Varianten der modularen Typableitung für Dart

Thomas S. Heinze[1], Anders Møller[2] und Fabio Strocco[2]

[1] Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, D-07743 Jena, Germany
t.heinze@uni-jena.de
[2] Aarhus Universitet, Åbogade 34, DK-8200 Aarhus N, Denmark
[amoeller,fstrocco]@cs.au.dk

Zusammenfassung. Um auch in dynamisch typisierten Programmiersprachen die Vorteile der statischen Typisierung zu nutzen, unterstützen moderne dynamische Sprachen oft Typannotationen. Zusätzlich kann eine statische Typableitung weitere Abschätzungen zu möglichen Laufzeittypen liefern. Da die Analyse auf die annotierten Typen zurückgreifen kann, bietet sich ein modularer Analyseansatz an, der aber die Sicherheit der Typannotationen berücksichtigen muss. In dieser Arbeit stellen wir Varianten der modularen Typableitung für die Sprache Dart vor.

1 Einführung und Motivation

Dynamische Programmiersprachen können auf vielfältige Weise von der Integration statischer Typen profitieren, sei es bei der Programmoptimierung, bei der Fehlersuche oder in Werkzeugen zur Programmentwicklung, wie der Codenavigation und -vervollständigung. Für ein dynamisch typisiertes Programm lassen sich Typen in Form von Typannotationen angeben, durch eine statische Typableitung abschätzen, oder mittels einer Kombination aus beidem bestimmen. Letzterer Ansatz ist in Verbindung mit Konzepten der optionalen oder graduellen Typisierung in modernen Sprachen wie *TypeScript*, *Python*, *Dart*, bei denen statisch als auch dynamisch typisierte Abschnitte gleichzeitig in einem Programm vorliegen können, vielversprechend. Die in einem Programm annotierten Typen lassen sich dabei zur Umsetzung einer *modularen Typableitung* ausnutzen.

In [4,5] haben wir zur Unterstützung der statischen Typprüfung von Dart-Programmen bereits Möglichkeiten der Typableitung vorgestellt. Ein Programm der objektorientierten, klassenbasierten und optional typisierten Sprache *Dart* [3] ist grundsätzlich dynamisch typisiert, das heißt verwendet implizit den dynamischen Typ `dynamic`. Allerdings können dem Programm auch Typen annotiert werden, mit denen entsprechende Typprüfungen zur Programmlaufzeit definiert werden.[1] In der Folge können für ein Dart-Programm zwei typbezogene Laufzeitfehler auftreten: Einerseits wird ein Fehler beim Zugriff auf eine undefinierte

[1] Dart unterstützt zwei Ausführungsmodi: Ein Programm kann in Übereinstimmung mit der optionalen Typisierung entweder vollständig dynamisch und unabhängig von den annotierten Typen ausgeführt werden (*Production Mode*), oder es erfolgen Typprüfungen zur Programmlaufzeit anhand der Typannotationen vergleichbar der graduellen Typisierung (*Checked Mode*). Wir beziehen uns hier auf letzteren Modus.

```
1: class Box<E> { E v; }
2: void main() {
3:   var x = new Box<dynamic>();
4:   x.v = new Object();
5:   Box<int> y = x;
6:   bool i = y.v.isEven;
7: }
```

```
1: class Sup { String f; }
2: class Sub extends Sup { Object f; }
3: void main() {
4:   Sub x = new Sub();
5:   x.f = new Object();
6:   Sup y = x;
7:   String s = y.f;
8: }
```

Abb. 1. Programmbeispiele für unsichere Typen in Dart

Methode beziehungsweise ein undefiniertes Feld ausgelöst, andererseits bei Vorliegen einer Unverträglichkeit von Laufzeittyp und annotiertem Typ.

Ausgangspunkt von [4,5] war, die statische Typprüfung hinsichtlich des Auftretens dieser zwei Laufzeitfehler zu unterstützen, indem insbesondere für die dynamisch typisierten Programmabschnitte sichere Abschätzungen zu möglichen Laufzeittypen abgeleitet werden. Grundsätzlich sind für eine solche Typableitung zwei Ansätze denkbar, wie in [5] unter den Begriffen *Flow Mode* und *Modular Mode* diskutiert. Zum einen kann der vollständige Fluss von Laufzeittypen im Programm nachvollzogen werden. Die Typableitung lässt sich dann in Form einer ANDERSEN-Analyse [12] realisieren. Zum anderen kann die Analyse aber die annotierten Typen als Spezifikationen auffassen und modular, das heißt ohne vollständige Analyse des Datenflusses im Programm, ableiten.

Gegenüber der Typabschätzung auf Grundlage einer vollständigen Analyse des Datenflusses, bietet ein solcher modularer Ansatz verschiedene Vorteile, insbesondere hinsichtlich Robustheit und Skalierbarkeit der Analyse [5]. Anders als für statisch typisierte Sprachen, ist eine sichere Typabschätzung mittels modularem Analyseansatz jedoch im Fall der dynamischen Typisierung nicht ohne Weiteres möglich. Für generische Typen stellt sich etwa das Problem, dass sich deren Typgarantien unter Verwendung von `dynamic` umgehen lassen. In Abbildung 1 ist auf der linken Seite ein entsprechendes Dart-Programm dargestellt. Die Zuweisung von `Box<dynamic>` an `y` ist aufgrund der Einordnung des dynamischen Typs `dynamic` in der Typhierarchie zulässig, so dass es zur Programmlaufzeit erst beim Zugriff auf das Feld `isEven` in Zeile 6 zu einem Fehler kommt, da der Laufzeittyp `Object` von `y.v` dieses nicht definiert. Bestehende Werkzeuge zur statischen Typprüfung für Dart (*dartanalyzer*) erkennen diesen Fehler nicht.[2]

In dieser Arbeit stellen wir überblicksweise drei verschiedene Ansätze zur modularen Typableitung vor: Der erste Ansatz vernachlässigt das Problem unsicherer Typen und entspricht einer naiven Übertragung des Prinzips der statischen Typprüfung für Programmiersprachen wie Java. Der zweite Ansatz wählt einen *pessimistischen* Zugang und geht von der Unsicherheit annotierter Typen aus. Der dritte Ansatz setzt schließlich ein *optimistisches* Verfahren unter Annahme sicherer Typen um, beinhaltet aber gleichzeitig eine vollständige Analyse des Datenflusses für die während der Analyse identifizierten unsicheren Typen.

[2] Mit Ausnahme des sogenannten *Strong Mode*, siehe dessen Diskussion in Abschnitt 4.

T ::= $N \mid E \mid$ `dynamic`
N ::= C`<`$\overline{T}$`>`
L ::= `class` C`<`$\overline{E}$`>` `extends` $N\,\{\,\overline{T}\,\overline{f};\,\}$
e ::= `new` $N() \mid e.f \mid x \mid x = e \mid e.f = e \mid (N)\,e$

x ... Variablenbezeichner
f ... Feldbezeichner
C, D ... Klassennamen
E ... Typparameter

Abb. 2. Betrachtetes Sprachfragment von Dart über Feld- und Variablenzugriffen

2 Naive modulare Typableitung für Dart

Zur Veranschaulichung des modularen Analyseansatzes wird das in Abbildung 2 dargestellte Sprachfragment, in Anlehnung an *Featherweight Java* [6], verwendet. Der Einfachheit halber beschränken wir uns auf Feld- und Variablenzugriffe, so dass das abgebildete Fragment neben Klassen- und Felddeklarationen lediglich Ausdrücke über Objektinstanziierungen, explizite Typumwandlungen, sowie lesenden und schreibenden Feld- beziehungsweise Variablenzugriff umfasst.[3] Dabei bezeichne $\overline{x}$ die eventuell leere Sequenz $x_1 \ldots x_n, n \geq 0$. Variablen sind grundsätzlich dynamisch typisiert, Typannotationen können aber über die Kombination aus Variablenzugriff und Typumwandlung simuliert werden, so etwa mittels `y = (Box<int>) x;` für die Zuweisung in Zeile 5 aus Abbildung 1.

Eine *modulare Typableitung* kann die Typannotationen nutzen, so dass etwa zur Abschätzung des Laufzeittyps für den Teilausdruck `y.v` aus Abbildung 1 auf den der Variable `y` annotierten Typ `Box<int>` zurückgegriffen und dementsprechend `int` abgeleitet wird. Für ein vollständig mit statischen Typen annotiertes Programm würde dieser Ansatz somit der Typprüfung für rein statisch typisierte Sprachen entsprechen. Sind in einem Programm dynamisch typisierte, das heißt mit `dynamic` ausgezeichnete, Abschnitte vorhanden, werden die im Programm instanziierten und annotierten Typen zusätzlich entlang des relevanten Datenflusses propagiert. Dieser grundlegende Ansatz lässt sich im Rahmen einer Andersen-Analyse [12], wie in Abbildung 3 angegeben, formalisieren.

Dazu werden den Ausdrücken und Feldern eines Programms Typvariablen zugewiesen, denen Mengen von Typen zugeordnet sind (Funktion $[\![]\!]$ in Abbildung 3). Wie zu erkennen, wird für abgeleitete Typen τ, σ, ρ dabei zwischen *konkreten Typen* N^C und *abstrakten Typen* N^A unterschieden, wobei letztere auch ihre jeweiligen Untertypen umfassen (vergleiche Relation $\trianglelefteq$ in Abbildung 3). Weiterhin werden Regeln zwischen den Typvariablen in Form von Mengenbeziehungen definiert. Als Lösung des dadurch charakterisierten Regelsystems ergibt sich dann die Abschätzung zu den möglichen Laufzeittypen des Programms.

Durch die Regeln wird prinzipiell zwischen mit Typen annotierten und dynamisch typisierten Variablen und Feldern unterschieden. Beim Zugriff auf erstere wird der Fluss der Typen nachvollzogen, für letztere wird hingegen der annotierte Typ abgeleitet. Beim Zugriff auf ein dynamisch typisiertes Feld f für einen abstrakten Typ τ (Prädikat $fdyn(\tau, f)$) ist dabei zu beachten, dass auch dessen Untertypen berücksichtigt werden, da diese das Feld überschreiben können.[4]

[3] Wir schließen zur Vereinfachung (ungebundene) Typparameter in Ausdrücken aus.
[4] Im Gegensatz zu Java werden Felder in der Programmiersprache Dart überschrieben.

$$\tau, \sigma, \rho ::= N^C \mid N^A \quad N^C \trianglelefteq N^C \quad N^C \trianglelefteq N^A \quad \frac{C\texttt{<}\overline{T}\texttt{>} <: D\texttt{<}\overline{T}\texttt{>}}{C\texttt{<}\overline{T}\texttt{>}^C \trianglelefteq D\texttt{<}\overline{T}\texttt{>}^A} \quad \frac{\mathit{ftype}(\tau, f) = \texttt{dynamic}}{\mathit{fdyn}(\tau, f)}$$

Instanziierung $\texttt{new}\ N()$:
$$N^C \in [\![\texttt{new}\ N()]\!]$$

Zuweisung $x = e$:
$$[\![e]\!] \subseteq [\![x]\!]$$

Typumwandlung $(N)\ e$:
$$N^A \in [\![(N)\ e]\!]$$

Feldzugriff $e.f$:
$$\frac{\tau \in [\![e]\!] \quad \mathit{ftype}(\tau, f) = N}{N^A \in [\![e.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = N}{N^A \in [\![e.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![C.f]\!] \subseteq [\![e.f]\!]}$$

Feldzugriff $e.f = e'$:
$$[\![e']\!] \subseteq [\![e.f\ =\ e']\!]$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{ftype}(\tau, f) = N}{N^A \in [\![C.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = N \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{N^A \in [\![C.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![e']\!] \subseteq [\![C.f]\!]}$$

Abb. 3. Modulare Typableitung unter Missachtung unsicherer Typen (*ftype* ermittelt für Felder den annotierten Typ, siehe Anhang A; <: steht für die Subtyprelation [3])

Wird wieder das Beispielprogramm auf der linken Seite von Abbildung 1 betrachtet, so ergibt sich etwa für den Feldzugriff `y.v` anhand der Regeln:[5]

$$\dfrac{\dfrac{\overline{\texttt{Box<int>}^A \in [\![\texttt{(Box<int>) x}]\!]}^{\text{Z5}} \quad \overline{[\![\texttt{(Box<int>) x}]\!] \subseteq [\![\texttt{y}]\!]}^{\text{Z5}}}{\texttt{Box<int>}^A \in [\![\texttt{y}]\!]}^{\text{Z6}} \quad \mathit{ftype}(\texttt{Box<int>}^A, \texttt{v}) = \texttt{int}}{\texttt{int}^A \in [\![\texttt{y.v}]\!]}$$

Werden für den Moment unsichere Typen außer Acht gelassen, liegen die Vorteile dieses modularen Analyseansatzes auf der Hand. Einerseits unterstützen die annotierten Typen die Skalierbarkeit der Typableitung, da nicht der vollständige Fluss der Typen entlang des Datenflusses nachvollzogen werden muss und sich somit die im schlechtesten Fall kubische Analyse [12], in Abhängigkeit vom Vorhandensein der Typannotationen, auf eine lineare Analyse reduziert. Andererseits sind die Analyseergebnisse für Programmkomponenten robuster gegenüber einem rein flussbasierten Ansatz, da sich Änderungen nur bis an die mit Typannotationen versehenen Komponentenschnittstellen auswirken.

Dem aufmerksamen Leser wird nicht entgangen sein, dass der naive Ansatz nicht zu einer sicheren Typabschätzung führt. Für das Programm auf der linken

[5] Regelanwendungen sind mit zugehörigen Zeilennummern im Programm angegeben.

Instanziierung $\texttt{new}\ N()$:
$$N^C \in [\![\texttt{new}\, N()]\!]$$

Zuweisung $x = e$:
$$[\![e]\!] \subseteq [\![x]\!]$$

Typumwandlung $(N)\ e$:
$$\frac{N = C\texttt{<}\overline{T}\texttt{>}}{C\texttt{<}\overline{\texttt{dynamic}}\texttt{>}^A \in [\![(N)\ e]\!]}$$

Feldzugriff $e.f$:
$$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = D\texttt{<}\overline{T}\texttt{>}}{D\texttt{<}\overline{\texttt{dynamic}}\texttt{>}^A \in [\![e.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![C.f]\!] \subseteq [\![e.f]\!]}$$

Feldzugriff $e.f = e'$:
$$[\![e']\!] \subseteq [\![e.f\ =\ e']\!]$$
$$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = D\texttt{<}\overline{T}\texttt{>} \quad \sigma = C\texttt{<}\overline{S}\texttt{>}^C}{D\texttt{<}\overline{\texttt{dynamic}}\texttt{>}^A \in [\![C.f]\!]}$$
$$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![e']\!] \subseteq [\![C.f]\!]}$$

Abb. 4. Pessimistischer Analyseansatz

Seite von Abbildung 1 ist – bedingt durch die Verwendung des Typs `dynamic` – das Aliasing in Zeile 5 zulässig und führt zu keiner Typverletzung. Dadurch werden die mittels Annotation `Box<int>` definierten Typgarantien für die Variable `y` umgangen, so dass für eine sichere Typabschätzung `int` nicht allein als Typ für den Feldzugriff `y.v` abgeleitet werden darf. Tatsächlich enthält das Feld zur Laufzeit ein `Object`, womit es, wie bereits erwähnt, dann in Zeile 6 zu einem Laufzeitfehler kommt. Neben generischen Typen liegt das Problem der unsicheren, das heißt nicht durch Laufzeitprüfungen geschützten Typannotationen in Dart auch für Funktionstypen und Schranken von Typparametern, sowie für das Überschreiben von Feldern und Methoden vor [3]. Für letzteres, vergleiche auch das zulässige Überschreiben von `f` im Dart-Programm auf der rechten Seite von Abbildung 1, für das es zur Laufzeit erst in Zeile 7 zu einem Typfehler kommt.

3 Pessimistischer und optimistischer Ansatz

Um eine sichere Typabschätzung auch für unsichere Typen zu ermöglichen, definieren wir einen *pessimistischen Analyseansatz* anhand der modifizierten Regeln in Abbildung 4. Wie zu erkennen, beziehen wir bei Feldzugriffen für abstrakte Typen nun prinzipiell deren Untertypen mit ein, nicht nur bei dynamisch typisierten Feldern. Für den Feldzugriff `y.f` im Beispiel auf der rechten Seite von Abbildung 1 würde sich somit anhand der modifizierten Regeln ergeben:

$$\dfrac{\dfrac{\dfrac{}{\texttt{Sup}^A \in [\![\texttt{(Sup) x}]\!]}\,\text{Z6} \quad \dfrac{}{[\![\texttt{(Sup) x}]\!] \subseteq [\![\texttt{y}]\!]}\,\text{Z6}}{\text{Z7}\ \ \texttt{Sup}^A \in [\![\texttt{y}]\!]} \quad \dfrac{\texttt{Sub} <: \texttt{Sup}}{\texttt{Sub}^C \trianglelefteq \texttt{Sup}^A} \quad \mathit{ftype}(\texttt{Sub}^C, \texttt{f}) = \texttt{Object}}{\texttt{Object}^A \in [\![\texttt{y.f}]\!]}$$

Instanziierung $\texttt{new}\ N()$: $$N^C \in [\![\texttt{new}\ N()]\!]$$

Zuweisung $x = e$: $$[\![e]\!] \subseteq [\![x]\!]$$

Typumwandlung $(N)\ e$: $$N^A \in [\![(N)\ e]\!]$$

$$\frac{\tau \in [\![e]\!] \quad \tau \trianglelefteq N^A \quad \mathit{unsafe}(\tau)}{\tau \in [\![(N)\ e]\!]} \quad (*)$$

Feldzugriff $e.f$: $$\frac{\tau \in [\![e]\!] \quad \mathit{ftype}(\tau, f) = N}{N^A \in [\![e.f]\!]}$$

$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = N}{N^A \in [\![e.f]\!]}$$

$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![C.f]\!] \subseteq [\![e.f]\!]}$$

$$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C \quad \rho \in [\![C.f]\!] \quad \rho \trianglelefteq \mathit{ftype}(\sigma, f)^A \quad \mathit{unsafe}(\rho)}{\rho \subseteq [\![e.f]\!]} \quad (*)$$

Feldzugriff $e.f = e'$: $$[\![e']\!] \subseteq [\![e.f\ =\ e']\!]$$

$$\frac{\tau \in [\![e]\!] \quad \mathit{ftype}(\tau, f) = N}{N^A \in [\![C.f]\!]}$$

$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{ftype}(\sigma, f) = N \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{N^A \in [\![C.f]\!]}$$

$$\frac{\tau \in [\![e]\!] \quad \mathit{fdyn}(\tau, f) \quad \sigma \trianglelefteq \tau \quad \mathit{fdyn}(\sigma, f) \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C}{[\![e']\!] \subseteq [\![C.f]\!]}$$

$$\frac{\tau \in [\![e]\!] \quad \sigma \trianglelefteq \tau \quad \sigma = C\texttt{<}\overline{T}\texttt{>}^C \quad \rho \in [\![e']\!] \quad \rho \trianglelefteq \mathit{ftype}(\sigma, f)^A \quad \mathit{unsafe}(\rho)}{\rho \subseteq [\![C.f]\!]} \quad (*)$$

Abb. 5. Optimistischer Ansatz (*unsafe* bestimmt unsichere Typen, siehe Anhang A)

so dass der Laufzeitfehler in Zeile 7 erkannt werden kann. Im naiven Ansatz wird stattdessen nur $\texttt{String}^A \in [\![\texttt{y.f}]\!]$ abgeleitet und dieser Fehler übersehen.

Gleichzeitig werden in den Regeln nun die Typargumente von annotierten generischen Typen grundsätzlich ignoriert, das heißt durch `dynamic` ersetzt, unabhängig davon ob der Laufzeittyp tatsächlich `dynamic` als Typargument definiert oder nicht. Auf diese Weise wird beim Zugriff auf ein generisches Feld nicht das Typargument als abstrakter Typ abgeleitet sondern stattdessen der Fluss der Typen in und aus dem Feld nachvollzogen. Für das Beispielprogramm auf der linken Seite von Abbildung 1 ergibt sich im Fall von `y.v` nun $\texttt{Box<dynamic>}^A \in [\![\texttt{y}]\!]$ sowie $\texttt{Object}^C \in [\![\texttt{Box.v}]\!]$ und damit schließlich auch $\texttt{Object}^C \in [\![\texttt{y.v}]\!]$.

Der pessimistische Analyseansatz führt derart zwar zu einer sicheren Typabschätzung, birgt aber Nachteile für Präzision und Skalierbarkeit. So werden die Typgarantien von unsicheren Typen grundsätzlich nicht für die Typableitung ausgenutzt. Dieser Verzicht führt aber gerade wieder zur vollständigen Analyse des Datenflusses, im Widerspruch zum modularen Analyseansatz. Der *optimistische Ansatz* zur modularen Typableitung versucht diese Nachteile zu vermeiden und geht zunächst von der Annahme sicherer Typen aus. Dadurch begründet

sich auch die Ähnlichkeit der zugehörigen Regeln in Abbildung 5 mit denen des naiven Ansatzes aus Abbildung 3. Tatsächlich entsprechen fast alle Regeln einander, einziger Unterschied sind die mit $(*)$ gekennzeichneten zusätzlichen Regeln. Aufgabe dieser ist es, bei Auftreten unsicherer Typen, bestimmt durch das Prädikat *unsafe*, die Analyse auf das Nachvollziehen des Datenflusses umzuschalten. Dazu wird das Prädikat als eine Art Filter verwendet, um nur die unsicheren Typen entlang des Datenflusses zu propagieren, ähnlich der Verwendung von Typfiltern [12]. Tritt kein unsicherer Typ während der Analyse auf, werden wie im naiven Analyseansatz – bei Vorliegen entsprechender Typannotationen – ausschließlich die annotierten Typen abgeleitet. Für das Beispielprogramm auf der linken Seite von Abbildung 1 ergibt sich somit zunächst ebenfalls $\texttt{Box<int>}^A \in [\![\texttt{y}]\!]$ (vergleiche die entsprechende Ableitung in Abschnitt 2). Zugleich wird nun aber auch $\texttt{Box<dynamic>}^C \in [\![\texttt{y}]\!]$ abgeleitet:

$$\dfrac{\dfrac{\dfrac{\texttt{Box<dynamic>}^C \in [\![\texttt{new Box<dynamic>()}]\!] \qquad [\![\texttt{new Box<dynamic>()}]\!] \subseteq [\![\texttt{x}]\!]}{\texttt{Box<dynamic>}^C \in [\![\texttt{x}]\!]}\,\text{Z3} \qquad \dfrac{\texttt{Box<dynamic>} <: \texttt{Box<int>}}{\texttt{Box<dynamic>}^C \trianglelefteq \texttt{Box<int>}^A} \qquad \mathit{unsafe}(\texttt{Box<dynamic>}^C)}{\texttt{Box<dynamic>}^C \in [\![\texttt{(Box<int>) x}]\!]}\,\text{Z5} \qquad \dfrac{}{[\![\texttt{(Box<int>) x}]\!] \subseteq [\![\texttt{y}]\!]}\,\text{Z5}}{\texttt{Box<dynamic>}^C \in [\![\texttt{y}]\!]}$$

und in der Folge ergeben sich für den Feldzugriff `y.v` die Typen `int` und `Object`.

Unter unsicheren Typen werden dabei Typen mit dynamischen Typargument verstanden, wie `Box<dynamic>` im Beispiel, als auch Typen, in denen Felder kontra- oder kovariant überschrieben werden. Der optimistische Ansatz kann derart eine sichere Typabschätzung gewährleisten, falls die in Dart zulässige Kovarianz generischer Typen unberücksichtigt bleibt. Zugleich profitiert der optimistische Analyseansatz von Typannotationen, ähnlich der Typprüfung in rein statisch und stark typisierten Sprachen, da annotierte sichere Typen eben ohne Nachvollziehen des Datenflusses abgeleitet werden können. Offen bleibt für den optimistischen Ansatz jedoch das Problem der Kovarianz generischer Typen, wie beispielsweise im Fall von `Box<Object> z = new Box<int>(); z.f = 1;` da sich dieses nicht allein lokal und effizient auf Typebene entscheiden lässt.[6]

4 Verwandte Arbeiten

Die Programmiersprache Dart bietet einen interessanten Ansatz zur Kombination von statischer und dynamischer Typisierung. Einerseits ist Dart im sogenannten *Production Mode* optional typisiert, in Übereinstimmung mit Gilad Brachas Konzept *Pluggable Type System* [2], wird also unabhängig von annotierten Typen ausgeführt. Andererseits definieren die Typannotationen im sogenannten *Checked Mode* Laufzeittypprüfungen, wodurch sich Dart in gewisser

[6] Denkbar wäre Kovarianz generischer Typen auszuschließen (analog *Strong Mode*).

Weise in das Spektrum der graduellen Typisierung [9,11] einreiht. Im Gegensatz zu dieser werden aber in Dart sowohl bei der statischen Typprüfung (*dartanalyzer*) als auch zur Programmlaufzeit keinerlei Garantien zur Typsicherheit von annotiertem Code gegeben. Wir beziehen uns hier auf *Checked Mode*.

Dart erfährt eine fortlaufende Weiterentwicklung. Wichtig ist in diesem Zusammenhang die geplante Umstellung auf den sogenannten *Strong Mode*[7]. Dieser definiert eine sichere Teilmenge von Dart, ergänzt durch Sprachneuerungen wie generische Methoden und Typableitung, mit dem Ziel ein sicheres Typsystem zu realisieren. Dadurch positioniert sich *Strong Mode* viel näher an der graduellen Typisierung als *Checked Mode*. Nicht mehr unterstützt werden unter anderem das ko-/kontravariante Überschreiben von Feldern, Kovarianz generischer Typen oder auch die Zuweisung von `Box<dynamic>` an die mit `Box<int>` annotierte Variable `y` aus Abbildung 1. Die Typableitung kann somit einfacher erfolgen, vergleichbar dem in Abschnitt 2 vorgestellten naiven Ansatz. Gleichzeitig geht aber Sprachflexibilität verloren, die unsere Analysen stattdessen unterstützt.

Statische Typableitung zur Unterstützung graduell typisierter Sprachen wird in einer Reihe von Forschungsarbeiten behandelt, unter anderem in [1,7,8,10]. Eine frühe Arbeit dazu [10], betrachtet die Kombination von gradueller Typisierung und Typableitung nach Hindley-Milner für funktionale Sprachen, ist aber für objektorientierte Sprachen wie Dart nicht geeignet [7]. In [7] wird eine Typableitung für *ActionScript* beschrieben. Im Gegensatz zu unseren Ansatz, werden keine Mengen von Laufzeittypen zur statischen Typprüfung abgeleitet, sondern Typannotationen für dynamisch typisierte Programmabschnitte, um die Anzahl an Typprüfungen zur Laufzeit zu verringern. Bestehende Typannotationen (sogenannte *Outflows*) bleiben dabei interessanterweise unberücksichtigt. *TypeScript* fügt Typen zu *JavaScript* hinzu und unterstützt den graduellen Typisierungsansatz ebenfalls mit einer statischen Typableitung und -prüfung. Die Analyse verfolgt einen ähnlichen modularen Ansatz wie wir, ist aber aufgrund des Typsystems von *TypeScript* unsicher [1], das heißt kann nicht alle Typfehler statisch finden. Mittels zusätzlicher Laufzeitprüfungen und einem modifizierten Typsystem wird in [8] eine sichere Variante von *TypeScript* beschrieben.

5 Zusammenfassung und Ausblick

Im vorliegenden Beitrag haben wir drei Ansätze zur modularen Typableitung für die optional typisierte Sprache Dart diskutiert. Eine naive Übertragung des Prinzips der statischen Typprüfung, der erste Ansatz, hat sich aufgrund des unsicheren Typsystems von Dart als ungeeignet herausgestellt. Der zweite pessimistische Ansatz führt zwar zu einer sicheren Typabschätzung, bedingt aber gleichzeitig eine Verminderung von Analysepräzision und Skalierbarkeit. Um diese Nachteile zu vermeiden, geht der dritte, optimistische Ansatz hingegen prinzipiell von sicheren Typen aus, beinhaltet zugleich aber die vollständige Analyse des Datenflusses für Verletzungen dieser Annahme beim Überschreiben von Feldern und bei der Verwendung von `dynamic` als Typargument generischer Typen.

[7] `https://www.dartlang.org/guides/language/sound-dart`, Zugriff am 20.8.2017

Offen und zukünftigen Arbeiten überlassen bleibt, für den optimistischen Ansatz das Problem der Kovarianz generischer Typen. Eine genaue Evaluation der drei Analyseansätze hinsichtlich deren Skalierbarkeit wird einen weiteren Schwerpunkt von Arbeiten bilden, wobei uns insbesondere der Zusammenhang zwischen Analyseaufwand und den Eigenschaften des Typsystems von Dart interessiert.

Literatur

[1] Bierman, Gavin ; Abadi, Martín ; Torgersen, Mads: Understanding TypeScript. In: *ECOOP 2014, Object-Oriented Programming, 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014, Proceedings*, Springer, 2014, S. 257–281

[2] Bracha, Gilad: *Plugable Type Systems.* OOPSLA'04 Workshop on Revival of Dynamic Languages. `http://bracha.org/pluggableTypesPosition.pdf`

[3] *Dart Programming Language Specification, 4th Edition.* Standard ECMA-408, 2015

[4] Heinze, Thomas S. ; Møller, Anders ; Strocco, Fabio: Statische Typableitung für die optional typisierte Sprache Dart. In: *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung, Bericht 2015-IX-1, Pörtschach am Wörthersee, 5.–7. Oktober 2015*, Technische Universität Wien, 2015, S. 260–265

[5] Heinze, Thomas S. ; Møller, Anders ; Strocco, Fabio: Type Safety Analysis for Dart. In: *DLS'16, Proceedings of the 12th Symposium on Dynamic Languages, Amsterdam, The Netherlands, November 1, 2016*, ACM, 2016, S. 1–12

[6] Igarashi, Atsushi ; Pierce, Benjamin ; Wadler, Philip: Featherweight Java: A Minimal Core Calculus for Java and GJ. In: *OOPSLA'99, Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, Denver, USA, November 1–5, 1999*, ACM, 1999, S. 132–146

[7] 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, Philadelphia, Pennsylvania, USA, January 22–28, 2012*, ACM, 2012, S. 481–494

[8] Rastogi, Aseem ; Swamy, Nikhil ; Fournet, Cédric ; Bierman, Gavin ; Vekris, Panagiotis: Safe & Efficient Gradual Typing for TypeScript. In: *POPL'15, Proceedings of the 42nd Annual ACM Symposium on Principles of Programming Languages, Mumbai, India, January 15–17, 2015*, ACM, 2015, S. 167–180

[9] Siek, Jeremy G. ; Taha, Walid: Gradual Typing for Functional Languages. In: *Proceedings of the 2006 Scheme and Functional Programming Workshop*, University of Chicago, 2006 (Technischer Bericht TR-2006-06), S. 81–92

[10] Siek, Jeremy G. ; Vachharajani, Manish: Gradual Typing with Unification-based Inference. In: *DLS'08, Proceedings of the 2008 Symposium on Dynamic Languages, Paphos, Cyprus, July 8, 2008*, ACM, 2008

[11] Siek, Jeremy G. ; Vitousek, Michael M. ; Cimini, Matteo ; Boyland, John T.: Refined Criteria for Gradual Typing. In: *1st Summit on Advances in Programming Languages, SNAPL'15, May 3–6, 2015, Asilomar, California, US*, Schloss Dagstuhli, Leibniz-Zentrum für Informatik, 2015 (LIPIcs 32), S. 274–293

[12] 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

A Hilfsfunktionen

$$\frac{\texttt{class}\, C\texttt{<}\overline{E}\texttt{>}\, \texttt{extends}\, N\, \{\, \overline{S}\, \overline{g};\, \}\qquad f \notin \overline{g}\qquad atype([\overline{T}/\overline{E}]N, f) = U}{atype(C\texttt{<}\overline{T}\texttt{>}, f) = U}$$

$$\frac{\texttt{class}\, C\texttt{<}\overline{E}\texttt{>}\, \texttt{extends}\, N\, \{\, \overline{S}\, \overline{g};\, \}\qquad f = g_i}{atype(C\texttt{<}\overline{T}\texttt{>}, f) = [\overline{T}/\overline{E}]S_i}$$

$$\frac{atype(N, f) = T}{ftype(N^A, f) = T}\qquad \frac{atype(N, f) = T}{ftype(N^C, f) = T}$$

$$dynarg(\texttt{dynamic})\qquad \frac{dynarg(T_i)}{dynarg(C\texttt{<}\overline{T}\texttt{>})}\qquad \frac{\texttt{class}\, C\texttt{<}\overline{E}\texttt{>}\, \texttt{extends}\, N\, \{\ldots\}\ \ dynarg(N)}{dynarg(C\texttt{<}\overline{T}\texttt{>})}$$

$$\frac{\texttt{class}\, C\texttt{<}\overline{E}\texttt{>}\, \texttt{extends}\, N\, \{\, \overline{S}\, \overline{f};\, \}\qquad atype(N, f_i) = U\qquad S_i \neq U}{variant(C\texttt{<}\overline{T}\texttt{>})}$$

$$\frac{\texttt{class}\, C\texttt{<}\overline{E}\texttt{>}\, \texttt{extends}\, N\, \{\ldots\}\qquad variant(N)}{variant(C\texttt{<}\overline{T}\texttt{>})}$$

$$\frac{dynarg(N)}{unsafe(N^A)}\qquad \frac{dynarg(N)}{unsafe(N^C)}\qquad \frac{variant(N)}{unsafe(N^A)}\qquad \frac{variant(N)}{unsafe(N^C)}$$