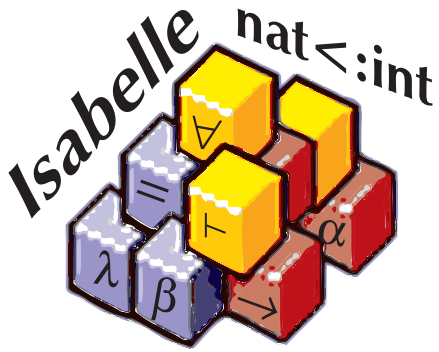
Erweiterung eines Typinferenzalgorithmus um Subtyping mit Coercions

Dmitriy Traytel

17. Mai 2010



Outline

Outline

Beispiele aus der Praxis

- Jeremy Avigad verifizierte 2004 in Isabelle:

```
(λx. pi x * ln (real x) / (real x)) ----> 1
```

Beispiele aus der Praxis

- Jeremy Avigad verifizierte 2004 in Isabelle:

```
(λx. pi x * ln (real x) / (real x)) ----> 1
```

- Dabei ging es um die Formalisierung des Primzahltheorems

$$\lim_{x \to \infty} \frac{\pi(x) \ln x}{x} = 1$$

($\pi(x)$ ist die Anzahl der Primzahlen, die kleiner als x sind)

Beispiele aus der Praxis

- Jeremy Avigad verifizierte 2004 in Isabelle:

```
(λx. pi x * ln (real x) / (real x)) ----> 1
```

- Dabei ging es um die Formalisierung des Primzahltheorems

$$\lim_{x \to \infty} \frac{\pi(x) \ln x}{x} = 1$$

($\pi(x)$ ist die Anzahl der Primzahlen, die kleiner als x sind)

- Johannes Hölzl verwendete 2008/2009 ca. 280 solcher `real`-Typcasts in den ersten 1500 Zeilen der HOL-Bibliothek `Approximation.thy`

Beispiele aus der Praxis

- Jeremy Avigad verifizierte 2004 in Isabelle:

```
(λx. pi x * ln (real x) / (real x)) ----> 1
```

- Dabei ging es um die Formalisierung des Primzahltheorems

$$\lim_{x \to \infty} \frac{\pi(x) \ln x}{x} = 1$$

($\pi(x)$ ist die Anzahl der Primzahlen, die kleiner als x sind)

- Johannes Hölzl verwendete 2008/2009 ca. 280 solcher `real`-Typcasts in den ersten 1500 Zeilen der HOL-Bibliothek `Approximation.thy`
- Beide berichteten über "Kopfschmerzen" ☺

Warum Typcasts Kopfschmerzen verursachen...

- Typcasts aus der Sicht des Programmierers:
  - tragen nicht zum logischen Inhalt des Quellcodes bei
  - erschweren dadurch die Lesbarkeit von Programmen

Warum Typcasts Kopfschmerzen verursachen...

- Typcasts aus der Sicht des Programmierers:
  - tragen nicht zum logischen Inhalt des Quellcodes bei
  - erschweren dadurch die Lesbarkeit von Programmen
- Typcasts aus der Sicht des Maschine:
  - sind essenziell für die Typ-Korrektheit

Warum Typcasts Kopfschmerzen verursachen...

- Typcasts aus der Sicht des Programmierers:
  - tragen nicht zum logischen Inhalt des Quellcodes bei
  - erschweren dadurch die Lesbarkeit von Programmen
- Typcasts aus der Sicht des Maschine:
  - sind essenziell für die Typ-Korrektheit
- Subtyping schließt diese Diskrepanz
  - bestimmte Castfunktionen werden als sog. Coercions definiert
  - Coercions werden automatisch eingefügt
  - Programmierer kann die Casts (meistens) weglassen

Outline

Sofortiges Einfügen von Coercions

- erste Idee: füge Coercions immer dann ein, wenn in einer Funktionsanwendung der Operator nicht zum Operand passt.
- resultiert in der folgenden Typ-Inferenz-Regel:

$$\frac{\Gamma \vdash \mathtt{t}_1 \rightsquigarrow \mathtt{u}_1 : T_{11} \to T_{12} \qquad \Gamma \vdash \mathtt{t}_2 \rightsquigarrow \mathtt{u}_2 : T_2 \qquad T_2 <:_{\mathtt{c}} T_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 \rightsquigarrow \mathtt{u}_1\ (\mathtt{c}\ \mathtt{u}_2) : T_{12}}$$

- wird in ähnlicher Form in Coq verwendet

Problematisches Beispiel

Beispiel: `f 0 True` vs. `f True 0`

- Signaturen: $\Sigma(\texttt{f}) = \alpha \to \alpha \to \mathbb{B}$, $\Sigma(\texttt{0}) = \mathbb{N}$ und $\Sigma(\texttt{True}) = \mathbb{B}$
- Subtypbeziehungen: $\mathbb{B} <:_{\texttt{nob}} \mathbb{N}$

Problematisches Beispiel

Beispiel: `f 0 True` vs. `f True 0`

- Signaturen: $\Sigma(\texttt{f}) = \alpha \to \alpha \to \mathbb{B}$, $\Sigma(\texttt{0}) = \mathbb{N}$ und $\Sigma(\texttt{True}) = \mathbb{B}$
- Subtypbeziehungen: $\mathbb{B} <:_{\texttt{nob}} \mathbb{N}$
- `f 0 True` wird korrekt zu `f 0 (nob True)` umgewandelt, denn
  - Funktionstyp $\mathbb{N} \to \mathbb{B}$ von `f 0` wird rekursiv inferiert
  - Argumenttyp von `True` ist $\mathbb{B}$

Problematisches Beispiel

Beispiel: `f 0 True` vs. `f True 0`

- Signaturen: $\Sigma(\texttt{f}) = \alpha \to \alpha \to \mathbb{B}$, $\Sigma(\texttt{0}) = \mathbb{N}$ und $\Sigma(\texttt{True}) = \mathbb{B}$
- Subtypbeziehungen: $\mathbb{B} <:_{\texttt{nob}} \mathbb{N}$
- `f 0 True` wird korrekt zu `f 0 (nob True)` umgewandelt, denn
  - Funktionstyp $\mathbb{N} \to \mathbb{B}$ von `f 0` wird rekursiv inferiert
  - Argumenttyp von `True` ist $\mathbb{B}$
- Typinferent von `f True 0` schlägt fehl, denn
  - Funktionstyp $\mathbb{B} \to \mathbb{B}$ von `f True` wird rekursiv inferiert
  - Argumenttyp von `0` ist $\mathbb{N}$
  - es gibt keine Coercion von $\mathbb{N}$ zu $\mathbb{B}$

Problematisches Beispiel

Beispiel: `f 0 True` vs. `f True 0`

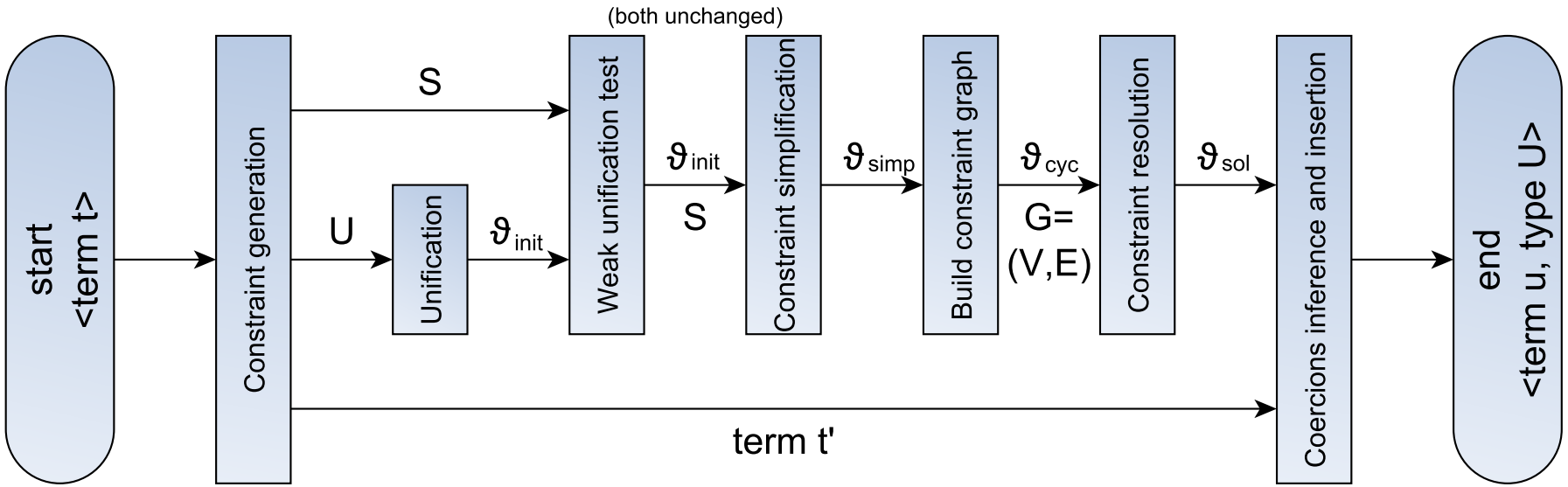
- Signaturen: $\Sigma(\texttt{f}) = \alpha \to \alpha \to \mathbb{B}$, $\Sigma(\texttt{0}) = \mathbb{N}$ und $\Sigma(\texttt{True}) = \mathbb{B}$
- Subtypbeziehungen: $\mathbb{B} <:_{\texttt{nob}} \mathbb{N}$
- `f 0 True` wird korrekt zu `f 0 (nob True)` umgewandelt, denn
  - Funktionstyp $\mathbb{N} \to \mathbb{B}$ von `f 0` wird rekursiv inferiert
  - Argumenttyp von `True` ist $\mathbb{B}$
- Typinferent von `f True 0` schlägt fehl, denn
  - Funktionstyp $\mathbb{B} \to \mathbb{B}$ von `f True` wird rekursiv inferiert
  - Argumenttyp von `0` ist $\mathbb{N}$
  - es gibt keine Coercion von $\mathbb{N}$ zu $\mathbb{B}$

- in der Reference-Manual von Coq heißt es:

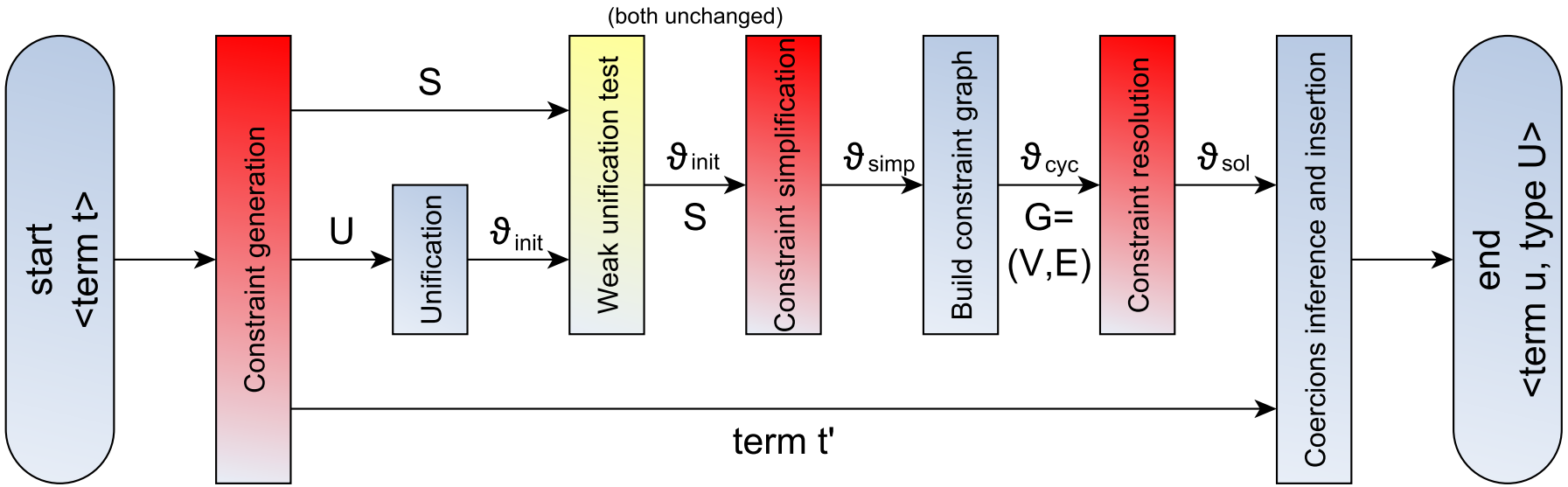
  This is "normal" behaviour of coercions.

Outline

Die Subtyping-Pipeline



Die Subtyping-Pipeline



Constraint-Generierung

Was wäre eine typtheoretische Arbeit ohne Inferenzregeln? ☺

$$\frac{\mathtt{x} : T \in \Gamma}{\Gamma \vdash \mathtt{x} : T\nabla(\emptyset, \emptyset)}\ \textsc{Var}$$

$$\frac{\Sigma(\mathtt{c}) = T}{\Gamma \vdash \mathtt{c}_{[\overline{\alpha} \mapsto \overline{\tau}]} : T[\overline{\alpha} \mapsto \overline{\tau}]\nabla(\emptyset, \emptyset)}\ \textsc{Const}$$

$$\frac{\Gamma, \mathtt{x} : T_1 \vdash \mathtt{t} : T_2\nabla(U, S)}{\Gamma \vdash \lambda\mathtt{x} : T_1 . \mathtt{t} : T_1 \to T_2\nabla(U, S)}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : T_1\nabla(U_1, S_1) \qquad \Gamma \vdash \mathtt{t}_2 : T_2\nabla(U_2, S_2) \qquad \overbrace{\alpha, \beta}^{\text{fresh}}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \beta\nabla(U_1 \cup U_2 \cup \{T_1 \doteq \alpha \to \beta\}, S_1 \cup S_2 \cup \{T_2 <: \alpha\})}\ \textsc{App}$$

Vereinfachung der Constraints

- Eingabe: Menge von Constraints zwischen beliebigen Typen
- Ausgabe: Menge von atomaten Constraints

Vereinfachung der Constraints

- Eingabe: Menge von Constraints zwischen beliebigen Typen
- Ausgabe: Menge von atomaten Constraints
- **Algorithmus:** wende wiederholt Regeln an bis alle Constraints atomar sind:
  1. $C\ \tau_1\ \tau_2\ \ldots\ \tau_n <: C\ \sigma_1\ \sigma_2\ \ldots\ \sigma_n \rightsquigarrow n$ neue Constraints oder Unifikationsprobleme:
  $$\forall i = 1 \ldots n \begin{cases} \tau_i <: \sigma_i & C \text{ ist kovariant im i-ten Argument} \\ \sigma_i <: \tau_i & C \text{ ist kontravariant im i-ten Argument} \\ \tau_i \doteq \sigma_i & \text{Varianz unbekannt } C \end{cases}$$
  Unifikationsprobleme können direkt gelöst werden. Resultierende Substitution auf alle Constrains anwenden.
  2. $\alpha <: C\ \tau_1\ \tau_2\ \ldots\ \tau_n \rightsquigarrow C\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n <: C\ \tau_1\ \tau_2\ \ldots\ \tau_n$ mit neuen Variablen $\alpha_1, \ldots, \alpha_n$. Wende Substitution $\{\alpha \mapsto C\ \alpha_1\ \alpha_2\ \ldots\ \alpha_n\}$ auf alle Constraints an. Analog für $C\ \tau_1\ \tau_2\ \ldots\ \tau_n <: \alpha$.
  3. $S <: T$ für Basistypen S und T "aufräumen".

Lösen der atomaren Constraints

- Eingabe: Constraint-Graph (Knoten beschriftet mit Typvariablen bzw. Basistypen, gerichtete Kanten repräsentieren die Constraints)
- Ausgabe: Zuweisung der Typvariablen, die alle Constraints erfüllt

Lösen der atomaren Constraints

- Eingabe: Constraint-Graph (Knoten beschriftet mit Typvariablen bzw. Basistypen, gerichtete Kanten repräsentieren die Constraints)
- Ausgabe: Zuweisung der Typvariablen, die alle Constraints erfüllt

Algorithmus (Idee):
Iteriere 2 Schritte:

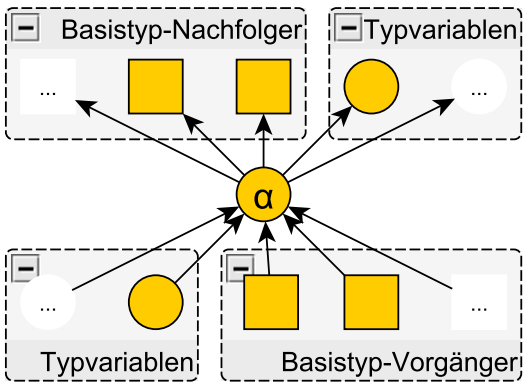
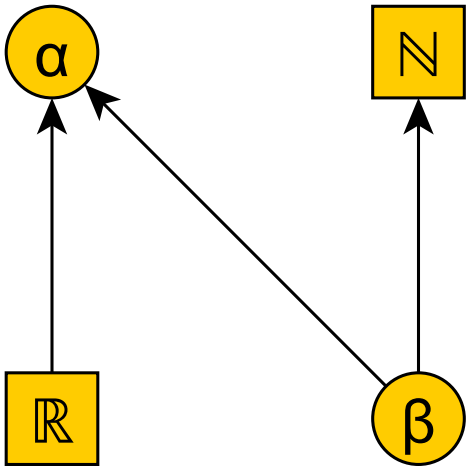


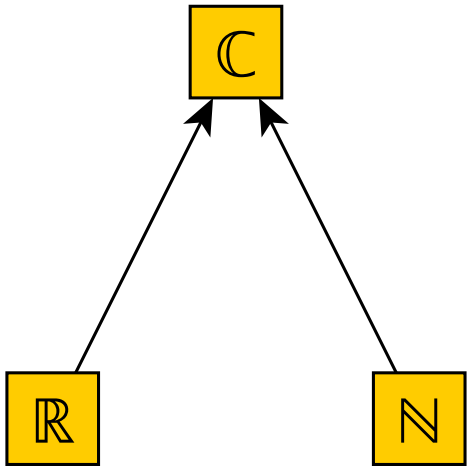
Abbildung: Sicht auf Variable α

1. Für alle Variablen α
   - Berechne den Schnitt aller Supertypen der Basistyp-Vorgänger von α
   - Weise α den "kleinsten" Typ aus dem berechneten Schnitt
   - Überprüfe ob der zugewiesene Typ Subtyp von allen Basistyp-Nachfolgern von α ist
2. Für alle Variablen α
   - Berechne den Schnitt aller Subtypen der Basistyp-Nachfolger von α
   - Weise α den "größten" Typ aus dem berechneten Schnitt

Noch ein problematisches Beispiel



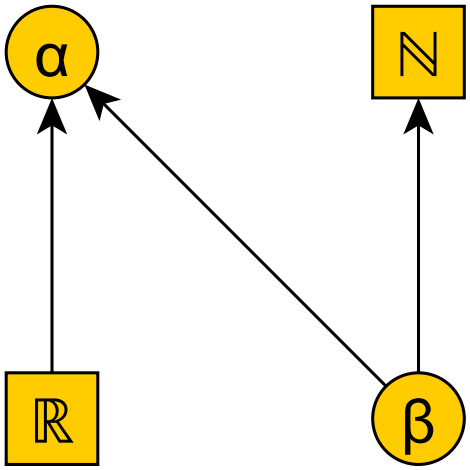
Constraint-Graph



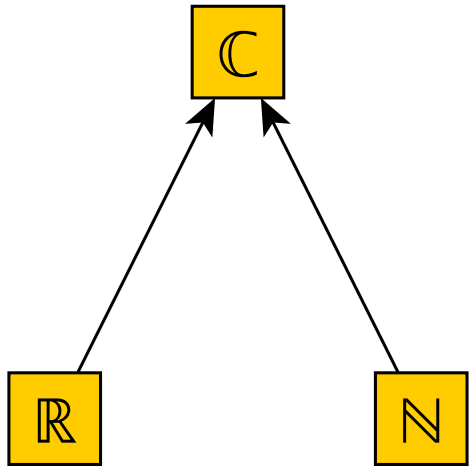
Subtyprelation

- Algorithmus weist im ersten Schritt α den Typ $\mathbb{R}$ zu

Noch ein problematisches Beispiel



Constraint-Graph

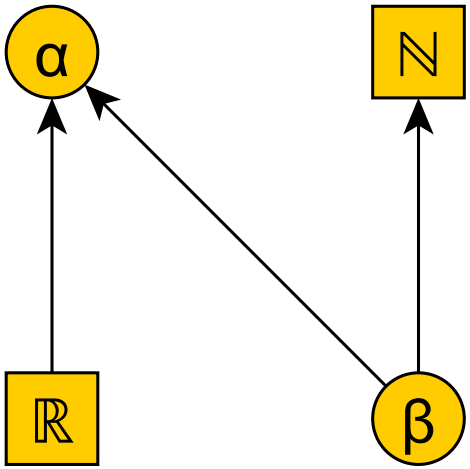


Subtyprelation

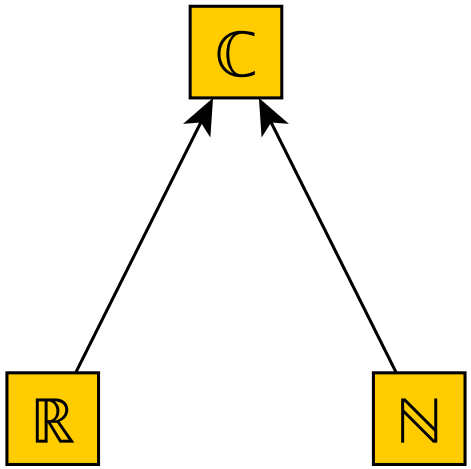
- Algorithmus weist im ersten Schritt α den Typ $\mathbb{R}$ zu
- für β gibt es dann aber keine gültige Zuweisung

$\Rightarrow$ Typinferenz schlägt fehl

Noch ein problematisches Beispiel



Constraint-Graph



Subtyprelation

- Algorithmus weist im ersten Schritt α den Typ $\mathbb{R}$ zu
- für β gibt es dann aber keine gültige Zuweisung

$\Rightarrow$ Typinferenz schlägt fehl

- Aber: die Substitution $\{\alpha \mapsto \mathbb{C},\ \beta \mapsto \mathbb{N}\}$ löst die Constraints

Fazit

- Algorithmus zur Lösung der atomaren Constraints ist polynomiell
- allgemeinste Version der Problemstellung ist NP-vollständig
- d.h. unser Algorithmus löst nur ein eingeschränktes Problem
- vorheriges Beispiel soll die Intuition für die Einschränkung wecken:

Fazit

- Algorithmus zur Lösung der atomaren Constraints ist polynomiell
- allgemeinste Version der Problemstellung ist NP-vollständig
- d.h. unser Algorithmus löst nur ein eingeschränktes Problem
- vorheriges Beispiel soll die Intuition für die Einschränkung wecken:

Behauptung

Algorithmus ist vollständig, falls die Subtyprelation eine disjunkte Vereinigung von Verbänden ist.

Outline

Zusammenspiel mit Typklassen

- eine andere Situation entsteht für die Variablen:

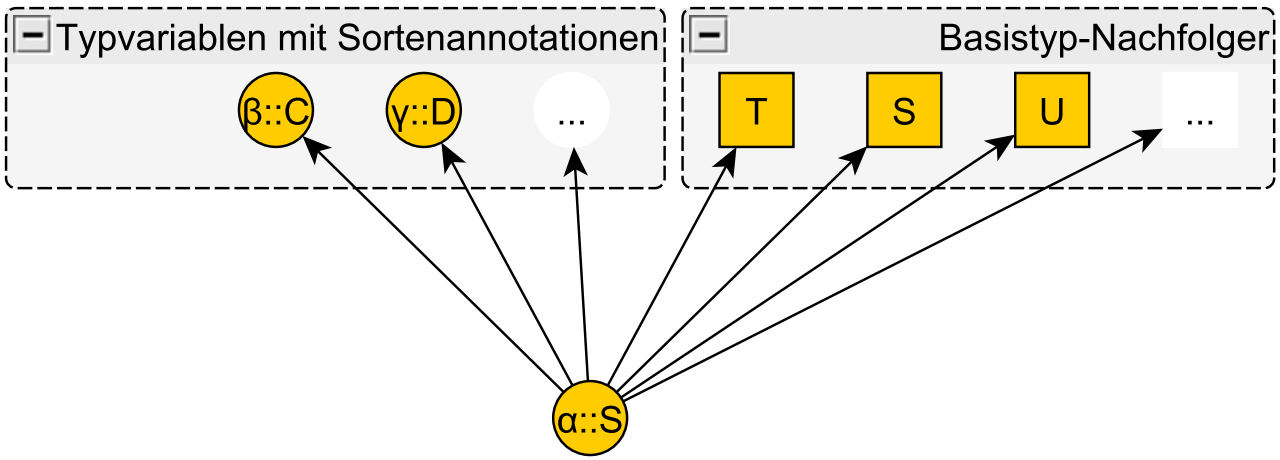


Abbildung: Sicht auf die Nachfolger der Variable α

Zusammenspiel mit Typklassen

- eine andere Situation entsteht für die Variablen:

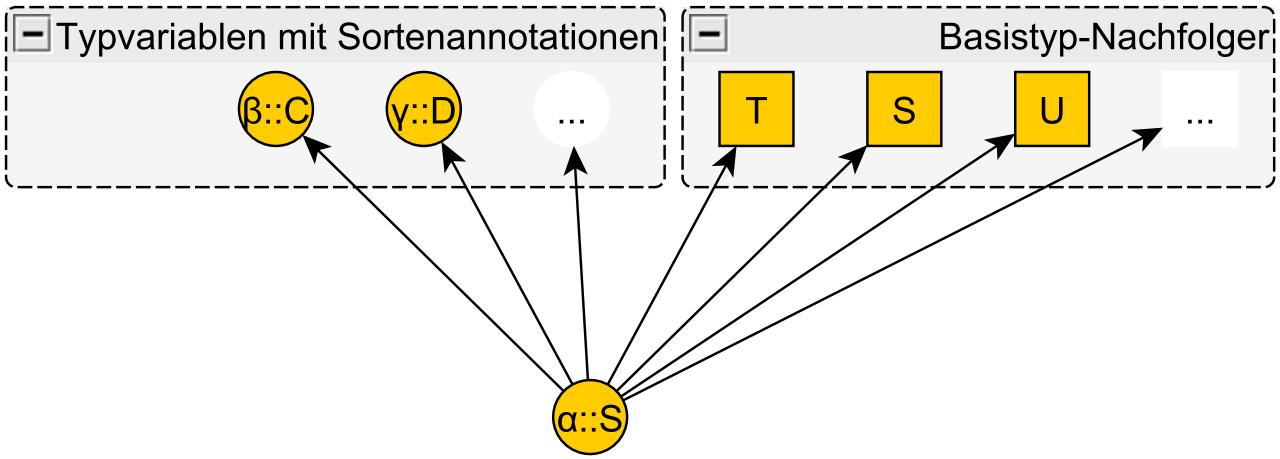


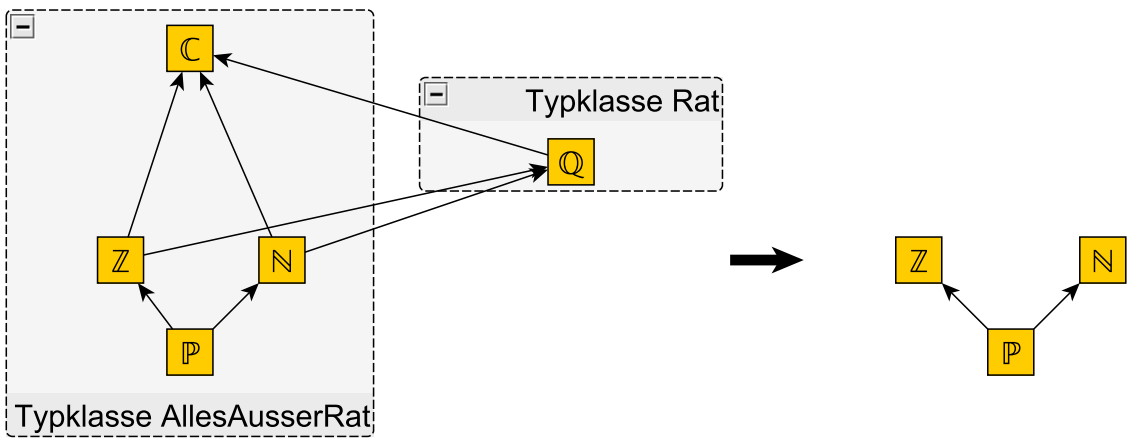
Abbildung: Sicht auf die Nachfolger der Variable α

- es reicht nicht mehr den Schnitt von Subtypen der Nachfolger zu berechenen (dual für Vorgänger)
- Typen, die nicht zur Klasse *S* gehören dürfen α nicht zugewiesen werden
- ebenfalls nicht Typen, die keinen Supertyp in der Klasse *C*, *D*, etc. haben

Das letzte problematische Beispiel

Zugegebenermaßen etwas konstruiert

- Gegeben die Constraints $\{\alpha :: \mathit{AllesAusserRat} <: \mathbb{C}, \alpha :: \mathit{AllesAusserRat} <: \beta :: \mathit{Rat}\}$ mit der folgenden Subtyprelation

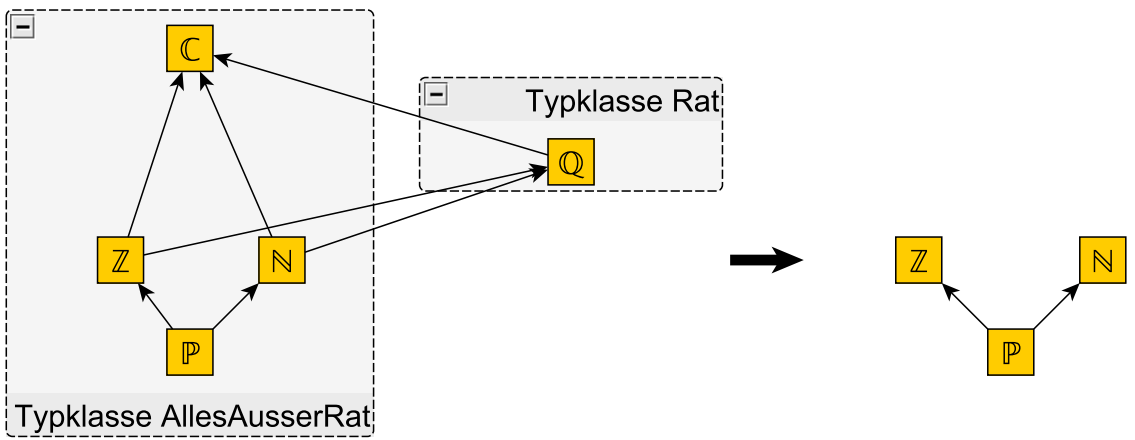


Subtyprelation Zuweisungskandidaten

Das letzte problematische Beispiel

Zugegebenermaßen etwas konstruiert

- Gegeben die Constraints $\{\alpha :: \mathit{AllesAusserRat} <: \mathbb{C}, \alpha :: \mathit{AllesAusserRat} <: \beta :: \mathit{Rat}\}$ mit der folgenden Subtyprelation



Subtyprelation Zuweisungskandidaten

- Zuweisungskandidaten (=Typen, die nach dem Bilden des Schnittes und dem Herausfiltern der mit den Typklassen kollidierenden Typen übrigbleiben) haben kein größtes Element

Fazit

- Typklassenzugehörigkeiten haben großen Einfluss auf die Zuweisungskandidaten
- sie können bewirken, dass die notwendigen Schranken nicht existieren
- klar ist: wenn die Subtyprelation eine disjunkte Vereinigung von linearen Ordnungen ist, dann ist der Algorithmus für beliebige Typklassenhierarchien vollständig
- eine schwächere Einschränkung ist möglich...

Fazit

- Typklassenzugehörigkeiten haben großen Einfluss auf die Zuweisungskandidaten
- sie können bewirken, dass die notwendigen Schranken nicht existieren
- klar ist: wenn die Subtyprelation eine disjunkte Vereinigung von linearen Ordnungen ist, dann ist der Algorithmus für beliebige Typklassenhierarchien vollständig
- eine schwächere Einschränkung ist möglich...
- ... aber schwierig zu formulieren

Outline

Q: Kann ich dank Subtyping die Typcasts immer weglassen?

- A: theoretisch ja, praktisch nein!

Q: Kann ich dank Subtyping die Typcasts immer weglassen?

- A: theoretisch ja, praktisch nein!
- Algorithmus korrekt und vollständig für "wohlgeformte" Subtyp-/Typklassen-Topologien

Q: Kann ich dank Subtyping die Typcasts immer weglassen?

- A: theoretisch ja, praktisch nein!
- Algorithmus korrekt und vollständig für "wohlgeformte" Subtyp-/Typklassen-Topologien
- Aber: nicht immer eindeutig, an welcher Stelle Coercions eingefügt werden müssen

Q: Kann ich dank Subtyping die Typcasts immer weglassen?

- A: theoretisch ja, praktisch nein!
- Algorithmus korrekt und vollständig für "wohlgeformte" Subtyp-/Typklassen-Topologien
- Aber: nicht immer eindeutig, an welcher Stelle Coercions eingefügt werden müssen

Beispiel: `sin(1 + 1)`

- Signaturen: $\Sigma(\texttt{sin}) = \mathbb{R} \to \mathbb{R}$, $\Sigma(+) = \alpha \to \alpha \to \alpha$ und $\Sigma(1) = \mathbb{N}$
- Subtypbeziehungen: $\mathbb{N} <:_{\texttt{ron}} \mathbb{R}$

Q: Kann ich dank Subtyping die Typcasts immer weglassen?

- A: theoretisch ja, praktisch nein!
- Algorithmus korrekt und vollständig für "wohlgeformte" Subtyp-/Typklassen-Topologien
- Aber: nicht immer eindeutig, an welcher Stelle Coercions eingefügt werden müssen

Beispiel: `sin(1 + 1)`

- Signaturen: $\Sigma(\texttt{sin}) = \mathbb{R} \to \mathbb{R}$, $\Sigma(+) = \alpha \to \alpha \to \alpha$ und $\Sigma(1) = \mathbb{N}$
- Subtypbeziehungen: $\mathbb{N} <:_{\texttt{ron}} \mathbb{R}$
- zwei mögliche Lösungen:
  - `sin(ron(1 + 1))`
  - `sin((ron 1) + (ron 1))`

Q: Welche Funktionen kann ich als Coercion deklarieren?

- A: theoretisch alle Funktionen auf Basistypen, die zusammen eine azyklische Subtyptopologie bilden

Q: Welche Funktionen kann ich als Coercion deklarieren?

- A: theoretisch alle Funktionen auf Basistypen, die zusammen eine azyklische Subtyptopologie bilden
- Vorsicht: uneindeutige Pfade in der Subtyptopologie können unangenehm sein

Q: Welche Funktionen kann ich als Coercion deklarieren?

- A: theoretisch alle Funktionen auf Basistypen, die zusammen eine azyklische Subtyptopologie bilden
- Vorsicht: uneindeutige Pfade in der Subtyptopologie können unangenehm sein

Beispiel

- Subtypbeziehungen:
  $\{\mathbb{A} <:_{\mathtt{boa}} \mathbb{B}, \mathbb{A} <:_{\mathtt{coa}} \mathbb{C}, \mathbb{B} <:_{\mathtt{dob}} \mathbb{D}, \mathbb{C} <:_{\mathtt{doc}} \mathbb{D}\}$

Q: Welche Funktionen kann ich als Coercion deklarieren?

- A: theoretisch alle Funktionen auf Basistypen, die zusammen eine azyklische Subtyptopologie bilden
- Vorsicht: uneindeutige Pfade in der Subtyptopologie können unangenehm sein

Beispiel

- Subtypbeziehungen:
  $\{\mathbb{A} <:_{\mathtt{boa}} \mathbb{B}, \mathbb{A} <:_{\mathtt{coa}} \mathbb{C}, \mathbb{B} <:_{\mathtt{dob}} \mathbb{D}, \mathbb{C} <:_{\mathtt{doc}} \mathbb{D}\}$
- zwei mögliche Coercions von $\mathbb{A}$ nach $\mathbb{D}$:
  - $\lambda \mathtt{x}\colon \mathbb{A}\,.\,\mathtt{dob}\ (\mathtt{boa}\ \mathtt{x}) : \mathbb{A} \to \mathbb{D}$
  - $\lambda \mathtt{x}\colon \mathbb{A}\,.\,\mathtt{doc}\ (\mathtt{coa}\ \mathtt{x}) : \mathbb{A} \to \mathbb{D}$

Q: Welche Funktionen kann ich als Coercion deklarieren?

- A: theoretisch alle Funktionen auf Basistypen, die zusammen eine azyklische Subtyptopologie bilden
- Vorsicht: uneindeutige Pfade in der Subtyptopologie können unangenehm sein

Beispiel

- Subtypbeziehungen:
  $\{\mathbb{A} <:_{\mathtt{boa}} \mathbb{B}, \mathbb{A} <:_{\mathtt{coa}} \mathbb{C}, \mathbb{B} <:_{\mathtt{dob}} \mathbb{D}, \mathbb{C} <:_{\mathtt{doc}} \mathbb{D}\}$
- zwei mögliche Coercions von $\mathbb{A}$ nach $\mathbb{D}$:
  - $\lambda \mathtt{x}\colon \mathbb{A}\,.\,\mathtt{dob}\ (\mathtt{boa}\ \mathtt{x}) : \mathbb{A} \to \mathbb{D}$
  - $\lambda \mathtt{x}\colon \mathbb{A}\,.\,\mathtt{doc}\ (\mathtt{coa}\ \mathtt{x}) : \mathbb{A} \to \mathbb{D}$
- problematisch, wenn beide Funktionen unterschiedlich

Q: Welchen Einfluss hat das Ganze auf die Performanz?

- A: kleiner Performanz-Vergleich von 4 "Umgebungen":
  1. ursprüngliche Typinferenz von Isabelle, unveränderte `Approximation.thy`.
  2. Subtyping-Algorithmus aktiviert, unveränderte `Approximation.thy`.
  3. Subtyping-Algorithmus aktiviert, Subtypbeziehungen definiert in `Approximation.thy`.
  4. Subtyping-Algorithmus aktiviert, Subtypbeziehungen definiert und Typcasts in teilweise entfernt in `Approximation.thy`

Q: Welchen Einfluss hat das Ganze auf die Performanz?

- A: kleiner Performanz-Vergleich von 4 "Umgebungen":
  1. ursprüngliche Typinferenz von Isabelle, unveränderte `Approximation.thy`.
  2. Subtyping-Algorithmus aktiviert, unveränderte `Approximation.thy`.
  3. Subtyping-Algorithmus aktiviert, Subtypbeziehungen definiert in `Approximation.thy`.
  4. Subtyping-Algorithmus aktiviert, Subtypbeziehungen definiert und Typcasts in teilweise entfernt in `Approximation.thy`

Umgebung	CPU Nutzzeit in *min* : *sec*
1	4 : 34.85
2	5 : 02.31
3	5 : 06.63
4	5 : 01.27

Danke für die Aufmerksamkeit!