

NPRG085: Theory of Programming Languages

Types for imperative language

Tomáš Petříček and Vít Šefl

✉ petricek@d3s.mff.cuni.cz | sefl@ksvi.mff.cuni.cz

🌐 [@tomasp.net](https://tomasp.net) | <https://tomasp.net>



Reminder

Operational semantics

OPERATORS

(op p) $\langle n_1 + n_2, s \rangle \rightarrow \langle n, s \rangle$ if $n = n_1 + n_2$
 (op gt) $\langle n_1 \geq n_2, s \rangle \rightarrow \langle \text{true}, s \rangle$ if $n_1 \geq n_2$
 (op gf) $\langle n_1 \geq n_2, s \rangle \rightarrow \langle \text{false}, s \rangle$ if $n_2 \geq n_1$

(op1) $\frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \rightarrow \langle e'_1 \text{ op } e_2, s' \rangle}$
 (op2) $\frac{\langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle}{\langle v \text{ op } e_2, s \rangle \rightarrow \langle v \text{ op } e'_2, s' \rangle}$

STORE

(defef) $\langle !l, s \rangle \rightarrow \langle n, s \rangle$ if $l \in \text{dom}(s) \wedge s(l) = n$
 (assign1) $\langle l := n, s \rangle \rightarrow \langle \text{skip}, s \cup \{l \mapsto n\} \rangle$ if $l \in \text{dom}(s)$

(assign2) $\frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle l := e, s \rangle \rightarrow \langle l := e', s' \rangle}$

CONDITIONALS

(if1) $\langle \text{if true then } e_1 \text{ else } e_2, s \rangle \rightarrow \langle e_1, s \rangle$
 (if2) $\langle \text{if false then } e_1 \text{ else } e_2, s \rangle \rightarrow \langle e_2, s \rangle$
 (if3) $\frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s \rangle \rightarrow \langle \text{if } e'_1 \text{ then } e_2 \text{ else } e_3, s' \rangle}$

SEQUENCING & WHILE

(seq1) $\langle \text{skip}; e, s \rangle \rightarrow \langle e, s \rangle$
 (seq2) $\frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1; e_2, s \rangle \rightarrow \langle e'_1; e_2, s' \rangle}$
 (while) $\langle \text{while } e_1 \text{ do } e_2, s \rangle \rightarrow \langle \text{if } e_1 \text{ then } e_2; (\text{while } e_1 \text{ do } e_2) \text{ else skip}, s \rangle$

Derivation rules for the full L1 language

$$\frac{\langle l_1 := 40, \{l_1 \mapsto 0\} \rangle \rightarrow \langle \text{skip}, \{l_1 \mapsto 40\} \rangle \quad (\text{assign1})}{\langle l_1 := 40; !l_1 + 2, \{l_1 \mapsto 0\} \rangle \rightarrow \langle \text{skip}; !l_1 + 2, \{l_1 \mapsto 40\} \rangle} \quad (\text{seq2})$$

$$\langle \text{skip}; !l_1 + 2, \{l_1 \mapsto 40\} \rangle \rightarrow \langle !l_1 + 2, \{l_1 \mapsto 40\} \rangle \quad (\text{seq1})$$

$$\frac{\langle !l_1, \{l_1 \mapsto 40\} \rangle \rightarrow \langle 40, \{l_1 \mapsto 40\} \rangle \quad (\text{deref})}{\langle !l_1 + 2, \{l_1 \mapsto 40\} \rangle \rightarrow \langle 40 + 2, \{l_1 \mapsto 40\} \rangle} \quad (\text{op1})$$

$$\langle 40 + 2, \{l_1 \mapsto 40\} \rangle \rightarrow \langle 42, \{l_1 \mapsto 40\} \rangle \quad (\text{opp})$$

Step-by-step derivation example

Sequence using individual $\rightarrow$ steps

Every $\rightarrow$ justified using a rule name!

Types

Motivation and background

Types

Different uses of types

-  Define what programs make sense
-  Prevent errors before execution
-  Guarantee other program properties
-  Useful information for the programmer
-  Guide compiler optimizations

Types across languages

TypeScript

- Editor tooling and documentation
- No strict formal guarantees

OCaml

- Guarantee safety properties
- Allow efficient compilation

Agda, Idris, Lean

- Use types to prove (program) properties!



```
$ cat test.c
#include <stdio.h>

int main(void) {
    char msg[] = "Hello world";
    char buf[4];

    buf[4] = 'J';
    printf("%s\n", msg);
    return 0;
}
```

What types guarantee?

No undefined behavior!

```
$ cat test.c
#include <stdio.h>

int main(void) {
    char msg[] = "Hello world";
    char buf[4];

    buf[4] = 'J';
    printf("%s\n", msg);
    return 0;
}

$ gcc test.c && ./test
Hello world
```

What types guarantee?

No undefined behavior!

Writing to a value out
of array bounds

C does not specify how
arrays are allocated

Types

What is correctness?



There is no undefined behavior

Out of bounds access, reading non-existent field, etc.



Expressing undefined behavior formally

It never happens that there is no reduction $\langle e, s \rangle \nrightarrow$



Beware of non-termination

There may be no v such that $\langle e, s \rangle \rightarrow^* \langle v, s' \rangle$



Beware of starting with valid s

Even the (**assign**) rule requires that $l \in \text{dom}(s)$

Types

Defining type systems

Typing judgement

Given a typing context Γ , the type of the expression e is τ .

$$\boxed{\Gamma \vdash e : \tau}$$

- Γ - locations in store, variables in scope
- e - expression we want to type check
- τ - type that we want to verify
- Relation defined using inference rules

Type checking only

- Know the context Γ , expression e and type τ !
- Looking for a derivation tree using typing rules
- Type inference & program synthesis harder!

Examples

Arguments of
correct types

References
are available

No mismatched
return values

$\vdash 1 + (\text{if } 2 \geq 1 \text{ then } 3 \text{ else } 4) : \text{int}$

$l : \text{intref} \vdash 1 + (2 + !l) : \text{int}$

$\nvdash \text{if } 42 \text{ then } 1 \text{ else } 2 : \text{int}$

$\vdash 1 + (4 \geq 2) : T \text{ for any } T$

$l_1 : \text{intref} \vdash !l_1 + !l_2 : \text{int}$

$\vdash \text{if } 1 \geq 0 \text{ then } 42 \text{ else skip} : \text{int}$

Syntax of types

Primitive types only

Contexts Γ are finite partial maps from location names $\mathbb{L}$ to location types T_{loc}

Location types are only ints (for now)

Types $T ::= \text{int} \mid \text{bool} \mid \text{unit}$

Loc types $T_{loc} ::= \text{intref}$

Contexts $\Gamma \in \mathbb{L} \rightarrow T_{loc}$
 $= l_1 : T_{loc_1} \mid \dots \mid l_n : T_{loc_n}$

(int) $\Gamma \vdash n : \text{int}$ for $n \in \mathbb{Z}$

(bool) $\Gamma \vdash b : \text{bool}$ for $b \in \{\text{true}, \text{false}\}$

(op +)
$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

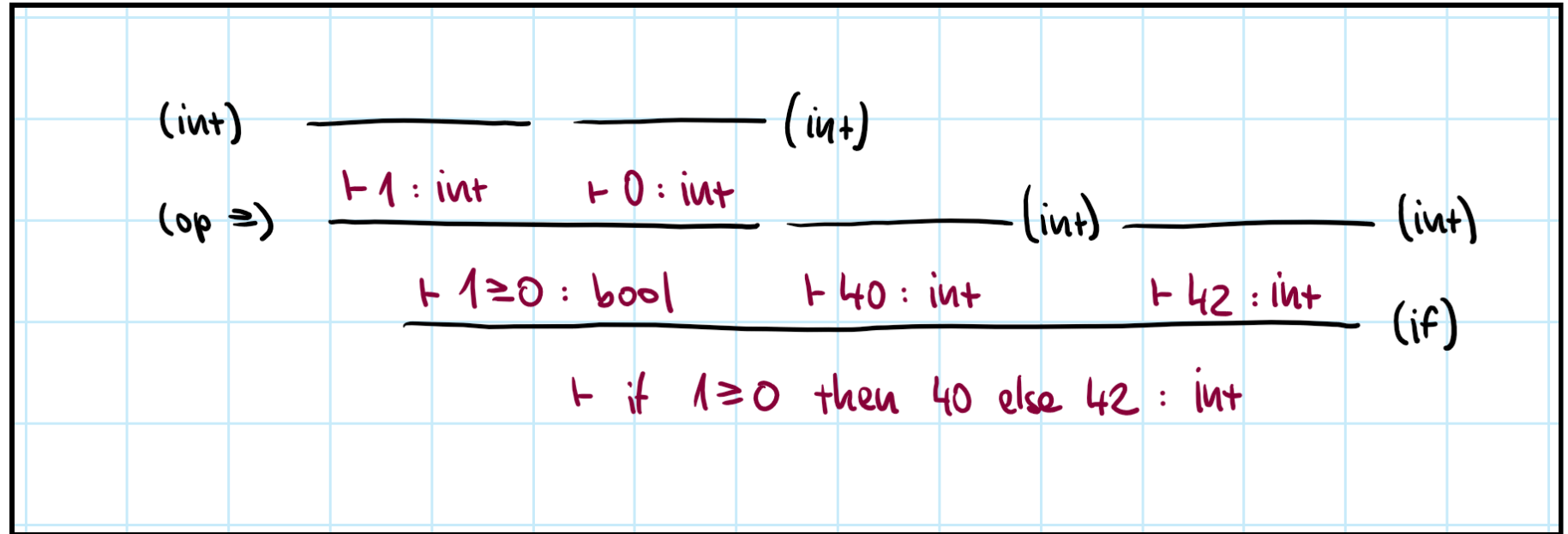
(op $\geq$)
$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \geq e_2 : \text{bool}}$$

(if)
$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T}$$

Typing rules for primitives, operators and if

Assumptions are other judgements or side-conditions

Example typing derivation



Look at the syntactic structure, choose a rule to apply
Repeat for all sub-expressions until done
Conditional expression can be of any type!

Store and sequencing

Store access
requires l is in Γ

No construct
modifies Γ (now)

$$\begin{array}{ll} \text{(assign)} & \frac{\Gamma(l) = \text{intref} \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash l := e : \text{unit}} \\ \text{(seq)} & \frac{\Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1; e_2 : T} \\ \text{(while)} & \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : \text{unit}} \\ \text{(deref)} & \frac{\Gamma(l) = \text{intref}}{\Gamma \vdash !l : \text{int}} \\ \text{(skip)} & \frac{}{\Gamma \vdash \text{skip} : \text{unit}} \end{array}$$

①

$$\begin{array}{c}
 \text{(int)} \quad \text{-----} \quad \text{(deref)} \\
 \text{(op +)} \quad \frac{l_1:\text{intref} \vdash 1:\text{int} \quad l_1:\text{intref} \vdash !l_1:\text{int}}{l_1:\text{intref} \vdash 1+!l_1:\text{int}} \quad \frac{\text{-----}}{l_1:\text{intref} \vdash 0:\text{int}} \quad \text{(int)} \\
 \hline
 l_1:\text{intref} \vdash (1+!l_1) \geq 0 : \text{bool} \quad \text{(op } \geq \text{)}
 \end{array}$$

②

$$\begin{array}{c}
 \text{(int)} \quad \text{-----} \quad \frac{l_1 \notin \text{dom}(\{ \})}{\vdash !l_1:\text{int}} \quad \text{X} \\
 \text{(op +)} \quad \frac{\vdash 0:\text{int} \quad \vdash !l_1:\text{int}}{\vdash 0+!l_1:\text{int}}
 \end{array}$$

Example #1: Arguments match and l_1 is in the context

Example #2: Accessing undefined location l_1

Typing rules

Interesting design alternatives

-  Syntax directed rules determined by structure
-  Context is a set, not ordered list
-  We never add to or remove from the store
-  Sequencing requires **skip** like the semantics

Properties

What is a good type system?

Well-typed programs do not go wrong

Well-typed programs do not get stuck

Theorem (Progress)

If $\Gamma \vdash e : \tau$ and $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ then either e is a value or there exist e', s' such that $\langle e, s \rangle \rightarrow \langle e', s' \rangle$.

Theorem (Preservation)

If $\Gamma \vdash e : \tau$ and $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ and $\langle e, s \rangle \rightarrow \langle e', s' \rangle$ then $\Gamma \vdash e' : \tau$ and $\text{dom}(\Gamma) \subseteq \text{dom}(s')$.

Well-typed programs do not get stuck

Theorem (Safety)

If $\Gamma \vdash e : \tau$ and $\text{dom}(\Gamma) \subseteq \text{dom}(s)$ and $\langle e, s \rangle \rightarrow^* \langle e', s' \rangle$
then either e' is a value or there exist e'', s'' such
that $\langle e', s' \rangle \rightarrow \langle e'', s'' \rangle$.

What this means

- Well-typed programs never get stuck
- Well-typed programs never reach undefined behavior
- They may still run forever

Conclusions

Types for imperative language

Conclusions

- 🧠 Types serve many purposes in practice
- ✅ Our models are about type safety
- 🚫 No undefined behavior / no stuck state
- 🛡️ Safety = Progress + Preservation

