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 $\lambda^{?}_{\rightarrow}$

- The syntax of $\lambda^{?}_{\rightarrow}$ is simple. We have:
  - variables x
  - ground types γ
  - constants c
  - types $\tau ::= \gamma \mid ? \mid \tau \rightarrow \tau$
  - expressions $e ::= c \mid x \mid \lambda x : \tau \,.\, e \mid e\ e \mid \lambda x \,.\, e \equiv \lambda x : ? \,.\, e$
- We indicate the unknown portions of a type with $?$, so a type $\mathrm{number} \times ?$ 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 λ 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 : number) (succ x)) #t)
  - This program is rejected because it's an application of a function of type number $\longrightarrow$ number to an argument of type boolean.
  - But ((lambda (x) (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 $\sigma_1 \longrightarrow \sigma_2 \sim \tau_1 \longrightarrow \tau_2$.
  - CUNL: Every type is consistent with $?$.
  - CUNR: $?$ is consistent with every type.
- Reflexive and symmetric, but not transitive.

Typing rules

- Rules for variables, constants, λ expressions: exactly like STLC.
- Rules for application:
  - (GAPP1) If the operator's type is $?$, the type of the entire expression is $?$.
  - (GAPP2) If the operator's type is $\tau \rightarrow \tau'$ and the operand's type is consistent with τ, then the type of the entire expression is τ'.
- (Theorem 1) For STLC terms (aka fully-annotated terms), $\vdash_G$ typing judgments are just like STLC typing judgments.
  - Proof sketch: throw out any typing rules that mention $?$. 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 $\vdash_G$ typing rules, either.

Run-time semantics

Adding explicit casts

- $\lambda^{?}_{\rightarrow}$ doesn't make programmers write explicit casts; instead, it inserts them itself, producing an intermediate language we call $\lambda^{\langle\tau\rangle}_{\rightarrow}$ ("lambda-cast"). $\lambda^{\langle\tau\rangle}_{\rightarrow}$ is the language we'll actually evaluate.
- The translation to $\lambda^{\langle\tau\rangle}_{\rightarrow}$ only requires casts to be inserted for certain kinds of application expressions:
  - (CAPP1) For function applications where the function's type is $?$, just insert a cast to $\tau_2 \longrightarrow ?$ where τ_2 is the argument's type.
  - (CAPP2) For function applications where the function's type is $\tau \longrightarrow \tau'$ and the argument's type is τ_2, which is ~ with τ, we just cast the argument to τ.
- $\lambda^{\langle\tau\rangle}_{\rightarrow}$'s typing rules are much like STLC's, but with a rule added for expressions containing an explicit cast.

Useful properties of $\lambda_{\rightarrow}^{\langle\tau\rangle}$

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

Run-time semantics of $\lambda_{\rightarrow}^{\langle\tau\rangle}$

- The result of evaluating a $\lambda_{\rightarrow}^{\langle\tau\rangle}$ 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 $\lambda_{\rightarrow}^{\langle\tau\rangle}$.

That canonical forms lemma that we said would be interesting

- Canonical forms lemmas always say something like "If v is of type τ, 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 $\lambda_{\rightarrow}^{\langle\tau\rangle}$

- $\lambda_{\rightarrow}^{\langle\tau\rangle}$ 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 γ, then e cast to γ can be evaluated for *n*+1 steps to produce v.
  - (ECSTE) If e evaluates to v and v has type σ, which is inconsistent with τ, then a cast of e to τ 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 ((lambda (x) (succ x)) #t) produces a *CastError* at run-time.
- Our higher-order example

  ((lambda (f : $?$ $\longrightarrow$ number) (f 1))
  (lambda (x : number) (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 τ under environment Γ...
  - ...and we extend Γ with a binding for a fresh variable, e still has type τ. (Pierce calls this *weakening*.)
  - ...and we remove something we don't need from the environment, e still has type τ.
  - ...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 τ.
- Lemma 7 (Substitution preserves typing). If e has type τ under Γ and we substitute some subexpression x of e with another subexpression e' of the same type as x, e still has type τ.

Finally, a proof of type safety

- Lemma 8 (Soundness of evaluation). If an $\lambda_{\rightarrow}^{\langle\tau\rangle}$ expression e is well-typed with type τ (which can include ?), it will evaluate to a result r, which will be either a value, a *CastError*, or a *KillError*.
- Theorem 2 (Type safety). If a $\lambda_{\rightarrow}^{?}$ expression e with type τ can be converted to a $\lambda_{\rightarrow}^{\langle\tau\rangle}$ expression e' with type τ, it will evaluate to a result r, which will be either a value, a *CastError*, or a *KillError*.
  - Proof: Lemma 3 (cast insertion produces well-typed $\lambda_{\rightarrow}^{\langle\tau\rangle}$ terms) followed by Lemma 8.

Adding references to $\lambda^{?}_{\rightarrow}$

- A couple of additions to the syntax:
  - types $\tau ::= \ldots \mid \mathit{ref}\ \tau$
  - expressions $e ::= \ldots \mid \mathit{ref}\ e \mid !e \mid e \leftarrow e$
    - $\mathit{ref}\ e$ 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:
  - (GDEREF1) If e's type is $?$ then $!e$'s type is $?$.
  - (GASSIGN1) If e_1's type is $?$ and e_2's type is τ, then $e_1 \leftarrow e_2$ has type $\mathit{ref}\ \tau$.
  - (GASSIGN2) If e_1's type is $\mathrm{ref}\ \tau$ and e_2's type is σ, and $\sigma \sim \tau$, then $e_1 \leftarrow e_2$ has type $\mathit{ref}\ \tau$.
- 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)