Gradual Typing for Delimited Continuations

Yusuke Miyazaki Taro Sekiyama * Atsushi Igarashi
Graduate School of Informatics, Kyoto University
{miyazaki, t-sekiym, igarashi}@fos.kuis.kyoto-u.ac.jp

1. Introduction

In this paper, we report our ongoing work on gradual typing for a language with delimited-control operators, *shift* and *reset* [4], which are known to be very powerful constructs [3, 5]. We base our gradual type system on the simple type system with so-called *answer-type modification* [1, 3]. We introduce the type dynamic (written $\star$), define the type consistency relation, modify the typing rules by using the type consistency relation, and give translation to insert explicit casts. The way we modify the typing rules is very similar to the Gradualizer [2] but it turns out that the Gradualizer is not directly applicable for generation of typing and cast insertion rules. We also discuss the properties of the obtained gradually typed language according to Siek et al.'s criteria [9] for gradually typed languages.

2. A Static Type System for shift/reset

Our gradually typed calculus $\lambda^G_{s/r}$ is based on the simply typed lambda calculus with shift and reset [3], which we call $\lambda^S_{s/r}$ and review here. The shift operator $\mathcal{S}k.\, e$ captures the current continuation up to the innermost surrounding reset operator $\langle\cdot\rangle$ as a function value, binds k to it, and executes e. The reduction rule for shift can be given as follows:

$$\langle E[\mathcal{S}k.\, e]\rangle \longrightarrow \langle \mathsf{let}\ k = \lambda x.\, \langle E[x]\rangle\ \mathsf{in}\ \mathrm{e}\rangle$$

(E is an evaluation context which does not contain a reset operator.) To give a type system for shift/reset, it is important to know an *answer type*—the return type of a current continuation, or the type of the expression inside the innermost surrounding reset (e.g., $E[\mathcal{S}k.\, e]$), because it will be the return type of the captured delimited continuation k.

Answer types can change during reduction. For example, after reduction of shift, the reset operator now surrounds e, which was the body of shift, and its type becomes a new answer type, which may have nothing to do with the old one. The type system is said to allow *answer-type modification* [1, 3] if it allows a new answer type to be different from the old one and keeps track of how answer types change. As a result, in $\lambda^S_{s/r}$, a type judgment takes the form $\Gamma;\, \alpha \vdash_S e : \tau;\, \beta$, which means that a term e is typed at type τ under a typing context Γ and it modifies its answer type α to β when evaluated. Also, a function type is now annotated with information on answer types and written $\sigma/\alpha \to \tau/\beta$. It represents a function from σ to τ, and the answer type is modified from α to β when it is applied.

We show representative typing rules below. For details, readers are referred to Asai and Kameyama [1], Danvy and Filinski [3].

$$\frac{\Gamma;\, \gamma \vdash_S e_1 : \tau_1/\alpha \to \tau_2/\beta;\, \delta \qquad \Gamma;\, \beta \vdash_S e_2 : \tau_1;\, \gamma}{\Gamma;\, \alpha \vdash_S e_1 e_2 : \tau_2;\, \delta}\ (\textsc{TApp})$$

$$\frac{\Gamma, k : \tau/\delta \to \alpha/\delta;\, \gamma \vdash_S e : \gamma;\, \beta}{\Gamma;\, \alpha \vdash_S \mathcal{S}k : \tau/\delta \to \alpha/\delta.\, e : \tau;\, \beta}\ (\textsc{TShift})$$

3. Gradualizing $\lambda^S_{s/r}$

Figure 1 shows the syntax of $\lambda^G_{s/r}$, which is obtained by adding the type dynamic $\star$ to $\lambda^S_{s/r}$. (B ranges over base types.) Abstractions and reset have unusual type annotations, which are explained later.

Our *type consistency*, which is a key notion in gradual typing, for $\lambda^G_{s/r}$ is very straightforward: it is given as the least compatible relation including $\star \sim \tau$. For example, $\mathsf{int}/\star \to \star/\mathsf{bool}$ is consistent with $\mathsf{int}/\mathsf{int} \to \star/\star$.

Typing Rules. One natural question that arises in deriving a gradual version of typing rules is where type consistency should be allowed for different occurrences of the same metavariable for types. Our approach to this question is basically the same as the Gradualizer [2], which starts with typing rules viewed as a moded logic program (so that one can build a straightforward type checking procedure by induction on the term structure) and insert consistency according to input/output modes (and other notions). In our setting, we set Γ, e and β in the type judgment form $\Gamma;\, \alpha \vdash_S e : \tau;\, \beta$ to be inputs and α and τ to be outputs.[1] There may be other possible mode specifications but it seems to us that, without having β as an input, building a simple type checking procedure would be difficult. Then, we analyze typing rules as in the Gradualizer.

Due to space restriction, we only show the resulting typing rules in Figure 1. The relation $\sigma \triangleright \tau_1/\alpha \to \tau_2/\beta$ in GTApp is called *matching* [9]. It is the least reflexive relation including $\star \triangleright \star/\star \to \star/\star$ and allows the function part of an application to have the dynamic type. The premises $\tau_1 \sim \tau_1'$ and $\beta \sim \beta'$ are derived from the fact that these occurrences are at output positions.[2] GTShift is similar to GTApp but we require σ to match $\tau/\delta \to \alpha/\delta$ with the same answer type δ to make cast insertion type-preserving.

Perhaps surprisingly, the Gradualizer would not yield a typing rule like GTShift. The Gradualizer replaces only type constructors in *output positions in premises* with fresh metavariables for types. It is found from this fact that the

* Current affiliation: IBM Research - Tokyo

[1] Actually, this mode specification works if annotations on abstraction and reset to specify β are added. Hence the unusual type annotations mentioned above.

[2] Those who are familiar with the Gradualizer may notice that conditions on β and β' will be not $\beta \sim \beta'$ but $\beta^{\mathrm{J}} = \beta \sqcup \beta'$. Since β^{J} does not occur elsewhere, this can be replaced with $\beta \sim \beta'$ without changing the meaning of the rule.

Types $\sigma, \tau, \alpha, \beta, \gamma, \delta ::= B \mid \tau/\tau \to \tau/\tau \mid \star$

Terms $e ::= x \mid c \mid \lambda^{\beta} x{:}\tau.\, e \mid e_1 e_2 \mid \mathcal{S}k{:}\sigma.\, e \mid \langle e \rangle^{\tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma;\, \alpha \vdash_G x : \tau;\, \alpha}\ (\textsc{GTVar})\qquad \Gamma;\, \alpha \vdash_G c : ty(c);\, \alpha\ (\textsc{GTConst})$$

$$\frac{\Gamma, x : \tau_1;\, \alpha \vdash_G e : \tau_2;\, \beta}{\Gamma;\, \gamma \vdash_G \lambda^{\beta} x{:}\tau_1.\, e : \tau_1/\alpha \to \tau_2/\beta;\, \gamma}\ (\textsc{GTAbs})$$

$$\frac{\Gamma;\, \gamma \vdash_G e_1 : \sigma;\, \delta \qquad \Gamma;\, \beta' \vdash_G e_2 : \tau_1';\, \gamma \qquad \sigma \rhd \tau_1/\alpha \to \tau_2/\beta \qquad \tau_1 \sim \tau_1' \qquad \beta \sim \beta'}{\Gamma;\, \alpha \vdash_G e_1 e_2 : \tau_2;\, \delta}\ (\textsc{GTApp})$$

$$\frac{\Gamma, k : \sigma;\, \gamma' \vdash_G e : \gamma;\, \beta \qquad \sigma \rhd \tau/\delta \to \alpha/\delta \qquad \gamma \sim \gamma'}{\Gamma;\, \alpha \vdash_G \mathcal{S}k{:}\sigma.\, e : \tau;\, \beta}\ (\textsc{GTShift})$$

$$\frac{\Gamma;\, \gamma' \vdash_G e : \gamma;\, \tau \qquad \gamma \sim \gamma'}{\Gamma;\, \alpha \vdash_G \langle e \rangle^{\tau} : \tau;\, \alpha}\ (\textsc{GTReset})$$

Figure 1. Syntax / Typing rules ($\Gamma;\, \alpha \vdash_G e : \tau;\, \beta$)

form of types of captured delimited continuations in the typing rule which the Gradualizer derives from TShift is restricted to be function types and cannot be metavariable σ for types because: (1) typing contexts in premises of typing rules are regarded as inputs, so the derived rule from TShift has premises whose typing contexts take the same form as TShift; and (2) the expression in the conclusion of the derived rule is the same as TShift (modulo metavariable renaming). We think that our typing rule is closer to a dynamically typed language in the sense that shift-bound variables can be used arbitrarily if they are given $\star$.

Translation to the Cast Calculus. The semantics of $\lambda^{G}_{s/r}$ is defined through translation into the intermediate language $\lambda^{C}_{s/r}$, (a subset of) the blame calculus with shift and reset proposed by Sekiyama et al. [7]. The target language extends the syntax of $\lambda^{G}_{s/r}$ with casts $f : \tau \Rightarrow^{\ell} \tau'$ which are tagged with blame labels ℓ. Casts preserve answer types:

$$\frac{\Gamma;\, \alpha \vdash_{CC} f : \tau;\, \beta \qquad \tau \sim \tau'}{\Gamma;\, \alpha \vdash_{CC} (f : \tau \Rightarrow^{\ell} \tau') : \tau';\, \beta}\ (\textsc{CTCast})$$

Figure 2 shows some of the cast insertion rules from $\lambda^{G}_{s/r}$ to $\lambda^{C}_{s/r}$. In these rules, we assume blame labels are all fresh. Basically, our rules insert casts where the type consistency relation appears. For example, CApp has two casts, one to adjust the actual argument type and the other to adjust one of the answer types of the function. Actually, we could merge the cast on f_2 to the other cast by $f_1 : \sigma \Rightarrow^{\ell_1} \tau_1'/\alpha \to \tau_2/\beta'$. We expect that our rule is practically better because two blame labels can distinguish a failure by argument mismatch and one by answer type mismatch. CShift is a little more involved. A cast on the continuation variable k' is needed because the target language does not allow a continuation variable bound by shift to have $\star$.

Interestingly, we could not apply the Gradualizer to derive cast insertion rules. As far as we understand, it will put a cast from σ to $\tau_1/\alpha \to \tau_2/(\beta \sqcup \beta')$ on f_1. To make the entire application well-typed, the left answer type β' for f_2 should be adjusted somehow to $\beta \sqcup \beta'$ but the Gradualizer fails to do it, although such adjustment is in fact possible by using shift and cast on the captured delimited continuation. Deriving translation rules for shift and reset will fail for similar reasons.

$$\frac{\Gamma;\, \gamma \vdash_{CC} e_1 \rightsquigarrow f_1 : \sigma;\, \delta \qquad \Gamma;\, \beta' \vdash_{CC} e_2 \rightsquigarrow f_2 : \tau_1';\, \gamma \qquad \sigma \rhd \tau_1/\alpha \to \tau_2/\beta \qquad \tau_1 \sim \tau_1' \qquad \beta \sim \beta'}{\Gamma;\, \alpha \vdash_{CC} e_1 e_2 \rightsquigarrow (f_1 : \sigma \Rightarrow^{\ell_1} \tau_1/\alpha \to \tau_2/\beta') \; (f_2 : \tau_1' \Rightarrow^{\ell_2} \tau_1) : \tau_2;\, \delta}\ (\textsc{CApp})$$

$$\frac{\Gamma, k : \sigma;\, \gamma' \vdash_{CC} e \rightsquigarrow f : \gamma;\, \beta \qquad \sigma \rhd \tau/\delta \to \alpha/\delta \qquad \gamma \sim \gamma'}{\Gamma;\, \alpha \vdash_{CC} \mathcal{S}k{:}\sigma.\, e \rightsquigarrow \mathcal{S}k'.\, (\lambda k.\, (f : \gamma \Rightarrow^{\ell_1} \gamma')) \; (k' : \tau/\delta \to \alpha/\delta \Rightarrow^{\ell_2} \sigma) : \tau;\, \beta}\ (\textsc{CShift})$$

$$\frac{\Gamma;\, \gamma' \vdash_{CC} e \rightsquigarrow f : \gamma;\, \tau \qquad \gamma \sim \gamma'}{\Gamma;\, \alpha \vdash_{CC} \langle e \rangle^{\tau} \rightsquigarrow \langle f : \gamma \Rightarrow^{\ell} \gamma' \rangle : \tau;\, \alpha}\ (\textsc{CReset})$$

Figure 2. Cast insertion rules ($\Gamma;\, \alpha \vdash_{CC} e \rightsquigarrow f : \tau;\, \beta$)

4. Properties

We are investigating correctness criteria for gradual typing [9]. At the time of writing, we have proved that (1) the typing relation is a conservative extension over that of $\lambda^{S}_{s/r}$; (2) type preservation of cast insertion; (3) type soundness; (4) blame-subtyping theorem; and (5) monotonicity w.r.t. precision [2] (which is also stated as a part of the gradual guarantee in Siek et al. [9]). It is mostly straightforward to adapt the statements to our setting.

There is a subtlety in precision $e_1 \sqsubseteq e_2$ (read "e_1 is more precise than e_2"), which intuitively means that e_2 is obtained by replacing some occurrences of types in e_1 with $\star$. However, the typing rule for shift requires the two answer types in the type declaration to be the same, so replacement of answer types should take place at once. For example, $\mathcal{S}k : \mathsf{bool}/\mathsf{int} \to \mathsf{bool}/\mathsf{int}.\, e \sqsubseteq \mathcal{S}k : \mathsf{bool}/\star \to \mathsf{bool}/\star.\, e$ but $\mathcal{S}k{:}\mathsf{bool}/\mathsf{int} \to \mathsf{bool}/\mathsf{int}.\, e \not\sqsubseteq \mathcal{S}k{:}\mathsf{bool}/\mathsf{int} \to \mathsf{bool}/\star.\, e$.

5. Related Work

Cimini and Siek [2] have proposed the Gradualizer, a method to automatically derive a gradually typed variant with cast insertion rules from a statically typed language, and demonstrated that it can be applied to various (simple) type systems. As we have already discussed, we cannot fully use the Gradualizer: although it can derive similar typing rules as ours, it fails to generate cast insertion rules.

Sekiyama et al. [7] propose the blame calculus with shift and reset, and prove its soundness and the blame theorem. This blame calculus is used in our intermediate language and its properties are used in our proofs.

6. Conclusion

We show a gradually typed language with delimited-control operators shift and reset and answer-type modification. We state that the language has some desirable properties as a gradually typed language.

We are currently working on proving remaining properties discussed in Siek et al. [9]. It is not very nice that our type system requires more annotations. Since the original type system by Danvy and Filinski has the principal type property [1], the next step would be to combine type inference and gradual typing [6, 8].

References

[1] K. Asai and Y. Kameyama. Polymorphic delimited continuations. In *Proc. of APLAS.*, pages 239–254, 2007.

[2] M. Cimini and J. G. Siek. The gradualizer: A methodology and algorithm for generating gradual type systems. In *Proc. of ACM POPL*, pages 443–455, 2016.

[3] O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, 1989.

[4] O. Danvy and A. Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.

[5] A. Filinski. Representing monads. In *Proc. of ACM POPL*, pages 446–457, 1994.

[6] R. Garcia and M. Cimini. Principal type schemes for gradual programs. In *Proc. of ACM POPL*, pages 303–315, 2015.

[7] T. Sekiyama, S. Ueda, and A. Igarashi. Shifting the blame - A blame calculus with delimited control. In *Proc. of APLAS*, pages 189–207, 2015.

[8] J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Proc. of Dynamic Languages Symposium (DLS)*, 2008.

[9] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL)*, pages 274–293, 2015.