1 Introduction

Type checking as seen earlier in this class is pessimistic by nature: if we cannot prove a program will never run into the problem we are checking for (e.g. getting stuck), it is not well-formed, and will hence be rejected by a type checker. This can be quite onerous, especially in cases where the type system at some point forces one to "forget" information that it will later need (e.g. storing a number in a place that accepts arbitrary values – in many cases, we know that whatever value we retrieve from there is a number, but the type checker does not). *Gradual typing* is the idea that sometimes we might tell the type checker to be optimistic instead, in which case it should only reject a program if it can prove that a program *will* run into the problem we are checking for (e.g. the program will get stuck).

2 $\lambda^{\to}_{?}$

We'll start by adding a special type **?** to the simply typed lambda calculus $\lambda^{\to}$.

$$\begin{array}{llcl} \text{terms} & e & ::= & n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \mid x \mid e_1\, e_2 \mid \lambda x : \tau.\, e \\ \text{types} & \tau, \sigma & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \boxed{?} \end{array}$$

Otherwise, the syntax of $\lambda^{\to}_{?}$ is exactly the same as that of $\lambda^{\to}$. The typing rules are also similar, only the rule for function application changes somewhat. It has three new components that we need to introduce. The first is the consistency relation $\sim$. It is defined as follows:

$$\tau \sim \tau \qquad \tau \sim ? \qquad ? \sim \tau \qquad \frac{\tau \sim \tau' \quad \sigma \sim \sigma'}{\tau \to \sigma \sim \tau' \to \sigma'}$$

The consistency relation represents a relaxed version of type equality: two types are consistent with each other if they *could* be equal if the occurrences of **?** in them are replaced appropriately. This is the central ingredient to making the type checker optimistic while still being able to warn us if two types do not match up at all. Note that $\sim$ is reflexive and symmetric, but not transitive – if it were, everything would be consistent with everything. The other two components are helper functions mapping function types to their domain and range, lifted to gradual types:

$$\mathbf{dom}_?(\tau) \triangleq \begin{cases} \sigma, & \text{if } \tau = \sigma \to \sigma', \\ ?, & \text{if } \tau = ?, \\ \text{undefined}, & \text{otherwise.} \end{cases} \qquad \mathbf{rng}_?(\tau) \triangleq \begin{cases} \sigma', & \text{if } \tau = \sigma \to \sigma', \\ ?, & \text{if } \tau = ?, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Now, we can define the typing rules:

$$\Gamma \vdash_? n : \mathsf{int} \qquad \Gamma \vdash_? \mathsf{true} : \mathsf{bool} \qquad \Gamma \vdash_? \mathsf{false} : \mathsf{bool} \qquad \Gamma \vdash_? \mathsf{null} : \mathsf{unit} \qquad \Gamma,\, x : \tau \vdash_? x : \tau$$

$$\frac{\Gamma \vdash_? e_0 : \tau \quad \Gamma \vdash_? e_1 : \sigma \quad \sigma \sim \mathbf{dom}_?(\tau)}{\Gamma \vdash_? e_0\, e_1 : \mathbf{rng}_?(\tau)} \qquad \frac{\Gamma,\, x : \sigma \vdash_? e : \tau}{\Gamma \vdash_? (\lambda x : \sigma.\, e) : \sigma \to \tau}$$

There are two big changes for function application. The first is that the type of the function itself might be **?**, so we cannot immediately match it to a function type constructed with $\rightarrow$. If the type of the function is indeed **?**, we will still optimistically assume that it is a function, and instruct the type checker to again be optimistic when it comes to its domain and range, which is what the helper functions **dom** and **rng** express. The second change is that the type of the argument does not have to be equal to the domain of the function; the two only have to be consistent. Thus, a **?** argument is accepted by any function, and a function that accepts a **?** accepts any argument.

3 Relationship to the λ-calculus and $\lambda^{\rightarrow}$

With the above rules, we can prove that for all expressions $e^{\rightarrow}$ of $\lambda^{\rightarrow}$, $\Gamma \vdash e : \tau \Leftrightarrow \Gamma \vdash_{\mathbf{?}} e^{\rightarrow} : \tau$. That is, fully typed programs that do not contain **?** are well-formed in $\lambda^{\rightarrow}_{\mathbf{?}}$ iff they are well-formed in $\lambda^{\rightarrow}$. Similarly, for any expression e of the (untyped) λ-calculus, let $e^{\mathbf{?}}$ be an expression in $\lambda^{\rightarrow}_{\mathbf{?}}$ that is obtained by the following translation:

$$
\begin{array}{rcl}
[\![n]\!] & \triangleq & n \\
[\![\mathsf{true}]\!] & \triangleq & \mathsf{true} \\
[\![\mathsf{false}]\!] & \triangleq & \mathsf{false} \\
[\![\mathsf{null}]\!] & \triangleq & \mathsf{null} \\
[\![\lambda x.\, e]\!] & \triangleq & \lambda x : \mathbf{?}.\, [\![e]\!] \\
[\![e_1\, e_2]\!] & \triangleq & ((\lambda x : \mathbf{?}.\, x)[\![e_1]\!])[\![e_2]\!]
\end{array}
$$

Every such $e^{\mathbf{?}}$ is well-formed in $\lambda^{\rightarrow}_{\mathbf{?}}$.

4 Semantics and Soundness

While the previous section suggests that for fully typed or fully untyped programs, the small-step operational semantics for the λ-calculus and $\lambda^{\rightarrow}$ would work just fine, the resulting language would have neither progress nor preservation. The reason for that is that the optimism we introduced may in some cases be unfounded – a program that *could* be safe does not always have to be safe. In some sense, we accepted that when we introduced optimism in the first place, but as it stands, we have no guarantees about how a program that is neither fully typed or fully untyped might behave. In order to get back some guarantees about the behavior of mixed programs, we need to add some run-time checks to a gradually typed program, creating an intermediate language called $\lambda^{\rightarrow}_{\langle\tau\rangle}$:

terms	e	$::=$	$n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \mid x \mid e_1\, e_2 \mid \lambda x : \tau.\, e \mid \langle \tau \Leftarrow \sigma \rangle e \mid \mathbf{box}^{\gamma}\, v$
values	v	$::=$	$n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \mid x \mid \lambda x : \tau.\, e \mid \langle \tau \rightarrow \sigma \Leftarrow \tau' \rightarrow \sigma' \rangle v \mid \mathbf{box}^{\gamma}\, v$
types	τ, σ	$::=$	$\mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \tau_1 \rightarrow \tau_2 \mid \mathbf{?}$
ground types	γ	$::=$	$\mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \mathbf{?} \rightarrow \mathbf{?}$

This language adds two new constructs. The first is the cast expression $\langle \tau \Leftarrow \sigma \rangle e$, where the expression e has type σ and is cast to type τ.The second new construct is a boxed value container for when a value is treated as **?**. The box remembers the original ground type γ of the value (functions are always boxed as $\mathbf{?} \rightarrow \mathbf{?}$, and **?** itself is never boxed). Note that we treat casts of function types as values – this is both for ease of notation and to highlight an important point later. With these constructs, we can give $\lambda^{\rightarrow}_{\langle\tau\rangle}$ pessimistic typing rules

that essentially treat **?** as the type of boxed values and allow casts between types that are consistent with each other:

$$\Gamma \vdash n : \mathsf{int} \qquad \Gamma \vdash \mathsf{true} : \mathsf{bool} \qquad \Gamma \vdash \mathsf{false} : \mathsf{bool} \qquad \Gamma \vdash \mathsf{null} : \mathsf{unit} \qquad \Gamma,\, x : \tau \vdash x : \tau$$

$$\frac{\Gamma \vdash e_0 : \tau \to \sigma \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\, e_1 : \sigma} \qquad \frac{\Gamma,\, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma .\, e) : \sigma \to \tau} \qquad \frac{\Gamma \vdash e : \sigma \quad \tau \sim \sigma}{\Gamma \vdash \langle \tau \Leftarrow \sigma \rangle e : \tau} \qquad \frac{\Gamma \vdash v : \gamma}{\Gamma \vdash \mathbf{box}^{\gamma}\ v : \mathbf{?}}$$

This language can now have a small-step CBV operational semantics that we can prove progress and preservation for (with the caveat that there can be run-time errors **error** representing a failed cast). First, evaluation contexts are extended so we can evaluate under casts and propagate errors:

$$E \ ::= \ E\, e \ \mid \ v\, E \ \mid \ \langle \tau \Leftarrow \sigma \rangle E \ \mid \ [\cdot]$$

$$\frac{e_0 \to e_1}{E[e_0] \to E[e_1]} \qquad \frac{e \to \mathbf{error}}{E[e] \to \mathbf{error}}$$

Then, a number of new reduction rules handle casting (first applicable rule applies exclusively):

$$\begin{aligned}
(\lambda x : \tau .\, e)\, v &\to e\{v/x\} \\
\langle \mathbf{?} \Leftarrow \mathbf{?} \rangle v &\to v \\
\langle \gamma \Leftarrow \gamma \rangle v &\to v \\
\langle \mathbf{?} \Leftarrow \gamma \rangle v &\to \mathbf{box}^{\gamma}\ v \\
\langle \tau \Leftarrow \mathbf{?} \rangle \mathbf{box}^{\gamma}\ v &\to \langle \tau \Leftarrow \gamma \rangle v && \text{if } \tau \sim \gamma \\
\langle \tau \Leftarrow \mathbf{?} \rangle \mathbf{box}^{\gamma}\ v &\to \mathbf{error} && \text{if } \tau \not\sim \gamma \\
(\langle \tau \to \sigma \Leftarrow \tau' \to \sigma' \rangle v_0)\, v_1 &\to \langle \sigma \Leftarrow \sigma' \rangle (v_0\, (\langle \tau' \Leftarrow \tau \rangle v_1)) \\
\langle \mathbf{?} \Leftarrow \tau \to \sigma \rangle v &\to \mathbf{box}^{\mathbf{?} \to \mathbf{?}}\ \langle \mathbf{?} \to \mathbf{?} \Leftarrow \tau \to \sigma \rangle v
\end{aligned}$$

The first two rules eliminate superfluous casts from a ground types and **?** to themselves. The third rule boxes values of ground types. The fourth and fifth rule handle casts from **?** – this is where run-time type errors are eventually caught. If the actual type γ of the boxed value is not consistent with type that the value is cast to, we raise an error. Such an inconsistency may happen in two cases[1]: either γ is $\mathbf{?} \to \mathbf{?}$ and τ is not a function type, or γ is not $\mathbf{?} \to \mathbf{?}$ and τ is not equal to γ. The sixth rule deals with the fact that it is in general impossible to prove something about the output of an arbitrary function, even when given information about the input. Thus, we cannot decide immediately whether a cast from one function type to another succeeds. Instead, we wrap the function (which is why a function cast is a value) and insert casts every time the function is called: one cast to make sure that the input to the function is what the function expected, and one cast to make sure that its output is what the caller expected. The last rule expresses that when we wrap a function, we remember that it was a function that we wrapped, but leave the details of its type to an inner cast.

The above typing and reduction rules give us progress and preservation as follows:

[1] $\tau = \mathbf{?}$ is handled by the first new rule, above

Progress For all e, if $\vdash e : \tau$, either $e = v$ for some v, or $e \to e'$ for some e', or $e \to \mathbf{error}$.

Preservation For all e, if $\Gamma \vdash e : \tau$ and $e \to e'$ for some e', $\Gamma \vdash e' : \tau$

Compared to the progress lemma of $\lambda^{\to}$, we just added the case where we might reduce to an error, while preservation stays the same.

4.1 Translation

We can now define the semantics of $\lambda^{\to}_{?}$ by translation to $\lambda^{\to}_{\langle\tau\rangle}$, inserting casts whenever the type-checker relies on optimism. The resulting type-directed translation rules are of the form $\Gamma \vdash^{\langle\tau\rangle}_{?} e \rightsquigarrow e' : \tau$, where an expression e in $\lambda^{\to}_{?}$ is translated to an expression e' in $\lambda^{\to}_{\langle\tau\rangle}$, having type τ under environment Γ in both languages.

$$\Gamma \vdash^{\langle\tau\rangle}_{?} n \rightsquigarrow n : \mathsf{int} \qquad \Gamma \vdash^{\langle\tau\rangle}_{?} \mathsf{true} \rightsquigarrow \mathsf{true} : \mathsf{bool} \qquad \Gamma \vdash^{\langle\tau\rangle}_{?} \mathsf{false} \rightsquigarrow \mathsf{false} : \mathsf{bool}$$

$$\Gamma \vdash^{\langle\tau\rangle}_{?} \mathsf{null} \rightsquigarrow \mathsf{null} : \mathsf{unit} \qquad \Gamma, x : \tau \vdash^{\langle\tau\rangle}_{?} x \rightsquigarrow x : \tau \qquad \frac{\Gamma, x : \sigma \vdash^{\langle\tau\rangle}_{?} e \rightsquigarrow e' : \tau}{\Gamma \vdash^{\langle\tau\rangle}_{?} (\lambda x : \sigma \,.\, e) \rightsquigarrow (\lambda x : \sigma \,.\, e') : \sigma \to \tau}$$

$$\frac{\Gamma \vdash^{\langle\tau\rangle}_{?} e_0 \rightsquigarrow e_0' : \tau \quad \Gamma \vdash^{\langle\tau\rangle}_{?} e_1 \rightsquigarrow e_1' : \sigma \quad \sigma \sim \mathbf{dom_?}(\tau)}{\Gamma \vdash^{\langle\tau\rangle}_{?} e_0\, e_1 \rightsquigarrow \langle \mathbf{dom_?}(\tau) \to \mathbf{rng_?}(\tau) \Leftarrow \tau \rangle e_0' \, \langle \mathbf{dom_?}(\tau) \Leftarrow \sigma \rangle e_1' : \mathbf{rng_?}(\tau)}$$

In this translation, almost all the work happens for applications (except that λ-abstractions translate their inner expressions recursively). The application rule inserts two casts to reflect the two optimistic assumptions the type checker might make: that the functor e_0 is a function and that the argument e_1 matches the expected argument type.

It follows from this translation that a well-formed program in $\lambda^{\to}_{?}$ translates to a well-formed program in $\lambda^{\to}_{\langle\tau\rangle}$, thus progress and preservation apply to $\lambda^{\to}_{?}$ under this translation semantics, giving us a basic notion of safety.