Final Exam, *Semantics and Types* (2023/24)

Andrzej Filinski

2 April 2024, 9:00 – 3 April 2024, 17:00

This 10-page exam consists of three independent parts, each containing 1–3 loosely related questions. The exam will be graded as a whole, with each question weighted as indicated next to it, for the initial score calculation. However, your answers should also demonstrate a satisfactory mastery of all relevant course learning outcomes; therefore, you should aim to answer all questions at least partially, rather than concentrating all your efforts on only some of them.

In the event of errors or ambiguities in the exam text, you are expected to state your assumptions as to the intended meaning, and proceed according to your chosen interpretation. In serious cases that you cannot resolve yourself, you may contact the course teacher directly at `andrzej@di.ku.dk` (*not* on the Absalon discussion forum), but do not expect an immediate reply. Any significant corrections or clarifications will be announced on Absalon in the first instance.

Your answers must be submitted through the Digital Exam system (`eksamen.ku.dk`) as a single PDF file. (Remember to also press the "Submit" button on the confirmation page, especially if re-uploading a revised version, or you may end up submitting nothing at all!) Note that the submission deadline is strictly enforced by a departmental exam administrator, so be sure to submit on time. If (and only if!) Digital Exam is unavailable around the submission deadline, you may also mail your answers to *both* `andrzej@di.ku.dk` and `uddannelse@di.ku.dk`, identifying them as such in the subject line.

You may submit scans of handwritten solutions, but any such scans should preferably be done with a proper flat-bed scanner; if you use a handheld mobile-phone camera or similar, make extra sure that *all parts* of *all pages* are well lit and clearly readable (and in particular, not reduced too much in size from the originals). Remember that, if you normally use a shared scanner, it may be unavailable close to the submission deadline, so plan accordingly. If you typeset your solutions in LaTeX, focus on correctness, rather than aesthetics.

Also, please make sure that all pages are numbered, and that you leave *at least* a 1 cm margin (preferably more) around *all* edges of the paper, especially if you submit scanned material. Finally, do set off some time to proofread your solutions against the question text, to avoid losing points on silly mistakes, such as simply forgetting to answer a subquestion.

Collaboration policy. This take-home exam is to be completed *100% individually*: for the duration of the entire exam period (and until **9:00 on 4 April**, since some students may have dispensations for extended exam time), you are not to communicate with any other person about academic matters related to the course (whether helping or receiving help); nor may you consult on-line resources beyond the course homepage. Any violations will be handled in accordance with Faculty of Science disciplinary procedures.

Happy working!

1 An imperative language with jumps

We consider a variant of the IMP language, called LIMP (Label-IMP). LIMP has exactly the same syntax and semantics of arithmetic and boolean expressions (a and b) as IMP in the notes, but where IMP uses structured control flow, LIMP control is expressed in terms of labels and jumps, much like in assembly language.

Specifically, in LIMP, we operate with the syntactic categories of *simple commands* (s) and *programs* (p), defined as follows:

$$\begin{array}{lcl} s & ::= & X := a \mid \textbf{if } b \textbf{ then goto } \ell \\ p & ::= & \textbf{halt} \mid \ell\colon p \mid s; p \end{array}$$

Here $\ell \in \textbf{Label}$ ranges over some infinite set of *labels*. The unconditional jump command **goto** ℓ can be thought of as syntactic sugar: **goto** $\ell \equiv$ **if true then goto** ℓ.

A LIMP program is said to be *well formed* if no label is declared twice. We will only consider well formed programs. (On the other hand, there is no *a priori* requirement that all labels used as the target of a **goto** are declared; an attempted jump to an undeclared label will just cause execution of the program to fail silently.)

The informal semantics of programs is the expected one: simple commands are normally executed in order, but if an **if** b **then goto** ℓ is executed, with b evaluating to **true** in the current state, then execution continues instead from the program point labelled by ℓ. If b evaluates to **false**, execution just falls through to the next command.

Formally, the big-step semantics of LIMP is expressed in terms of a new judgment $\boxed{p_0 \vdash \langle p, \sigma\rangle \downarrow \sigma'}$, which says that, in the context of a complete program p_0, executing the program fragment p from a starting state σ, terminates in σ'. For expressing the semantics of jumps, we also need an auxiliary judgment $\boxed{p_0 \vdash \ell \Downarrow p}$, which says that, in the complete program p_0, the remainder of the program after label ℓ is p. The rules are given in Figure 1.

1.1 Compiling IMP to LIMP

We may think of IMP as a high-level language, which can be compiled or translated to the low-level language LIMP. Informally, every IMP command c is translated to a sequence $[\![c]\!]$ of LIMP simple commands and label declarations, as shown in Figure 2. In the figure, the labels ℓ and ℓ' used in the translation of each **if**- and **while**-command must be "fresh", i.e., not used anywhere else in the program. A final **halt** statement is added after the compilation of a complete program.

To formalize the translation, and especially the fresh-label generation, let $\textbf{Lab} = \{\ell_0, \ell_1, \ldots\}$ be an infinite enumeration of distinct labels. We then define the main program compilation judgment $\boxed{[\![c]\!]^{\text{top}} \rightsquigarrow p}$ in terms of an auxiliary judgment $\boxed{[\![c]\!] \mathbin{@} p \rightsquigarrow^{i}_{i'} p'}$, which builds up the compiled program *from the end*.

Intuitively, the latter judgment says that the translation of c, *prepended* to some already constructed LIMP program p, is p'. Also, i is the first unused label before the translation, while i' is the next free one after the translation; in other words, the translation of c itself declares the labels $\ell_i, \ell_{i+1}, ..., \ell_{i'-1}$. This means that, if p is well formed and only declares labels less than ℓ_i, then p' will also be well formed and only declare labels less than $\ell_{i'}$. The formal compilation rules are given in Figure 3.

Judgment $\boxed{p_0 \vdash \langle p, \sigma\rangle \downarrow \sigma'}$:

$$\text{P-Halt} : \frac{}{p_0 \vdash \langle \mathbf{halt}, \sigma\rangle \downarrow \sigma} \qquad \text{P-Lab} : \frac{p_0 \vdash \langle p_1, \sigma\rangle \downarrow \sigma'}{p_0 \vdash \langle \ell\colon p_1, \sigma\rangle \downarrow \sigma'}$$

$$\text{P-Assign} : \frac{\langle a, \sigma\rangle \downarrow n \qquad p_0 \vdash \langle p_1, \sigma[X \mapsto n]\rangle \downarrow \sigma'}{p_0 \vdash \langle X := a; p_1, \sigma\rangle \downarrow \sigma'}$$

$$\text{P-IfGotoT} : \frac{\langle b, \sigma\rangle \downarrow \mathbf{true} \qquad p_0 \vdash \ell \Downarrow p_2 \qquad p_0 \vdash \langle p_2, \sigma\rangle \downarrow \sigma'}{p_0 \vdash \langle \mathbf{if}\ b\ \mathbf{then\ goto}\ \ell; p_1, \sigma\rangle \downarrow \sigma'}$$

$$\text{P-IfGotoF} : \frac{\langle b, \sigma\rangle \downarrow \mathbf{false} \qquad p_0 \vdash \langle p_1, \sigma\rangle \downarrow \sigma'}{p_0 \vdash \langle \mathbf{if}\ b\ \mathbf{then\ goto}\ \ell; p_1, \sigma\rangle \downarrow \sigma'}$$

Judgment $\boxed{p_0 \vdash \ell \Downarrow p}$:

$$\text{L-LabS} : \frac{}{\ell\colon p_1 \vdash \ell \Downarrow p_1} \qquad \text{L-LabD} : \frac{p_1 \vdash \ell \Downarrow p}{\ell'\colon p_1 \vdash \ell \Downarrow p}\ (\ell' \neq \ell) \qquad \text{L-Simple} : \frac{p_1 \vdash \ell \Downarrow p}{s; p_1 \vdash \ell \Downarrow p}$$

Figure 1: Big-step operational semantics of LIMP

c	$[\![c]\!]$
skip	(empty sequence)
$X := a$	$X := a;$
$c_0; c_1$	$[\![c_0]\!]$ $[\![c_1]\!]$
if b **then** c_0 **else** c_1	**if** b **then goto** ℓ; $[\![c_1]\!]$ **goto** ℓ'; ℓ: $[\![c_0]\!]$ ℓ':
while b **do** c_0	ℓ: **if** $\neg b$ **then goto** ℓ'; $[\![c_0]\!]$ **goto** ℓ; ℓ':

Figure 2: Compilation from IMP to LIMP (informally)

$\boxed{[\![c]\!] \mathbin{@} p \rightsquigarrow^i_{i'} p'}$:

$$\text{C-Skip} : \frac{}{[\![\mathbf{skip}]\!] \mathbin{@} p \rightsquigarrow^i_i p} \qquad \text{C-Assign} : \frac{}{[\![X := a]\!] \mathbin{@} p \rightsquigarrow^i_i X := a; p}$$

$$\text{C-Seq} : \frac{[\![c_1]\!] \mathbin{@} p \rightsquigarrow^i_{i''} p'' \qquad [\![c_0]\!] \mathbin{@} p'' \rightsquigarrow^{i''}_{i'} p'}{[\![c_0; c_1]\!] \mathbin{@} p \rightsquigarrow^i_{i'} p'}$$

$$\text{C-If} : \frac{[\![c_0]\!] \mathbin{@} \ell_{i+1}\colon p \rightsquigarrow^{i+2}_{i''} p'' \qquad [\![c_1]\!] \mathbin{@} \mathbf{goto}\ \ell_{i+1}; \ell_i\colon p'' \rightsquigarrow^{i''}_{i'} p'''}{[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!] \mathbin{@} p \rightsquigarrow^i_{i'} \mathbf{if}\ b\ \mathbf{then\ goto}\ \ell_i; p'''}$$

$$\text{C-While} : \frac{[\![c_0]\!] \mathbin{@} \mathbf{goto}\ \ell_i; \ell_{i+1}\colon p \rightsquigarrow^{i+2}_{i'} p''}{[\![\mathbf{while}\ b\ \mathbf{do}\ c_0]\!] \mathbin{@} p \rightsquigarrow^i_{i'} \ell_i\colon \mathbf{if}\ \neg b\ \mathbf{then\ goto}\ \ell_{i+1}; p''}$$

$\boxed{[\![c]\!]^{\text{top}} \rightsquigarrow p}$

$$\text{CT-Com} : \frac{[\![c]\!] \mathbin{@} \mathbf{halt} \rightsquigarrow^0_{i'} p}{[\![c]\!]^{\text{top}} \rightsquigarrow p}$$

Figure 3: Compilation from IMP to LIMP (formally)

We now want to prove that the translation preserves the semantics of IMP programs. Specifically, we want to show:

Theorem 1 *Suppose* $[\![c]\!]^{\text{top}} \rightsquigarrow p$. *If* $\langle c, \sigma\rangle \downarrow \sigma'$, *then* $p \vdash \langle p, \sigma\rangle \downarrow \sigma'$.

(The other direction also holds, but we will not show it here.)
To show the theorem, we need a little more machinery. We write $p_0 \sqsupseteq p_1$ when p_1 is a *suffix* of p_0, i.e., if p_1 can be obtained by removing zero or more simple commands and/or label declarations from the start of p_0. In particular, $p_1 \vdash \ell \Downarrow p$ evidently implies $p_0 \vdash \ell \Downarrow p$ (as long as p_0 is well formed). We also observe that, if $[\![c]\!] \mathbin{@} p \rightsquigarrow^i_{i'} p'$, then $p' \sqsupseteq p$. (The proof is a simple induction on the translation derivation, using that $\sqsupseteq$ is transitive.)

Lemma 2 *Let* p *be such that all* ℓ_j *declared in* p *have* $j < i$, *and suppose* $[\![c]\!] \mathbin{@} p \rightsquigarrow^i_{i'} p'$ *(by* $\mathcal{C}$*), and* $p_0 \sqsupseteq p'$. *If* $\langle c, \sigma\rangle \downarrow \sigma''$ *(by* $\mathcal{E}$*), and* $p_0 \vdash \langle p, \sigma''\rangle \downarrow \sigma'$ *(by* $\mathcal{P}$*), then* $p_0 \vdash \langle p', \sigma\rangle \downarrow \sigma'$ *(by some* $\mathcal{P}'$*).*

We can paraphrase the lemma as: "Executing the IMP command c, and then the LIMP program fragment p (in the context of some larger program p_0), is equivalent to executing the LIMP program p' obtained by prepending the translation of c to p."

Proof. By induction on the derivation $\mathcal{E}$. We show the cases for EC-Assign and EC-IfT:

- Case $\mathcal{E} = \text{EC-Assign}\dfrac{\begin{array}{c}\mathcal{E}_0\\ \langle a, \sigma\rangle \downarrow n\end{array}}{\langle X := a, \sigma\rangle \downarrow \sigma[X \mapsto n]}$, so $c = (X := a)$, and $\sigma'' = \sigma[X \mapsto n]$.

  In this case, the compilation derivation must look as follows:

$$\mathcal{C} = \text{C-Assign}\frac{}{[\![X := a]\!] \mathbin{@} p \rightsquigarrow^i_i X := a; p}$$

and in particular $p' = (X := a; p)$. Since $\mathcal{P}$ is then a derivation of $p_0 \vdash \langle p, \sigma[X \mapsto n]\rangle \downarrow \sigma'$, we can directly take:

$$\mathcal{P}' = \textsc{p-Assign}\frac{\overset{\mathcal{E}_0}{\langle a, \sigma\rangle \downarrow n} \qquad \overset{\mathcal{P}}{p_0 \vdash \langle p, \sigma[X \mapsto n]\rangle \downarrow \sigma'}}{p_0 \vdash \langle X := a; p, \sigma\rangle \downarrow \sigma'}$$

- Case $\mathcal{E} = \textsc{ec-IfT}\dfrac{\overset{\mathcal{E}_0}{\langle b, \sigma\rangle \downarrow \mathbf{true}} \qquad \overset{\mathcal{E}_1}{\langle c_0, \sigma\rangle \downarrow \sigma''}}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma\rangle \downarrow \sigma''}$, so $c = \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1$.

  Here we must have:

$$\mathcal{C} = \textsc{c-If}\frac{\overset{\mathcal{C}_0}{[\![c_0]\!] \mathbin{@} \ell_{i+1}\colon p \rightsquigarrow_{i''}^{i+2} p''} \qquad \overset{\mathcal{C}_1}{[\![c_1]\!] \mathbin{@} \mathbf{goto}\ \ell_{i+1}; \ell_i\colon p'' \rightsquigarrow_{i'}^{i''} p'''}}{[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!] \mathbin{@} p \rightsquigarrow_{i'}^{i} \mathbf{if}\ b\ \mathbf{then\ goto}\ \ell_i; p'''},$$

  so $p' = \mathbf{if}\ b\ \mathbf{then\ goto}\ \ell_i; p'''$. Let

$$\mathcal{P}_0 = \textsc{p-Lab}\frac{\overset{\mathcal{P}}{p_0 \vdash \langle p, \sigma''\rangle \downarrow \sigma'}}{p_0 \vdash \langle \ell_{i+1}\colon p, \sigma''\rangle \downarrow \sigma'}$$

  By IH on $\mathcal{E}_1$ with $\mathcal{C}_0$ and $\mathcal{P}_0$, we then get a derivation $\mathcal{P}_0'$ of $p_0 \vdash \langle p'', \sigma\rangle \downarrow \sigma'$.
  We can now take $\mathcal{L} = \textsc{l-LabS}\dfrac{}{\ell_i\colon p'' \vdash \ell_i \Downarrow p''}$, and since (using the observation above on $\mathcal{C}_1$),

$$p_0 \sqsupseteq p' = \mathbf{if}\ b\ \mathbf{then\ goto}\ \ell_i; p''' \sqsupseteq p''' \sqsupseteq \mathbf{goto}\ \ell_{i+1}; \ell_i\colon p'' \sqsupseteq \ell_i\colon p''$$

  then also $p_0 \vdash \ell_i \Downarrow p''$ by some $\mathcal{L}_0$. And thus, we can take:

$$\mathcal{P}' = \textsc{p-IfGotoT}\frac{\overset{\mathcal{E}_0}{\langle b, \sigma\rangle \downarrow \mathbf{true}} \qquad \overset{\mathcal{L}_0}{p_0 \vdash \ell_i \Downarrow p''} \qquad \overset{\mathcal{P}_0'}{p_0 \vdash \langle p'', \sigma\rangle \downarrow \sigma'}}{p_0 \vdash \langle \mathbf{if}\ b\ \mathbf{then\ goto}\ \ell_i; p''', \sigma\rangle \downarrow \sigma'}$$

Question 1.1 (20%)

a. Complete the missing cases in Lemma 2 (i.e., for EC-Skip, EC-Seq, EC-IfF, EC-WhileF, and EC-WhileT).

b. Show Theorem 1.

2 A program logic for LIMP

Note: the questions in this part can be answered completely independently of the ones in Section 1.1.

We now consider formal verification of LIMP programs, intended as part of a proof-carrying code system for a (relatively) low-level language with unstructured control flow, similar to machine code. We still use the big-step semantics of programs from Figure 1 (which again shares the semantics of arithmetic and boolean expressions with standard IMP).

The syntax and semantics of assertions A are also exactly like in IMP. We can then specify when, in the context of the complete LIMP program p_0, the program fragment p semantically satisfies a partial-correctness specification:

$$p_0 \models \{A\}\, p\, \{B\} \iff \forall \sigma, \sigma'.\, \sigma \models A \wedge p_0 \vdash \langle p, \sigma\rangle \downarrow \sigma' \Rightarrow \sigma' \models B\,.$$

An *annotation* of a LIMP program p_0 is a function Δ, assigning an assertion to every label declared in p_0. Given an annotation of the program, and a desired postcondition B, we can define two judgments, $\boxed{\Delta/B \vdash \{A\}\, p}$ and $\boxed{\Delta/B \Vdash \{A\}\, p}$, with the rules given in Figure 4. Note that, since Δ and B are the same throughout the derivation, we may state them once and for all, and elide them from the derivation itself.

The intuitive reading of $\Delta/B \vdash \{A\}\, p$ is that A is a *sufficient* precondition on the initial state, for B to hold in the final state (if the program terminates at all). $\Delta/B \Vdash \{A\}\, p$ says that A is also the *weakest* such precondition, consistent with the annotations of the program. Rule V-Weak says that A is a sufficient precondition precisely when it implies the weakest precondition A'. Rule W-Lab then says that the designated precondition for a label must imply the weakest precondition of the program fragment following the label. Rule W-Halt simply says that the weakest precondition of **halt** is precisely the postcondition B of the entire program; and W-Assign is just the usual rule for assignments from Hoare logic, pushing the weakest precondition A_1 of p_1 backwards through $X := a$. In rule W-IfGoto, the weakest precondition is just strong enough to cover both the possibility of taking the jump to ℓ (in which case, we must ensure that $\Delta(\ell)$ holds, and the possibility of falling through to the next statement (in which case we must ensure that *its* precondition holds).

Note that the weakest precondition A is uniquely determined by p, B, and Δ. That is, if $\Delta/B \Vdash \{A\}\, p$ and $\Delta/B \Vdash \{A'\}\, p$, then $A = A'$. This also means that all of the semantic conditions to verify in the uses of V-Weak are uniquely determined by the program, its pre- and postcondition, and the chosen annotation for the labels.

2.1 Verification of a summation program

Consider the following complete program S (the line numbers in the first column are just for reference):

```
1        s := 0;
2    loop:
3        s := s + x;
4        if x = 1 then goto done;
5        x := x − 1;
6        goto loop;
7    done:
8        s := s + s;
9        halt
```

$\boxed{\Delta/B \vdash \{A\}\, p}$:

$$\textsc{V-Weak} : \frac{\models A \Rightarrow A' \qquad \Delta/B \Vdash \{A'\}\, p}{\Delta/B \vdash \{A\}\, p}$$

$\boxed{\Delta/B \Vdash \{A\}\, p}$:

$$\textsc{W-Halt} : \frac{}{\Delta/B \Vdash \{B\}\, \mathbf{halt}} \qquad \textsc{W-Lab} : \frac{\Delta/B \vdash \{\Delta(\ell)\}\, p_1}{\Delta/B \Vdash \{\Delta(\ell)\}\, \ell\colon p_1}$$

$$\textsc{W-Assign} : \frac{\Delta/B \Vdash \{A_1\}\, p_1}{\Delta/B \Vdash \{A_1[a/X]\}\, X := a; p_1}$$

$$\textsc{W-IfGoto} : \frac{\Delta/B \Vdash \{A_1\}\, p_1}{\Delta/B \Vdash \{(b \Rightarrow \Delta(\ell)) \wedge (\neg b \Rightarrow A_1)\}\, \mathbf{if}\ b\ \mathbf{then\ goto}\ \ell; p_1}$$

Figure 4: Program logic for LIMP

(We may note that, since the exit is from the *middle* of the loop, this LIMP program couldn't be the result of compiling any IMP program.)

Also, let $PRE \equiv (\wedge x = n)$, and $POST \equiv (s = n \times (n+1))$. (Note that the location n, representing the initial value of x, is intentionally never used or modified in the program itself.) We want to verify that S is partially correct with respect to these pre- and post-conditions:

Question 2.1 (15%) Find a suitable annotation Δ for the labels in S, and show that $\Delta/POST \vdash \{PRE\}\, S$. To reduce the syntactic overhead, you are allowed to use the following simplified proof rule for unconditional jumps:

$$\textsc{W-Goto} : \frac{\Delta/B \Vdash \{A_1\}\, p_1}{\Delta/B \Vdash \{\Delta(\ell)\}\, \mathbf{goto}\ \ell; p_1}$$

(This rule is also easily shown to be sound.) *Hint:* As usual, once you have determined the annotation assertions for the labels, work *backwards* through the program to determine the assertion that must hold at each program point, and the corresponding subderivations in the proof.
You may refer to the program fragment starting at line i as S_i. For example, S_7 is the fragment "*done*: $s := s + s$; **halt**", and S_1 is the whole program. Make it clear what the derivation is for every S_i, and why the semantic implications in the uses of V-Weak (i.e., the verification conditions) hold. You may also use a full-annotation proof style, with an explicit precondition for every line, as long as you explain clearly how it can be converted to a formal derivation using the two proof judgments ($\vdash$ and $\Vdash$).

2.2 Soundness of program logic

We want to show that the program logic for LIMP is sound with respect to the operational semantics. That is, we want to show the following theorem:

Theorem 3 (Soundness for programs) *If* $\Delta/B \vdash \{A\}\, p$ *then* $p \models \{A\}\, p\, \{B\}$.

For showing the theorem, we will use the following lemmas:

Lemma 4 (Label lookup) *If* $\Delta/B \Vdash \{A_0\}\, p_0$ *and* $p_0 \vdash \ell \Downarrow p$, *then* $\Delta/B \vdash \{\Delta(\ell)\}\, p$.

Proof. By induction on the lookup derivation.

Lemma 5 (Soundness for fragments) *If* $\Delta/B \Vdash \{A_0\}\, p_0$ *and* $\Delta/B \vdash \{A\}\, p$, *then* $p_0 \models \{A\}\, p\, \{B\}$.

Proof. Let $\mathcal{W}_0$ be the derivation of $\Delta/B \Vdash \{A_0\}\, p_0$, and $\mathcal{V}$ of $\Delta/B \vdash \{A\}\, p$. To show $p_0 \models \{A\}\, p\, \{B\}$, let σ and σ' be given, with $\sigma \models A$ and $p_0 \vdash \langle p, \sigma\rangle \downarrow \sigma'$ by some $\mathcal{P}$; we must show $\sigma' \models B$. We first note that $\mathcal{V}$ must have the following shape:

$$\mathcal{V} = \textsc{v-Weak}\frac{\models A \Rightarrow A' \qquad \overset{\mathcal{W}}{\Delta/B \Vdash \{A'\}\, p}}{\Delta/B \vdash \{A\}\, p}.$$

In particular, from the first premise and $\sigma \models A$, we also know $\sigma \models A'$. The proof is now by induction on $\mathcal{P}$. A sample case is the following:

- Case $\mathcal{P} = \textsc{p-Assign}\dfrac{\overset{\mathcal{E}}{\langle a, \sigma\rangle \downarrow n} \qquad \overset{\mathcal{P}_1}{p_0 \vdash \langle p_1, \sigma[X \mapsto n]\rangle \downarrow \sigma'}}{p_0 \vdash \langle X := a; p_1, \sigma\rangle \downarrow \sigma'}$, so $p = (X := a; p_1)$. Then $\mathcal{W}$ must have shape:

  $$\mathcal{W} = \textsc{w-Assign}\frac{\overset{\mathcal{W}_1}{\Delta/B \Vdash \{A_1\}\, p_1}}{\Delta/B \Vdash \{A_1[a/X]\}\, X := a; p_1}$$

  so $A' = A_1[a/X]$, and thus $\sigma \models A_1[a/X]$. By Lemma 3.6($\Rightarrow$) on $\mathcal{E}$, this means that $\sigma[X \mapsto n] \models A_1$. By V-Weak on $\mathcal{W}_1$, using the trivial implication $\models A_1 \Rightarrow A_1$, we get a corresponding $\mathcal{V}_1$ of $\Delta/B \vdash \{A_1\}\, p_1$. But then, by IH on $\mathcal{P}_1$ with $\mathcal{V}_1$, we get the required $\sigma' \models B$.

- (The remaining cases are omitted.) ∎

Note now that Theorem 3 is an immediate corollary of Lemma 5 (taking $p_0 = p$ and $A_0 = A'$ from the use of V-Weak ending the derivation of $\Delta/B \vdash \{A\}\, p$).

Question 2.2 (15%)

a. Show Lemma 4.

b. Show Lemma 5. (You don't need to consider the extra rule W-Goto, which is essentially just a special case of W-IfGoto.)

3 FUN with lists

Consider an extension of the FUN language from the notes with a facility for working with finite lists. Specifically, we extend the syntax of terms, canonical forms, and types:

$$\begin{array}{rcl} t & ::= & \cdots \mid [] \mid t_1 :: t_2 \mid \mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,(t_0) \\ c & ::= & \cdots \mid [] \mid c_1 :: c_2 \\ \tau & ::= & \cdots \mid \mathbf{list}\,(\tau_0) \end{array}$$

Here, [] represents the empty list, while $t_1 :: t_2$ is the list with head t_1 and tail t_2. Syntactically, :: is considered right-associative; e.g., $\overline{1} :: \overline{2} :: []$ parses as $\overline{1} :: (\overline{2} :: [])$. For **fold**, the variables x and y, which must be different, are bound in t_c. $\mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,(t_0)$ corresponds to `foldr (\`x y `->` t_c`)` t_n t_0 in Haskell, or `List.foldBack (fun` x y `->` t_c`)` t_0 t_n in F#, but note that the order in which the various subterms are evaluated may be different. Using **fold**, many computations on lists can be expressed without explicit recursion; for example, we could define the list-append operation as $t_1 \mathbin{@} t_2 \equiv \mathbf{fold}\,(t_2, x.y.x :: y)\,(t_1)$. (It is also possible to define the **head**, and even **tail**, operations.)

The big-step, small-step, and typing rules for the new constructs are shown in Figure 5.

Question 3.1 (20%) Prove that the equivalence between the big-step and small-step semantics still holds for the extended language. Specifically:

a. Show Theorem 4.2 for the new cases.

b. Show Lemma 4.4 for the new cases.

(The proof of Lemma 4.3 extends immediately.)

Question 3.2 (18%) The existing proof (sketches) of Weakening (Lemma 4.9), Substitution (Lemma 4.10), and Progress (Lemma 4.8) extend quite straightforwardly. For the remaining two key properties of FUN, there is a little more work to do:

a. Prove Preservation (Lemma 4.11) for the new cases. As usual, you don't need to cover the context rules explicitly.

b. Prove Termination (Theorem 4.13) for the new cases. Start by defining the predicate $\models^\mathrm{c} c : \tau$ for the case $\tau = \mathbf{list}\,(\tau_0)$; make sure this is a proper inductive definition.

Hint: in the proof case for $\mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,(t_0)$, you will probably want to do an *inner* induction on the length (or structure) of the list that $t_0[s]$ evaluates to.

Question 3.3 (12%) Finally, we may consider some properties of the typing judgment itself, independent of the operational semantics:

a. Give the equivalent constraint-typing rules (judgment $\boxed{\hat{\Gamma} \vdash^i t : \hat{\tau} \mid^{i'} C}$ from Chapter 6) for the new constructs.

b. Let t be a term and n a variable. Consider the following purported typing equivalence:

$$\Gamma \vdash \mathbf{let}\ n \Leftarrow []\ \mathbf{in}\ t : \tau \iff^{?} \Gamma \vdash t[[]/n] : \tau$$

For each direction of the biimplication, either give a proof that it holds for *all* instantiations of Γ, t, and τ; or give a *specific* counterexample, for which you show that it does not.

Judgment $\boxed{t \downarrow c}$:

$$\text{E-Nil} : \frac{}{[] \downarrow []} \qquad \text{E-Cons} : \frac{t_1 \downarrow c_1 \qquad t_2 \downarrow c_2}{t_1 :: t_2 \downarrow c_1 :: c_2} \qquad \text{E-FoldN} : \frac{t_0 \downarrow [] \qquad t_\mathrm{n} \downarrow c}{\mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,(t_0) \downarrow c}$$

$$\text{E-FoldC} : \frac{t_0 \downarrow c_1 :: c_2 \qquad \mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,(c_2) \downarrow c' \qquad t_\mathrm{c}[c_1/x][c'/y] \downarrow c}{\mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,(t_0) \downarrow c}$$

Judgment $\boxed{t \to t'}$:

$$\text{(no rules for [])} \qquad \text{S-Cons1} : \frac{t_1 \to t_1'}{t_1 :: t_2 \to t_1' :: t_2} \qquad \text{S-Cons2} : \frac{t_2 \to t_2'}{c_1 :: t_2 \to c_1 :: t_2'}$$

$$\text{S-Fold1} : \frac{t_0 \to t_0'}{\mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,(t_0) \to \mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,(t_0')}$$

$$\text{S-FoldN} : \frac{}{\mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,([]) \to t_\mathrm{n}}$$

$$\text{S-FoldC} : \frac{}{\mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,(c_1 :: c_2) \to \mathbf{let}\ y \Leftarrow \mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,(c_2)\ \mathbf{in}\ t_\mathrm{c}[c_1/x]}$$

Judgment $\boxed{\Gamma \vdash t : \tau}$:

$$\text{T-Nil} : \frac{}{\Gamma \vdash [] : \mathbf{list}\,(\tau_0)} \qquad \text{T-Cons} : \frac{\Gamma \vdash t_1 : \tau_0 \qquad \Gamma \vdash t_2 : \mathbf{list}\,(\tau_0)}{\Gamma \vdash t_1 :: t_2 : \mathbf{list}\,(\tau_0)}$$

$$\text{T-Fold} : \frac{\Gamma \vdash t_\mathrm{n} : \tau \qquad \Gamma[x \mapsto \tau_0][y \mapsto \tau] \vdash t_\mathrm{c} : \tau \qquad \Gamma \vdash t_0 : \mathbf{list}\,(\tau_0)}{\Gamma \vdash \mathbf{fold}\,(t_\mathrm{n}, x.y.t_\mathrm{c})\,(t_0) : \tau}$$

Figure 5: Big-step, small-step, and typing rules for lists