Modular and Automated Type-Soundness Verification for Language Extensions

Sebastian Erdweg[1] and Florian Lorenzen[2]

[1] TU Darmstadt [2] TU Berlin

Whenever code generation is used to abstract from low-level details or to provide high-level interfaces to software developers, type errors in generated code jeopardize the abstraction barrier: First, error messages are in terms of generated code and thus expose programmers to low-level details that should be hidden. Second, manual inspection of generated code may be necessary to identify the cause of the type error. Third, since a type error in generated code may be caused by either a defective generator or by invalid generator input, manual inspection of the generator may be necessary to identify the generator's contract and whether the input adheres to that contract. Type errors in generated code present a serious usability threat for abstractions implemented via code generation.

We address this problem in the context of code generators that extend a base language with new language constructs by translation into other constructs of the base language. Such code generators are sometimes referred to as desugarings. Many compilers employ desugarings to transform programs of the input language into a core language, so that subsequent compiler phases can focus on fewer language constructs. Moreover, macro systems empower regular programmers to introduce new language constructs via desugaring transformations. Despite wide-spread application of desugarings, few existing compilers and no existing macro system can *guarantee the absence of type errors in desugared code.*

To this end, we present SoundExt, a formalism for soundly extending a base language with new language constructs. SoundExt statically and modularly validates a language extension and guarantees that desugared code does not contain type errors. More specifically, for each a language extension, SoundExt requires the definition of (i) an extended syntax, (ii) type rules for checking programs that use the extended syntax, and (iii) a desugaring transformation that translates a program of the extended syntax into a base-language program. SoundExt then derives proof obligations for each user-defined type rule: For all programs permitted by the type rule, the desugared version of these programs must have the same type. SoundExt synthesizes the corresponding proof for each type rule by instrumenting the inference engine with additional axioms that correspond to the assumptions of the user-supplied type rules. We exemplify our methodology in following Section. We have verified that the validity of each derived proof obligation entails the following high-level property:

$$\Gamma \vdash_{ext} e : T \;\wedge\; e \rightsquigarrow^* e' \;\wedge\; e' \in Base \quad\Rightarrow\quad \Gamma \vdash_{base} e' : T$$

That is, given a program e that is well-typed in the extended type system, if this program desugars into a base-language program e', then the desugared

$$\text{T-Var}\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-Abs}\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1 . e : T_1 \to T_2}$$

$$\text{T-App}\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

Fig. 1. Type rules of the simply-typed lambda calculus.

program is well-typed in the base type system. In other words, type checking the user-supplied program is sufficient to ensure the absence of type errors in desugared code.

To demonstrate the expressiveness of SoundExt, we instantiated the formalism for SugarFomega, a syntactically extensible variant of System F_ω. Besides standard lambda and type abstraction, SugarFomega features variants, records, and higher-order iso-recursive types as well as SugarJ-like macros with flexible syntax [1,3,4]. Using this macro system, SugarFomega programmers can introduce new language constructs at the level of expressions, types, and kinds. Accordingly, programmers define additional "type" rules using type and kind judgments. We have implemented language extensions of SugarFomega for *let* expressions, monadic *do* blocks (no implicit dictionary passing), and even algebraic data types. SoundExt modularly validated each of these extensions and guarantees that they are sound with respect to the type system of System F_ω.

Illustrating example

Figure 1 shows the standard type rules of the simply-typed lambda calculus as they, for example, appear in Pierce's Types and Programming Languages [5]. The soundness of the corresponding type system is an established fact in the programming-language community. However, when extending such a base language, one has to manually reestablish the soundness theorem for the extended language [6]. As we demonstrate with SoundExt, we can automatically verify the soundness of the extended type system for language extensions that are defined through translation into the base language.

For example, consider the extension of the simply-typed lambda calculus with *let* expressions:

$$e ::= \ldots \mid \text{let } x : T = e \text{ in } e$$

We can rewrite *let* expressions to the simply-typed lambda calculus using the following desugaring:

$$\mathsf{desugar\text{-}let} : (\text{let } x : T = e_1 \text{ in } e_2) \rightsquigarrow (\lambda x : T . e_2)\ e_1$$

Since we want to avoid type errors in generated code, we extend the type system of the simply-typed lambda calculus with a type rule for *let* expressions:

$$\text{T-Let}\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \text{let } x : T_1 = e_1 \text{ in } e_2 : T_2}$$

In the hope of preventing type errors in generated code, we use the extended type system to validate a user program prior to any desugaring. This way, the rewrite rule desugar-let will only be applied to a *let* expression that has been checked by T-Let. For example, the expression

$$\text{let } n : \text{Nat} = 17 \text{ in } n + n$$

is well-typed since 17 has type Nat and the judgment $n : \text{Nat} \vdash n + n : \text{Nat}$ holds. Therefore, it is safe to apply the rewrite rule, that is, the rewriting generates a well-typed expression:

$$(\lambda n : \text{Nat}.n + n)\ 17$$

Conceptually, there are two sources of possible errors. First, the rewrite rule may be defective and produce ill-typed or wrongly typed code, even though the input *let* expression was well-typed according to T-Let. Second, the type rule may be defective and admit *let* expressions that are not well-typed. For example, a defective rewrite rule $(\text{let } x : T = e_1 \text{ in } e_2) \rightsquigarrow (e_2\ e_1)$ would translate above *let* expression into the ill-typed expression $(n + n)\ 17$. Conversely, suppose we forgot the first premise in the definition of the type rule T-Let. This defective type rule would admit the expression $(\text{let } f : \text{Nat} \to \text{Nat} = 17 \text{ in } f\ 0)$, which the (correct) rewrite rule desugar-let translates into the ill-typed program $(\lambda f : \text{Nat} \to \text{Nat}.f\ 0)\ 17$. Technically, these two sources of errors are two sides of the same coin: To guarantee the absence of type errors in generated code, we must ensure that *the rewrite rule and the type rule are correct with respect to each other*.

To this end, we can read the type rule T-Let as a contract for the rewrite rule desugar-let: The rewrite rule may assume all input adheres to the type rule T-Let, and the rewrite rule must produce an expression of the type declared in the type rule. In essence, the rewrite rule must be type-preserving with respect to the type rules. If this holds, we can type check the user program once before desugaring and know that the desugared program has the same type.

To verify that user-defined rewrite rules preserve types according to the user-defined type rules, we proceed as follows. We symbolically apply the rewrite rule to the subject of the corresponding type rule. For example, for the *let* extension we obtain the type rule T-Let':

$$\text{T-Let'}\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash (\lambda x : T_1.e_2)\ e_1 : T_2}$$

We could use this type rule to validate the code generated by desugar-let. Instead, we want to show that this type rule is *admissible*, that is, for all expressions typeable through applications of T-Let', there is a derivation in the type system without T-Let' given the same context yielding the same type. Accordingly, we do not need T-Let' to type check the generated code, because the other type rules already are expressive enough.

SoundExt automatically infers proof obligations for the admissibility of derived type rules. But, SoundExt also automatically verifies these proof obligations. In fact we can reuse the type system for checking the admissibility of

a derived type rule if (i) we interpret all metavariables in the assumptions and conclusion as constants that only unify with themselves and (ii) we temporarily add the assumptions of the derived type rule as axioms to the system. A type rule then is admissible if we can find a derivation of the modified conclusion given the additional axioms.

For example, for T-Let' we try to find a derivation for $\Gamma \vdash (\lambda x : T_1 . e_2)\ e_1 : T_2$ given the axioms (Ax1) $\Gamma \vdash e_1 : T_1$ and (Ax2) $\Gamma, x : T_1 \vdash e_2 : T_2$. Indeed, we can infer the following derivation, where all occurring metavariables in fact are constants. For instance, it is not possible to derive $\Gamma \vdash 0 : T_1$ using Ax1.

$$\text{T-App}\dfrac{\text{T-Abs}\dfrac{\text{Ax2}\dfrac{}{\Gamma, x : T_1 \vdash e_2 : T_2}}{\Gamma \vdash \lambda x : T_1 . e_2 : T_1 \to T_2} \qquad \text{Ax1}\dfrac{}{\Gamma \vdash e_1 : T_1}}{\Gamma \vdash (\lambda x : T_1 . e_2)\ e_1 : T_2}$$

This derivation shows that the type rule T-Let' is admissible. As consequence, given an expression that is well-typed according to the user-defined type rule T-Let, we know that the expression generated by desugar-let has the same type as the original *let* expression. Accordingly, no type errors can emerge from the code generated by the desugaring rule.

Outlook

SoundExt modularly checks language extensions to ensure desugared code is well-typed. As long as extensions are not syntactically overlapping, SoundExt supports *incremental extension* [2] (one extension desugars into code of another extension) and *extension unification* [2] (independent extensions can be unified into a single extension). We have formally verified the soundness of SoundExt and the composability of SoundExt extensions.

References

1. S. Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction.* PhD thesis, Philipps-Universiät Marburg, 2013.
2. S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, pages 7:1–7:8. ACM, 2012.
3. S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 391–406. ACM, 2011.
4. S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of Haskell Symposium*, pages 149–160. ACM, 2012.
5. B. C. Pierce. *Types and programming languages.* MIT press, 2002.
6. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.