklasser, men ingen har `Object` som superklass.	Felmeddelande: `ERROR: Out of local stack`	Rekursiva steg hittar aldrig `Object` och använder då hela stacken.
Två klasser, `A` och `B`, där `A` anropar `super()` med två argument i konstruktorn och har `B` som superklass men superklassen har inga fältdeklarationer.	Alla klasser är OK.	
Två klasser, `A` och `B`, där `A` anropar `super()` med två argument i konstruktorn och har `B` som superklass men superklassen har inga fältdeklarationer.	Endast klassen `B` är OK.	Första klassen försöker göra något som den inte kan göra, helt enkelt.
`A <: B <: Object` där båda har en metod var och `A` använder `B`s metod.	Alla klasser är OK.	
`A <: B <: Object` där `A` har en metod och försöker anropa en metod som `B` inte har.	Endast klassen `B` är OK.	Eftersom `A` försöker göra något otillåtet är den inte okej.

Kapitel 5

Diskussioner

5.1 Resultatdiskussion

Det kan alltid diskuteras hur bra en implementation. Implementationen i detta projekt är testad, men det är svårt att testa för alla möjliga fall eftersom det finns oändligt många. Men med de test som redovisats borde vara tillräckliga för att visa att implementationen är en fungerande till en början, som en prototyp. Implementationen är gjord i Prolog som bygger på logik, så de inferensregler som är angivna i FJ-dokumentationen är enkla att föra över till Prolog.

Dock finns det en svårighet i Prolog med att man kan skriva regler rekursiva på ett sätt så att interpretatorn kan hitta samma lösning ett oändligt antal gånger. Det kan lösas med snitt, som i koden skrivs `!`. När interpretatorn lyckas satisfiera villkoren före ett snitt kan den efter snittet inte gå tillbaka för att försöka finna nya lösningar.[2] Problemet med oändligt många lösningar uppstod i `issuper/2` och löstes med ett snitt. Det kan dock sedermera leda till att en del lösningar som borde vara korrekta blir tolkade som inkorrekta.

Fördelar med snittet är att man kan spara tid och resurser om man vet att bara en lösning kan hittas.[2] Detta har gjorts efter varje uttrycksregel just eftersom man bara kan hamna i en regel. Det enda undantaget är T-Invk samt T-Field i implementationen eftersom `F` i `dot(E0, F)` kan vara `(M, Emakron)`. Men eftersom `member(F, Fmakron)` antagligen inte går igenom om `F` är en atom.

Reduktionsreglerna finns i Featherweight Java, men används inte i de övriga inferensreglerna. De är mer av följder av de andra reglerna och är därför inte implementerade. Om en vidareutveckling av implementationen sker kan man tänka sig att reduktionsreglerna kan användas som genvägar för att snabba upp programmet.

Featherweight Java-kod kan med hjälp av en parser översättas till de klasslistor som kan användas i Prolog-implementationen. Det kan användas som en del i en kompilator för att se om koden kan kompileras. Dessutom skulle FJ stegvis eventuellt kunna utökas, och så också denna implementation, och med lite mer arbete kanske bli ett, i någon mening, riktigt programmeringsspråk.

5.2 Referensdiskussion

Den huvudsaklig referenser i projektet är dokumentet som beskriver Featherweight Java (referens [1]). För just Featherweight Java finns inga relevanta artiklar som ger något utöver det.

Det mesta av Prologkunskaperna har jag med mig från kursen "Programmeringsparadigm", kurskod DD1361, tagen höstterminen 2008. De påståenden som gjorts i rapporten angående Prolog har backats upp med referensen till boken "Programming in Prolog" (referens [2]) som är en bred bok om Prologspråket. Jag har inte lyckats hitta andra tillräckligt bra referenser för att kunna använda dem.

Angående betydelsen av ordet 'makron' som används i Prologimplementationen för att likna den notation som används i FJ-dokumentet refereras Wikipedia eftersom det var där jag först fann ordet. Wikipedia får kritik för att gemene man kan redigera artiklar och möjligen skriver felaktig information, och därför har en ytterligare referens införskaffats. Dessutom är det av en mycket liten betydelse för projektet eftersom ordet kan bytas ut mot nästa vilket annat ord som helst.

Litteraturförteckning

[1] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: A minimal core calculus for Java and GJ," in *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Oct. 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), maj 2001.

[2] W. F. Clocksin and C. Mellish, *Programming in Prolog*. Springer, fifth ed., 2003. Kan läsas hos Google Böcker: `http://books.google.com/books?id=VjHk2Cjrti8C`.

[3] Dictionary.com, "Dictionary.com Unabridged." `http://dictionary.reference.com/browse/macron`, april 2010.

[4] Wikipedia, "Streck (diakritiskt tecken) — Wikipedia." `http://sv.wikipedia.org/w/index.php?title=Streck_(diakritiskt_tecken)&oldid=11419491`, april 2010.

Bilaga A

FJ-implementationen

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Prologimplementation av Featherweight Java av Johan Öhlin (johanohl@kth.se)
%  som ett kandidatexamensarbete 2010. Handledare och examinatör är Mads Dam
%  (mfd@kth.se). Hela arbetet kommer finnas på http://www.d.kth.se/~johanohl/
%  när det är färdigt.
%
%  Det är programmet är skapat med "SWI-Prolog version 5.6.64 for amd64" i
%  Ubuntu 9.10.
%  SWI-Prolog är fri mjukvara och kan hittas på:  http://www.swi-prolog.org/
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Prolog implementation of Featherweight Java by Johan Öhlin (johanohl@kth.se)
%  as a part of a bachelor's essay 2010. Supervisor and examinator is Mads Dam
%  (mfd@kth.se). The essay will be available on http://www.d.kth.se/~johanohl/
%  when finsihed, but only in Swedish.
%
%  This program is made for "SWI-Prolog version 5.6.64 for amd64" on Ubuntu
%  9.10.
%  SWI-Prolog is free software and can be found on:  http://www.swi-prolog.org/
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Fältuppslag.

fields('Object', []) :- !.
fields(Namn, Retur) :-
        classtable(class(Namn, Superklassnamn, Deklarationer, _, _)),
        fields(Superklassnamn,  Superretur),
        flatten([Deklarationer|Superretur], Retur), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%  Metodtypsuppslag.
mtype(_, 'Object', _) :- !, fail.
% Om m finns i M
mtype(Metod, Klass, reduction(Bmakron, B)) :-
        classtable(class(Klass, _, _, _, Metoder)),
        member(method(Metod, B, Argument, _), Metoder),
        helper(Argument, Bmakron, _).
% Om m inte finn i M
mtype(Metod, Klass, reduction(Bmakron, B)) :-
        classtable(class(Klass, Superklass, _, _, Metoder)),
        \+ member(method(Metod, B, _, _), Metoder),
        mtype(Metod, Superklass, reduction(Bmakron, B)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Metodinnehåll.
mbody(_, 'Object', _) :- !, fail.
mbody(Metodnamn, Klassnamn, dot(Xmakron,E)) :-
        classtable(class(Klassnamn, _, _, Metoder, _)),
        member(method(Metodnamn, _, Argument, E), Metoder),
        helper(Argument, _, Xmakron).
mbody(Metodnamn, Klassnamn, dot(Xmakron,E)) :-
        \+ classtable(class(Klassnamn, Superklass, _, Metoder, _)),
        member(method(Metodnamn, _, _, E), Metoder),
        mbody(Metodnamn, Superklass, dot(Xmakron,E)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Uttryckregler.

% T-Var
% Basfall
expression(_, [], []).
% Kollar om X av typen G finns i miljölistan Gamma.
expression(Gamma, [X|XX], [G|GG]) :-
        member((G, X), Gamma),
        expression(Gamma, XX, GG).
% Rekrusivt med listor
expression(Gamma, [X|XX], [G|GG]) :-
        expression(Gamma, X, G),
        expression(Gamma, XX, GG).

% T-Invk
expression(Gamma, dot(E0, (M, Emakron)), C) :-
        expression(Gamma, [E0], [C0]),
        mtype(M, C0, reduction(Dmakron, C)),
        expression(Gamma, Emakron, Cmakron),
        issuper(Cmakron, Dmakron), !.

% T-Field
expression(Gamma, dot(E0, F), GammaX) :-
```

```
        \+ is_list(E0),
        expression(Gamma, [E0], [C0]),
        fields(C0, CFmakron),
        helper(CFmakron, Cmakron, Fmakron),
        member(GammaX, Cmakron),
        member(F, Fmakron), !.

% T-New
expression(Gamma, new(C, Emakron), C) :-
        fields(C, DFmakron),
        helper(DFmakron, Dmakron, _),
        expression(Gamma, Emakron, Cmakron),
        issuper(Cmakron, Dmakron), !.

% T-UCast
expression(Gamma, cast(C, E0), C) :-
        expression(Gamma, [E0], [D]),
        issuper(D, C), !.

% T-DCast
expression(Gamma, cast(C, E0), C) :-
        expression(Gamma, [E0], [D]),
        issuper(C, D),
        \+ C == D, !.

% T-SCast
expression(Gamma, cast(C, E0), C) :-
        expression(Gamma, [E0], [D]),
        \+ issuper(D, C),
        \+ issuper(C, D), !,
        format('\nWarning: Stupid cast.\n\n').

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Metodkontroll

% För lista, samt basfall.
isok([], 'Object') :- !.
isok([], _).
isok([M|Mlist], C) :-
    isok(M, C),
    isok(Mlist, C).

% Om mtype(m,D) = Dmakron → D
isok(method(M, C0, Argument, Ex0), C) :-
        classtable(class(C, D, _, _, _)),
        mtype(M, D, reduction(Dmakron, D0)),
        Dmakron == Cmakron, D0 == C0,
        helper(Argument, Cmakron, Xmakron),
        flatten([Xmakron, 'this'], Xex),
```

```
        flatten([Cmakron, C], Cex),
        helper(Gamma, Cex, Xex),
        expression(Gamma, Ex0, E0),
        issuper(E0, C0).

% ... och annars...
isok(method(M, C0, Argument, Ex0), C) :-
        classtable(class(C, D, _, _, _)),
        \+ mtype(M, D, reduction(_, _)),
        helper(Argument, Cmakron, Xmakron),
        flatten([Xmakron, 'this'], Xex),
        flatten([Cmakron, C], Cex),
        helper(Gamma, Cex, Xex),
        expression(Gamma, Ex0, E0),
        issuper(E0, C0).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Klasskontroll

% Basfall
isok(class(Namn, Superklass, _,
        constructor(Namn, _, [(super, Gmakron)|_]),
     Metoder)) :-
    fields(Superklass, DGmakron),
    helper(DGmakron, _, Gmakron),
    isok(Metoder, Namn).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Hjälpmetoder.
%  Får en lista av par och skapar två listor av dem.
helper([], [], []) :- !.
helper([(A,B)|Rest], [A|ReturnA], [B|ReturnB]) :-
        helper(Rest, ReturnA, ReturnB), !.

%  Får en klasstabell, och två klasser och kollar om C <: D rekrusivt.
issuper([C|CRest], [D|DRest]) :-
        issuper(C, D),
        issuper(CRest, DRest).
issuper(_, 'Object').
issuper(C, C).
issuper(C, D) :-
        classtable(class(C, D, _, _, _)).
issuper('Object', 'Object') :- !.
```

Bilaga B

Trace

```
?- trace, classtable(X), isok(X).
   Call: (8) classtable(_G335) ? creep
   Exit: (8) classtable(class('A', 'B', [], constructor('A', [], [ (super,
   Call: (8) isok(class('A', 'B', [], constructor('A', [], [ (super, [])]),
   Call: (9) fields('B', _L229) ? creep
   Call: (10) classtable(class('B', _G443, _G444, _G445, _G446)) ? creep
   Exit: (10) classtable(class('B', 'Object', [], constructor('B', [], [ (s
   Call: (10) fields('Object', _L249) ? creep
   Exit: (10) fields('Object', []) ? creep
   Call: (10) lists:flatten([[]], _L229) ? creep
   Exit: (10) lists:flatten([[]], []) ? creep
   Exit: (9) fields('B', []) ? creep
   Call: (9) helper([], _L246, []) ? creep
   Exit: (9) helper([], [], []) ? creep
   Call: (9) isok([], 'A') ? creep
   Exit: (9) isok([], 'A') ? creep
   Exit: (8) isok(class('A', 'B', [], constructor('A', [], [ (super, [])]),
X = class('A', 'B', [], constructor('A', [], [ (super, [])]), []) ;
   Redo: (8) classtable(_G335) ? creep
   Exit: (8) classtable(class('B', 'Object', [], constructor('B', [], [ (su
   Call: (8) isok(class('B', 'Object', [], constructor('B', [], [ (super, [
   Call: (9) fields('Object', _L229) ? creep
   Exit: (9) fields('Object', []) ? creep
   Call: (9) helper([], _L246, []) ? creep
   Exit: (9) helper([], [], []) ? creep
   Call: (9) isok([], 'B') ? creep
   Exit: (9) isok([], 'B') ? creep
   Exit: (8) isok(class('B', 'Object', [], constructor('B', [], [ (super, [
X = class('B', 'Object', [], constructor('B', [], [ (super, [])]), []) ;
   Redo: (8) classtable(_G335) ? creep
   Exit: (8) classtable(class('Pair', 'Object', [ ('A', fst), ('B', snd)],
   Call: (8) isok(class('Pair', 'Object', [ ('A', fst), ('B', snd)], constr
   Call: (9) fields('Object', _L213) ? creep
   Exit: (9) fields('Object', []) ? creep
```

```
35     Call: (9) helper([], _L230, []) ? creep
36     Exit: (9) helper([], [], []) ? creep
37     Call: (9) isok([method(setfst, 'Pair', [ ('B', newfst)], new('Pair', [do
38     Call: (10) isok(method(setfst, 'Pair', [ ('B', newfst)], new('Pair', [do
39     Call: (11) classtable(class('Pair', _G529, _G530, _G531, _G532)) ? creep
40     Exit: (11) classtable(class('Pair', 'Object', [ ('A', fst), ('B', snd)],
41     Call: (11) mtype(setfst, 'Object', reduction(_G630, _G631)) ? creep
42     Call: (12) fail ? creep
43     Fail: (12) fail ? creep
44     Fail: (11) mtype(setfst, 'Object', reduction(_G630, _G631)) ? creep
45     Redo: (10) isok(method(setfst, 'Pair', [ ('B', newfst)], new('Pair', [do
46     Call: (11) classtable(class('Pair', _G529, _G530, _G531, _G532)) ? creep
47     Exit: (11) classtable(class('Pair', 'Object', [ ('A', fst), ('B', snd)],
48     Call: (11) mtype(setfst, 'Object', reduction(_G630, _G631)) ? creep
49     Call: (12) fail ? creep
50     Fail: (12) fail ? creep
51     Fail: (11) mtype(setfst, 'Object', reduction(_G630, _G631)) ? creep
52     Call: (11) helper([ ('B', newfst)], _L255, _L256) ? creep
53     Call: (12) helper([], _G631, _G634) ? creep
54     Exit: (12) helper([], [], []) ? creep
55     Exit: (11) helper([ ('B', newfst)], ['B'], [newfst]) ? creep
56     Call: (11) lists:flatten([[newfst], this], _L257) ? creep
57     Exit: (11) lists:flatten([[newfst], this], [newfst, this]) ? creep
58     Call: (11) lists:flatten([['B'], 'Pair'], _L258) ? creep
59     Exit: (11) lists:flatten([['B'], 'Pair'], ['B', 'Pair']) ? creep
60     Call: (11) helper(_L259, ['B', 'Pair'], [newfst, this]) ? creep
61     Call: (12) helper(_G661, ['Pair'], [this]) ? creep
62     Call: (13) helper(_G667, [], []) ? creep
63     Exit: (13) helper([], [], []) ? creep
64     Exit: (12) helper([ ('Pair', this)], ['Pair'], [this]) ? creep
65     Exit: (11) helper([ ('B', newfst), ('Pair', this)], ['B', 'Pair'], [newf
66     Call: (11) expression([ ('B', newfst), ('Pair', this)], new('Pair', [dot
67     Call: (12) fields('Pair', _L275) ? creep
68     Call: (13) classtable(class('Pair', _G673, _G674, _G675, _G676)) ? creep
69     Exit: (13) classtable(class('Pair', 'Object', [ ('A', fst), ('B', snd)],
70     Call: (13) fields('Object', _L297) ? creep
71     Exit: (13) fields('Object', []) ? creep
72     Call: (13) lists:flatten([[ ('A', fst), ('B', snd)]], _L275) ? creep
73     Exit: (13) lists:flatten([[ ('A', fst), ('B', snd)]], [ ('A', fst), ('B'
74     Exit: (12) fields('Pair', [ ('A', fst), ('B', snd)]) ? creep
75     Call: (12) helper([ ('A', fst), ('B', snd)], _L276, _L295) ? creep
76     Call: (13) helper([ ('B', snd)], _G784, _G787) ? creep
77     Call: (14) helper([], _G790, _G793) ? creep
78     Exit: (14) helper([], [], []) ? creep
79     Exit: (13) helper([ ('B', snd)], ['B'], [snd]) ? creep
80     Exit: (12) helper([ ('A', fst), ('B', snd)], ['A', 'B'], [fst, snd]) ? c
81     Call: (12) expression([ ('B', newfst), ('Pair', this)], [dot(new('Pair',
82     Call: (13) lists:member((_G795, dot(new('Pair', [cast('A', newfst), dot(
83     Fail: (13) lists:member((_G795, dot(new('Pair', [cast('A', newfst), dot(
```

```
 84     Redo: (12) expression([ ('B', newfst), ('Pair', this)], [dot(new('Pair',
 85     Call: (13) expression([ ('B', newfst), ('Pair', this)], dot(new('Pair',
 86     Call: (14) expression([ ('B', newfst), ('Pair', this)], [new('Pair', [ca
 87     Call: (15) lists:member((_G801, new('Pair', [cast('A', newfst), dot(this
 88     Fail: (15) lists:member((_G801, new('Pair', [cast('A', newfst), dot(this
 89     Redo: (14) expression([ ('B', newfst), ('Pair', this)], [new('Pair', [ca
 90     Call: (15) expression([ ('B', newfst), ('Pair', this)], new('Pair', [cas
 91     Call: (16) fields('Pair', _L366) ? creep
 92     Call: (17) classtable(class('Pair', _G805, _G806, _G807, _G808)) ? creep
 93     Exit: (17) classtable(class('Pair', 'Object', [ ('A', fst), ('B', snd)],
 94     Call: (17) fields('Object', _L388) ? creep
 95     Exit: (17) fields('Object', []) ? creep
 96     Call: (17) lists:flatten([[ ('A', fst), ('B', snd)]], _L366) ? creep
 97     Exit: (17) lists:flatten([[ ('A', fst), ('B', snd)]], [ ('A', fst), ('B'
 98     Exit: (16) fields('Pair', [ ('A', fst), ('B', snd)]) ? creep
 99     Call: (16) helper([ ('A', fst), ('B', snd)], _L367, _L386) ? creep
100     Call: (17) helper([ ('B', snd)], _G916, _G919) ? creep
101     Call: (18) helper([], _G922, _G925) ? creep
102     Exit: (18) helper([], [], []) ? creep
103     Exit: (17) helper([ ('B', snd)], ['B'], [snd]) ? creep
104     Exit: (16) helper([ ('A', fst), ('B', snd)], ['A', 'B'], [fst, snd]) ? c
105     Call: (16) expression([ ('B', newfst), ('Pair', this)], [cast('A', newfs
106     Call: (17) lists:member((_G927, cast('A', newfst)), [ ('B', newfst), ('P
107     Fail: (17) lists:member((_G927, cast('A', newfst)), [ ('B', newfst), ('P
108     Redo: (16) expression([ ('B', newfst), ('Pair', this)], [cast('A', newfs
109     Call: (17) expression([ ('B', newfst), ('Pair', this)], cast('A', newfst
110     Call: (18) expression([ ('B', newfst), ('Pair', this)], [newfst], [_G933
111     Call: (19) lists:member((_G933, newfst), [ ('B', newfst), ('Pair', this)
112     Exit: (19) lists:member(('B', newfst), [ ('B', newfst), ('Pair', this)])
113     Call: (19) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
114     Exit: (19) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
115     Exit: (18) expression([ ('B', newfst), ('Pair', this)], [newfst], ['B'])
116     Call: (18) issuper('B', 'A') ? creep
117     Call: (19) classtable(class('B', 'A', _G941, _G942, _G943)) ? creep
118     Fail: (19) classtable(class('B', 'A', _G941, _G942, _G943)) ? creep
119     Fail: (18) issuper('B', 'A') ? creep
120     Redo: (19) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
121     Fail: (19) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
122     Redo: (19) lists:member((_G933, newfst), [ ('B', newfst), ('Pair', this)
123     Fail: (19) lists:member((_G933, newfst), [ ('B', newfst), ('Pair', this)
124     Redo: (18) expression([ ('B', newfst), ('Pair', this)], [newfst], [_G933
125     Call: (19) expression([ ('B', newfst), ('Pair', this)], newfst, _G933) ?
126     Fail: (19) expression([ ('B', newfst), ('Pair', this)], newfst, _G933) ?
127     Redo: (18) expression([ ('B', newfst), ('Pair', this)], [newfst], [_G933
128     Fail: (18) expression([ ('B', newfst), ('Pair', this)], [newfst], [_G933
129     Redo: (17) expression([ ('B', newfst), ('Pair', this)], cast('A', newfst
130     Call: (18) expression([ ('B', newfst), ('Pair', this)], [newfst], [_G933
131     Call: (19) lists:member((_G933, newfst), [ ('B', newfst), ('Pair', this)
132     Exit: (19) lists:member(('B', newfst), [ ('B', newfst), ('Pair', this)])
```

```
133     Call: (19) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
134     Exit: (19) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
135     Exit: (18) expression([ ('B', newfst), ('Pair', this)], [newfst], ['B'])
136     Call: (18) issuper('A', 'B') ? creep
137     Call: (19) classtable(class('A', 'B', _G941, _G942, _G943)) ? creep
138     Exit: (19) classtable(class('A', 'B', [], constructor('A', [], [ (super,
139     Exit: (18) issuper('A', 'B') ? creep
140     Call: (18) 'A'=='B' ? creep
141     Fail: (18) 'A'=='B' ? creep
142     Exit: (17) expression([ ('B', newfst), ('Pair', this)], cast('A', newfst
143     Call: (17) expression([ ('B', newfst), ('Pair', this)], [dot(this, snd)]
144     Call: (18) lists:member((_G955, dot(this, snd)), [ ('B', newfst), ('Pair
145     Fail: (18) lists:member((_G955, dot(this, snd)), [ ('B', newfst), ('Pair
146     Redo: (17) expression([ ('B', newfst), ('Pair', this)], [dot(this, snd)]
147     Call: (18) expression([ ('B', newfst), ('Pair', this)], dot(this, snd),
148     Call: (19) expression([ ('B', newfst), ('Pair', this)], [this], [_G961])
149     Call: (20) lists:member((_G961, this), [ ('B', newfst), ('Pair', this)])
150     Exit: (20) lists:member(('Pair', this), [ ('B', newfst), ('Pair', this)]
151     Call: (20) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
152     Exit: (20) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
153     Exit: (19) expression([ ('B', newfst), ('Pair', this)], [this], ['Pair']
154     Call: (19) fields('Pair', _L434) ? creep
155     Call: (20) classtable(class('Pair', _G968, _G969, _G970, _G971)) ? creep
156     Exit: (20) classtable(class('Pair', 'Object', [ ('A', fst), ('B', snd)],
157     Call: (20) fields('Object', _L531) ? creep
158     Exit: (20) fields('Object', []) ? creep
159     Call: (20) lists:flatten([[ ('A', fst), ('B', snd)]], _L434) ? creep
160     Exit: (20) lists:flatten([[ ('A', fst), ('B', snd)]], [ ('A', fst), ('B'
161     Exit: (19) fields('Pair', [ ('A', fst), ('B', snd)]) ? creep
162     Call: (19) helper([ ('A', fst), ('B', snd)], _L435, _L436) ? creep
163     Call: (20) helper([ ('B', snd)], _G1079, _G1082) ? creep
164     Call: (21) helper([], _G1085, _G1088) ? creep
165     Exit: (21) helper([], [], []) ? creep
166     Exit: (20) helper([ ('B', snd)], ['B'], [snd]) ? creep
167     Exit: (19) helper([ ('A', fst), ('B', snd)], ['A', 'B'], [fst, snd]) ? c
168     Call: (19) lists:member(_G955, ['A', 'B']) ? creep
169     Exit: (19) lists:member('A', ['A', 'B']) ? creep
170     Call: (19) lists:member(snd, [fst, snd]) ? creep
171     Exit: (19) lists:member(snd, [fst, snd]) ? creep
172     Exit: (18) expression([ ('B', newfst), ('Pair', this)], dot(this, snd),
173     Call: (18) expression([ ('B', newfst), ('Pair', this)], [], _G956) ? cre
174     Exit: (18) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
175     Exit: (17) expression([ ('B', newfst), ('Pair', this)], [dot(this, snd)]
176     Exit: (16) expression([ ('B', newfst), ('Pair', this)], [cast('A', newfs
177     Call: (16) issuper(['A', 'A'], ['A', 'B']) ? creep
178     Call: (17) issuper('A', 'A') ? creep
179     Exit: (17) issuper('A', 'A') ? creep
180     Call: (17) issuper(['A'], ['B']) ? creep
181     Call: (18) issuper('A', 'B') ? creep
```

```
182     Call: (19) classtable(class('A', 'B', _G1092, _G1093, _G1094)) ? creep
183     Exit: (19) classtable(class('A', 'B', [], constructor('A', [], [ (super,
184     Exit: (18) issuper('A', 'B') ? creep
185     Call: (18) issuper([], []) ? creep
186     Exit: (18) issuper([], []) ? creep
187     Exit: (17) issuper(['A'], ['B']) ? creep
188     Exit: (16) issuper(['A', 'A'], ['A', 'B']) ? creep
189     Exit: (15) expression([ ('B', newfst), ('Pair', this)], new('Pair', [cas
190     Call: (15) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
191     Exit: (15) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
192     Exit: (14) expression([ ('B', newfst), ('Pair', this)], [new('Pair', [ca
193     Call: (14) fields('Pair', _L321) ? creep
194     Call: (15) classtable(class('Pair', _G1107, _G1108, _G1109, _G1110)) ? c
195     Exit: (15) classtable(class('Pair', 'Object', [ ('A', fst), ('B', snd)],
196     Call: (15) fields('Object', _L383) ? creep
197     Exit: (15) fields('Object', []) ? creep
198     Call: (15) lists:flatten([[ ('A', fst), ('B', snd)]], _L321) ? creep
199     Exit: (15) lists:flatten([[ ('A', fst), ('B', snd)]], [ ('A', fst), ('B'
200     Exit: (14) fields('Pair', [ ('A', fst), ('B', snd)]) ? creep
201     Call: (14) helper([ ('A', fst), ('B', snd)], _L322, _L323) ? creep
202     Call: (15) helper([ ('B', snd)], _G1218, _G1221) ? creep
203     Call: (16) helper([], _G1224, _G1227) ? creep
204     Exit: (16) helper([], [], []) ? creep
205     Exit: (15) helper([ ('B', snd)], ['B'], [snd]) ? creep
206     Exit: (14) helper([ ('A', fst), ('B', snd)], ['A', 'B'], [fst, snd]) ? c
207     Call: (14) lists:member(_G795, ['A', 'B']) ? creep
208     Exit: (14) lists:member('A', ['A', 'B']) ? creep
209     Call: (14) lists:member(fst, [fst, snd]) ? creep
210     Exit: (14) lists:member(fst, [fst, snd]) ? creep
211     Exit: (13) expression([ ('B', newfst), ('Pair', this)], dot(new('Pair',
212     Call: (13) expression([ ('B', newfst), ('Pair', this)], [dot(this, snd)]
213     Call: (14) lists:member((_G1229, dot(this, snd)), [ ('B', newfst), ('Pai
214     Fail: (14) lists:member((_G1229, dot(this, snd)), [ ('B', newfst), ('Pai
215     Redo: (13) expression([ ('B', newfst), ('Pair', this)], [dot(this, snd)]
216     Call: (14) expression([ ('B', newfst), ('Pair', this)], dot(this, snd),
217     Call: (15) expression([ ('B', newfst), ('Pair', this)], [this], [_G1235]
218     Call: (16) lists:member((_G1235, this), [ ('B', newfst), ('Pair', this)]
219     Exit: (16) lists:member(('Pair', this), [ ('B', newfst), ('Pair', this)]
220     Call: (16) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
221     Exit: (16) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
222     Exit: (15) expression([ ('B', newfst), ('Pair', this)], [this], ['Pair']
223     Call: (15) fields('Pair', _L343) ? creep
224     Call: (16) classtable(class('Pair', _G1242, _G1243, _G1244, _G1245)) ? c
225     Exit: (16) classtable(class('Pair', 'Object', [ ('A', fst), ('B', snd)],
226     Call: (16) fields('Object', _L440) ? creep
227     Exit: (16) fields('Object', []) ? creep
228     Call: (16) lists:flatten([[ ('A', fst), ('B', snd)]], _L343) ? creep
229     Exit: (16) lists:flatten([[ ('A', fst), ('B', snd)]], [ ('A', fst), ('B'
230     Exit: (15) fields('Pair', [ ('A', fst), ('B', snd)]) ? creep
```

```
231     Call: (15) helper([ ('A', fst), ('B', snd)], _L344, _L345) ? creep
232     Call: (16) helper([ ('B', snd)], _G1353, _G1356) ? creep
233     Call: (17) helper([], _G1359, _G1362) ? creep
234     Exit: (17) helper([], [], []) ? creep
235     Exit: (16) helper([ ('B', snd)], ['B'], [snd]) ? creep
236     Exit: (15) helper([ ('A', fst), ('B', snd)], ['A', 'B'], [fst, snd]) ? c
237     Call: (15) lists:member(_G1229, ['A', 'B']) ? creep
238     Exit: (15) lists:member('A', ['A', 'B']) ? creep
239     Call: (15) lists:member(snd, [fst, snd]) ? creep
240     Exit: (15) lists:member(snd, [fst, snd]) ? creep
241     Exit: (14) expression([ ('B', newfst), ('Pair', this)], dot(this, snd),
242     Call: (14) expression([ ('B', newfst), ('Pair', this)], [], _G1230) ? cr
243     Exit: (14) expression([ ('B', newfst), ('Pair', this)], [], []) ? creep
244     Exit: (13) expression([ ('B', newfst), ('Pair', this)], [dot(this, snd)]
245     Exit: (12) expression([ ('B', newfst), ('Pair', this)], [dot(new('Pair',
246     Call: (12) issuper(['A', 'A'], ['A', 'B']) ? creep
247     Call: (13) issuper('A', 'A') ? creep
248     Exit: (13) issuper('A', 'A') ? creep
249     Call: (13) issuper(['A'], ['B']) ? creep
250     Call: (14) issuper('A', 'B') ? creep
251     Call: (15) classtable(class('A', 'B', _G1366, _G1367, _G1368)) ? creep
252     Exit: (15) classtable(class('A', 'B', [], constructor('A', [], [ (super,
253     Exit: (14) issuper('A', 'B') ? creep
254     Call: (14) issuper([], []) ? creep
255     Exit: (14) issuper([], []) ? creep
256     Exit: (13) issuper(['A'], ['B']) ? creep
257     Exit: (12) issuper(['A', 'A'], ['A', 'B']) ? creep
258     Exit: (11) expression([ ('B', newfst), ('Pair', this)], new('Pair', [dot
259     Call: (11) issuper('Pair', 'Pair') ? creep
260     Exit: (11) issuper('Pair', 'Pair') ? creep
261     Exit: (10) isok(method(setfst, 'Pair', [ ('B', newfst)], new('Pair', [do
262     Call: (10) isok([], 'Pair') ? creep
263     Exit: (10) isok([], 'Pair') ? creep
264     Exit: (9) isok([method(setfst, 'Pair', [ ('B', newfst)], new('Pair', [do
265     Exit: (8) isok(class('Pair', 'Object', [ ('A', fst), ('B', snd)], constr
266  X = class('Pair', 'Object', [ ('A', fst), ('B', snd)], constructor('Pair',
267     Redo: (11) issuper('Pair', 'Pair') ? creep
268     Call: (12) classtable(class('Pair', 'Pair', _G1382, _G1383, _G1384)) ? c
269     Fail: (12) classtable(class('Pair', 'Pair', _G1382, _G1383, _G1384)) ? c
270     Fail: (9) isok([method(setfst, 'Pair', [ ('B', newfst)], new('Pair', [do
271     Fail: (8) isok(class('Pair', 'Object', [ ('A', fst), ('B', snd)], constr
272  false.
```