Statische Typableitung für die optional typisierte Sprache Dart

Thomas S. Heinze, Anders Møller und Fabio Strocco

Aarhus Universitet, Institut for Datalogi, Åbogade 34, DK-8200 Aarhus N, Denmark
[t.heinze,amoeller,fstrocco]@cs.au.dk

1 Einführung und Motivation

Statisch typisierte Sprachen wie Java erleichtern die Programmierung durch die Möglichkeit, Programmierfehler bereits zur Übersetzungszeit aufzufinden, zu beheben und für die eigentliche Programmausführung weitestgehend auszuschließen. Zudem unterstützen die im Programm enthaltenen Typinformationen die Umsetzung hilfreicher Programmierwerkzeuge, wie beispielsweise Autovervollständigung und automatische Programmrestrukturierung. Gleichzeitig lassen sich die Typinformationen auch für die Übersetzung selbst, insbesondere zur Laufzeitoptimierung ausnutzen. Diesen Vorteilen statisch typisierter Sprachen stehen jedoch Einschränkungen hinsichtlich der Flexibilität der Programmierung gegenüber. Im Gegensatz dazu bieten die häufig zur Webprogrammierung genutzten dynamisch typisierten Programmiersprachen, namentlich JavaScript, eine höhere Flexibilität. Verbunden mit dieser ist nun aber das mögliche Auftreten von Typverletzungen zur Programmlaufzeit, sowie eine nur eingeschränkte Unterstützung von Programmierwerkzeugen und Programmoptimierungen.

Mit der Zunahme der Komplexität von Webanwendungen ergibt sich für die Webprogrammierung mit dynamischen Sprachen der Wunsch, auf die Vorteile statischer Typen zurückzugreifen, ohne dabei jedoch auf die Flexibilität der dynamischen Sprachen zu verzichten. Als ein Ansatz zur Verbindung der Vorteile von statischer und dynamischer Typisierung wurde die *optionale Typisierung* vorgeschlagen [2]. Diese erlaubt dem Programmierer selbst zu entscheiden, welche Teile eines Programms mit Typinformationen versehen werden, also statisch typisiert sind, und für welche Programmteile die Typen erst zur Laufzeit bestimmt werden sollen. Eine Umsetzung für ein solch optionales Typsystem bietet die Programmiersprache *Dart* [3]. In einem Dart-Programm kann einer Variablen entweder ein statischer Typ annotiert werden, oder aber die Variable wird als dynamisch typisiert ausgezeichnet und damit auch als zuweisungskompatibel zu jeder anderen Variablen, unabhängig von deren Typ. Verbunden mit der Möglichkeit zur selektiven Typisierung ist somit auch eine Lockerung der Typsicherheit eines Programms, da sich für die dynamisch typisierten Programmteile entsprechende Garantien nicht ohne Weiteres angeben lassen. Im Unterschied zu dem vergleichbaren Ansatz der graduellen Typisierung werden ferner auch keinerlei Garantien (etwa das Blame-Theorem [6]) zur Typsicherheit für die statisch typisierten Programmteile allein gegeben.

```
class Pair<E> {
    E left, right;
    void apply(Function f) {
        f(right); f(left);
    }
}
...
Pair<String> p =
        new Pair<String>();
//      new Pair<dynamic>();
Pair<Object> a = p;
a.left = 1;
print(p.left.length);
```

a) Kovarianz generischer Typen

```
class Type {}
class Subtype extends Type {
    bool field;
}
...
Pair<Type> p = new Pair<Type>();
p.right = new Subtype();
p.right.field = true;
p.left = new Subtype();
p.left.field = false;
p.apply((var e) {
    print(e.field);
});
```

b) Inkonsistente Typannotation

Abb. 1. Zwei Beispiele zur Verwendung generischer Typen in Dart: Sowohl in a) als auch in b) wird von der bestehenden Typprüfung kein Fehler angezeigt.

Um dennoch bereits während der Übersetzungszeit Aussagen zur Typsicherheit eines Dart-Programms treffen zu können, schlagen wir die Anwendung einer statischen Programmanalyse zur Typinferenz vor. Mit Hilfe einer solchen Analyse können sichere Abschätzungen zu den möglichen Typen in den dynamisch typisierten Programmteilen abgeleitet werden, die anschließend die Grundlage zur Prüfung der Typsicherheit bilden. Die Spracheigenschaften von Dart stellen dabei jedoch verschiedene Herausforderungen an Entwurf und Konzeption der Analyse, insbesondere im Hinblick auf die notwendigerweise *sichere* Ableitung der Typen, auf die im Folgenden überblicksweise eingegangen werden soll.

2 Optionale Typisierung in Dart

Die objektorientierte Programmiersprache Dart [3] wurde als Alternative zu JavaScript eingeführt, mit dem Ziel die Programmierung von Webanwendungen durch den optionalen Einsatz statischer Typen zu unterstützen. Dart weist Ähnlichkeiten mit statisch typisierten Sprachen auf, so erinnern Syntax und Klassenkonzept an Java. Im Gegenzug erlaubt der auch als Vorgabe verwendete Typ `dynamic` eine dynamische Programmierung analog JavaScript.

Dart-Programme lassen sich auf einer virtuellen Maschine ausführen oder werden nach JavaScript übersetzt. In Übereinstimmung mit dem Konzept der optionalen Typisierung wird dabei zwischen zwei verschiedenen Ausführungsmodi unterschieden. Im ersten *Produktionsmodus (Production Mode)* erfolgt die Programmausführung vollständig dynamisch und somit unabhängig von den im Programm deklarierten Typen. Im zweiten *Entwicklungsmodus (Checked Mode)* erfolgen hingegen eine Reihe von Typprüfungen auf Grundlage der vorhandenen Typannotationen. Zwischen zwei typbezogenen Laufzeitfehlern kann unterschieden werden. Einerseits führt der Zugriff auf eine nicht vorhandene Metho-

de beziehungsweise auf ein nicht vorhandenes Feld zu einem Fehler (*Message Not Understood*). Andererseits liegt eine Typverletzung (*Subtype Violation*) vor, falls beispielsweise einem statisch typisierten Feld ein Wert außerhalb von dessen Definitionsbereich zugewiesen wird. Können erstgenannte Fehler in beiden Ausführungsmodi auftreten, beruhen letztgenannte Fehler auf den annotierten statischen Typen und sind somit auf den Entwicklungsmodus begrenzt.

Das Typsystem von Dart ist dabei bewusst als nicht korrekt entworfen worden [3]. So können etwa generische Klassen kovariant sein (vergleiche Reihungstypen in Java). In Abbildung 1 ist auf der linken Seite ein Programm mit einer Zuweisung zwischen kovarianten generischen Typen (`Pair<Object> a = p`) dargestellt. Offenbar führt die Programmausführung zu einem Laufzeitfehler, der aber von der bestehenden statischen Typprüfung trotz vollständig deklarierter statischer Typen nicht identifiziert wird. Im Produktionsmodus kommt es zum Fehler beim Zugriff auf das Feld `length` in der letzten Anweisung (*Message Not Understood*), im Entwicklungsmodus wird als Ursache dafür zumindest eine Typverletzung in Zuweisung `a.left = 1` signalisiert. Wird das Programm wie im Kommentar angegeben modifiziert, indem der dynamische Typ als Typargument verwendet wird, ist selbst das nicht mehr möglich.

3 Statische Typableitung für Dart

Eine Analyse zur statischen Typableitung muss diese Eigenschaften der Sprache Dart berücksichtigen. Insbesondere stellt sich die grundlegende Frage nach dem Umgang mit Typannotationen. Werden diese ignoriert, ergibt sich eine Typableitung in Übereinstimmung mit der Ausführung eines Programms im Produktionsmodus. Voraussetzung dafür ist jedoch das Vorliegen des vollständigen Programms. Sonst kann beispielsweise für die Parameter einer öffentlichen Methode keine Aussage zu den möglichen Typen getroffen werden. Entsprechend schwer gestaltet sich in diesem Fall die Analyse von Bibliotheken oder Programmteilstücken (siehe auch [1,4]). Im anderen Fall führt die Berücksichtigung von Typannotationen zu einer modularen Analyse. Anhand der für die Programmschnittstelle deklarierten statischen Typen sind nun Aussagen zu den einlaufenden Typen möglich – unter Annahme der Ausführung im Entwicklungsmodus, um die Korrektheit der Typannotationen zu gewährleisten. Gleichzeitig ist auch eine Reduktion des Analyseaufwands in Abhängigkeit vom Vorhandensein statischer Typannotationen zu erwarten, da zur Abschätzung des Typs eines Elements nun nicht mehr aufwändig der Objektfluss nachvollzogen werden muss, sondern direkt auf den deklarierten Typ zurückgegriffen werden kann.

Die von uns vorgeschlagene statische Analyse zur Typinferenz für Dart unterstützt beide Ansätze und damit sowohl den Produktions- als auch den Entwicklungsmodus. Die Typableitung beruht auf dem Verfahren der Zeigeranalyse [5], nur dass anstatt einer Überabschätzung für die Ziele von Zeigern eine Überabschätzung für die möglichen Typen abgeleitet wird. Zu diesem Zweck werden den Ausdrücken und Elementen eines Dart-Programms Typvariablen zugeordnet, die auf Mengen von Typen verweisen (vergleiche Funktion $[\![]\!]$ in

$Type = (ConcreteType \cup DeclaredType \cup ExternType) \setminus \{dynamic\}$
$[\![]\!]: Expression \cup Element \to \mathcal{P}(Type)$
$type: Element \to DeclaredType \cup ExternType$

Zuweisung `x=e`:	$type(\texttt{x}) \neq dynamic \Rightarrow \{type(\texttt{x})\} \subseteq [\![\texttt{x}]\!]$
	$type(\texttt{x}) = dynamic \Rightarrow [\![\texttt{e}]\!] \subseteq [\![\texttt{x}]\!]$
	$t \in [\![\texttt{e}]\!] \wedge t \in ExternType \Rightarrow \{t\} \subseteq [\![\texttt{x}]\!]$
Instanz `new T()`:	$\{T\} \subseteq [\![\texttt{new T()}]\!]$
`literal` vom Typ T:	$\{T\} \subseteq [\![\texttt{literal}]\!]$
Feldzugriff `x.f`:	$t \in [\![\texttt{x}]\!] \wedge t' \leq: t \wedge type(t'.\texttt{f}) = dynamic \Rightarrow [\![t'.\texttt{f}]\!] \subseteq [\![\texttt{x.f}]\!]$
	$t \in [\![\texttt{x}]\!] \wedge t' \leq: t \wedge type(t'.\texttt{f}) \neq dynamic \Rightarrow \{type(t'.\texttt{f})\} \subseteq [\![\texttt{x.f}]\!]$
	$t \in [\![\texttt{x}]\!] \wedge t' \leq: t \wedge s \in [\![t'.\texttt{f}]\!] \wedge s \in ExternType \Rightarrow \{s\} \subseteq [\![\texttt{x.f}]\!]$
	$t \in [\![\texttt{x}]\!] \wedge t \in ExternType \Rightarrow \{Object^E\} \subseteq [\![\texttt{x.f}]\!]$
Feldzugriff `x.f=e`:	$t \in [\![\texttt{x}]\!] \wedge t' \leq: t \wedge type(t'.\texttt{f}) = dynamic \Rightarrow [\![\texttt{e}]\!] \subseteq [\![t'.\texttt{f}]\!]$
	$t \in [\![\texttt{x}]\!] \wedge t' \leq: t \wedge s \in [\![\texttt{e}]\!] \wedge s \in ExternType \Rightarrow \{s\} \subseteq [\![t'.\texttt{f}]\!]$

Abb. 2. Auswahl von (vereinfachten) Regeln zur statischen Typableitung

Abbildung 2). Weiterhin werden Regeln zwischen den Typvariablen in Form von Teilmengenbeziehungen definiert. Als Lösung des dadurch charakterisierten Regelsystems ergibt sich eine konservative Abschätzung zu den Typen im Programm. Wie in Abbildung 2 ersichtlich, wird dabei zwischen *konkreten Typen* $t \in ConcreteType$ und *Deklarationstypen* $t \in DeclaredType$ unterschieden, da letztere auch alle im analysierten Programm deklarierten Untertypen $t' <: t$ umfassen. Eine dritte Kategorie bilden die *externen Typen*, wobei ein externer Typ $t \in ExternType$ ebenfalls seine Untertypen $t' <: t$ umfasst, nur das im Gegensatz zu den Deklarationstypen auch unbekannte Untertypen dazu zählen. Auf diese Weise sollen die von außerhalb einlaufenden Typen repräsentiert sein. Dies ist insofern wichtig, als dass die Methoden und Felder eines Typs in den von ihm abgeleiteten Untertypen überschrieben werden können. Da sich für die überschriebenen Felder und Methoden der dynamische Typ deklarieren lässt, folgt, dass etwa für den Feldzugriff über einen externen und damit unbekannten Typ keine Aussagen mehr zum Feldtyp möglich sind. Dies wird in Abbildung 2 durch gesonderte Regeln für den Fluss externer Typen modelliert.

Zur Unterstützung beider Ausführungsmodi sind die Regeln zur Typableitung in Abbildung 2 durch die Funktion *type* parametrisiert. Diese Funktion bildet für ein Programmelement, in Abhängigkeit von dessen deklariertem Typ, auf einen Deklarationstyp oder einen externen Typ ab. Grundlegende Idee ist nun, im Fall des Produktionsmodus für jedes Programmelement e unabhängig von dessen tatsächlich deklarierten Typ $type(e) = dynamic$ zu setzen. Im Fall des Entwicklungsmodus wird hingegen $type(e)$ auf den tatsächlich deklarierten Typ, oder für ein öffentliches Element auf den diesem entsprechenden externen Typ gesetzt. Grundsätzlich sind weitere Parametrisierungen möglich, etwa eine in der Typannotationen nur für öffentliche Elemente berücksichtigt werden.

Die Regeln entsprechen für den Produktionsmodus den Erwartungen (es gilt überall $type(e) = dynamic$). Im Wesentlichen werden die für Instanziierungsausdrücke und Literale erzeugten Typen, analog einer Zeigeranalyse, entlang des Datenflusses propagiert. Angewendet auf das Programmbeispiel auf der linken Seite von Abbildung 1 ergibt sich unter anderem: $[\![\texttt{new Pair<String>()}]\!] \subseteq [\![p]\!]$, $\{Pair^C\} \subseteq [\![\texttt{new Pair<String>()}]\!]$, $t \in [\![\texttt{a}]\!] \Rightarrow [\![\texttt{1}]\!] \subseteq [\![t\texttt{.left}]\!]$, $[\![\texttt{p}]\!] \subseteq [\![\texttt{a}]\!]$, $\{int^C\} \subseteq [\![\texttt{1}]\!]$, $t \in [\![\texttt{p}]\!] \Rightarrow [\![t\texttt{.left}]\!] \subseteq [\![\texttt{p.left}]\!]$. Die Lösung $[\![\texttt{p.left}]\!] = \{int^C\}$ erlaubt der folgenden Typprüfung für `print(p.left.length)` einen Fehler zu identifizieren, da der konkrete Typ int^C kein Feld `length` definiert (zur besseren Unterscheidung der Typkategorien verwenden wir Hochstellungen ^{C},^{D} ,^{E}).

Interessanter ist die Betrachtung der Typableitung für den Entwicklungsmodus. Vereinfachend soll für das Beispiel aus Abbildung 1 im Folgenden angenommen werden, dass keine öffentlichen Elemente definiert, und damit keine externen Typen abzuleiten sind. Wird, wie oben bereits angesprochen, in einem ersten Ansatz für jedes definierte Element e die Funktion $type(e)$ auf den jeweiligen deklarierten Typ gesetzt, ergibt sich für das Beispielprogramm unter anderem: $\{Pair\texttt{<}Object^D\texttt{>}^D\} \subseteq [\![\texttt{a}]\!]$, $\{Object^D\} \subseteq [\![\texttt{a.left}]\!]$. Mit dieser Lösung kann der sich für das Beispiel im Entwicklungsmodus ergebende Fehler (*Subtype Violation*) jedoch nicht nachvollzogen werden. Grund hierfür ist in der Kovarianz generischer Typen zu suchen (siehe auch Abschnitt 2), die für die Typannotation `Pair<Object>` von `a` berücksichtigt werden muss. Gleiches gilt, falls das modifizierte Beispiel betrachtet wird, indem `dynamic` an Stelle von `String` als Typargument auftritt. Auch dann kann der Typannotation, in diesem Fall `Pair<String>` von `p`, nicht vertraut werden, wobei der Grund nun nicht in der Kovarianz generischer Typen sondern im dynamischen Typargument liegt.

Um sichere Ergebnisse auch unter Berücksichtigung der optionalen Typisierung und des inkorrekten Typsystems von Dart zu ermöglichen, erfolgt eine Verfeinerung mit Hilfe einer weiteren Parametrisierungsfunktion *sound*. Anstatt für ein Programmelement e mit deklariertem statischen Typ ($type(e) \neq dynamic$) einfach nur diesen Typ zu nutzen, werden für "unsichere" Typen zusätzlich auch die entlang des Datenfluss propagierten Typen berücksichtigt, analog dem Vorgehen für den dynamischen Typ. Als Bedingung für die entsprechenden Regeln ergibt sich somit $type(e) = dynamic \vee \neg sound(type(e))$. Für das Beispielprogramm ergibt sich dieses Mal: $\{Pair\texttt{<}String^D\texttt{>}^C\} \subseteq [\![\texttt{new Pair<String>()}]\!]$, $[\![\texttt{p}]\!] \subseteq [\![\texttt{a}]\!]$, $[\![\texttt{new Pair<String>()}]\!] \subseteq [\![\texttt{p}]\!], \{Pair\texttt{<}Object^D\texttt{>}^D\} \subseteq [\![\texttt{a}]\!], \{String^D\} \subseteq [\![\texttt{a.left}]\!]$ $\{Object^D\} \subseteq [\![\texttt{a.left}]\!]$, $\{Pair\texttt{<}String^D\texttt{>}^D\} \subseteq [\![\texttt{p}]\!]$. Auf Grundlage dieser Typabschätzung kann die Typverletzung in `a.left = 1` identifiziert werden und analog auch der sich für das modifizierte Beispiel ergebende Laufzeitfehler.

Denkbar wäre hier ebenfalls gewesen, auf die für den Produktionsmodus abgeleiteten Typen zurückzugreifen, da diese zum gleichen Ergebnis der Typprüfung geführt hätten. Jedoch ist das nicht immer der Fall. Auf der rechten Seite von Abbildung 1 ist ein weiteres Dart-Programm angegeben. Betrachtet werden soll darin der Feldzugriff `e.field`, dabei handelt es sich bei `e` um einen dynamisch typisierten Parameter einer anonymen Funktion, die der Methode `apply` übergeben wird. Für dieses Beispiel tritt sowohl im Entwicklungs- als auch im Produktionsmodus kein Laufzeitfehler auf, da der Parameter `e` jeweils auf eine

Instanz der Klasse `Subtype` verweist. Der betrachtete Feldzugriff `e.field` erfolgt somit auf definierten Feldern. Allerdings wirft die gewählte Typannotation im Beispiel Fragen auf. Zwar entspricht der konkrete Typ der Felder `left` und `right` dem Typ `Subtype`, als deklarierter Typ ergibt sich aber über das Typargument der Instanz `Pair<Type>` der Typ `Type`. Da für diesen das Feld `field` nicht definiert ist, liegt eine zumindest im statischen Sinn inkonsistente Typdeklaration vor, die lediglich durch Verwendung des dynamischen Typs für den Parameter `e` geheilt wird. Situationen wie diese stellen keine unmittelbaren Fehler dar, können aber auf Entwurfsschwächen in einem Programm hinweisen und sollten sich daher ebenfalls identifizieren lassen. Dies ist aber nur möglich, falls die deklarierten Typen in die Typableitung mit einbezogen werden.

4 Zusammenfassung

Der vorliegende Beitrag beschreibt überblicksweise eine Analyse zur statischen Typableitung für die Sprache Dart. Die Eigenschaften von Dart, insbesondere die Möglichkeit zur optionalen Typisierung und das bewusst inkorrekt definierte Typsystem müssen bei Entwurf und Konzeption berücksichtigt werden und führen für die beschriebene Analyse zu einem parametrisierten Entwurf. Dieser gestattet sowohl die sichere Typableitung für ein vollständig vorliegendes Dart-Programm unter Annahme der Programmausführung im Produktionsmodus, als auch eine modulare Typableitung bei Berücksichtigung der in einem Programm enthaltenen Typannotationen analog des Entwicklungsmodus.

Literatur

[1] Allen, Nicholas ; Krishnan, Padmanabhan ; Scholz, Bernhard: Combining Type-Analysis with Points-To Analysis for Analyzing Java Library Source-Code. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, June 14, 2015, Portland, OR, USA*, ACM, 2015, S. 13–18

[2] Bracha, Gilad ; Griswold, David: Strongtalk: Typechecking Smalltalk in a Production Environment. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference, Washington, DC, USA, September 26 – October 1, 1993, Proceedings*, ACM, 1993, S. 215–230

[3] *Dart Programming Language Specification.* Ecma Intl., Standard ECMA-408, 2015

[4] Rastogi, Aseem ; Chaudhuri, Avik ; Hosmer, Basil: The Ins and Outs of Gradual Type Inference. In: *POPL'12, Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 25-27, 2012, Philadelphia, PA, USA*, ACM, 2012, S. 481–494

[5] Sridharan, Manu ; Chandra, Satish ; Dolby, Julian ; Fink, Stephen J. ; Yahav, Eran: Alias Analysis for Object-Oriented Programs. In: *Aliasing in Object-Oriented Programming.* Springer, 2013 (LNCS 7850), S. 196–232

[6] Wadler, Philip ; Findler, Robert B.: Well-Typed Programs Can't Be Blamed. In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009, Proceedings*, Springer, 2009 (LNCS 5502), S. 1–16