Modular and Automated Type-Soundness Verification for Language Extensions

Florian Lorenzen
TU Berlin, Germany

Sebastian Erdweg
TU Darmstadt, Germany

Abstract

Language extensions introduce high-level programming constructs that protect programmers from low-level details and repetitive tasks. For such an abstraction barrier to be sustainable, it is important that no errors are reported in terms of generated code. A typical strategy is to check the original user code prior to translation into a low-level encoding, applying the assumption that the translation does not introduce new errors. Unfortunately, such assumption is untenable in general, but in particular in the context of extensible programming languages, such as Racket or SugarJ, that allow regular programmers to define language extensions.

In this paper, we present a formalism for building and automatically verifying the type-soundness of syntactic language extensions. To build a type-sound language extension with our formalism, a developer declares an extended syntax, type rules for the extended syntax, and translation rules into the (possibly further extended) base language. Our formalism then validates that the user-defined type rules are sufficient to guarantee that the code generated by the translation rules cannot contain any type errors. This effectively ensures that an initial type check prior to translation precludes type errors in generated code. We have implemented a core system in PLT Redex and we have developed a syntactically extensible variant of System F_ω that we extend with *let* notation, monadic *do* blocks, and algebraic data types. Our formalism verifies the soundness of each extension automatically.

1. Introduction

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.

In this paper, we address above 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 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 present the details of our methodology in Section 3, where we also show that the validity of each derived proof obligation entails the following high-level property:

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

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 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 instantiate 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 [5, 8, 9]. 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 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.

In summary, we make the following contributions:

- We present the design of SoundExt, a formalism for type-sound language extensibility.
- SoundExt automatically verifies the type-soundness of language extensions by deriving type rules for generated programs and checking the admissibility of these type rules.
- To check admissibility of type rules, SoundExt instruments the type checker with additional axioms on non-unifiable metavariables.
- We present a formalization of SoundExt based on PLT Redex and prove that the desugarings of successfully verified language extensions adhere to type preservation and progress.
- We verify that SoundExt extensions compose soundly as long as there is no syntactic overlap.
- We implement the extensible language SugarFomega based on SoundExt and show how SoundExt enables us to soundly extend SugarFomega with *let* expressions, monadic *do* blocks, and algebraic data types.

2. 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 [19]. 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 [28]. As we demonstrate with SoundExt in this paper, 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:

$$e ::= \ldots \mid \mathrm{let}\, x\!:\!T = e\, \mathrm{in}\, e$$

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

$$\mathsf{desugar\text{-}let} : (\mathrm{let}\, x\!:\!T = e_1\, \mathrm{in}\, e_2) \dashrightarrow (\lambda x\!:\!T.e_2)\; e_1$$

Since we want to detect and report type errors prior to desugaring, we extend the type system of the simply-typed lambda calculus with a type rule for *let* expressions:

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

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

$$\mathrm{let}\, n\!:\!\mathrm{Nat} = 17\, \mathrm{in}\, n + n$$

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

$$(\lambda n\!:\!\mathrm{Nat}.n+n)\; 17$$

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 $(\mathrm{let}\, x\!:\!T = e_1\, \mathrm{in}\, e_2) \dashrightarrow (e_2\;\, e_1)$ 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 $(\mathrm{let}\; f\!:\!\mathrm{Nat} \to \mathrm{Nat} = 17\, \mathrm{in}\, f\, 0)$, which the (correct) rewrite rule desugar-let translates into the ill-typed program $(\lambda f\!:\!\mathrm{Nat} \to \mathrm{Nat}.f\, 0)\; 17$. 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*.

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

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

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

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':

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

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.

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

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.

$$\begin{array}{rlr} t & ::= x \mid (c\,\overline{t}) & \textit{Terms} \\ J & ::= t \vdash t : t & \textit{Judgments} \\ I & ::= \forall \overline{x}.\ \overline{J} \Rightarrow J & \textit{Inference rules} \\ R & ::= t \dashrightarrow t & \textit{Rewrite rules} \\ E & ::= \mathit{ext}\ \overline{I}\ \overline{R} & \textit{Extensions} \\ B & ::= \overline{I} & \textit{Base systems} \end{array}$$

Figure 2. Syntax for declaring extensions.

In the following section we describe the approach followed by SoundExt formally and proof that the admissibility of derived type rules indeed entails type preservation of desugarings.

3. SoundExt

A language extension defines new syntax, type rules for that syntax, and rewritings of that syntax into the base language. SoundExt provides a language for the declaration of such extensions and a verification procedure for checking type soundness of extensions. We verify that our verification procedure is sufficient to guarantee that no type errors are in desugared code.

Our formalization is based on an implementation of SoundExt in PLT Redex [10].[1] However, whereas our PLT Redex implementation contains a concrete rewrite engine and inference engine, we parameterize our formalization over these engines and formulate minimal assumptions. Moreover, like the PLT Redex implementation, our formalization ignores all aspects of concrete syntax and assumes programmers write abstract syntax trees directly; our implementation of SugarFomega (see next section) supports the declaration of concrete syntax for extensions.

3.1 A language for declaring extensions

We define abstract syntax for the declaration of extensions as shown in Figure 2. We use a generic term representation to generalize over the expression, types, and contexts of any particular language. Essentially, a term t is an abstract syntax tree consisting of constructor applications $(c\,\overline{t})$ (overlines denote sequences) and metavariables x. We use these terms for declaration of judgments J. A judgment $t_1 \vdash t_2 : t_3$ relates a subject t_2 to an object t_3 under a context t_1. We conventionally choose metavariable names that resemble type checking as in $\Gamma \vdash e : T$.

Multiple judgments make up an inference rule, where all but the last judgments are premises and free metavariables must be explicitly quantified. For example, we can encode the type rule T-Abs for lambda abstraction in this format:

$$\begin{aligned} I_{\text{ABS}} = \forall\, \Gamma\ x\ e\ T_1\ T_2.\quad & (\texttt{Bind}\ \Gamma\ x\ T_1) \vdash e : T_2 \\ \Rightarrow\quad & \Gamma \vdash (\texttt{abs}\ x\ T_1\ e) : (\texttt{Fun}\ T_1\ T_2) \end{aligned}$$

For axioms, that is, inference rules without premises, we simply write $\forall \overline{x}.\ J$; for inference rules that do not bind any metavariables, we drop the quantifier $\overline{J} \Rightarrow J$.

Similar to inference rules, we can use terms to specify rewritings $e_1 \dashrightarrow e_2$, where e_1 may contain metavariables to pattern match on a term and e_2 is a generation template that can use the metavariables bound during pattern matching. Based on these definitions, we define an extension as a list of inference rules and rewrite rules.

The base language B only consists of inference rules but no rewrite rules. We do not model the dynamic semantics of base languages since we are only interested in static guarantees.

For simplicity, we assume that all rewrite rules of an extension are overlap-free, that is, there is no program to which two different rewrite rules can apply. Furthermore, rewrite rules of an extension must fail for pure base-language programs. We describe how to lift these restrictions in Section 3.6.

[1] Implementation available online: `http://sugarj.org/fomega`

$$\text{R-MATCH}\ \frac{\mathit{rewrite}\ \overline{R}\ e = e'}{e \leadsto_{\overline{R}} e'} \qquad \text{R-CON}\ \frac{\begin{array}{c} e_2 \leadsto_{\overline{R}} e_2' \\ e_1 \not\leadsto_{\overline{R}} e_1'\ \forall e_1 \in \overline{e_1} \end{array}}{c\ \overline{e_1}\ e_2\ \overline{e_3} \leadsto_{\overline{R}} c\ \overline{e_1}\ e_2'\ \overline{e_3}}$$

Figure 3. Specification of a left-to-right, one-step rewriting.

3.2 Assumptions about the rewrite engine

SoundExt is not tied to a specific rewriting algorithm. We assume a function $\mathit{rewrite}\ \overline{R}\ e$ that performs a single rewriting step by one of the rules in $\overline{R}$ or returns *fail* if no rule is applicable. Since we want to apply the rewrite function to subjects of inference rules that contain metavariables, we must require that rewriting is closed under metavariable substitution:

Assumption 1 (Metavariable substitution-invariance)**.** *If* $\mathit{rewrite}\ \overline{R}\ e[\overline{x}] = e'[\overline{x}]$*, then* $\mathit{rewrite}\ \overline{R}\ e[\,\overline{t}\,] = e'[\,\overline{t}\,]$.

We write $e[\overline{x}]$ to denote a term e in which at most the metavariables $\overline{x}$ appear free, and $e[\,\overline{t}\,]$ for the substitutions of these metavariables by the terms $\overline{t}$.

This assumption requires that rewriting is independent of metavariable occurrences. For example, consider the following two rewrite rules:

$$\begin{array}{ll} \mathsf{plus1} : (\texttt{Plus}\ (\texttt{Suc}\ m)\ n) & \dashrightarrow (\texttt{Suc}\ (\texttt{Plus}\ m\ n)) \\ \mathsf{plus2} : (\texttt{Plus}\ m\ n) & \dashrightarrow n \end{array}$$

Given the input term $(\texttt{Plus}\ x\ \texttt{Zero})$ with metavariable x, the rewrite function might do pattern matching to apply plus2 and return Zero. However, if we substitue $(\texttt{Suc}\ y)$ for x, rule plus1 becomes applicable and yields $(\texttt{Suc}\ (\texttt{Plus}\ y\ \texttt{Zero}))$. This violates Assumption 1 because applying the same substitution to the old result does not yield the new result. However, it is easy to recover this problem by expanding the pattern matching of plus2:

$$\mathsf{plus2'} : (\texttt{Plus}\ \texttt{Zero}\ n) \dashrightarrow n$$

Given this rule, the rewrite function would fail for the input term $(\texttt{Plus}\ x\ \texttt{Zero})$, which is consistent with Assumption 1. In general, a rewrite engine should arrange that if a rewrite rule inspects a subtree, this subtree cannot match a metavariable. In our PLT Redex implementation, we assume that user-defined rewrite rules already adhere to the above properties.

Based on the function *rewrite*, we define the function $e_1 \leadsto_{\overline{R}} e_2$ that performs a top-down, left-to-right, one-step rewriting of e_1 using $\mathit{rewrite}\ \overline{R}$. The function $e_1 \leadsto_{\overline{R}} e_2$ adheres to the specification given in Figure 3. We write $e \leadsto^*_{\overline{R}} e'$ for the reflexive, transitive closure of $e \leadsto_{\overline{R}} e'$.

Finally, we require that *rewrite* does not succeed on terms of the base language. As we will later show, this ensures that the structure of base-language programs is preserved by $\leadsto_{\overline{R}}$.

Assumption 2 (Rewrite fails on base-language constructors)**.** *For any base-language constructor c,* $\mathit{rewrite}\ \overline{R}\ (c\,\overline{e}) = \mathit{fail}$.

3.3 Assumptions about the inference engine

SoundExt requires developers to declare static analyses for extensions via inference rules. To reason about the user-defined inference rules and to proof their soundness, SoundExt employs an inference engine to compute derivations. As for the rewriting, SoundExt does not require any specific inference engine. Instead, we assume an inference engine that, given inference rules $\overline{I}$ and a judgment J, checks if J can be derived using the rules $\overline{I}$. If successful, the inference engine yields a concrete derivation $\overline{I} \succ J$. Without going into any detail, we require correctness of the inference engine:

Assumption 3 (Correctness)**.** *If the inference engine yields a derivation $\overline{I} \succ J$, then the derivation only consists of valid applications of the rules in $\overline{I}$.*

Since we want to use the inference engine not only for checking concrete programs but also for mechanically proving entailment between judgments, we require proper handling of metavariables by the inference engine. Specifically, like for the rewrite engine, we require that the inference of derivations is closed under metavariable substitution (in the context of logical consequences this property is known as *structurality* [15]):

Assumption 4 (Metavariable substitution-invariance)**.**
If $\overline{I[\overline{x}]} \succ J[\overline{x}]$, then $\overline{I[\overline{t}\,]} \succ J[\overline{t}\,]$.

This requirement means that if the inference engine is able to derive $J[\overline{x}]$ using $\overline{I[\overline{x}]}$ without any knowledge about the free metavariables $\overline{x}$, then a similar derivation must exist when instantiating the free metavariables with concrete terms. For example, if the inference engine can deduce a derivation for the identity function

$$(I_{\text{VAR}}\ I_{\text{ABS}}\ I_{\text{APP}}) \succ (\Gamma \vdash (\texttt{abs}\ x\ T\ x) : T)$$

then a derivation must exist for any replacements of the free metavariables Γ, x, and T.[2]

Recall that our inference rules quantify bound metavariables, as is visible in the inference rule for lambda abstraction I_{ABS} shown in Section 3.1. SoundExt capitalizes the fact that inference rules can also contain free metavariables to install assumptions about specific metavariables. In contrast to bound metavariables, free metavariables in an inference rule may not be instantiated when applying the inference rule. For example, when deriving

$$(\forall \Gamma.\ \Gamma \vdash e : T) \succ \emptyset \vdash e : T$$

only the bound variable Γ gets instantiated to $\emptyset$ whereas e and T must match literally. Accordingly, we could not derive $\emptyset \vdash e_2 : T$ from the same inference rule, because e does not literally match e_2.

Finally, we require the inference engine to satisfy the cut rule:

Assumption 5 (Cut)**.** *If $\overline{I} \succ J_1$ and $\overline{I}\ J_1 \succ J_2$, then $\overline{I} \succ J_2$.*

In our PLT Redex implementation, we do not use quantifiers in inference rules but implicitly quantify over all free variables. To encode non-unifiable metavariables, we wrap such metavariables in $(\texttt{symbol}\ x)$ nodes. Otherwise, our inference engine performs a simple proof-tree search and respects both metavariable substitution-invariance and the cut rule.

3.4 Extension soundness

Using the rewrite engine and the inference engine, we define a verification procedure for the soundness of extensions relative to a base system. The basic idea is to show that rewrite rules are type-preserving. We do so by (i) deriving an inference rule that admits exactly the programs generated by the rewrite rule but requires the original type, and (ii) verifying that this derived rule is admissible, that is, all programs it admits were already admitted by the original inference rules. Together, this shows that all programs generated by the rewrite rule are admitted by the original inference rules, and thus must be well-typed. The full verification procedure is shown in Figure 4.

[2] Note that there is only a single syntactic category in SoundExt: terms t. As far as SoundExt or the inference engine are concerned, any term can be used to represent any object-language construct, such as a variable.

An extension $ext\ \overline{I}\ \overline{R}$ is sound with respect to a base system B if for all typing rules $I \in \overline{I}$ the following steps succeed:

Let $I = \forall \overline{x}.\ \overline{J[\overline{x}]} \Rightarrow (\Gamma[\overline{x}] \vdash e[\overline{x}] : T[\overline{x}])$.
Let $\overline{y}$ be fresh and distinct variables with $len\ \overline{y} = len\ \overline{x}$.

1. Perform one-step rewriting of rule's subject with extension's rewrite rules:
$$e[\overline{x}] \rightsquigarrow_{\overline{R}} e'[\overline{x}]$$
2. Show that derived rule $I_D = \forall \overline{x}.\ \overline{J[\overline{x}]} \Rightarrow (\Gamma[\overline{x}] \vdash e'[\overline{x}] : T[\overline{x}])$ is admissible in $B\ \overline{I}$:
$$B\ \overline{I}\ \overline{J[\overline{y}]} \succ (\Gamma[\overline{y}] \vdash e'[\overline{y}] : T[\overline{y}])$$

Figure 4. The SoundExt verification procedure.

In Step 1, we rewrite the subject of the inference rule I by applying the $rewrite$ function. Since the subject of an inference rule typically includes metavariables, this amounts to a symbolic rewriting of the rule's subject. Due to Assumption 1 on the rewrite engine, the rewritten subject $e'[\overline{x}]$ captures all programs that can ever be generated by the rewrite rules $\overline{R}$ from programs admitted by I.

In Step 2, we define a derived inference rule I_D that uses the rewritten subject $e'[\overline{x}]$ but has the same premises, context, and object. The derived rule admits exactly those programs that can be generated by $\overline{R}$ from programs admitted by I. By showing that I_D is admissible in $B\ \overline{I}$, we ensure that generated programs have derivations in $B\ \overline{I}$ without I_D.

To verify the admissibility of I_D, we instantiate its bound variables with fresh variables $\overline{y}$. We then install the premises of the derived rule as axioms $\overline{J[\overline{y}]}$ into the inference engine and try to derive the conclusion of the derived rule $\Gamma[\overline{y}] \vdash e'[\overline{y}] : T[\overline{y}]$. If this derivation succeeds, it means the rules from B and $\overline{I}$ proof that the premises of the derived rule entail the conclusion of the derived rule for any $\overline{y}$. Intuitively, we can use this derivation to eliminate any application of the derived rule I_D.

For example, for the *let* extension shown in Section 2, we proof the admissibility of the derived rule T-Let' as follows:

$$\begin{aligned}&(I_{\text{VAR}}\ I_{\text{ABS}}\ I_{\text{APP}}\ I_{\text{LET}}\ \ (\Gamma' \vdash e_1' : T_1')\ \ (\Gamma', x' : T_1' \vdash e_2' : T_2'))\\ &\succ\ \ (\Gamma' \vdash (\texttt{app}\ (\texttt{abs}\ x'\ T_1'\ e_2')\ e_1') : T_2')\end{aligned}$$

Note that the same unbound, fresh variables Γ', x', e_1', e_2', T_1', and T_2' appear in the axioms and the goal, but not freely in any of the other rules. Since the variables in the axioms are not quantified, the inference engine may only apply axioms to terms derived from the goal (this condition is part of Assumption 3). Note also that we permit the usage of an extension's own inference rules $\overline{I}$ in the admissibility check. This enables extensions to desugar into recursive occurrences of the same extension without requiring the exact loop invariant as a premise of the inference rule.

Our PLT Redex implementation follows the verification procedure exactly, with the exception that instead of substituting fresh variables we "lock" metavariables in $(\texttt{symbol}\ x)$ nodes.

3.5 Metatheory

The goal of our verification procedure for extension soundness is to ensure that desugarings cannot generate code with type errors. The following is our main theorem:

Theorem 1 (Preservation)**.** *Let $ext\ \overline{I}\ \overline{R}$ be a sound extension with respect to B. If $B\ \overline{I} \succ (\Gamma \vdash e : T)$, $e \rightsquigarrow^*_{\overline{R}} e'$, and $e' \in term\ B$, then $B \succ (\Gamma \vdash e' : T)$.*

That is, given a derivation for $\Gamma \vdash e : T$ in the extended system, we apply rewriting steps to the program e until it is desugared into a term of the base language. We show that the resulting base term is well-typed in the base system B and satisfies the judgment $\Gamma \vdash e' : T$.

Before turning to the proof of Theorem 1, we first proof a few important lemmas about the rewrite engine, the inference engine, and sound extensions.

Lemma 1. *If $e_1[\overline{x}] \rightsquigarrow_{\overline{R}} e_2[\overline{x}]$, then $e_1[\,\overline{t}\,] \rightsquigarrow_{\overline{R}} e_2[\,\overline{t}\,]$.*

Proof. Straightforward by induction on the derivation $e_1 \rightsquigarrow_{\overline{R}} e_2$, using Assumption 1. □

Lemma 2. *Let $e[\overline{x}]$ be a base-language term and $e_1 = e[\,\overline{t_1}\,]$. If $e_1 \rightsquigarrow_{\overline{R}} e_2$, then $e_2 = e[\,\overline{t_2}\,]$ where $t_1 \rightsquigarrow_{\overline{R}} t_2$ for exactly one $(t_1, t_2) \in \overline{(t_1, t_2)}$ and $t_1 = t_2$ for all others.*

Proof. By induction on the derivation $e_1 \rightsquigarrow_{\overline{R}} e_2$, using that *rewrite* fails for terms that start with a base-language constructor (Assumption 2) and that $e \rightsquigarrow_{\overline{R}} e$ preserves base-language constructors. □

The next lemma is the crucial property, which states that a single rewrite step using a sound extension preserves types.

Lemma 3. *Let ext $\overline{I}$ $\overline{R}$ be a sound extension with respect to B. If $B\ \overline{I} \succ (\Gamma \vdash e : T)$ and $e \rightsquigarrow_{\overline{R}} e'$, then $B\ \overline{I} \succ (\Gamma \vdash e' : T)$.*

Proof. By induction on the derivation $B\ \overline{I} \succ (\Gamma \vdash e : T)$.

Base case: The derivation consists of the application of an axiom

$$I_0 = \forall \overline{x}.\ \Gamma_0[\overline{x}] \vdash e_0[\overline{x}] : T_0[\overline{x}].$$

We have $e = e_0[\,\overline{t}\,]$, $\Gamma = \Gamma_0[\,\overline{t}\,]$, and $T = T_0[\,\overline{t}\,]$ for some substitute $\overline{t}$ of $\overline{x}$.

If $I_0 \in \overline{I}$, extension soundness entails $e_0[\overline{x}] \rightsquigarrow_{\overline{R}} e_0'[\overline{x}]$ and $B\ \overline{I} \succ (\Gamma_0[\overline{y}] \vdash e_0'[\overline{y}] : T_0[\overline{y}])$ for fresh $\overline{y}$. By Lemma 1 and $e_0[\,\overline{t}\,] = e$ we get $e_0'[\,\overline{t}\,] = e'$. By Assumption 4 we can substitute $\overline{t}$ for $\overline{y}$ and get $B\ \overline{I} \succ (\Gamma \vdash e' : T)$ as required.

If $I_0 \in B$, then $e_0[\overline{x}]$ is a base language term. By Lemma 2 we have $e' = e_0[\,\overline{t'}\,]$. Accordingly, we get a derivation of $B\ \overline{I} \succ (\Gamma \vdash e' : T)$ by instantiating I_0 with $\overline{x} = \overline{t'}$.

Step case: Assume the last rule applied in the derivation is

$$I_0 = \forall \overline{x}.\ \overline{J_0[\overline{x}]} \Rightarrow \Gamma_0[\overline{x}] \vdash e_0[\overline{x}] : T_0[\overline{x}].$$

We have $e = e_0[\,\overline{t}\,]$, $\Gamma = \Gamma_0[\,\overline{t}\,]$, and $T = T_0[\,\overline{t}\,]$ for some substitute $\overline{t}$ of $\overline{x}$, and $B\ \overline{I} \succ J_0[\,\overline{t}\,]$ for all $J_0[\,\overline{t}\,] \in \overline{J_0[\,\overline{t}\,]}$ (Assumption 3).

If $I_0 \in \overline{I}$, extension soundness entails $e_0[\overline{x}] \rightsquigarrow_{\overline{R}} e_0'[\overline{x}]$ and $B\ \overline{I}\ \overline{J_0[\overline{y}]} \succ (\Gamma_0[\overline{y}] \vdash e_0'[\overline{y}] : T_0[\overline{y}])$. By Lemma 1 and $e_0[\,\overline{t}\,] = e$ we get $e_0'[\,\overline{t}\,] = e'$. By Assumption 4 we can substitute $\overline{t}$ for $\overline{y}$ and get $B\ \overline{I}\ \overline{J_0[\,\overline{t}\,]} \succ (\Gamma \vdash e' : T)$. Repeated application of the cut rule (Assumption 5) yields the required $B\ \overline{I} \succ (\Gamma \vdash e' : T)$.

If $I_0 \in B$, then $e_0[\overline{x}]$ is a base language term. By Lemma 2 we have $e' = e_0[\,\overline{t'}\,]$ and $t \rightsquigarrow_{\overline{R}} t'$ for one $(t, t') \in \overline{(t, t')}$ and $t = t'$ for all others. By the induction hypothesis we get $B\ \overline{I} \succ J_0[\,\overline{t'}\,]$ since the extended syntax t is irrelevant in contexts and types and can be replaced by t. Accordingly, we get a derivation of $B\ \overline{I} \succ (\Gamma \vdash e' : T)$ by instantiating I_0 with $\overline{x} = \overline{t'}$. □

We can scale type preservation to the reflexive, transitive closure of the rewrite relation $\rightsquigarrow_{\overline{R}}$:

Lemma 4. *Let ext $\overline{I}$ $\overline{R}$ be a sound extension with respect to B. If $B\ \overline{I} \succ (\Gamma \vdash e : T)$ and $e \rightsquigarrow^*_{\overline{R}} e'$, then $B\ \overline{I} \succ (\Gamma \vdash e' : T)$.*

Proof. Straightforward by induction on the derivation $e \rightsquigarrow^*_{\overline{R}} e'$, using Lemma 3. □

Finally, we can proof Theorem 1:

Proof of Theorem 1. By Lemma 4, we get $B\ \overline{I} \succ (\Gamma \vdash e' : T)$. Since we assume $e' \in term\ B$ and require rewrite rules to fail for pure base terms, the derivation tree of $\Gamma \vdash e' : T$ cannot contain an application of a rule from $\overline{I}$ because extension soundness requires a successful rewrite for a rule's subject. Accordingly, the derivation tree of $\Gamma \vdash e' : T$ only contains instantiations of rules from B, hence $B \succ (\Gamma \vdash e' : T)$ as required. □

In addition to our main preservation theorem, the verification procedure also ensures progress of desugarings:

Theorem 2 (Progress)**.** *Let ext $\overline{I}$ $\overline{R}$ be a sound extension with respect to B. If $B\ \overline{I} \succ (\Gamma \vdash e : T)$, then either $B \succ (\Gamma \vdash e : T)$ or there is an e' such that $e \rightsquigarrow_{\overline{R}} e'$.*

Proof. By induction on the derivation $B\ \overline{I} \succ (\Gamma \vdash e : T)$.

Base case: Either the axiom is from the base system B, or by Step 1 of extension soundness and Lemma 1 there is an e' with $e \rightsquigarrow_{\overline{R}} e'$.

Step case: If the last applied rule is from $\overline{I}$, Step 1 of extension soundness and Lemma 1 ensure there is an e' with $e \rightsquigarrow_{\overline{R}} e'$. Otherwise, assume the last applied rule is from the base system B. By the induction hypothesis all subderivations are either derivable in B or their subject can be desugared. If there is at least one subderivation of the latter kind, then the corresponding extended syntax must already be part of the current subject e because base rules cannot pattern match on extended syntax. Thus, by the definition of $\rightsquigarrow_{\overline{R}}$ the term e can be desugared. Otherwise all subderivations are derivable in B, and so is the current judgment. □

This concludes our correctness proof for the verification procedure of extension soundness employed by SoundExt. We have shown that our verification procedure guarantees well-typing of generated base-language programs.

3.6 Extension composition and overlapping definitions

As long as extensions are not syntactically overlapping, SoundExt supports *incremental extension* [6] (one extension desugars into code of another extension) and *extension unification* [6] (independent extensions can be unified into a single extension).

For incremental extension, we assume a sound extension $ext\ \overline{I_1}\ \overline{R_1}$ with respect to base system B. A second extension $ext\ \overline{I_2}\ \overline{R_2}$ can be defined on top of them by desugaring into the extended base system. We then require that $ext\ \overline{I_2}\ \overline{R_2}$ is a sound extension with respect to the extended base system $(B\ \overline{I_1})$. Since we desugar programs e from the double-extended language consecutively $e \rightsquigarrow^*_{\overline{R_2}} e' \rightsquigarrow^*_{\overline{R_1}} e''$, twice applying our soundness result from Theorem 1 proofs that the final result e'' does not contain type errors if the original program e is well-typed.

For extension unification, we assume two extensions $ext\ \overline{I_1}\ \overline{R_1}$ and $ext\ \overline{I_2}\ \overline{R_2}$ both of which are sound with respect to the base system B. We can compose these extensions into a unified extension $ext\ \overline{I_1 I_2}\ \overline{R_1 R_2}$. The unified extension is sound with respect to the base system B, because (i) for every type rule there is a successful rewriting and (ii) the proof of admissibility for a type rule remains valid in the unified extension. Accordingly, our verification procedure for extension soundness can be applied modularly to different extensions before composing them.

In case of a syntactic overlap, the composition of two sound extensions may be unsound if the type rule of one extension admits a term that is transformed by a desugaring of the other extension. To retain soundness, we have to cross-check all involved type rules for all potential desugarings, so that for any concrete rewriting the generated code is guaranteed to be free of type errors. This amounts to strengthening the contracts on desugarings by combining the expectations formulated in multiple type rules. If this cross-checking fails, the composition must fail. We have not implemented overlap detection in PLT Redex or SugarFomega and therefore our implementations cannot guarantee sound composition. As alternative to the detection of overlaps and the reverification of soundness at composition time, we could restrict the syntactic flexibility of extensions (as in traditional macro systems) to guarantee unambiguous syntax [21].

3.7 Summary

We achieve type-sound language extensibility by requiring *small-step desugarings* that are applicable to the subject of corresponding type rules. This way, we can derive type rules that exactly admit those programs that can be generated from well-typed programs. By showing that these derived rules are admissible, we ensure that the generated programs are well-typed themselves and do not require further type checking. In particular, we have shown that extension soundness entails that desugarings preserve types and never get stuck.

4. SugarFomega

We have implemented a sound syntactically extensible programming language SugarFomega[3] by combining the base language System F_ω, with SugarJ-like syntactic extensibility, and SoundExt.

SugarJ [5, 8, 9] enables the syntactic extension of a base language like Java with arbitrary new syntax, given programs of that syntax can be desugared into the base language. SugarJ (and SugarFomega) employ the SDF2 syntax formalism [25] for definition of extended syntax and the Stratego rewrite language [27] for the definition of desugaring rules. SugarJ organizes language extensions in modules of the base language itself, so that regular module-import statements activate extensions locally by bringing the extended grammar and desugarings into the scope of the current module.

SugarJ also provides support for program analyses that annotate the input program with metadata. This information is used in error messages of the compiler as well as in our Eclipse-based extensible IDE [7], for example, to show type information in hover help. We employ the support for analyses to implement the F_ω type checker and the verification procedure for sound extensions.

4.1 The base language System F_ω

SugarFomega is based on System F_ω and primarily augments it with a simple module system that is amenable to SugarJ's module-based extension mechanism. We also add a few practical features: pair kinds, type synonyms, records, variants, higher-order iso-recursive types, and some primitive types. With the exception of higher-order recursive types and the module system, SugarFomega is fairly standard and all its components can be found in textbooks [19].

Modules and signatures. A SugarFomega module may import other modules and contains a sequence of value or type definitions that may be public. For example, we define a type synonym for polymorphic pairs, a swap function for pairs, and a private test expression that is not exported:

```
module Data.Pair
import Foo
public type Pair = \A::*. \B::*. {fst:A, snd:B}
public val  swap = \A::*. \B::*. \p:Pair A B.
                    {fst=p!snd, snd=p!fst}
val test = swap [Nat] [Bool] {fst=1, snd=false}
```

[3] Implementation available online: `http://sugarj.org/fomega`

Kind judgments `~C |- ~T :: ~K`

```
~C |- ~T1 :: (~K => *) => ~K => *
~C |- ~T2 :: ~K
================================= K-Mu
~C |- mu (~T1, ~T2) :: *
```

Type judgments `~C |- ~e : ~T`

```
~C |- ~T :: *
~C |- ~T ~> mu (~T1, ~T2)
~C |- ~e : ~S
~C |- ~S ~> mu (~T1, ~T2)
~C |- ~T2 :: ~K
================================== T-Unfold
~C |- unfold [~T] ~e :
      ~T1 (\A::~K. mu (~T1, A)) ~T2
```

Figure 5. Kind and type rule for higher-order iso-recursive types.

The definition of a module is valid if a signature is derivable, that is, if each type (value) definition is well-kinded (well-typed). The initial context for the derivation of a signature is build from the signatures of imported modules.

Higher-order recursive types. SugarFomega implements a form of higher-order iso-recursive types similar to Crary and Weirich [3]. Figure 5 shows the kind rule for our recursive type operator `mu` and the type rule for `unfold`, which performs a one-step unfolding of a recursive type. A recursive type is a pair `mu` (T_1, T_2) of types where T_2 is of kind K and turns the higher-kinded recursive type operator T_1 into a proper type.

For example, in SugarFomega, we can define a polymorphic list with the following type synonyms:

```
type LRec = \List::* => *. \A::*.
  <Nil: {}, Cons: {hd:A, tl:List A}>
type List = \A::*. mu (LRec, A)
```

The isomorphism between, for example, `List Nat` and its one-step unfolding is witnessed by `fold` and `unfold` according to their type rules:

$$\begin{tikzcd} \texttt{mu (LRec, Nat)} \arrow[d, shift right, "\texttt{unfold [List Nat]}"'] \\ \texttt{<Nil: \{\}, Cons: \{hd:Nat, tl:mu (LRec, Nat)\}>} \arrow[u, shift right, "\texttt{fold [List Nat]}"'] \end{tikzcd}$$

4.2 The SugarFomega type checker

To define the type system of F_ω, we implemented a domain-specific language for the declaration of inference rules:

$Premise_1$
...
$Premise_n$
========== $RuleName$
$Conclusion$

Our inference rules are layout-sensitive so that premises and conclusions can span multiple lines, given they adhere to the offside rule.

The conclusion of a rule as well as each premise is a judgment. Whereas our formalization of SoundExt in Section 3 was limited to type judgments, for System F_ω we generalized SoundExt to support multiple kinds of judgments. The most important judgments are:

Γ	|-	*defs*	==>	σ	Signature judgments
Γ	|-	e	:	T	Type judgments
Γ	|-	T	::	K	Kind judgments
Γ	|-	T	~>	T'	Normalization judgments

For example, the kind and type rule for recursive types in Figure 5 are implemented in our language for inference rules, where metavariables are prefixed by a tilde ~. In fact, we implemented most of the F_ω type system through the declaration of inference rules.

We compile the inference rules to executable Stratego code that is called during the analysis phase of SugarFomega. The compiler has hard-wired knowledge about how to compile the different judgments. For example, our compiler makes the following common assumptions about input and output positions of judgments:

Input		Input		Output
Γ	|-	*defs*	==>	σ
Γ	|-	e	:	T
Γ	|-	T	::	K
Γ	|-	T	~>	T'

The code generated from different inference rules has a similar structure: First, we check that the rule's input matches the input expected by the conclusion. Second, we check the premises in order of appearance. Third, we compute the conclusion's output. For signature, kind, and type judgments, we store the output of a conclusion as annotations of the corresponding subject, so that we can reuse the signature, kind, or type during desugaring and in editor services such as hover help. Should an error occur, we annote them to the defective term instead of annotating a signature, kind, or type. Furthermore, note that the order of premises is significant: Before a metavariable can be used as input of a premise or as output of the conclusion, it must be introduced in an output position of a premise or as input of the conclusion.

Our compiler translates each rule into a certain Stratego strategy. For example, the compiler translates kind rules into definitions of `annotate-kind` and type rules translate into definitions of `annotate-type`. We eagerly normalize types and use Stratego's support for backtracking to explore all possible derivations (a compiler that produces *efficient* code for finding derivations is orthogonal to the work presented in this paper and left for future work).

We illustrate the compilation of inference rules by example of the SugarFomega type rule for *let* expressions (~ marks a metavariable, ~% marks a metavariable only for identifiers):

```
~C |-  ~e1  : ~T1                 // (P1)
~C |-  ~S  ::  *                  // (P2)
~C |-  ~S  ~> ~U                  // (P3)
~C |-  ~T1 ~> ~U                  // (P4)
(~C;~%x:~S) |-  ~e2 : ~T2         // (P5)
==================================== T-Let
~C |- (let ~%x : ~S = ~e1 in ~e2) : ~T2
```

Figure 6 presents a simplified version of the Stratego code generated for `T-Let`:

- Line 2 matches the input of the conclusion of rule `T-Let` using abstract syntax. Only if the input is a *let* expression, the type derivation continues with an instantiation of rule `T-Let`.
- Lines 3 and 4 implement premise `(P1)`. The type of `e1` is computed and annotated by a recursive call. We require the successful typing of `e1` by retrieving its type into variable `T1` using `get-type`.

```
1 annotate-type =
2     ?(C, node@Let(x, S, e1, e2))
3   ; <annotate-type> (C, e1)
4   ; <get-type> e1 => T1
5   ; <annotate-kind> (C, S)
6   ; <get-kind> S => tmp1
7   ; <kind-eq-star> tmp1 => msgs1
8   ; <add-context-errors> (msgs1, S)
9   ; <norm> (C, S) => U
10  ; <norm> (C, T1) => tmp2
11  ; <type-eq> (U, tmp2) => msgs2
12  ; <add-context-errors> (msgs2, T1)
13  ; <annotate-type> (CtxBindVar(C, x, S), e2)
14  ; <get-type> e2 => T2
15  ; <put-type> (T2, node)
```

Figure 6. Stratego code generated for `T-Let`.

- Lines 5 to 8 implement premise `(P2)`. The kind of `S` is computed, annotated, and retrieved into a temporary variable `tmp1`. `(P2)` demands `S` to be of kind `*`, which we check using `kind-eq-star` in line 7. `kind-eq-star` returns a possibly empty list of error messages that we annotate to `S` in line 8.
- Lines 9 to 12 handle premises `(P3)` and `(P4)`. The normalization judgment of `(P3)` compiles into a call of the type-normalization strategy `norm`. During translation of `(P4)`, the compiler recognizes that the output `U` appears as output of a preceding judgment, namely `(P3)`. Therefore, the normal form of `T1` is stored in a temporary variable `tmp2`. The call of `type-eq` in line 11 checks that the types bound to `U` and `tmp2` indeed are equal (up to renaming of bound variables). `type-eq` returns a possibly empty list of error messages that we annotate to type `T1`.
- Lines 13 and 14 implement `(P5)`, similar to the translation of `(P1)` but with an enriched context that contains the binding `~%x:~S` represented by the term `CtxBindVar(C, x, S)`.
- Finally, line 15 annotates the resulting type to the *let* expression.

We use our domain-specific language for inference rules to implement the whole type system of System F_ω: kinding, typing, type normalization, and deriving module signatures. We provide SugarFomega programmers with the same language to define inference rules for language extensions.

4.3 Writing language extensions in SugarFomega

SugarFomega allows users to define language extensions in regular base-language modules. A SugarFomega extension takes the following form:

```
module Name
syntax { SDF2 definitions }
desugaring Strategy name { Stratego rewrite rules }
typing { Typing rules }
```

That is, a SugarFomega extension consists of an extended syntax, desugaring transformations from the extended syntax into the (possibly further extended) base language F_ω, and type rules for programs of the extended syntax. For example, Figure 7 displays the definition of a SugarFomega extension for *let* expressions.

- The **`syntax`** section adds a new production to the nonterminal `Expr` that is defined in the base grammar of SugarFomega. The extension declares that a *let* expression can be used whenever a `Expr` is expected. The annotation `{cons("Let")}` declares that a *let* expression is represented by an abstract syntax tree node of name `Let`.

```
module extensions.Let

syntax {
  context-free syntax
    "let" ID ":" Type "=" Expr "in" Expr
      -> Expr {cons("Let")}
}

desugaring desugar-let {
  desugar-let :
    |[ let ~%x : ~T = ~e1 in ~e2 ]| ->
      |[ (\ ~%x : ~T. ~e2) ~e1 ]|
}

typing {
  ~C |-  ~e1  : ~T1
  ~C |-  ~S  ::  *
  ~C |-  ~S  ~> ~U
  ~C |-  ~T1 ~> ~U
  (~C; ~%x:~S) |-  ~e2 : ~T2
  ====================================== T-Let
  ~C |- (let ~%x : ~S = ~e1 in ~e2) : ~T2
}
```

Figure 7. Declaration of a type-sound extension for *let* expressions.

```
1  verify-context-rule =
2    ?TypingRule(premises, TypingJudgment(C, e, T))
3    if <desugar-one-step> e => e-des then
4      with-scoped-axioms(
5          <activate-dynamic-axioms> premises
6        ; <annotate-type> (C, e-des)
7        ; <get-type <+ !TyUnknown> e-des => U
8        ; <norm> (C, T) => T'
9        ; <norm> (C, U) => U'
10     )
11     ; <collect-all-errors> e-des => errs
12     ; if <not(is-empty)> errs then
13         !errs
14       else if <not(type-eq)> (T', U') then
15         !["Type mismatch"]
16       else
17         ![]
18       end end
19   else
20     !["Could not desugar conclusion"]
21   end
```

Figure 8. SugarFomega verification procedure for type rules.

- The `desugaring` section defines a Stratego rule `desugar-let` that implements the necessary rewriting. We use concrete syntax [26] for SugarFomega code in the declaration of rewrite rules. Like in type rules, a tilde ~ marks a metavariable. Often, it is useful to decompose a desugaring into multiple strategies. The identifier following the keyword `desugaring` names the entry point of desugaring that is called by the SugarFomega compiler.
- The `typing` section defines a type rule for *let* expressions as discussed in the previous subsections.

Since the language extension for *let* expressions is represented by a regular SugarFomega module, we can activate the extension by importing the module. The SugarFomega system then loads the grammar extensions, applies the desugarings to parsed code, and performs static analysis according to predefined and user-defined inference rules.

```
module test.Let
import extensions.Let

val test = let p : {x:Nat, y:Nat} = {x=1, y=2}
           in  p!x
```

4.4 Verifying type-soundness of extensions

SugarFomega verifies the soundness of each language extension using a modified version of SoundExt (Section 3) that supports all judgments required in the type system of System F_ω and knows about the input and output positions of judgments. Moreover, due to type-level abstraction in System F_ω, we require equivalent types everywhere we required syntactically equal types before. As usual, we check type equivalence by comparing the normal forms of types. Specifically, we modified the verification procedure for SugarFomega as follows:

- *Type rule*: Rewrite expression and assert equality of the normal forms of the expect and the actual type.
- *Kind rule*: Rewrite type and assert equality of the expect and the actual kind.
- *Signature rule*: Rewrite definitions and assert equality of the expected and the actual signature.
- *Type-normalization rule*: Rewrite input type and assert equality of the normal forms of the input and the output type.

We have not yet formally proofed that these criterions entail type preservation. But we are very confident that they indeed do since we can in principle encode the different judgments into SoundExt's judgment form by auxiliary constructors.

Like in Section 3, we use the inference engine to verify the soundness of user-defined inference rules. Since we compile inference rules to Stratego, we use Stratego's support for dynamically scoped rewrite rules [2] to activate axioms dynamically. For example, the strategy `activate-dynamic-axiom` activates an axiom for a type judgment:

```
activate-dynamic-axiom =
    ?TypingJudgment(C, e, T)
  ; rules(Dynamic-Annotate-Type :+
            (C, e) -> <put-type> (T, e))
```

Given a type judgment, `activate-dynamic-axiom` extends the definition of `Dynamic-Annotate-Type` for the input `(C, e)`. For example, for the premise like `~C |- ~e1 : ~T1` from `T-Let`, `activate-dynamic-axiom` activates the following rule:

```
Dynamic-Annotate-Type :
  (C@Metavar("C"), e@Metavar("e1"))
  -> <put-type> (Metavar("T1"), e)
```

As required, this rule only accepts the specific term `Metavar("e1")`, and not any other expression. We hook `Dynamic-Annotate-Type` into the regular type checking routine `annotate-type` by extending it as follows:

```
annotate-type = Dynamic-Annotate-Type
```

That is, whenever a type judgment is checked, the type checker will consider `Dynamic-Annotate-Type`, too.

We show the Stratego code for verifying the soundness of a type rule in Figure 8. This implementation corresponds to the verification procedure of SoundExt with the above-mentioned modifications. In SugarFomega, the verification procedure acts as a static analysis for checking modules that define language extensions.

- Strategy `verify-context-rule` gets a typing rule as input and produces a possibly empty list of error messages (lines 1 and 2).
- In line 3, `verify-context-rule` tries to desugar the expression `e` of the conclusion one step. If this fails, an error message is emitted and the verification fails (line 20).

- If the desugaring is successful, line 5 enables the premises of the type rule as additional axioms. To prevent that axioms are active outside the verification procedure, we scope them using the strategy `with-scoped-axioms` (line 4).
- While the additional axioms are active, we type check the desugared expression `des-e` (line 6). We retrieve the annotated type in line 7 or, if type checking fails, use `TyUnknown` instead.
- To make the expected and the actual type comparable, we normalize them in lines 8 and 9.
- Afterwards, we collect all errors that have occurred during type checking of `des-e` (line 11). If we find some errors, we return these errors and verification fails.
- Otherwise, we compare the normalized expected type `T'` and the normalized actual type `U'` (line 14). If they are not equal, we return an error message and verification fails.
- Otherwise, no errors are emitted and the verification succeeds.

We have implemented similar verification procedures for the other rules used in SugarFomega, in particular, kinding, type normalization, and deriving module signatures. As we demonstrate in the following section, SugarFomega permits users to integrate even sophisticated language features as language extensions, for which SugarFomega modularly, statically, and automatically verifies type-soundness.

5. Case studies

In the previous sections, we used *let* expressions as an exemplary language extensions that is easy to define and that SoundExt was able to prove sound. In the present section, we further demonstrate the applicability of SoundExt by extending SugarFomega with language extensions for monadic *do* blocks and algebraic data types.

5.1 Monadic *do* blocks

Monads enable programmers to abstract over different computational flavors, such as state-full computation or non-deterministic computation [17]. Due to this abstraction, it is possible to define combinators that are independent of concrete computational flavors. For example, the function `liftM2` lifts any pure function of type `A -> B -> C` into a monadic function `M A -> M B -> M C` for any monad `M`.

In SugarFomega, we can encode a monad as a type operator $M :: * \Rightarrow *$ with associated bind and return functions:

$$\begin{array}{lll} bind & : & \forall A :: *. \forall B :: *. M\ A \to (A \to M\ B) \to M\ B \\ return & : & \forall A :: *. A \to M\ A \end{array}$$

To support convenient programming with monads, we have developed a language extension of SugarFomega that introduces *do* notation similar to Haskell. We provide type rules for *do* blocks that detect type errors prior to desugaring. SugarFomega verifies the soundness of our extension to guarantee no type errors occur in desugared code.

Here is a simple *do* block as supported by our extension of SugarFomega:

```
do [T] {
    x:T1 <- e1
  ; y:T2 <- e2
  ; e3 }
```

A *do* block starts with a keyword `do`, followed by the result type of the computation that the *do* block represents. Inside a *do* block occurs a semicolon-separated list of computational statements, each of which may bind their result to a local variable. The last statement of a *do* block must be an expression that returns a value

```
desugaring desugar-do {
  desugar-do :
  |[ do [~T] { ~e } ]| ->
    |[ \M::*=>*.
       \bind: forall A::*. forall B::*.
              M A -> (A -> M B) -> M B.
       \return: forall A::*. A -> M A. ~e ]|

  desugar-do :
  |[ do [~T] { ~%x : ~S <- ~e1; ~stmts } ]| ->
    |[ \M::*=>*.
       \bind: forall A::*. forall B::*.
              M A -> (A -> M B) -> M B.
       \return: forall A::*. A -> M A.
       bind [~S] [~T] ~e1
         (\~%x:~S. do [~T] { ~stmts } [M] bind return) ]|
}
```

Figure 9. Desugaring rules for *do* blocks.

corresponding to the result type of the *do* block. Note that the type annotations in our encoding of *do* blocks are not strictly necessary; in Section 6 we discuss how to drop them while retaining automatically verifiable type-soundness.

As shown in Figure 9, we desugar a *do* block into an expression that is parameterized over a monad, that is, over a type operator `M` and functions `bind` and `return`. The first desugaring rule handles *do* blocks that only contain a final expression, whereas the second desugaring rule handles *do* blocks with multiple statements. In contrast to *let* expressions, the desugaring of *do* blocks requires recursion. We trigger recursion in the second desugaring rule by generating code that contains a residual *do* block, to which we propagate `M`, `bind`, and `return`.

SugarFomega only allows programmers to write programs that can be type checked before desugaring. Accordingly, we require type rules for *do* blocks. In Figure 10, we display the type rule for *do* blocks with multiple statements. The type of the conclusion clearly demonstrates our choice to implement *do* blocks as polymorphic functions. The premises `(P3)` and `(P7)` express the essential context conditions of *do* blocks:

- The right-hand side `~e1` of a monadic binding must be well-typed in a context that provides `bind` and `return`.
- The subsequent statements `~stmts` must be well-typed in the same context enriched by the bound variable `~%x`.

SugarFomega automatically verifies that these requirements are sufficient to guarantee that the desugared code is well-typed. For example, this allows us to define `liftM2` using *do* notation:

```
module MonadLift
import extensions.DoBlock
val liftM2 = \M::*=>*.
  \bind: forall A::*. forall B::*.
         M A -> (A -> M B) -> M B.
  \return: forall A::*. A -> M A.
  \A::*. \B::*. \C::*.
  \f: A -> B -> C. \m1: M A. \m2: M B.
    do [C] {
        x:A <- m1
      ; y:B <- m2
      ; return [C] (f x y)
    } [M] bind return
```

5.2 Algebraic data types

The type system of System F_ω is very flexible and expressive, but the encoding of tree-like data structures in terms of variants, records, and recursive types is cumbersome. To this end, we use SugarFomega

```
(~C; M::*=>*) |-  ~S :: *                                                           // (P1)
(~C; M::*=>*) |-  ~T :: *                                                           // (P2)
(~C; M::*=>*; bind: forall A::*. forall B::*. M A -> (A -> M B) -> M B              // (P3)
            ; return: forall A::*. A -> M A)   |-   ~e1 : ~R1
(~C; M::*=>*) |-  ~R1 ~> M ~U1                                                      // (P4)
(~C; M::*=>*) |-  ~U1 :: *                                                          // (P5)
(~C; M::*=>*) |-  ~S ~> ~U1                                                         // (P6)
(~C; M::*=>*; bind: forall A::*. forall B::*. M A -> (A -> M B) -> M B              // (P7)
            ; return: forall A::*. A -> M A; ~%x:~S)   |-   do [~T] { ~stmts } :
    forall M::*=>*. (forall A::*. forall B::*. M A -> (A -> M B) -> M B) -> (forall A::*. A -> M A) -> M ~T
======================================================================================= T-DoCons
~C |- do [~T] { ~%x : ~S <- ~e1; ~stmts } :
  forall M::*=>*. (forall A::*. forall B::*. M A -> (A -> M B) -> M B) -> (forall A::*. A -> M A) -> M ~T
```

Figure 10. Typing rule for a *do* block with multiple statements.

to define a type-sound extension for algebraic data types like the following definition for polymorphic lists:

```
data List (A::*) = Nil {}
                 | Cons {hd: A, tl: List A}
```

This definition is desugared into a type synonym `List` and constructor functions `Nil` and `Cons` that type-normalize to the following definitions:

```
type List = \A::*. mu (\List::*=>*. \A::*.
  <Nil:{}, Cons:{hd:A, tl:List A}>, A)

val Nil = \A::*. fold [List A]
  (<Nil={}> as <Nil:{}, Cons:{hd:A, tl:List A}>)

val Cons = \A::*. \hd:A. \tl:List A. fold [List A]
  (<Cons={hd=hd, tl=tl}> as
     <Nil:{}, Cons:{hd:A, tl:List A}>)
```

To achieve the desugaring of algebraic data types, we need to exercise two features of SugarFomega not discussed in detail before: *abstract intermediate program representations* for which no concrete syntax exists and *type-level language extensions*.

Abstract intermediate program representations. Since the verification procedure of SoundExt and SugarFomega require a rewriting to apply to the subject of a type rule, language extensions must be desugared in small-step fashion. For example, for *do* blocks the second desugaring created a residual *do* block, which is transformed in a later iteration of the fix-point desugaring. In a sense, we got lucky for *do* blocks, because the residual program could be described by concrete syntax.

More generally, and for algebraic data types in particular, it is not always possible to describe residual programs via concrete syntax (which users would be allowed to write, too). For this reason, SugarFomega allows desugaring transformations and inference rules to use abstract intermediate program representations instead of concrete syntax. Note that this was not a problem for SoundExt because we only used abstract syntax anyway.

An extension has to declare the abstract-syntax nodes it wants to use as part of the desugaring:

```
desugaring desugar-adt {
  signature constructors
    ADT-DCON:        ID * DataParams *
                     ID * Decls * DataCons -> Term
    ADT-DCON-FIELDS: Decls -> Term
    ADT-TABS:        DataParams * Expr -> Term
  ...
}
```

Afterwards, the extension can use these nodes in place of concrete syntax by prefixing them with `M~`. For example, we use an abstract intermediate node `ADT-TABS` to prefix the code generated for a constructor with the type parameters of the algebraic data type (a nested occurrence of `|[ ]|` escapes back to concrete syntax):

```
desugar-data :
  |[ M~ADT-TABS(|[]|, e) |]
  -> |[ ~e ]|

desugar-data :
  |[ M~ADT-TABS( |[ (~%X::~K) ~params ]| , e) ]|
  -> |[ \~%X::~K. M~ADT-TABS(params, e) ]|

~C |- ~e : ~T
===================================== T-ADT-TABS1
~C |- M~ADT-TABS(|[]|, e) : ~T

(~C; ~%X::~K) |- M~ADT-TABS(params, e) : ~T
========================================== T-ADT-TABS2
~C |- M~ADT-TABS(|[ (~%X::~K) ~params ]|, e)
   :  forall ~%X::~K. ~T
```

In our case study for algebraic data types, we exclusively represent intermediate data-type fragments via abstract syntax. To this end, we employ 19 different abstract-syntax nodes for the generation of the type synonym and the constructor functions.

Type-level and kind-level language extensions. As described in Section 4.2 and 4.4, SugarFomega supports syntactic sugar at all levels: kinds, types, and expressions. However, an extension is only usable if there are inference rules that can be employed for the validation of a program using the extended syntax. For expression-level extensions, we have shown for *let* expressions and *do* blocks that type rules can provide the necessary validation. For type-level and kind-level extensions, other means are necessary.

For type-level extensions, there are two possibilities for the specification of valid usage patterns of extended types: using kind rules only or also adding type-normalization rules. These alternatives have an interesting consequence when using an abstract intermediate representation for extended types: The extended types can be introduced nominally or structurally. For example, consider the following kind rule that specifies the kind of the residual type `ADT-ABS`.

```
(~C;~%X::~K1) |- M~ADT-ABS(params, T) :: ~K2
========================================== K-ADT-ABS2
~C |- M~ADT-ABS(|[ (~%X::~K1) ~params ]|, T)
   :: ~K1 => ~K2
```

Since SugarFomega performs type checking prior to desugaring, this kind rule only applies to residual types `ADT-ABS`, and nothing else. Compare this with the following type-normalization rule that specifies to what the residual type `ADT-ABS` normalizes:

```
~C |- M~ADT-ABS(params, T) ~> ~T2
========================================= N-ADT-ABS2
```

```
~C |- M~ADT-ABS(|[ (~%X::~K) ~params ]|, T)
   ~> \~%X::~K. ~T2
```

Since type normalization happens every time two types are compared, this type-normalization rule permits the usage of ADT-ABS wherever a type `\~X::~K. ~T` is expected.

For our case study of algebraic data types, the difference is whether values of the generated type synonym can be defined only with the generated constructors (nominal) or with any expression of the right type (structural). Note that kind rules are required in most cases anyway, because during verification of type rules with kind-judgment premises, there is typically not enough information on types to perform type normalization.

For kind-level extensions, there is no such choice because there are no explicit sorts or super-kinds in SugarFomega. However, we added context-free kind normalization to permit the generation of kinds depending on the input program. For example, we use the following kind-normalization rule to permit the usage of the residual kind ADT-CK as a higher-order kind (used to describe the type arguments of an algebraic data-type constructor such as List):

```
M~ADT-CK(params) =::=> ~K2
================================ KN-ADT-PK2
M~ADT-CK(|[ (~%X::~K) ~params ]|)
=::=> ~K => ~K2
```

In our SugarFomega implementation of algebraic data types, we use kind rules, type-normalization rules, and kind-normalization rules. However, we found that our normalization rules always resemble the rewriting of the desugaring rules. As consequence, the verification is trivial because the desugaring of the normalization input yields the normalization output. Moreover, this is obvious boilerplate that we hope to remedy in a future version of our domain-specific language for inference engines.

Summary. Algebraic data types are by far the most sophisticated language extension we have implemented in SugarFomega. The whole extension comprises 19 abstract-syntax nodes, 34 desugaring rules, 4 module-signature derivation rules, 7 type rules, 13 kind rules, 21 type-normalization rules, and 4 kind-normalization rules.

6. Type reconstruction

As briefly mentioned before, quite a few of the type annotations we currently require are not necessary, because the defining expressions of the involved variables are statically available. For example, the language extension for *let* expressions (Figure 7) requires a type annotation for the bound variable, because such annotation is required in the desugared lambda abstraction. To reduce the number of required type annotations, we started to experiment with a *type-of* metalanguage construct that permits the reuse of an expression's type. For example, with the *type-of* construct, we can support *let* expressions without type annotations:

```
desugaring desugar-let {
  desugar-let :
    |[ let ~%x = ~e1 in ~e2 ]| ->
      |[ (\~%x:$(typeof ~e1). ~e2) ~e1 ]|
}
```

This desugaring is only applicable if `~e1` is well-typed, in which case the type of `~e1` is copied into the generated program. We are currently evaluating different alternatives for implementing the *type-of* construct, such that *type-of* is supported during the verification of extensions and while type checking user programs. Moreover, we are currently investigating how to support *type-of* constructs that are transitively generated, for example, by another extension that produces *let* expressions.

7. Related work

Most approaches for language extensibility have in common that type checking is only performed on the full expansion of the extended syntax, that is, after desugaring. There is no validation of transformation rules to ensure they produce well-typed code of the required type. Even in Typed Scheme [24], a gradually typed Scheme dialect, macros are untyped and type checking operates on fully desugared core syntax. While some systems [4, 22] require the transformation itself to be a well-typed function, the employed program representations are not typed enough to make sufficient statements about the typing properties of the generated code. On the other hand, existing highly typed program representations as known, for example, from dependently typed languages such as Agda or from similar encodings in Haskell, have not yet been adapted by approaches for language extensibility.

Herman's λ_m-calculus [14] augments a Scheme-like macro system with signatures for macro definitions that describe their binding structure. These signatures allow to verify if a macro generates code that matches the signature. This, for example, means that an argument which is bound by the macro must be used at a binding position in the generated code. Macro applications are validated along the same lines. The λ_m-calculus not only enables the definition of α-equivalence for unexpanded Scheme programs, but moreover binding signatures improve the robustness of macros a lot. They are, however, restricted to static scoping aspect of macros, and do not scale to the static typing of terms.

MacroML [13] is an extension of the ML programming language with the possibility to define generative and binding macros in a statically typed setting. The semantics of a MacroML program is defined by a translation into the multi-stage programming language Meta-ML [23] and a type preservation result of this translation is proved. In contrast to MacroML, SoundExt allows desugarings to inspect and decompose user programs using pattern matching. Furthermore, MacroML macros that introduce new binding constructs have to be variations of predefined binding structures like lambda abstraction to unambiguously identify binding and bound occurrences of variables. In particular, MacroML macros must receive the variables they bind as arguments. In contrast, SoundExt allows type rules to freely specify extension-specific scoping of identifiers via the context in judgments. For example, a simple SugarFomega extension `openpair` could bring the components of a pair into the current scope by binding variables `fst` and `snd`. The corresponding type rule would clearly document the bound variables `fst` and `snd` and their scope. Such macro could not be defined in MacroML.

Ziggurat [11, 12] is a metalanguage to extend other programming languages through Scheme-like macros and to attach static analyses like type checking or termination analysis to the new syntax. In Ziggurat, nodes of the syntax tree are represented as objects with a parsing and a rewrite function in an an object-oriented system called lazy delegation. Lazy delegation permits the implementation of static analyses explicitly for a node by implementing the corresponding method, or implicitly by delegating analysis to the objects generated during desugaring. However, static analyses themselves are not validated against desugarings and there is also the danger that a malicious analysis of a subclass overrides a sound analysis of a super class.

Pluggable type systems [1] extend an existing type systems with additional constraints. Pluggable type systems may reject programs that, for example, do not comply to a certain design pattern or introduce new types, for example a non-null type in Java to check that `null` references are never dereferenced. Implementations of pluggable type systems like JavaCOP [16] or the Checker Framework [18] provide infrastructure to implement type analyses and to integrate them into the semantic analysis phase of the compiler. The soundness of a pluggable type system is not verified mechanically, this is the implementer's responsibility. But JavaCOP has another interesting approach with respect to soundness: testing for soundness violations. Their test harness supports the instrumentation of Java byte code with runtime checks to detect "stuck expressions" that the pluggable type system is supposed to prevent statically.

Roberson et al. [20] propose model checking to automatically proof the soundness of a type system for all program states of at most some finite size. Their technique is similar to ours: Identify well-typed intermediate programs, perform one step of reduction, and proof that the resulting program is also well-typed. However, the most important difference is that Roberson et al. inspect concrete program states and apply sophisticated pruning techniques to avoid state-space explosion, whereas SOUNDEXT performs symbolic rewriting and reasoning. Moreover, Roberson et al. tackle the more general problem of proofing the soundness of an entire type system (for a bounded size of the program state). Accordingly, their reduction relation is a small-step operation semantics, whereas SOUNDEXT uses a small-step program transformation.

8. Conclusion and future work

We presented SOUNDEXT, a formalism for syntactically extending a base language without affecting type soundness. In particular, SOUNDEXT verifies language extension to guarantee that desugarings adhere to type preservation and progress. Accordingly, code generated from desugarings cannot contain type errors, which would break important abstraction barriers for programmers.

We applied SOUNDEXT to the advanced base language System F_ω, resulting in SugarFomega, and successfully added nontrivial extensions for *do* blocks and algebraic data types in a type-sound way. In future work, we hope to use SOUNDEXT to validate the desugarings and type rules of language extensions currently in use in compilers such as GHC, which will require solid support for type reconstruction and the generation of more efficient code from inference rules.

References

[1] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004. Available at `http://bracha.org/pluggableTypesPosition.pdf`, accessed at Mar. 26 2013.

[2] M. Bravenboer, A. v. Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1-2):123–178, 2006.

[3] K. Crary and S. Weirich. Flexible type analysis. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 233–248. ACM, 1999.

[4] D. de Rauglaudre. Camlp4 reference manual. `http://caml.inria.fr/pub/docs/manual-camlp4/index.html`, accessed Mar. 26 2013., 2003.

[5] S. Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universiät Marburg, 2013.

[6] 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.

[7] S. Erdweg, L. C. L. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. Growing a language environment with editor libraries. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 167–176. ACM, 2011.

[8] 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.

[9] 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.

[10] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[11] D. Fisher and O. Shivers. Static analysis for syntax objects. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 111–121. ACM, 2006.

[12] D. Fisher and O. Shivers. Building language towers with ziggurat. *Functional Programming*, 18(5-6):707–780, 2008.

[13] S. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proceedings of International Conference on Functional Programming (ICFP)*. ACM, 2001.

[14] D. Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, Massachusetts, 2012.

[15] J. Łoś and R. Suszko. Remarks on sentential logics. *Indagationes Mathematicae*, 20:177–183, 1958.

[16] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble. JavaCOP: Declarative pluggable types for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):4:1–4:37, 2010.

[17] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[18] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 201–212. ACM, 2008.

[19] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[20] M. Roberson, M. Harries, P. T. Darga, and C. Boyapati. Efficient software model checking of soundness of type systems. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 493–504. ACM, 2008.

[21] A. Schwerdfeger and E. Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 199–210. ACM, 2009.

[22] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proceedings of Haskell Workshop*, pages 1–16. ACM, 2002.

[23] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[24] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 395–406. ACM, 2008.

[25] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[26] E. Visser. Meta-programming with concrete object syntax. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, volume 2487 of *LNCS*, pages 299–315. Springer, 2002.

[27] E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 13–26. ACM, 1998.

[28] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.