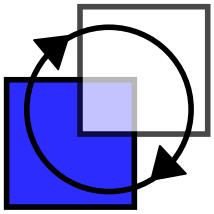
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Prof. Bernd Finkbeiner, Ph.D.
Jana Hofmann, M.Sc.
Reactive Systems Group



Programmierung 1 (WS 2020/21)
Übungsblatt J

Lesen Sie im Buch Kapitel 11.7 - 12.

Hinweis: Über Aufgaben, die mit 🤔 markiert sind, müssen Sie eventuell etwas länger nachdenken. Falls Ihnen keine Lösung einfällt - kein Grund zur Sorge. Kommen Sie in die Office Hour, unsere Tutor:innen helfen gerne.

Laufzeit rekursiver Prozeduren

Aufgabe J.1

Bestimmen Sie die Best- und die Worst-Case-Laufzeit der folgenden Prozedur:

$$
\begin{aligned}
&gcd : \mathbb{N}_+ \times \mathbb{N}_+ \to \mathbb{N}_+ \\
&gcd\ (x, x) = x \\
&gcd\ (x, y) = gcd\ (x - y, y) && \text{für } x > y \\
&gcd\ (x, y) = gcd\ (x, y - x) && \text{für } x < y
\end{aligned}
$$

Dabei soll die Größenfunktion $\lambda(x, y).\ \max\{x, y\}$ zugrunde gelegt werden.

(a) Geben Sie für beliebige Größen $n \geq 1$ Argumente an, für die die Laufzeit minimal beziehungsweise maximal ist.

(b) Geben Sie die Best-Case-Laufzeitfunktion und deren Komplexität an.

(c) Geben Sie die Worst-Case-Laufzeitfunktion und deren Komplexität an.

Aufgabe J.2

Geben Sie die Komplexitäten der im Folgenden rekursiv definierten Funktionen $f \in \mathbb{N} \to \mathbb{N}_+$ möglichst einfach an.

(a) $f\ n = \text{if } n < 30 \text{ then } 1 \text{ else } f\ (n - 6) + 3n + 7n^2$

(b) $f\ n = \text{if } n < 7 \text{ then } 1 \text{ else } 3\ f\ (n - 1) + n \lfloor \log n \rfloor + 5$

(c) $f\ n = \text{if } n < 4 \text{ then } 1 \text{ else } f\ \left(\left\lfloor \frac{n}{3} \right\rfloor\right) + 13$

(d) $f\ n = \text{if } n < 2 \text{ then } 1 \text{ else } 2\, f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 3n + \lfloor \log n \rfloor + 5$

Aufgabe J.3

Geben Sie die Komplexitäten der durch die folgenden Gleichungen rekursiv definierten **Funktionen** $f \in \mathbb{N} \to \mathbb{N}$ an:

(a) $f\ n = \text{if } n = 0 \text{ then } 0 \text{ else } f\ (n - 1)$

(b) $f\ n = \text{if } n = 0 \text{ then } 1 \text{ else } f\ (n - 1)$

(c) $f\ n = \text{if } n = 0 \text{ then } 0 \text{ else } f\ (n - 1) + 1$

Aufgabe J.4
Geben Sie die Komplexitäten der durch die folgenden Gleichungen rekursiv definierten **Prozeduren** $f : \mathbb{N} \to \mathbb{N}$ an:

(a) $f\ n = \text{if } n = 0 \text{ then } 0 \text{ else } f\ (n-1)$

(b) $f\ n = \text{if } n = 0 \text{ then } 1 \text{ else } f\ (n-1)$

(c) $f\ n = \text{if } n = 0 \text{ then } 0 \text{ else } f\ (n-1) + 1$

Vergleichen Sie Ihre Ergebnisse mit den Ergebnissen aus Aufgabe J.3.

Aufgabe J.5
Wir betrachten die aus früheren Aufgaben bereits bekannte Prozedur zur Berechnung natürlicher Quadratwurzeln: 🤔

$$p : \mathbb{N}^2 \to \mathbb{N}$$
$$p\ (n, x) = \text{if } n^2 \leq x \text{ then } p\ (n+1, x) \text{ else } n - 1$$

(a) Geben Sie die Laufzeit von p für ein Argument (n, x) an.

(b) Bestimmen Sie die Laufzeitfunktion der Prozedur p gemäß der Größenfunktion

$$\lambda(n, x).\ \max\{0, x + 1 - n\}$$

(c) Geben Sie die Komplexität von p gemäß der Laufzeitfunktion aus (b) an.

Aufgabe J.6
Seien zwei Prozeduren wie folgt gegeben:

$p : \mathbb{N} \to \mathbb{N}$
$p\ n = \text{if } n < 5 \text{ then } n \text{ else } p\ (n-2)$

$q : \mathbb{N} \to \mathbb{N}$
$q\ n = \text{if } n < 12 \text{ then } 3n \text{ else } q\ (n-1) + q\ (n-1) + p\ n$

Sie sollen die Laufzeitfunktionen und die Komplexitäten der Prozeduren für die Größenfunktion $\lambda n.\ n$ bestimmen.

(a) Beschreiben Sie die Laufzeitfunktion von p rekursiv.

(b) Beschreiben Sie die Laufzeitfunktion von q rekursiv. Machen Sie dabei von der Laufzeitfunktion für p Gebrauch, um die für die Anwendung der Funktion p anfallenden Nebenkosten einzubringen.

(c) Geben Sie die Komplexität der Prozedur p an.

(d) Geben Sie die Komplexität der Prozedur q an.

Aufgabe J.7

(a) Geben Sie möglichst einfache Prozeduren $\mathbb{N} \to \{0\}$ an, die für die Größenfunktion $\lambda n.\ n$ die Komplexitäten $\mathcal{O}(\log n)$ und $\mathcal{O}(n \log n)$ haben. Für $\mathcal{O}(\log n)$ sollen keine Nebenkosten anfallen.

(b) Geben Sie möglichst einfache Prozeduren $\mathbb{N} \to \{0\}$ an, die für die Größenfunktion $\lambda n.\ n$ die Komplexitäten $\mathcal{O}(1)$, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$ und $\mathcal{O}(n^3)$ haben. Für $\mathcal{O}(1)$ und $\mathcal{O}(n)$ sollen keine Nebenkosten anfallen. Für $\mathcal{O}(n^2)$ und $\mathcal{O}(n^3)$ sollen lineare, beziehungsweise quadratische Nebenkosten anfallen.

(c) Geben Sie möglichst einfache Prozeduren $\mathbb{N} \to \{0\}$ mit den Komplexitäten $\mathcal{O}(2^n)$ und $\mathcal{O}(3^n)$ an. 🤔

Statische und dynamische Semantik

Aufgabe J.8
Geben Sie (in SML) Deklarationen an, die den Bezeichner e an die abstrakte Darstellung des Ausdrucks `fn f : int → int ⇒ fn n : int ⇒ if n <= 0 then 1 else n * f (n − 1)` binden.

Hinweis: Gehen Sie dabei schrittweise vor und beginnen Sie mit der Deklaration des Teilausdrucks `n <= 0` durch `val e1 = Opr(Leq, Id "n", Con(IC 0))`.

Aufgabe J.9
In den Abschnitten 2.4 und 3.8 haben wir definiert, was wir unter offenen und geschlossenen Ausdrücken und den freien Variablen eines Ausdrucks verstehen wollen.

Schreiben Sie eine Prozedur `closed : exp → bool`, die testet, ob ein Ausdruck geschlossen ist. Schreiben Sie zuerst eine Hilfsprozedur `closed' : exp → id list → bool`, die testet, ob alle freien Bezeichner eines Ausdrucks in einer Liste vorkommen.

Aufgabe J.10
Geben Sie einen Ableitungsbaum (Inferenzbaum) für die folgenden Aussagen an. Begründen Sie jeden Schritt mit der angewendeten Inferenzregel.

(a)
$$[x \coloneqq \mathit{int}] \vdash \text{fn } f : \mathit{int} \to \mathit{bool} \Rightarrow \text{fn } y : \mathit{int} \Rightarrow f\ (2 \cdot x + y) : (\mathit{int} \to \mathit{bool}) \to (\mathit{int} \to \mathit{bool})$$

(b)
$$\emptyset \vdash (\text{fn } x : \mathit{int} \to \mathit{int} \Rightarrow x\ 5)(\text{fn } y : \mathit{int} \Rightarrow y + 3) \triangleright 8$$

(c)
$$[y \coloneqq 4, z \coloneqq 2] \vdash (\text{fn } x : \mathit{int} \Rightarrow x + 5)\ y * 5 \triangleright 45$$

Aufgabe J.11
Erweitern Sie unsere Elaborierung von `F` um 2 unäre Operatoren:

(a) $\sim$ `: int → int`, welche das additive Inverse für ganze Zahlen berechnet.

(b) `! : bool → bool`, welche die boolesche Negation berechnet.

Lösen Sie diese Aufgaben Schritt für Schritt:

- Erweitern Sie formal die abstrakte Grammatik für die abstrakte Syntax von F. (Siehe Seite 243, Abbildung 12.2)
- Erweitern Sie nun die Typdeklarationen für die abstrakte Syntax von F.
- Geben Sie die Inferenzregeln für die statische Semantik der neuen unären Operatoren an.
- Erweitern Sie nun die Deklaration der Prozedur `elab`.

Aufgabe J.12
Betrachten Sie erneut Aufgabe L.5. Erweitern Sie nun ebenfalls die dynamische Semantik um die entsprechenden unären Operatoren.

Aufgabe J.13
Sie sollen F um Paare erweitern. Die abstrakte Syntax und die Menge der Werte sollen dazu wie folgt erweitert werden:

$$\begin{aligned} t \in \mathit{Ty} &= \cdots \mid t_1 * t_2 \\ e \in \mathit{Exp} &= \cdots \mid (e_1, e_2) \mid \mathit{fst}\ e \mid \mathit{snd}\ e \\ v \in \mathit{Val} &= \mathbb{Z} \cup \mathit{Pro} \cup (\mathit{Val} \times \mathit{Val}) \end{aligned}$$

(a) Geben Sie die für die statische Semantik zusätzlich erforderlichen Inferenzregeln an.

(b) Geben Sie die für die dynamische Semantik zusätzlich erforderlichen Inferenzregeln an.

(c) Erweitern Sie die Implementierung um die für Paare erforderlichen Konstrukte. Dafür müssen die Deklarationen von `ty`, `exp`, `elab`, `value` und `eval` geändert werden.

Aufgabe J.14

Die Prozedur `eval empty` liefert für viele Ausdrücke Ergebnisse, für die `elab empty` Fehler meldet. Geben Sie für jeden der von `elab` behandelten Fehler einen Ausdruck an, der bei `eval` keinen Fehler erzeugt.

Aufgabe J.15

(a) Erweitern Sie die Deklarationen der Typen `exp` und `value` um einen Konstruktor für rekursive Abstraktionen.

(b) Erweitern Sie die Prozedur `elab` um eine Regel für rekursive Abstraktionen.

(c) Erweitern Sie die Prozedur `eval` um eine Regel für rekursive Abstraktionen. Erweitern Sie zudem die Fallunterscheidung in der Regel für Prozeduranwendungen um die Anwendung rekursiver Prozeduren. Erproben Sie Ihre erweiterte Prozedur `eval` mit einer rekursiven Prozedur, die Fakultäten berechnet.

(d) Geben Sie einen zulässigen Ausdruck `e` an, sodass die Auswertung von `eval empty e` divergiert.

kNobelpreis

Sie sind an den Lösungen für kNobelaufgabe H interessiert? Der/die Aufgabensteller:in bietet ein kNobeltutorium für alle Interessierte an:

Donnerstag, 21.1. um 16:15
Zoom-Link:
https://cs-uni-saarland-de.zoom.us/j/98947936655?pwd=TXdBOXUxSmI2SzRDeWNsellnUEhGQT09

Dieter Schlau möchte mit guten Vorsätzen ins neue Jahr starten. Da in der Vergangenheit das ein oder andere Programm von ihm Fehler aufwies, möchte er 2021 nur noch garantiert korrekten Code zur Anwendung bringen oder zumindest davor warnen, wenn ein Problem auftritt. Dafür hat er sich eine Formelschreibweise überlegt, um auszudrücken wie sich das Programm verhalten sollte. Solche Formeln nennt er Spezifikation. Seine Programme liefern während der Laufzeit Informationen darüber in welchem Zustand sie sich gerade befinden. Dies geschieht in der Form von Aussagen (propositions), die in jedem Programmschritt ausgegeben werden, ähnlich wie ein log.

Daraus ergibt sich eine Spur (trace) eines Programmablaufes. Diese könnte zum Beispiel für ein Programm, welches einfache Berechnungen durchführt, so aussehen:
`{"start"}, {"calculate"}, {"calculate"}, {"print_text", "print_result"}, {}, {"terminate"}`
In seinen Formeln kann Dieter nun aufschreiben wie die Spur aussehen sollte. Dafür hat er eine Reihe von Werkzeugen:

- Einfache Formeln z.B. *start* gilt zum Beispiel im ersten Schritt der obigen Spur. *true* gilt immer.
- Mit dem Next-Operator $\mathcal{X}a$ beschreibt er, dass die Aussage a im nächsten Schritt gelten soll. So gilt $\mathcal{X}\,calculate$ in obiger Spur.
- Der Until-Operator $a\ \mathcal{U}\ b$ beschreibt, dass a solange gelten muss bis im nächsten Schritt b gilt. b muss aber irgendwann erfüllt werden. Zum Beispiel erfüllt unsere Spur $\mathcal{X}(calculate\ \mathcal{U}\ print\_result)$, aber auch $true\ \mathcal{U}\ start$, aber nicht $true\ \mathcal{U}\ false$.
- Huch? Es gibt auch ein *false*? Damit Dieter auch alle Operatoren so miteinander kombinieren kann wie er möchte, gibt es auch die bekannten logischen Operatoren Konjunktion $a \wedge b$ und Negatio n$\neg a$. Sie verhalten sich wie aus der Mathematik gewohnt. Es gilt also $false = \neg true$.
- Finally $\mathcal{F}\ a = true\ \mathcal{U}\ a$ bedeutet, dass a irgendwann in der Spur vorkommen muss.
- Globally $\mathcal{G}\ a = \neg(\mathcal{F}\neg a)$ heißt, dass a immer gelten muss.

Natürlich kann man mit DeMorgan auch noch eine Disjunktion $\vee$ bauen.

Um Dieters Formeln ein bisschen besser zu verstehen hat er sich folgende Beispiele überlegt:

Die Aussage $false\ \mathcal{U}\ true$ gilt immer. Denn wenn für die Formel $a\ \mathcal{U}\ b$ direkt b gilt, dann gibt es keine vorherigen Schritte, in denen a gelten müsste.

Außerdem hat Dieter die witzige Beobachtung gemacht, dass sowohl $\neg a\ \mathcal{U}\ \neg b$ als auch $a\ \mathcal{U}\ b$ auf der Spur `{"a","b"}` gelten.

$\neg(a\ \mathcal{U}\ b)$ bedeutet, dass b niemals auftreten wird oder 'zu spät' auftritt, also erst nachdem bereits in einem vorherigen Schritt a nicht mehr gegolten hat. $\mathcal{F}(\mathcal{G}\ a) = \mathit{true}\ \mathcal{U}(\neg(\mathit{true}\ \mathcal{U}\ \neg a))$ bedeutet, dass ab einem beliebigen Zeitpunkt dauerhaft a gelten muss.

Dieter Schlau wurde vor kurzem damit beauftragt eine Schaltung für die Ampel eines Fußgängerüberwegs zu programmieren. Zur Überprüfung seines Codes benötigt er noch eine Spezifikation und bittet Sie um Hilfe.

Aufgabe J.16
Formulieren Sie alle notwendigen Eigenschaften einer Ampel an einem Fußgängerüberweg, sodass sowohl Autofahrer als auch Fußgänger garantiert ungefährdet zum Zuge kommen und der normale Rhythmus der Autoampel eingehalten wird. Der Zustand für die Ampel der Autos wird durch `"car_red"`, `"car_yellow"` oder `"car_green"` beschrieben und `"ped_red"`, `"ped_green"` die Ampel für die Fußgänger.

Aufgabe J.17
Nun möchte Dieter Schlau prüfen ob seine Programm seine Spezifikationen erfüllen. Seine Programme haben die Form:

```
type (''s, 'α) program = (''s → ''s) * ''s * (''s → 'α list)
```

Hierbei stellt `'α` die Aussagen und `''s` die Zustände des Systems dar.
Sei `(d, s0, l) : (''s, 'α) program` ein Programm.

- `s0` ist der Startzustand des Programms.
- `d` ist die Übergangsfunktion. Diese gibt an welcher Zustand auf welchen folgt.
- `l` gibt an welche Aussagen in welchem Zustand gelten.

Die (unendliche) Spur eines solchen Programms ist die Folge

$$l(s_0), l(d(s_0)), l(d(d(s_0))), l(d^3(s_0)), l(d^4(s_0)) \ldots$$

Zudem sollen die Zustände des Programms sich irgendwann wiederholen, d.h.

$$\exists i \in \mathbb{N}_0.\ \exists j > i.\ d^i(s_0) = d^j(s_0)$$

Schreiben Sie nun eine Prozedur `|= :(''s, 'α) program → 'α spec → bool`, die prüft ob ein Programm eine Spezifikation erfüllt.

Aufgabe J.18
Dieter Schlau hat in der Zwischenzeit zwei Ampeln gebaut. Prüfen Sie nun ob/welche diese/r Ampeln gemäß Teil 1 richtig sind/ist. Benutzen sie hierfür Ihre Prozedur aus Teil 2.

Bonus für Mutige
Dieter hat ein Idee. Was wäre wenn er – statt seine Programme selber zu schreiben – richtige Programme aus seinen Spezifikationen generieren könnte und herausfinden könnte wenn das unmöglich ist (ein Beispiel hierfür ist die Spezifikation *false*)? Helfen Sie Dieter hierbei. **Warnung:** Diesen Bonus komplett zu lösen ist sehr schwer. Sie können aber gerne versuchen das Problem teilweise zu lösen.