Gradual Typing for Functional Languages

Jeremy Siek and Walid Taha
(presented by Lindsey Kuper)
Introduction
What we want

• Static and dynamic typing: both are useful!  (If you’re here, I assume you agree.)

• So, we want a type system that...
  • ...lets us choose the degree to which we want to annotate programs with types.
  • ...lets us write programs in a dynamically typed style (no explicit coercions to/from type Dynamic).
  • ...uses type annotations for static type checking, not just improving run-time performance.
  • ...behaves just like a static type system on completely annotated programs.

• Siek and Taha’s gradual type system fulfills all these desires.
Contributions

• Siek and Taha present $\lambda_{\rightarrow}^?$ ("lambda-dyn") and show that:
  • $\lambda_{\rightarrow}^?$, with its gradual type system, is equivalent to the STLC for fully-annotated programs. (Theorem 1)
  • They extend the language with references and assignment to show that it’s suitable for imperative languages as well.
  • $\lambda_{\rightarrow}^?$ is type safe: if evaluation terminates, the result is either a value of the expected type or a cast error, but not a type error. (Theorem 2)
    • On the way to Theorem 2, they prove an interesting result about $\lambda_{\rightarrow}^?$: the run-time cost of dynamism in the language is “pay-as-you-go”.
• The proofs are all mechanically verified with Isabelle.
Introduction to Gradual Typing
Syntax of $\lambda?$

- The syntax of $\lambda?$ is simple. We have:
  - variables $x$
  - ground types $\gamma$
  - constants $c$
  - types $T ::= \gamma \mid ? \mid T \rightarrow T$
  - expressions $e ::= c \mid x \mid \lambda x : T . e \mid e \ e \mid \lambda x . e \equiv \lambda x : ? . e$

- We indicate the unknown portions of a type with $\ ?$, so a type $\text{number } x \ ?$ is a pair of a number and an element of unknown type.

- Programming in a dynamically-typed style in this language is easy. Just leave off the type annotations on parameters. (A $\lambda$ with no parameter type annotation is sugar for one that has parameter type $\ ?$.)
What the type system does: Easy first-order example

• “The job of the type system is to reject programs that have inconsistencies in the known parts of types.”

• \(((\lambda (x : \text{number}) (\text{succ } x)) \ #t)\)
  
  • This program is rejected because it’s an application of a function of type \text{number} \rightarrow \text{number} to an argument of type \text{boolean}.

• But \(((\lambda (x) (\text{succ } x)) \ #t)\) is accepted by the static type system (and the type error is caught at run-time).
What the type system does: Fancy higher-order example

map : (number → number) × number list → number list

(map (lambda (x) (succ x)) (list 1 2 3))

• We’d like (lambda (x) (succ x)) to be accepted by our type system, but it has type ? → number and map expects type number → number. How do we design the type system to not reject this program?

• Intuition: require known portions of the two types ? → number and number → number to be equal; ignore the unknown parts.

• In effect, we’re delaying comparison of unknown parts until run-time.

• Analogy with partial functions: two partial functions are consistent when every element in the domain of both functions maps to the same result.
Type consistency rules

• Also known as *compatibility* rules.

• Just four simple rules:
  
  - **CREFL**: Every type is consistent with itself.
  
  - **CFUN**: If $\sigma_1 \sim \tau_1$ and $\sigma_2 \sim \tau_2$, then $\sigma_1 \rightarrow \sigma_2 \sim \tau_1 \rightarrow \tau_2$. 
  
  - **CUNL**: Every type is consistent with $\Box$.
  
  - **CUNR**: $\Box$ is consistent with every type.

• Reflexive and symmetric, but not transitive.
Typing rules

• Rules for variables, constants, $\lambda$ expressions: exactly like STLC.

• Rules for application:
  • (GAPP1) If the operator’s type is $\tau$, the type of the entire expression is $\tau$.
  • (GAPP2) If the operator’s type is $\tau \rightarrow \tau'$ and the operand’s type is consistent with $\tau$, then the type of the entire expression is $\tau'$.

• (Theorem 1) For STLC terms (aka fully-annotated terms), $\Gamma \vdash \tau$ typing judgments are just like STLC typing judgments.

• Proof sketch: throw out any typing rules that mention $\tau$. We’re left with the STLC’s typing rules.

• (Corollary 1) If an STLC term isn’t well-typed under STLC typing rules, it isn’t well-typed under $\Gamma \vdash \tau$ typing rules, either.
Run-time semantics
Adding explicit casts

- \( \lambda \) doesn’t make programmers write explicit casts; instead, it inserts them itself, producing an intermediate language we call \( \lambda^{\langle \tau \rangle} \) (“lambda-cast”). \( \lambda^{\langle \tau \rangle} \) is the language we’ll actually evaluate.

- The translation to \( \lambda^{\langle \tau \rangle} \) only requires casts to be inserted for certain kinds of application expressions:
  
  - (\text{CAPP1}) For function applications where the function’s type is \( ? \), just insert a cast to \( T_2 \rightarrow ? \) where \( T_2 \) is the argument’s type.
  
  - (\text{CAPP2}) For function applications where the function’s type is \( T \rightarrow T' \) and the argument’s type is \( T_2 \), which is \( \sim \) with \( T \), we just cast the argument to \( T \).

- \( \lambda^{\langle \tau \rangle} \)’s typing rules are much like STLC’s, but with a rule added for expressions containing an explicit cast.
Useful properties of $\lambda^{(\tau)}$

- Lemma 1. Inversion lemmas for $\lambda^{(\tau)}$'s typing rules. (These lemmas, which “invert” the typing rules, come in handy for some of the other lemmas.)
- Lemma 2. Every $\lambda^{(\tau)}$ expression has a unique type.
- Lemma 3. Cast insertion produces well-typed $\lambda^{(\tau)}$ terms.
- Lemma 4. Cast insertion does nothing to STLC terms.
Run-time semantics of $\lambda^{\langle \tau \rangle}$

- The result of evaluating a $\lambda^{\langle \tau \rangle}$ term can either be a value, a `CastError`, a `TypeError`, or a `KillError`.

- There are two kinds of run-time type errors: those that cause undefined behavior (like what happens when we have a buffer overflow in C) and those that are caught by the run-time system (like in Scheme). We say that the former are `TypeError`s and the latter are `CastError`s.

- We need `KillError` because of a technicality in the type safety proof. It could have been avoided if we’d been using a small-step semantics rather than a big-step semantics for $\lambda^{\langle \tau \rangle}$. 
That canonical forms lemma that we said would be interesting

- Canonical forms lemmas always say something like “If $v$ is of type $T$, then it must be...”.
- For instance, if $v$ is of type `boolean`, then it must be either `#t` or `#f`.
- Handy for compiler optimizations: we can use an efficient unboxed representation for every value whose type is completely known at compile time.
- And we get these efficient representations proportionally to the amount that we use type annotations in our programs: we “pay as we go” for efficiency.
Evaluation of $\lambda^{\langle \tau \rangle}$

- $\lambda^{\langle \tau \rangle}$ has an operational semantics defined in *big-step* style, where each rule completely evaluates the expression to a result. For instance...

- (ECSTG) If we evaluate $e$ for $n$ steps, producing a result $v$, and $v$ (unboxed if necessary) has type $\chi$, then $e$ cast to $\chi$ can be evaluated for $n+1$ steps to produce $v$.

- (ECSTE) If $e$ evaluates to $v$ and $v$ has type $\sigma$, which is inconsistent with $\tau$, then a cast of $e$ to $\tau$ will result in a run-time *CastError*.

- This big-step semantics is unusual and prevents us from using a more typical progress-and-preservation-style proof of type safety.
Examples

• Our original example \(((\text{lambda} \ (x) \ (\text{succ} \ x)) \ #t)\) produces a \textit{CastError} at run-time.

• Our higher-order example

\[((\text{lambda} \ (f : \ ? \rightarrow \text{number}) \ (f \ 1)))\]
\((\text{lambda} \ (x : \text{number}) \ (\text{succ} \ x)))\)

evaluates to the result 2.
A few more lemmas on the way to type safety

- Lemma 6 (Environment Expansion and Contraction). If a term $e$ has type $\tau$ under environment $\Gamma$...
  - ...and we extend $\Gamma$ with a binding for a fresh variable, $e$ still has type $\tau$. (Pierce calls this weakening.)
  - ...and we remove something we don’t need from the environment, $e$ still has type $\tau$.
  - ...and we swap in a new store typing for the old one, as long as they agree on the types of all locations, $e$ still has type $\tau$.

- Lemma 7 (Substitution preserves typing). If $e$ has type $\tau$ under $\Gamma$ and we substitute some subexpression $x$ of $e$ with another subexpression $e'$ of the same type as $x$, $e$ still has type $\tau$. 
Finally, a proof of type safety

- Lemma 8 (Soundness of evaluation). If an $\lambda^{(T)}$ expression $e$ is well-typed with type $T$ (which can include $\text{?}$), it will evaluate to a result $r$, which will be either a value, a $\text{CastError}$, or a $\text{KillError}$.

- Theorem 2 (Type safety). If a $\lambda^{?}$ expression $e$ with type $T$ can be converted to a $\lambda^{(T)}$ expression $e'$ with type $T$, it will evaluate to a result $r$, which will be either a value, a $\text{CastError}$, or a $\text{KillError}$.

- Proof: Lemma 3 (cast insertion produces well-typed $\lambda^{(T)}$ terms) followed by Lemma 8.
Adding references to \( \lambda \)

• A couple of additions to the syntax:
  - types \( \mathcal{T} ::= \ldots \mid \mathsf{ref} \mathcal{T} \)
  - expressions \( e ::= \ldots \mid \mathsf{ref} \mathcal{E} \mid !e \mid e \leftarrow e \)
    - \( \mathsf{ref} \mathcal{E} \) creates; \( !e \) dereferences; \( e \leftarrow e \) assigns and returns the value of the expression on the left after the assignment has happened.

• Interesting typing rules:
  - \((\mathsf{GDEREF1})\) If \( e \)'s type is \( ? \) then \( !e \)'s type is \( ? \).
  - \((\mathsf{GASSIGN1})\) If \( e_1 \)'s type is \( ? \) and \( e_2 \)'s type is \( \mathcal{T} \), then \( e_1 \leftarrow e_2 \) has type \( \mathsf{ref} \mathcal{T} \).
  - \((\mathsf{GASSIGN2})\) If \( e_1 \)'s type is \( \mathsf{ref} \mathcal{T} \) and \( e_2 \)'s type is \( \mathcal{\sigma} \), and \( \mathcal{\sigma} \sim \mathcal{T} \), then \( e_1 \leftarrow e_2 \) has type \( \mathsf{ref} \mathcal{T} \).
  - Types of locations can’t change, or type safety is compromised.
Related work

• We’re probably reading these two papers within the next 1-2 weeks:
  • Quasi-static typing (Section 3)
  • Abadi et al.’s language of explicit casts (Section 6)

• Languages with some degree of gradual typing, previously not formalized: Cecil, Boo, Bigloo, proposed extensions to VB.NET/C# and Java, ... (and since the paper came out: Typed Racket, and maybe also JavaScript)

• Languages with optional type annotations for run-time performance improvement only: Common Lisp, Dylan, ...

• Soft Typing: type inference for run-time performance improvement

• Lots of others!
Conclusion
Main points

• It’s no fun to start writing code in a dynamic language only to have to translate to a static language midway through. Ideally, you could keep the same language, and the language would have a type system that supports gradual addition of static types. Gradual typing gives us that.

• In $\lambda^?\rightarrow$, all programs are type-safe in the sense that non-type-safe actions can’t be completed, either because of static type checking or because of run-time exceptions.

• $\lambda^?\rightarrow$ is pay-as-you-go: the degree to which one or the other mechanism enforces the type safety of a particular program corresponds to the degree to which that program has type annotations. (And we get as much efficiency as we pay for, too.)
Possible directions for future work

• Add support for lists, arrays, ADTs, implicit coercions (such as between numeric types in Scheme) to a gradual type system.

• Investigate relationship between gradual typing and...
  • ...parametric polymorphism.
  • ...Hindley-Milner type inference.

• Incorporate gradual typing into a mainstream dynamic language (Python?) and find out if it really benefits programmer productivity.

• Everything else we’re going to talk about in this course...
(exit)