Theory

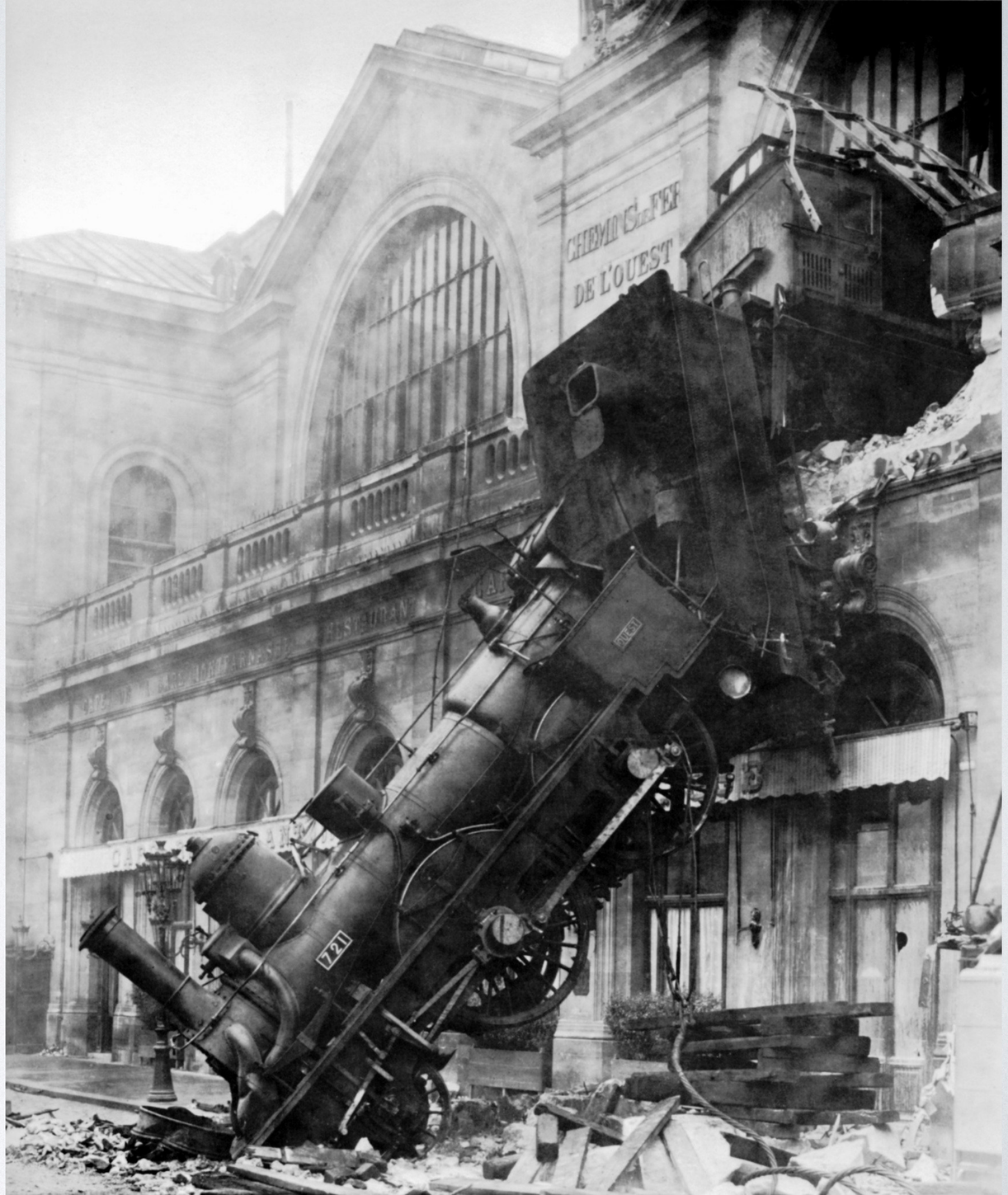
Static Vs. Dynamic as a Continuum

Guarantees

- Or: "what bugs are you hoping the compiler will detect?"
- Slogan: a well-typed program can't go wrong
- But what does "wrong" mean?

A Well-Typed Program Can't Go Wrong

- Every operation should have well-defined behavior (semantics)
- What does **1+2** mean? It should mean 3. If it means 42, "hello", or SEGFAULT, it has gone wrong.
- This program went wrong.



Programs Going Wrong

- `printf("%d", *NULL);`
- The above program has *undefined behavior*
- Why do we have languages that allow undefined programs?
- "Don't do that" isn't good enough; people do *that* all the time.

Syntax vs. Semantics

- **Syntax** specifies *what* is a program
- **Semantics** specify what a program *means*
  - **Static semantics** specify what a program means to the compiler
  - **Dynamic semantics** specify what a program *does* when it runs
- "Furiously sleep ideas green colorless" is ungrammatical (Chomsky)
- "Colorless green ideas sleep furiously" is grammatical but doesn't mean anything

Syntax vs. Semantics

- `1// : parse error` — syntax error
- `1/2` ⇓ .5 — dynamic semantics
- `1/0` ⇓ Error — dynamic semantics
- `fun fib (n: int) = …` — syntax OK; functions are values
- `fib("oops") : ill-typed` — syntax OK; type error (static semantics)

Guarantees and Proofs

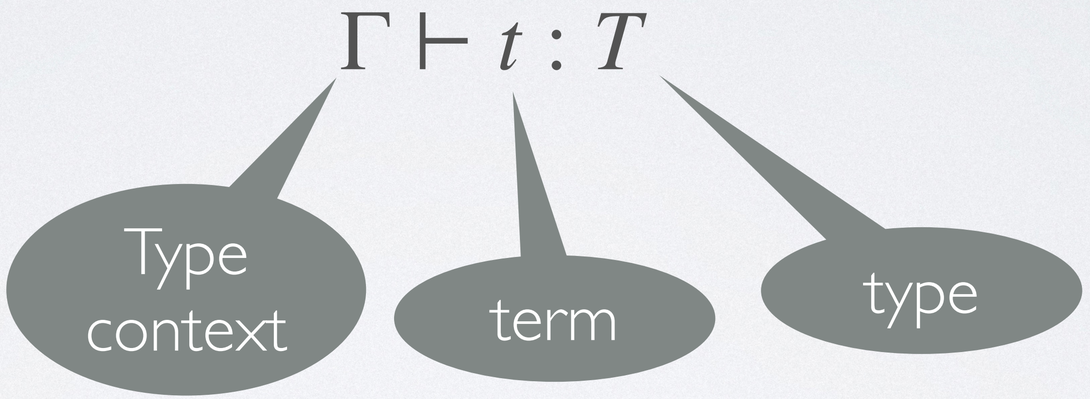
- Type soundness: a program will never escape its dynamic semantics
- Relationship between static and dynamic semantics is typically proved with two lemmas: progress and preservation
  - Progress: if a term is well-typed and not a value, it will take a step
  - Preservation: if a term takes a step, the type doesn't change

Guarantees

- When does the dynamic semantics say the result is "ERROR"?
  - Strong guarantees: almost never (bugs are caught by compiler)
  - Weak guarantees: very frequently (bugs are caught at run time)
- This is the essence of static vs. dynamic typing.
  - Not a binary question!
- If the behavior is sometimes undefined, the language is unsound (some say *broken*)
  - e.g. C, C++, …

Judgments

- $\vdash$ means "entails" or "proves"



A type context tracks types of variables that are in scope

Simply Typed Lambda Calculus

syntax

$$\begin{aligned} \mathrm{t} &::= \mathrm{x} \\ &\mid \lambda \mathrm{x} : T \,.\, \mathrm{t} \\ &\mid \mathrm{t}_1 \ \mathrm{t}_2 \end{aligned}$$

static semantics

$$\boxed{\Gamma \vdash \mathrm{t} : T}$$

$$\Gamma ::= \cdot \mid \Gamma, x : T$$

$$\frac{\mathrm{x} : T \in \Gamma}{\Gamma \vdash \mathrm{x} : T}\ \textsc{T-Var}$$

$$\frac{\Gamma, \mathrm{x} : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda \mathrm{x} : T_1 . t_2 : T_2 \to T_2}\ \textsc{T-Abs}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \textsc{T-App}$$

Example Derivations

$$\frac{\mathrm{x} : T \in \Gamma}{\Gamma \vdash \mathrm{x} : T}\ \text{T-VAR} \qquad \frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \text{T-APP}$$

$$\frac{\Gamma, \mathrm{x} : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda \mathrm{x} : T_1 . t_2 : T_2 \to T_2}\ \text{T-ABS}$$

$$\frac{\mathrm{x} : \mathrm{int} \in \cdot, \mathrm{x} : \mathrm{int}}{\cdot, \mathrm{x} : \mathrm{int} \vdash \mathrm{x} : \mathrm{int}}\ \text{T-VAR}$$

Example Derivations

$$\frac{\mathrm{x} : T \in \Gamma}{\Gamma \vdash \mathrm{x} : T}\ \text{T-VAR} \qquad \frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \text{T-APP}$$

$$\frac{\Gamma, \mathrm{x} : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda \mathrm{x} : T_1 . t_2 : T_2 \to T_2}\ \text{T-ABS}$$

$$\frac{\dfrac{\dfrac{\mathrm{x} : \mathrm{int} \in \cdot, \mathrm{y} : \mathrm{int}, \mathrm{x} : \mathrm{int}}{\cdot, \mathrm{y} : \mathrm{int}, \mathrm{x} : \mathrm{int} \vdash \mathrm{x} : \mathrm{int}}\ \text{T-VAR}}{\cdot, \mathrm{y} : \mathrm{int} \vdash \lambda \mathrm{x} : \mathrm{int} . \mathrm{x} : \mathrm{int} \to \mathrm{int}}\ \text{T-ABS} \qquad \dfrac{\mathrm{y} : \mathrm{int} \in \cdot, \mathrm{y} : \mathrm{int}}{\cdot, \mathrm{y} : \mathrm{int} \vdash \mathrm{y} : \mathrm{int}}\ \text{T-VAR}}{\cdot, \mathrm{y} : \mathrm{int} \vdash (\lambda \mathrm{x} : \mathrm{int} . \mathrm{x})\ \mathrm{y} : \mathrm{int}}\ \text{T-APP}$$

Type Soundness

- "A well-typed program can't go wrong"
- Want: if a program is well-typed, then it *either*:
  - evaluates to a value
  - runs forever
- Never:
  - If a program is well-typed, it might exhibit *undefined behavior*

Proving Type Soundness

- Progress: if $\Gamma \vdash t : T$ then either t is a value or $t \mapsto t'$ (for some t')
- Preservation: if $\Gamma \vdash t : T$ and $t \mapsto t'$ then $\Gamma \vdash t' : T$

Gradual Typing

- What if this is TOO HARD for the user?
- What if the user doesn't want to deal with all these types?
- Suppose we allow the user to decide whether to get their guarantee at run time or at compile time.

Gradual Typing

- What is the *gradual guarantee?* Siek et al.:
- The programmer should be able to conveniently evolve code from being statically typed to dynamically typed, and vice versa.
- When **removing** type annotations, a well-typed program will continue to be well-typed (with no need to insert explicit casts) and a correctly running program will continue to do so.
- When **adding** type annotations, if the program remains well typed, the only possible change in behavior is a trapped error due to a mistaken annotation.

Dynamic Semantics

Dynamic Semantics $\boxed{f \longmapsto f}$

$$(\lambda x{:}T.\, f)\ v \longmapsto [x := v]f \qquad \text{(Beta)}$$

$$v : B \Rightarrow^{\ell} B \longmapsto v \qquad \text{(IdBase)}$$

$$v : \star \Rightarrow^{\ell} \star \longmapsto v \qquad \text{(IdStar)}$$

$$v : G \Rightarrow^{\ell_1} \star \Rightarrow^{\ell_2} G \longmapsto v \qquad \text{(Succeed)}$$

$$v : G_1 \Rightarrow^{\ell_2} \star \Rightarrow^{\ell_2} G_2 \longmapsto \mathtt{blame}_{G_2}\, \ell_2 \quad \text{if } G_1 \neq G_2 \qquad \text{(Fail)}$$

$$(v_1 : T_1 \to T_2 \Rightarrow^{\ell} T_3 \to T_4)\ v_2 \longmapsto v_1\ (v_2 : T_3 \Rightarrow^{\ell} T_1) : T_2 \Rightarrow^{\ell} T_4 \qquad \text{(AppCast)}$$

$$v : T \Rightarrow^{\ell} \star \longmapsto v : T \Rightarrow^{\ell} G \Rightarrow^{\ell} \star \quad \text{if } T \neq \star, T \neq G, T \sim G \qquad \text{(Ground)}$$

$$v : \star \Rightarrow^{\ell} T \longmapsto v : \star \Rightarrow^{\ell} G \Rightarrow^{\ell} T \quad \text{if } T \neq \star, T \neq G, T \sim G \qquad \text{(Expand)}$$

$$F[f] \longmapsto F[f'] \quad \text{if } f \longmapsto f' \qquad \text{(Cong)}$$

$$F[\mathtt{blame}_{T_1}\, \ell] \longmapsto \mathtt{blame}_{T_2}\, \ell \quad \text{if } \vdash F : T_1 \Rightarrow T_2 \qquad \text{(Blame)}$$

Gradual Typing

- What i

e has a more precise type (T) than e' does (T')

e is more precise than e'

▶ **Theorem 5** (Gradual Guarantee). *Suppose* $e \sqsubseteq e'$ *and* $\vdash e : T$.

1. $\vdash e' : T'$ *and* $T \sqsubseteq T'$.
2. *If* $e \Downarrow v$, *then* $e' \Downarrow v'$ *and* $v \sqsubseteq v'$.
   If $e \Uparrow$ *then* $e' \Uparrow$.
3. *If* $e' \Downarrow v'$, *then* $e \Downarrow v$ *where* $v \sqsubseteq v'$, *or* $e \Downarrow \texttt{blame}_T\, l$.
   If $e' \Uparrow$, *then* $e \Uparrow$ *or* $e \Downarrow \texttt{blame}_T\, l$.

e evaluates to something more precise than e' does

if e runs forever, so does e'

-

if e' evaluates to v', either e evaluates to something more precise than v', or we have a bad cast at location l