Final Exam, *Semantics and Types* (2022/23)

Andrzej Filinski

28 March 2023, 9:00 – 29 March 2023, 17:00

This 16-page exam consists of five independent parts, each containing one or two loosely related questions. The exam will be graded as a whole, with all parts weighted approximately equally in 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, or on Discord), 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 "Confirm" 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. 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 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, 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 Imperative semantics

[The notation and definitions in this part refer to the Notes Chapter 2.]

We extend IMP with the following additional command forms:

$$c ::= \cdots \mid \textbf{check}\ b \mid \textbf{pick}\ c_0\ \textbf{or}\ c_1 \mid \textbf{iterate}\ c_0$$

(The syntax and semantics of arithmetic and boolean expressions are unmodified.) Informally, **check** b verifies that the boolean expression b evaluates to **true**; otherwise, the command fails. **pick** c_0 **or** c_1 executes either c_0 or c_1, and **iterate** c_0 executes the command c_0 some number of times (possibly none).

Formally, we extend the big-step operational semantics with the following rules for the judgment $\boxed{\langle c, \sigma\rangle \downarrow \sigma'}$:

$$\text{EC-CheckT} : \frac{\langle b, \sigma\rangle \downarrow \textbf{true}}{\langle \textbf{check}\ b, \sigma\rangle \downarrow \sigma} \qquad \text{(no rule for } \textbf{check}\ b \text{ when } b \text{ is false)}$$

$$\text{EC-PickL} : \frac{\langle c_0, \sigma\rangle \downarrow \sigma'}{\langle \textbf{pick}\ c_0\ \textbf{or}\ c_1, \sigma\rangle \downarrow \sigma'} \qquad \text{EC-PickR} : \frac{\langle c_1, \sigma\rangle \downarrow \sigma'}{\langle \textbf{pick}\ c_0\ \textbf{or}\ c_1, \sigma\rangle \downarrow \sigma'}$$

$$\text{EC-IterZ} : \frac{}{\langle \textbf{iterate}\ c_0, \sigma\rangle \downarrow \sigma} \qquad \text{EC-IterM} : \frac{\langle c_0, \sigma\rangle \downarrow \sigma'' \qquad \langle \textbf{iterate}\ c_0, \sigma''\rangle \downarrow \sigma'}{\langle \textbf{iterate}\ c_0, \sigma\rangle \downarrow \sigma'}$$

Note that the extended language is highly nondeterministic. However, it is possible to use the new constructs in a restricted way, to simulate the standard conditionals and loops of IMP. In particular, one can easily show that, for any b, c_0 and c_1,

$$\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1 \sim \textbf{pick}\ (\textbf{check}\ b; c_0)\ \textbf{or}\ (\textbf{check}\ \neg b; c_1)$$

by considering the possible execution derivations for each side. More interestingly, we can also simulate **while**-loops:

Question 1 Show *both* directions of the following equivalence:

$$\textbf{while}\ b\ \textbf{do}\ c_0 \sim \textbf{iterate}\ (\textbf{check}\ b; c_0); \textbf{check}\ \neg b$$

Hint: use induction on derivations, with a suitable IH for each direction.

2 Hoare logic

[The notation and definitions in this part refer to the Notes Chapter 3.]

For this question, we consider the syntax of arithmetic expressions in IMP to be extended with two additional constructs:

$$a ::= \cdots \mid a_0 \mathbin{//} a_1 \mid a_0 \mathbin{\%} a_1$$

Syntactically, // and % are left-associative, with the same precedence as $\times$. Their semantics is analogous to the other arithmetic operations, with // computing the integer quotient, and % computing the remainder. Formally, for all integers n, and $d > 0$, we have $0 \leq n \mathbin{\%} d < d$, and

$$n = (n \mathbin{//} d) \times d + n \mathbin{\%} d \tag{2.1}$$

This also means that, for every m,

$$(m \times d + n) \mathbin{\%} d = n \mathbin{\%} d \tag{2.2}$$

(The behavior of // and % when d is negative or zero is unspecified, and not relevant for the following.) Consider now the following program *MOD9*:

$s := t;$
while $10 \leq s$ **do**
  $(t := 0;$
   while $\neg(s = 0)$ **do**
     $(t := t + s \mathbin{\%} 10;$
      $s := s \mathbin{//} 10);$
   $s := t);$
if $s + 1 = 10$ **then**
  $s := 0$
else
  skip

Question 2 Prove the following partial correctness assertion:

$$\{t = i \wedge 0 \leq i\}\ \mathit{MOD9}\ \{s = i \mathbin{\%} 9\}$$

Use the "fully annotated program" style. Make it very clear exactly where you use semantic reasoning in the derivation, and why the implications involved are valid. (You may use equation (2.2) without separate proof.)
Hint: Start by determining the invariants. Use $(s + t) \mathbin{\%} 9 = i \mathbin{\%} 9$ as the *core* of the invariant of the inner loop. (You may need to extend it with additional conjuncts, to make the full derivation go through.)

3 Functional semantics

[The notation and definitions in this part refer to the Notes Chapter 4.]

Consider the extension of FUN with an alternative conditional:

$$t ::= \cdots \mid \textbf{pif } t_0 \textbf{ then } t_1 \textbf{ else } t_2$$

Its big-step semantics, $\boxed{t \downarrow c}$, is given by the following three rules:

$$\text{E-PIFT} : \frac{t_0 \downarrow \textbf{true} \qquad t_1 \downarrow c}{\textbf{pif } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \downarrow c} \qquad \text{E-PIFF} : \frac{t_0 \downarrow \textbf{false} \qquad t_2 \downarrow c}{\textbf{pif } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \downarrow c}$$

$$\text{E-PIFX} : \frac{t_1 \downarrow c \qquad t_2 \downarrow c}{\textbf{pif } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \downarrow c}$$

The first two are standard, but the third one says that, if both branches evaluate to exactly the same canonical form, then the whole term can also immediately evaluate to that canonical form, without necessarily evaluating the test t_0. Note that this semantics is still deterministic. (This can be shown by a simple extension of the standard determinism proof by induction on derivations.)

The small-step semantics, $\boxed{t \to t'}$, of **pif** consists of three context rules and three computation rules:

$$\text{S-PIF0} : \frac{t_0 \to t_0'}{\textbf{pif } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to \textbf{pif } t_0' \textbf{ then } t_1 \textbf{ else } t_2}$$

$$\text{S-PIF1} : \frac{t_1 \to t_1'}{\textbf{pif } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to \textbf{pif } t_0 \textbf{ then } t_1' \textbf{ else } t_2}$$

$$\text{S-PIF2} : \frac{t_2 \to t_2'}{\textbf{pif } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to \textbf{pif } t_0 \textbf{ then } t_1 \textbf{ else } t_2'}$$

$$\text{S-PIFT} : \frac{}{\textbf{pif true then } t_1 \textbf{ else } t_2 \to t_1} \qquad \text{S-PIFF} : \frac{}{\textbf{pif false then } t_1 \textbf{ else } t_2 \to t_2}$$

$$\text{S-PIFX} : \frac{}{\textbf{pif } t_0 \textbf{ then } c \textbf{ else } c \to c}$$

This semantics is *not* deterministic at the single-step level, because a term may be reduced in different ways by overlapping rules. However, we can show that multi-step reduction to a canonical form remains deterministic, and in fact agrees with the big-step semantics:

Question 3.1 Extend Theorem 4.2 in the Notes (i.e., if $t \downarrow c$, then $t \to^* c$) with all the relevant cases related to the **pif** construct.

Question 3.2 Extend Lemma 4.4 in the Notes (i.e., if $t \to t'$ and $t' \downarrow c$, then $t \downarrow c$) with all the relevant cases related to the **pif** construct.

4 Type systems

[The notation and definitions in this part refer to the Notes Chapter 4]

Consider the following language, a tiny subset of FUN:

$$t ::= \overline{n} \mid x \mid t_0 + t_1 \mid \lambda x.\, t_0 \mid t_1\, t_2$$

As usual, the canonical forms c are the numerals and lambda-abstractions. The big-step operational semantics is also a restriction of FUN's, specifically consisting of the rules E-Num, E-Plus, E-Lam, and E-App. Similarly, the small-step semantics consists of rules S-Plus1, S-Plus2, S-Plus, S-App1, S-App2, and S-App.

The type syntax, on the other hand, contains an extension:

$$\tau ::= \mathbf{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \,\&\, \tau_2$$

The typing judgment $\boxed{\Gamma \vdash t : \tau}$ includes the standard FUN rules T-Num, T-Var, T-Plus, T-Lam, and T-App. The "&" type constructor is used to type terms that have *both* of the component types; the new typing rules are:

$$\text{T-Both} : \frac{\Gamma \vdash t : \tau_1 \qquad \Gamma \vdash t : \tau_2}{\Gamma \vdash t : \tau_1 \,\&\, \tau_2} \qquad \text{T-Left} : \frac{\Gamma \vdash t : \tau_1 \,\&\, \tau_2}{\Gamma \vdash t : \tau_1} \qquad \text{T-Right} : \frac{\Gamma \vdash t : \tau_1 \,\&\, \tau_2}{\Gamma \vdash t : \tau_2}$$

For example, the identity function $I = \lambda x.\, x$ has (among others) the type:

$$(\mathbf{int} \to \mathbf{int}) \,\&\, ((\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to \mathbf{int})),$$

meaning that it can be safely used on both integers and integer-to-integer functions. Moreover, $\omega = \lambda x.\, x\, x$ has (again, among others) the type $(\tau \,\&\, (\tau \to \tau)) \to \tau$ for any τ, whereas it was clearly untypable in the original FUN type system. And in particular, we now also have $[] \vdash \omega\, I : \mathbf{int} \to \mathbf{int}$. (But $\Omega = \omega\, \omega$ remains untypable.)

We say that a type τ *covers* another type τ', if $\tau = \cdots \,\&\, \tau' \,\&\, \cdots$. More precisely, we define the judgment $\boxed{\tau \sqsubseteq \tau'}$ by the rules:

$$\text{C-Eq} : \frac{}{\tau \sqsubseteq \tau} \qquad \text{C-Left} : \frac{\tau_1 \sqsubseteq \tau'}{\tau_1 \,\&\, \tau_2 \sqsubseteq \tau'} \qquad \text{C-Right} : \frac{\tau_2 \sqsubseteq \tau'}{\tau_1 \,\&\, \tau_2 \sqsubseteq \tau'}$$

Question 4.1 Prove Progress for the language, i.e., if $[] \vdash t : \tau$ then either t is a canonical form, or there exists a t' such that $t \to t'$.
Hint: Prove first the following canonical-forms lemma: If $[] \vdash c : \tau$, and $\tau \sqsubseteq \mathbf{int}$, then $c = \overline{n}$ for some n; and if $\tau \sqsubseteq \tau_1 \to \tau_2$, then $c = \lambda x.\, t$ for some x and t.

(Preservation also holds, but you are not asked to prove that.)

Question 4.2 Prove Termination for the language, i.e., if $[] \vdash t : \tau$, then there exists a c such that $t \downarrow c$.
Hint: Use the proof strategy of Theorem 4.13, taking $\models^{\mathrm{c}} c : \tau_1 \,\&\, \tau_2$ as the *conjunction* of the semantic typing conditions for τ_1 and τ_2. If some proof cases are *identical* to the ones in the notes, you may simply say so, rather than copying them into your answer.

5 Denotational semantics

[The notation and definitions in this part refer to the Notes Chapter 7.]

We extend the syntax of IMP with a simple exception facility, by adding two new command forms:

$$c ::= \cdots \mid \textbf{throw}\ a \mid \textbf{try}\ c_0\ \textbf{catch}\ X : c_1$$

Informally, **throw** a signals an error condition, identified by the *non-zero* number that a evaluates to. (If a evaluates to 0, **throw** a behaves like a **skip** instead.) **try** c_0 **catch** X : c_1 first tries to execute c_0; if this succeeds (i.e., if c_0 does not throw an exception), the whole command behaves just like c_0. But if c_0 throws some exception $n \neq 0$, the execution of c_0 is aborted, and the handler c_1 is executed instead, with X set to n.

Formally, we define the operational semantics of commands of the extended language in terms of a new judgment form $\boxed{\langle c, \sigma\rangle \downarrow \langle \sigma', n\rangle}$, given by the rules:

$$\text{EC-Skip} : \frac{}{\langle \textbf{skip}, \sigma\rangle \downarrow \langle \sigma, 0\rangle} \qquad \text{EC-Assign} : \frac{\langle a, \sigma\rangle \downarrow n_0}{\langle X := a, \sigma\rangle \downarrow \langle \sigma[X \mapsto n_0], 0\rangle}$$

$$\text{EC-SeqN} : \frac{\langle c_0, \sigma\rangle \downarrow \langle \sigma'', 0\rangle \qquad \langle c_1, \sigma''\rangle \downarrow \langle \sigma', n\rangle}{\langle c_0; c_1, \sigma\rangle \downarrow \langle \sigma', n\rangle} \qquad \text{EC-SeqE} : \frac{\langle c_0, \sigma\rangle \downarrow \langle \sigma', n\rangle}{\langle c_0; c_1, \sigma\rangle \downarrow \langle \sigma', n\rangle}(n \neq 0)$$

$$\text{EC-IfT} : \frac{\langle b, \sigma\rangle \downarrow \textbf{true} \qquad \langle c_0, \sigma\rangle \downarrow \langle \sigma', n\rangle}{\langle \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, \sigma\rangle \downarrow \langle \sigma', n\rangle} \qquad \text{EC-IfF} : \frac{\langle b, \sigma\rangle \downarrow \textbf{false} \qquad \langle c_1, \sigma\rangle \downarrow \langle \sigma', n\rangle}{\langle \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, \sigma\rangle \downarrow \langle \sigma', n\rangle}$$

$$\text{EC-WhileTN} : \frac{\langle b, \sigma\rangle \downarrow \textbf{true} \qquad \langle c_0, \sigma\rangle \downarrow \langle \sigma'', 0\rangle \qquad \langle \textbf{while}\ b\ \textbf{do}\ c_0, \sigma''\rangle \downarrow \langle \sigma', n\rangle}{\langle \textbf{while}\ b\ \textbf{do}\ c_0, \sigma\rangle \downarrow \langle \sigma', n\rangle}$$

$$\text{EC-WhileTE} : \frac{\langle b, \sigma\rangle \downarrow \textbf{true} \qquad \langle c_0, \sigma\rangle \downarrow \langle \sigma', n\rangle}{\langle \textbf{while}\ b\ \textbf{do}\ c_0, \sigma\rangle \downarrow \langle \sigma', n\rangle}(n \neq 0) \qquad \text{EC-WhileF} : \frac{\langle b, \sigma\rangle \downarrow \textbf{false}}{\langle \textbf{while}\ b\ \textbf{do}\ c_0, \sigma\rangle \downarrow \langle \sigma, 0\rangle}$$

$$\text{EC-Throw} : \frac{\langle a, \sigma\rangle \downarrow n}{\langle \textbf{throw}\ a, \sigma\rangle \downarrow \langle \sigma, n\rangle} \qquad \text{EC-TryN} : \frac{\langle c_0, \sigma\rangle \downarrow \langle \sigma', 0\rangle}{\langle \textbf{try}\ c_0\ \textbf{catch}\ X : c_1, \sigma\rangle \downarrow \langle \sigma', 0\rangle}$$

$$\text{EC-TryE} : \frac{\langle c_0, \sigma\rangle \downarrow \langle \sigma'', n'\rangle \qquad \langle c_1, \sigma''[X \mapsto n']\rangle \downarrow \langle \sigma', n\rangle}{\langle \textbf{try}\ c_0\ \textbf{catch}\ X : c_1, \sigma\rangle \downarrow \langle \sigma', n\rangle}(n' \neq 0)$$

For the denotational semantics we modify the functionality of the meanings of commands analogously, so that we now have:

$$\mathcal{C}[\![c]\!] : \Sigma \to (\Sigma \times \mathbf{Z})_\perp\,,$$

with a sample defining equation reading,

$$\mathcal{C}[\![c_0; c_1]\!] = \lambda\sigma.\, \mathcal{C}[\![c_0]\!]\sigma \star \lambda(\sigma_1, n_1).\, \mathit{cond}(\mathit{eq}(n_1, 0), \mathcal{C}[\![c_1]\!]\sigma_1, \eta\,(\sigma_1, n_1))$$

Also, Theorem 7.8 for the extended langauge says: $\mathcal{C}[\![c]\!]\sigma = \lfloor(\sigma', n)\rfloor \Leftrightarrow \langle c, \sigma\rangle \downarrow \langle \sigma', n\rangle$.

Question 5 Complete the definition of $\mathcal{C}[\![c]\!]$ for *all* the remaining command forms; then prove (the corresponding modifications of) Lemmas 7.3 and 7.4 for *just* the new constructs (i.e., **throw** and **try**), towards showing Theorem 7.8.

(End of exam text.)