Theory
Static Vs. Dynamic as a Continuum

- Detect more bugs at compile time
- Force programmer to prove safety

Static

- Leave programmer alone
- Wait until run time to find bugs

Dynamic
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
- `1/2 ↓ 0.5`
- `1/0 ↓ Error`
- `fun fib (n: int) = ...`
- `fib("oops") : ill-typed`
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"

$\Gamma \vdash t : T$

A type context tracks types of variables that are in scope
Simply Typed Lambda Calculus

**Syntax**

\[ t ::= x \]

\[ | \lambda x : T . t \]

\[ | t_1 t_2 \]

**Static Semantics**

\[ \Gamma ::= \cdot \quad | \quad \Gamma, x : T \]

\[
\begin{align*}
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} & \quad \text{T-VAR} \\
\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} & \quad \text{T-APP} \\
\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_2 \rightarrow T_2} & \quad \text{T-ABS}
\end{align*}
\]
Example Derivations

\[
\begin{align*}
\text{T-VAR} & \quad \frac{x : T \in \Gamma \quad \text{T-VAR}}{\Gamma \vdash x : T} \\
\text{T-APP} & \quad \frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \\
\text{T-Abs} & \quad \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_2 \rightarrow T_2}
\end{align*}
\]

\[
\begin{align*}
\text{T-VAR} & \quad \frac{x : \text{int} \in \cdot, x : \text{int} \quad \text{T-VAR}}{\cdot, x : \text{int} \vdash x : \text{int}}
\end{align*}
\]
Example Derivations

\[
\begin{align*}
\Gamma &\vdash x : T \quad \text{T-VAR} \\
\Gamma &\vdash t_1 : T_{11} \to T_{12} \quad \Gamma &\vdash t_2 : T_{11} \\
\Gamma &\vdash t_1 \ t_2 : T_{12} \quad \text{T-APP} \\
\Gamma, x : T_1 &\vdash t_2 : T_2 \\
\Gamma &\vdash \lambda x : T_1.t_2 : T_2 \to T_2 \quad \text{T-Abs}
\end{align*}
\]

\[
\begin{align*}
x &\in \cdot, y : \text{int}, x : \text{int} \\
\cdot, y : \text{int}, x &\vdash x : \text{int} \quad \text{T-VAR} \\
\cdot, y : \text{int} &\vdash \lambda x : \text{int}.x : \text{int} \to \text{int} \quad \text{T-Abs} \\
\cdot, y : \text{int} &\vdash (\lambda x : \text{int}.x) \ y : \text{int} \quad \text{T-App}
\end{align*}
\]
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

\[(\lambda x : T. f) v \mapsto [x := v] f\]
\[v : B \Rightarrow^\ell B \mapsto v\]
\[v : * \Rightarrow^\ell * \mapsto v\]
\[v : G \Rightarrow^\ell_1 * \Rightarrow^\ell_2 G \mapsto v\]
\[v : G_1 \Rightarrow^\ell_2 * \Rightarrow^\ell_2 G_2 \mapsto \text{blame}_{G_2} \ell_2 \quad \text{if } G_1 \neq G_2\]
\[(v_1 : T_1 \Rightarrow^\ell T_2 \Rightarrow^\ell T_3 \Rightarrow^\ell T_4) v_2 \mapsto v_1 (v_2 : T_3 \Rightarrow^\ell T_1) : T_2 \Rightarrow^\ell T_4\]
\[v : T \Rightarrow^\ell * \mapsto v : T \Rightarrow^\ell G \Rightarrow^\ell * \quad \text{if } T \neq *, T \neq G, T \sim G\]
\[v : * \Rightarrow^\ell T \mapsto v : * \Rightarrow^\ell G \Rightarrow^\ell T \quad \text{if } T \neq *, T \neq G, T \sim G\]
\[F[f] \mapsto F[f'] \quad \text{if } f \mapsto f'\]
\[F[\text{blame}_{T_1} \ell] \mapsto \text{blame}_{T_2} \ell \quad \text{if } \vdash F : T_1 \Rightarrow T_2\]
Gradual Typing

• What is the gradual guarantee?

**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 \( e \Downarrow \text{blame}_T l \).
   If \( e' \Uparrow \), then \( e \Uparrow \) or \( e \Downarrow \text{blame}_T l \).

5.1 GTLC

As discussed above, the Gradually Typed Lambda Calculus \([50]\) satisfies the gradual guarantee (the proof is in Section 6).

5.2 GTLC with Mutable References

Siek and Taha \([50]\) treat mutable references as invariant in their type system, disallowing implicit casts that change the pointed-to type. Consider that design in relation to the lattice of programs in Figure 7. The program at the top is well-typed because \( \text{Ref int} \) may be implicitly cast to \( \text{anything can} \). The program at the bottom is well-typed; it contains no...

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