Sound Type-Dependent Syntactic Language Extension

Florian Lorenzen
TU Berlin, Germany

Sebastian Erdweg
TU Darmstadt, Germany

Abstract

Syntactic language extensions can introduce new facilities into a programming language while requiring little implementation effort and modest changes to the compiler. It is typical to desugar language extensions in a distinguished compiler phase after parsing or type checking, not affecting any of the later compiler phases. If desugaring happens before type checking, the desugaring cannot depend on typing information and type errors are reported in terms of the generated code. If desugaring happens after type checking, the code generated by the desugaring is not type checked and may introduce vulnerabilities. Both options are undesirable.

We propose a system for syntactic extensibility where desugaring happens after type checking and desugarings are guaranteed to only generate well-typed code. A major novelty of our work is that desugarings operate on typing derivations instead of plain syntax trees. This provides desugarings access to typing information and forms the basis for the soundness guarantee we provide, namely that a desugaring generates a *valid* typing derivation. We have implemented our system for syntactic extensibility in a language-independent fashion and instantiated it for a substantial subset of Java, including generics and inheritance. We provide a sound Java extension for Scala-like for-comprehensions.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]; I.2.2 [*Automatic Programming*]: Program transformation; D.3.2 [*Language Classifications*]: Extensible languages

Keywords Language extensibility, type soundness, type-dependent desugaring, automatic verification, metaprogramming, macros

1. Introduction

Syntactic language extensions provide essential abstractions for managing the complexity of software. Numerous examples of embedded domain-specific languages like Hydra [12], XML in Scala [22], or PLT Redex [8], and well-known syntactic sugar like do-notation in Haskell [21], enhanced for-statements in Java [13], or list-comprehension syntax in many languages demonstrate the necessity and the usefulness of syntactic extensions. Therefore, it is little surprise that numerous programming systems provide facilities that enable programmers to define and integrate their own custom syntactic extensions. Extensible systems include Racket [10], SugarJ [6], TemplateHaskell with quasiquoting [19, 26], and Scala macros [1].

Each syntactic extension defines a *desugaring transformation* that translates code of the extended language to code of the base language at compile time. This paper is concerned with interactions between the desugaring of extensions and the type checking of the base language. Consider the following simplified compilation pipeline:

$$\begin{tikzcd}[row sep=tiny]
 & & & \text{compile} \\
\text{ext. parse} \arrow[r] & \text{desugar} \arrow[r] & \text{type check} \arrow[ur] \arrow[dr] & \\
 & & & \textit{type error}
\end{tikzcd}$$

First, an extended parser reads the code that uses an extension. Second, the desugaring eliminates the extension by translation to the base language. Third, the type checker of the base languages checks the desugared code for type errors. Finally, if no type error occurred, the compiler does the rest of the processing. The advantage of this approach is its simplicity, only requiring an extensible parser and leaving the type checker and backend compiler unaffected. However, this comes at a high price.

1. Type errors are reported relative to the generated code. This exposes implementation details of the extension and violates the abstraction barrier that the extension tried to establish. To fix a type error, manual inspection of the generated code is often necessary, which is tedious and time consuming.
2. Type errors in generated code either result from a defective desugaring transformation *or* from a defective user program. To clarify the situation, it may be necessary to inspect the implementation of a desugaring transformation in order to identify the implicit contract that the user program must adhere to.
3. No type information is available to the desugaring transformation, which effectively prevents type-directed desugarings and extensions with inferred types.

To eliminate these problems, we propose the following compilation pipeline instead:

$$\begin{tikzcd}[row sep=tiny]
 & & \text{desugar} \arrow[r] & \text{compile} \\
\text{ext. parse} \arrow[r] & \text{ext. type check} \arrow[ur] \arrow[dr] & & \\
 & & \textit{type error} &
\end{tikzcd}$$

Instead of applying desugarings directly after parsing and before type checking, we propose to type check the extended program first and only desugar it afterwards. This design has three important implications: First, the type checker has to be extensible. Second, each syntactic extension has to extend the type checker, making the contract for user programs explicit. Third, since no further type checking occurs after desugaring, the desugaring transformation of each extension must ensure type soundness so that the well-typedness of the extended program carries over to the desugared program and the overall compilation pipeline is sound.

We present a language-independent system for type-sound syntactic extensibility called SoundX that implements the second pipeline from above. Given the grammar and typing rules of a base

```
// base-language syntax
lexical syntax
  [a-zA-Z][a-zA-Z0-9]*      -> ID
  [0-9][0-9]*               -> NUM

context-free syntax
  ID                        -> Term
  "λ" ID ":" Type "." Term -> Term
  Term Term                 -> Term
  NUM                       -> Term
  Term "+" Term             -> Term

  "Nat"                     -> Type
  Type "→" Type             -> Type
```

```
// metalanguage syntax
context-free syntax
  ∅                        -> Env
  Env "," ID ":" Type -> Env

judgement forms
  { Env "⊢" Term ":" Type }
  { ID ":" Type "∈" Env }

variables
  "t" [a-zA-Z0-9]* -> Term
  "x" [a-zA-Z0-9]* -> ID
  "T" [a-zA-Z0-9]* -> Type
  "Γ" [a-zA-Z0-9]* -> Env
  "n" [a-zA-Z0-9]* -> Num
```

```
// typing rules
Lookup:                      Abs:
 ----------                     Γ,x:T1 ⊢ t2 : T2
 x:T ∈ Γ,x:T                 ----------------------
                             Γ ⊢ λx:T1. t2 : T1 → T2
LookupSkip:
 (x≠y) (x:T ∈ Γ)             App:
 ---------------             Γ ⊢ t1 : T11 → T12
   x:T ∈ Γ,y:S               Γ ⊢ t2 : T11
                             -----------------------
Var:                              Γ ⊢ t1 t2 : T12
  x:T ∈ Γ
 ---------                   Add:
 Γ ⊢ x : T                   (Γ ⊢ t1 : Nat) (Γ ⊢ t2 : Nat)
                             -----------------------------
Nat:                               Γ ⊢ t1 + t2 : Nat
 ----------
 Γ ⊢ n : Nat
```

Figure 1. The SoundX base-language definition for the simply-typed lambda calculus with natural numbers $\lambda_\to$.

language, SoundX provides syntactic extensibility where a syntactic extension can affect any syntactic category of the base language (e. g., terms, types, statements, or class declarations). To provide access to type information, SoundX desugarings operate on typing derivations. That is, a desugaring receives a typing derivation of the extended program as input and yields a derivation of the desugared program as output. The resulting derivation is not checked for soundness later on. Instead, SoundX features a verification procedure that modularly and automatically verifies for each extension that the desugaring only produces valid derivations. This way, SoundX satisfies the following important proposition for any base language B, extension X, and program p:

If p is well-typed in $B \cup X$ and p desugars to p', then p' is well-typed in B.

Other systems for syntactic extensibility often follow a hybrid approach that interleaves desugaring and type checking. For example, Scala first type checks the arguments of a macro invocation, then expands the macro definition, and finally type checks the expanded macro body, which may find errors in the generated code [1]. In contrast, we propose to perform all type checking before any desugaring takes place and to verify the type-soundness of desugarings to prevent type errors in generated code altogether. Existing systems with sound desugaring after type checking [11, 17, 18] do not provide access to type information in the transformation and act upon terms instead of derivations.

In the remainder of this paper, we make the following contributions:

- We present SoundX, a language-independent system for type-sound syntactic extensibility supporting type-dependent desugarings (Section 2).
- We develop a novel derivation-centered desugaring procedure that transforms valid typing derivations of the extended language into valid typing derivations of the base language (Section 3).
- We present a procedure for automatically verifying the soundness of extensions. We show that the derivation desugaring and the verification procedure form a sound system, which we prove through preservation and progress theorems (Section 4).
- To demonstrate the expressiveness and applicability of our approach, we have instantiated SoundX with a subset of Java as base language, including generics and inheritance. We extend this base language with syntax for polymorphic pairs and Scala-like for-comprehensions (Section 5).

2. Exemplary Base Language and Extensions

SoundX is language-independent and supports the extension of arbitrary base languages defined with SoundX. A SoundX base-language definition consists of a context-free grammar and inductively defined typing rules over user-defined judgements.

2.1 $\lambda_\to$ Base Language

Figure 1 displays the SoundX base-language definition for the simply-typed lambda calculus with natural numbers $\lambda_\to$. The first column defines the context-free syntax of $\lambda_\to$, where we have omitted precedence and associativity annotations. For syntax, SoundX employs the syntax formalism SDF2 [29], which looks similar to EBNF but the defined nonterminal occurs on the right-hand side of a production. Internally, SDF2 uses a generalised LR parser that supports grammar extensions.

The second column of Figure 1 defines $\lambda_\to$-specific metalanguage syntax used for defining the typing rules of $\lambda_\to$. Besides typing environments `Env`, we define the syntax of judgements needed in the typing rules of $\lambda_\to$, namely the usual ternary typing judgement (`_ ⊢ _ : _`) and a lookup judgement to retrieve the type of a variable from the typing environment. Because typing environments and judgement forms are user-defined, SoundX can easily accommodate languages with other typing environments (e. g., subtyping, linear typing) and other judgement forms (e. g., type normalisation, class checking). The only built-in judgements of SoundX are equality and inequality. Finally, as common in the type-system literature, we define naming conventions to unambiguously associate metavariables with the syntactic sorts they range over.

The third column of Figure 1 lists the typing rules of $\lambda_\to$ using the judgements, typing environments, and metavariables declared in second column and the syntax defined in the first column. SoundX typing rules take the form of inductively defined inference rules with a sequence of newline-separated or parenthesised premises and a single consequence. The typing rules of $\lambda_\to$ are standard [15, 24]. Generally, SoundX assumes that the typing rules of the base language form a sound type system; the soundness of extensions is always relative to the base language. However, proving type soundness for the base language is not in scope of SoundX at the moment, which is why we do not model the dynamic semantics of base languages.

2.2 $\lambda_\to$ Extension: Let-Expressions With Inferred Types

To illustrate the expressive power of SoundX extensions, we use an example from Pierce's textbook [24, Chapter 11.5]. Pierce adds let-expressions to the simply-typed lambda-calculus with the following typing rule:

```
(Γ ⊢ t1 : T1) (Γ,x:T1 ⊢ t2 : T2)
--------------------------------Let
    Γ ⊢ let x=t1 in t2 : T2
```

He continues with the observation that it is not possible to implement let-expressions by the usual desugaring `let x=t1 in t2 ~~~> (λx:T1.t2) t1` at the term level since the typing annotation `T1` for the bound variable `x` does not occur in the left-hand side. Pierce concludes "that this information comes from the typechecker" and that "we should regard it [the `let` desugaring] as a transformation on *typing derivations* [. . .]." In his book, Pierce calls syntactic extensions that incorporate type information in their desugaring "a

```
context-free syntax
  "let" Bindings "in" Term -> Term

  ID "=" Term              -> Binding
  Binding                  -> Bindings
  Binding ";" Bindings     -> Bindings

variables
  "bs" [a-zA-Z0-9]* -> Bindings

inductive definitions and desugarings
  Let1:
    (Γ ⊢ t1 : T1) (Γ,x:T1 ⊢ t2 : T2)
    --------------------------------
     Γ ⊢ [ let x = t1 in t2 ] : T2
    ~~~> (λx:T1. t2) t1
  Let2:
    (Γ ⊢ t1 : T1) (Γ,x:T1 ⊢ let bs in t2 : T2)
    ------------------------------------------
        Γ ⊢ [ let x = t1; bs in t2 ] : T2
    ~~~> (λx:T1. let bs in t2) t1
```

Figure 2. $\lambda_\rightarrow$ extension: Let-expressions with inferred types.

```
context-free syntax
  "Pair" Type            -> Type

  "(" Term "," Term ")" -> Term
  Term "." "1"          -> Term
  Term "." "2"          -> Term

desugarings
  { Pair T ~~~> (T → T → T) → T }

inductive definitions and desugarings
  Fst:                          Snd:
    Γ ⊢ t : Pair T                Γ ⊢ t : Pair T
    --------------                --------------
    Γ ⊢[ t.1 ] : T                Γ ⊢[ t.2 ] : T
    ~~~> t (λa:T.λb:T.a)          ~~~> t (λa:T.λb:T.b)
  Pair:
    (Γ ⊢ t1 : T) (Γ ⊢ t2 : T)
    -------------------------
    Γ ⊢ [ (t1,t2) ] : Pair T
    ~~~> (λa:T.λb:T.λs:T→T→T. s a b) t1 t2
```

Figure 3. $\lambda_\rightarrow$ extension: Pairs at the type and term level.

little less derived" because their semantics is derived but their typing must be built into the base language.

While SoundX also supports fully-derived syntactic extensions whose desugarings do not require type information, the novel feature of SoundX is its support for syntactic extensions with desugaring transformations on typing derivations. To exemplify desugarings on typing derivations, we provide an extension of $\lambda_\rightarrow$ with let-expressions that allow multiple bindings scoped from left to right. This extension is a generalisation of the example by Pierce, where we require a recursive desugaring in addition to type information.

The full extension for let-expressions appears in Figure 2. First, we extend the syntax of terms to allow let-expressions with a list of one or more bindings. Second, we introduce a new naming scheme `bs` for metavariables ranging over bindings. Finally, we define the typing rules and the desugaring of let-expressions.

We define the typing rules and desugarings in a combined form, where the typing rules act as a guard for the desugarings. That is, a guarded desugaring applies to instances of the typing rule in the typing derivation. Importantly, all metavariables occurring in the typing rule can be used in the right-hand side of the desugaring. For example, this allows us to fill in the necessary type annotation on `x` in rule `Let1`, which handles let-expression with a single binding. Besides the insertion of the inferred type, the desugaring of rule `Let1` is standard. The brackets `[ ]` in the conclusion of the typing rule mark which part of a typing derivation a desugaring replaces. The brackets must fully enclose exactly one child of the used judgement (e. g., Γ, `t`, or `T` in the typing judgement of $\lambda_\rightarrow$). Roughly, the result of the desugaring is a derivation where the constructs inside the brackets are replaced by the desugared form on the right-hand side of the arrow in the desugaring rule.

The second rule `Let2` is responsible for let-expressions with more than one binding. Its second premise recursively brings the bound variables into the scope of the bound expressions and the body of the let-expression. The desugaring is equivalent to `Let1` except for recursion: The body of the abstraction on the right-hand side of the desugaring contains a residual let-expression. SoundX applies desugarings iteratively and the residual let-expression leads to a recursive desugaring based on derivation of the second premise of `Let2`.

When desugaring a typing derivation, it is important to ensure that the desugared derivation is valid with respect to the base language. SoundX verifies for each guarded desugaring that the desugaring yields a judgement for which a derivation can be automatically reconstructed. Roughly, this is the case if the desugared conclusion follows from the premises. For example, for `Let2` we have to prove that the following rule is derivable:

```
(Γ ⊢ t1 : T1) (Γ,x:T1 ⊢ let bs in t2 : T2)
------------------------------------------
      Γ ⊢ (λx:T1. let bs in t2) t1 : T2
```

This rule is derivable through applications of `Abs` and `App` from the base language. SoundX rejects desugarings that cannot be shown to yield reconstructable derivations.

2.3 $\lambda_\rightarrow$ Extension: Pairs at the Type and Term Level

Our second example extends $\lambda_\rightarrow$ with pairs at the level of types and terms. Similar to a Church encoding, we encode pairs using functions. However, due to the limited expressiveness of $\lambda_\rightarrow$, both components of a pair have to be of the same type.

Figure 3 shows the code that implements pairs. We add syntax for the pair type constructor `Pair T`, for pair construction `(t1,t2)`, and for the selection of pair components `t.1` and `t.2`. We desugar a pair type `Pair T` to a function type `(T→T→T) → T` that expects a selector function. Since parametric polymorphism is not available in $\lambda_\rightarrow$, we cannot use the usual Church encoding and instead require the components of a pair to have the same type. Our desugaring of Pair types is purely syntactic and does not require any context information. We call such desugarings *universal* since they are applicable whenever their left-hand pattern matches syntactically.

The typing rules and desugaring of pair expressions are now mostly straightforward. The selection of pair components `t.1` and `t.2` requires `t` to be of type `Pair T` for some `T`. A component selection desugars to a function application where we pass an appropriate selector function to `t`. Note that the bound variables `a` and `b` in the selector function are concrete base-language names; they are not metavariables because they do not match the naming scheme declared in Figure 1. On the other hand, the type annotation `T` is a metavariable whose value we extract from the typing derivation.

For pair construction, we require `t1` and `t2` to have the same type `T`. Basically, we desugar a pair construction into `λs:T→T→T. s t1 t2`, that is, a function that takes a selector and applies it to the components. However, such desugaring captures free occurrences of variable name `s` in `t1` or `t2`. Accordingly, `t1` in the desugared program would have to be well-typed in the extended context `Γ,s:T→T→T ⊢ t1 : T`, which we cannot guarantee given the premises of typing rule. Therefore, the verification procedure of SoundX rejects this desugaring as unsound. To avoid variable capture, we desugar a pair construction into an eta-expanded variant of `λs:T→T→T. s t1 t2`, which is a sound desugaring. Alternatively, we could have generated a fresh name as discussed in Section 5.2.

We introduced the extension for pairs because it illustrates a different class of guarded desugarings. While we can verify the soundness of `Let1`, `Let2`, and `Pair` by checking if the desugared conclusion follows from the non-desugared premises, this check fails for `Fst` and `Snd` because they rely on the desugared form of their premise: We cannot conclude `(Γ ⊢ t (λa:T.λb:T.a) : T)` from `(Γ ⊢ t : Pair T)`, unless `t` is known to be a function. In contrast to the other rules, `Fst` and `Snd` require that their premises are desugared first, so that (Γ ⊢ t : (T→T→T)→T). To support all extensions, SoundX features two desugaring strategies that we describe in the next section.

3. Desugaring Typing Derivations

A novelty of SoundX is that desugarings transform the typing derivation of the extended program. This imposes three challenges:

Propagation. Syntactic constructs such as terms and types occur duplicate in a typing derivation. When desugaring a construct, the desugared form must be propagated through the derivation toward the root.

Validity. When desugaring replaces a syntactic construct in a typing derivation, the derivation is likely to become corrupted and must be reconstructed to a valid derivation.

Traversal order. Desugarings on plain terms are typically applied in either bottom-up (leaves to root) or top-down (root to leaves) order. Desugarings on typing derivations require both top-down and bottom-up application depending on the occurrence of extended constructs in the premises of typing rules.

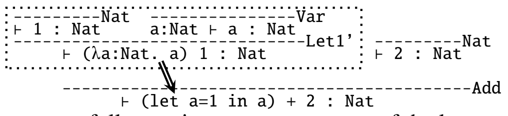
In the following, we present our solution for each challenge.

3.1 Propagation of Desugared Forms

Consider the following typing derivation of the judgement ∅ ⊢ (let a=1 in a) + 2 : Nat.

```
:..................................................:
:---------Nat   ---------------Var                 :
:⊢ 1 : Nat      a:Nat ⊢ a : Nat                    :
:------------------------------Let1:  ---------Nat
:      ⊢ let a=1 in a : Nat        :  ⊢ 2 : Nat
:..................................:
      ---------------------------------------------Add
            ⊢ (let a=1 in a) + 2 : Nat
```

The derivation contains the subexpression (let a=1 in a) twice: Once in the root node and once in the boxed subderivation starting at rule Let1. The desugarings for let-expressions apply to instances of the typing rules Let1 and Let2, which serve as guards. Accordingly, the desugaring of Let1 matches the boxed subderivation and rewrites the conclusion to (λa:Nat. a) 1, yielding an intermediate derivation of the following form:



```
:..................................:
:---------Nat   ---------------Var :
:⊢ 1 : Nat      a:Nat ⊢ a : Nat    :
:------------------------------Let1':  ---------Nat
:     ⊢ (λa:Nat. a) 1 : Nat         :  ⊢ 2 : Nat
:...................................:

      ---------------------------------------------Add
            ⊢ (let a=1 in a) + 2 : Nat
```

We have successfully rewritten one occurrence of the let-expression (for now ignoring the validity of the desugared boxed subderivation), but we have yet to propagate this desugaring to the rest of the derivation. In particular, as indicated by the arrow, we need to propagate the desugared form to the root of the tree, so that we get the fully desugared term (λa:Nat. a) 1 + 2. Generally, we propagate desugared forms toward the root of the derivation in a stepwise fashion. We call this propagation step *forwarding*.

To propagate the desugared form of our example, forwarding tries to find an alternative instantiation of the typing rule Add at the root of the derivation. The new instantiation of Add must be such that the premises match the desugared conclusions of the subderivations. That is, we assert the following equations on the syntax of judgements:

$$\Gamma \vdash \texttt{t1} : \texttt{Nat} = \emptyset \vdash (\lambda\texttt{a:Nat. a})\ \texttt{1} : \texttt{Nat}$$
$$\Gamma \vdash \texttt{t2} : \texttt{Nat} = \emptyset \vdash \texttt{2} : \texttt{Nat}$$

Matching the premises yields the following substitution:

$$\sigma = \{\Gamma \mapsto \emptyset,\ \texttt{t1} \mapsto (\lambda\texttt{a:Nat. a})\ \texttt{1},\ \texttt{t2} \mapsto \texttt{2}\}$$

With this we can create a new instance of typing rule Add by applying the substitution to the premises and conclusion (metavariables that do not occur in any premise are instantiated in a later step, see Section 4.5). By construction, the new premises now correspond to the conclusions of the subderivations. And, importantly, any desugared forms mentioned by the subderivations have been forwarded to the new conclusion. For our example, forwarding yields the following derivation with the fully desugared conclusion as desired:

```
---------Nat   ---------------Var
⊢ 1 : Nat      a:Nat ⊢ a : Nat
------------------------------Let1'   ---------Nat
      ⊢ (λa:Nat. a) 1 : Nat           ⊢ 2 : Nat
      -------------------------------------------Add
              ⊢ (λa:Nat. a) 1 + 2 : Nat
```

In general, forwarding can fail, namely if there is no matcher for the system of equations, that is, if the subderivations do not simultaneously match the premises of the typing rule. However, this only happens if multiple premises share a metavariable that represents an extended construct, which is desugared inconsistently into different base-language constructs by the respective subderivations. As we later show, this is the only way a desugaring can fail.

3.2 Reestablishing Validity for Desugared Conclusions

In our example in the previous subsection, we applied the desugaring of Let1 to the subderivation starting at rule Let1, yielding the following intermediate derivation:

```
---------Nat   ---------------Var
⊢ 1 : Nat      a:Nat ⊢ a : Nat
------------------------------Let1'
      ⊢ (λa:Nat. a) 1 : Nat
```

We annotated the last step in the derivation with rule name Let1', but actually no such rule exists; the above derivation is not valid in the base language. This is no surprise as we just changed the term in the conclusion without accommodating for this change in any way. Only if we can reconstruct a valid derivation for the desugared conclusion, we know that the desugared program is well-typed in the base language.

In general, such reconstruction is not always possible. For example, let us consider what happens if the desugaring for Let1 generated the conclusion ⊢ (λa:Nat. 1) a : Nat, accidentally swapping 1 and a. There is no chance of reconstructing a valid derivation for this conclusion because the reference to a is illegal in the empty environment. If we want to guarantee that desugarings only produce well-typed code, we must reject definitions like this one.

SoundX statically verifies the type soundness of a desugaring and rejects those desugarings that do not yield a provably derivable desugared conclusion. That is, for any guarded desugaring, SoundX automatically proves that the desugared conclusion is derivable from the premises of the typing rule. If the proof succeeds, it is guaranteed that the reconstruction of a valid derivation after desugaring succeeds for any instance of the typing rule. We present the details of the verification procedure in Section 4.4; for now it suffices that SoundX rejects the bogus desugaring for Let1 (because the desugared conclusion is not derivable) but accepts both extensions presented in the previous section. Thus, for our example derivation from above, we can indeed reconstruct a valid derivation for the desugared conclusion as follows. Note that the reconstruction is always local to the last rule that appears in the derivation. Technically, we use backward chaining with backtracking to reconstruct the derivation.

```
    ---------------Var
    a:Nat ⊢ a : Nat
-----------------------Abs   ---------Nat
⊢ λa:Nat. a : Nat → Nat      ⊢ 1 : Nat
-------------------------------------App
        ⊢ (λa:Nat. a) 1 : Nat
```

3.3 A Desugaring Traversal for Derivations

A guarded desugaring rewrites the conclusion of corresponding type-rule instances in the derivation. After each rewrite, we reestablish the validity of the derivation as discussed in Section 3.2 and we forward the desugared forms toward the root as discussed in Section 3.1. However, it turns out there is an intricate interaction between revalidation, forwarding, and the order in which desugarings are applied.

Traditionally, when desugaring a plain syntax tree, it is common to apply desugarings either bottom-up (leaves to root) or top-down (root to leaves). Because our forwarding also works bottom-up, it seems intuitive to desugar derivations bottom-up as well. Indeed, a bottom-up traversal is a valid strategy for many guarded desugarings. In fact, it is valid for all of our example desugarings except for `Let2`, because forwarding would fail.

Consider an extended version of the let-expression from before, where we bind an additional variable `b`:

```
                                 ...
 ---------Nat  ---------------------------------Let1
 ⊢ 0 : Nat      b:Nat ⊢ let a=1 in a : Nat
 -----------------------------------------------Let2
            ⊢ let b=0; a=1 in a : Nat
```

If we desugar this derivation bottom-up (leaves to root), the desugaring of `Let1` and subsequent revalidation yields the following intermediate derivation:

```
                                 ...
---------Nat  ----------------------------------App
⊢ 0 : Nat      b:Nat ⊢ (λa:Nat. a) 1 : Nat
-------------------------------------------------Let2
           ⊢ let b=0; a=1 in a : Nat
```

Next, forwarding tries to match the desugared conclusion of the inner let against the second premise of typing rule `Let2`:

```
Γ,x:T1 ⊢ let bs in t2 : T2 = b:Nat ⊢ (λa:Nat.a) 1 : Nat
```

This match cannot succeed because the desugared conclusion cannot contain a let-expression. For this reason, guarded desugarings that use extended syntax in their premises cannot be applied bottom-up. That is, unless it is possible to desugar the premise using a universal desugaring first, as in the case of `Fst` and `Snd`, where it is possible to perform forwarding after desugaring the `Pair` type constructor.

If a bottom-up traversal sometimes is not possible, let us consider a top-down traversal instead. In a top-down traversal, we first rewrite the root conclusion, then reestablish validity for the desugared conclusion, and finally continue desugaring the subderivations. To propagate desugared forms from subderivations, the top-down traversal still requires a subsequent bottom-up forwarding. For our example let-expression, desugaring and revalidating the root yields the following:

```
           ...
  --------------------------Let1
   b:Nat ⊢ let a = 1 in a : Nat
-------------------------------------Abs  ---------Nat
⊢ (λb:Nat. let a = 1 in a) : Nat→Nat     ⊢ 0 : Nat
-------------------------------------------------------App
         ⊢ (λb:Nat. let a = 1 in a) 0 : Nat
```

The recursive desugaring rewrites the conclusion of `Let1` into `(b:Nat ⊢ (λa:Nat. a) 1 : Nat)` as before. Does forwarding succeed now? It does because we have already eliminated the surrounding let-expression. Instead of forwarding the desugared term into rule `Let2` with its premise in extended syntax, we forward the desugared term through the already-desugared derivation via base-language rules `Abs` and `App`. This forwarding succeeds and provides us the fully desugared derivation:

```
            ...
  -------------------------App
   b:Nat ⊢ (λa. a) 1 : Nat
---------------------------------Abs  ---------Nat
⊢ (λb:Nat. (λa. a) 1) : Nat→Nat      ⊢ 0 : Nat
----------------------------------------------------App
        ⊢ (λb:Nat. (λa. a) 1) 0 : Nat
```

Unfortunately, sometimes a top-down traversal is not possible. Consider the following derivation of `⊢ (3,7).1 : Nat`.

```
---------Nat  ---------Nat
⊢ 3 : Nat     ⊢ 7 : Nat
-------------------------Pair
   ⊢ (3,7) : Pair Nat
   ------------------Fst
     ⊢ (3,7).1 : Nat
```

If we desugar this derivation top-down, we receive the desugared root conclusion `(⊢ (3,7) (λa:T.λb:T.a) : Nat)`. Next we need to reestablish a valid derivation for this conclusion, but such a derivation does not exist: We have no means to show that `(3,7)` has a function type, which is necessary for the application term to be typeable. The reason why the `Fst` desugaring cannot be applied top-down is that it relies on the desugared form of `(3,7)`; `Fst` and `Snd` require a bottom-up desugaring traversal.

SoundX Desugaring Traversal. In summary, the desugaring traversal of SoundX combines a top-down with a bottom-up traversal into a down-up traversal. On the way down (to the leaves), we apply top-down desugarings as shown above. On the way up (to the root), we apply forwarding and bottom-up desugarings as shown above. Whenever a desugaring rewrites a conclusion, we immediately reestablish validity.

N	Names
$E ::= N \mid N\,\vec{E}$	S-expressions
$J ::= \vdash_N \vec{E}$	Judgements, named N
$I ::= \vec{J} \rightharpoondown^{N} J$	Inference rules, named N
$U ::= E \rightsquigarrow E$	Universal desugarings
$G ::= I[E] \rightsquigarrow E$	Guarded desugarings
$B ::= (\vec{N}, \vec{I})$	Base language definitions
$X ::= (\vec{N}, \vec{I}, \vec{G}, \vec{U})$	Extensions
$\nabla ::= !J \mid \vec{\nabla} \Rightarrow^{N} J$	Derivations

Figure 4. Abstract syntax of SoundX.

The fact that different desugarings require different traversal directions also materialises in the verification procedure, which decides a desugaring's traversal direction and enforces a corresponding soundness criterium. We formalize the desugaring process of SoundX and its verification procedure in the following section and prove that the desugaring of verified extensions is sound.

4. Metatheory

In this section, we define SoundX as a formal system, describe its verification procedure and the precise details of the derivation desugaring. Finally, we establish soundness by showing that derivation desugaring satisfies preservation and progress theorems.

4.1 Notation and Abstract Syntax

We use the following notational conventions for lists. We write $\vec{a} = \langle a_1, \ldots, a_n \rangle = \langle a_{1..n} \rangle$ for a list of a elements. We liberally write $\langle a, \vec{a} \rangle$, $\langle \vec{a}, a \rangle$, and $\langle \vec{a}_1, \vec{a}_2 \rangle$ for prepending, appending, and list literals, respectively. We write $a \in \vec{a}$ to check membership, $\vec{a}_1 \subseteq \vec{a}_2$ to check if all elements in $\vec{a}_1$ are in $\vec{a}_2$ independent of order, and $\vec{a}_1 \not\cap \vec{a}_2$ to check if the elements of the two lists are disjoint.

For the formalisation of SoundX, we use an abstract syntax for judgements, inference rules, desugarings, derivations, etc. as shown in Figure 4. We assume a countably-infinite set of atomic names N. We capture all elements of the base-language and extensions by the single syntactic sort of s-expressions E. We use expressions E to model terms, types, statements, typing environments, and all other programming constructs. We write $vars(a)$ to recursively retrieve all names N that occur as atoms within s-expressions inside of a, and we write $cons(a)$ to retrieve all names N that occur as function symbols within s-expressions inside of a.

Each judgement J has a name and judges over a list of argument expressions $\vec{E}$. We denote inference rules I as $(\vec{J} \rightharpoondown^{N} J)$ where $\vec{J}$ are the premises, J is the conclusion, and N is the name of the rule. For example, we can represent typing rule `Var` of $\lambda_{\rightarrow}$ from Figure 1 using the following abstract syntax:

$$\langle \vdash_{\texttt{lookup}} \langle \texttt{x}, \texttt{T}, \Gamma \rangle \rangle \rightharpoondown^{\texttt{Var}} (\vdash_{\texttt{typed}} \langle \Gamma, \texttt{x}, \texttt{T} \rangle)$$

A universal desugaring U is a simple rewrite rule $(E \rightsquigarrow E')$ that matches E and produces E'. A guarded desugaring $(I[E] \rightsquigarrow E')$ is similar but uses an inference rule for matching. As explained in the previous section, a guarded desugaring has to mark which part of a judgement the desugaring replaces with brackets. In our abstract syntax, we use the abbreviation $I[E] := (\vec{J} \rightharpoondown^{N} (\vdash_{N_c} \vec{E}_1[E]\vec{E}_2))$, such that the guarded desugaring $(I[E] \rightsquigarrow E')$ replaces E by E' where $I[E]$ occurs in a derivation.

We represent an abstract base-language definition B by the constructors and inference rules it provides. A base-language definition $B = (\vec{N}, \vec{I})$ is well-formed (B ok) if all inference rules $\vec{I}$ have unique names and their constructors are listed in $\vec{N}$, $cons(\vec{I}) \subseteq \vec{N}$.

$$\text{V-ASSUMPTION}\quad \frac{}{\langle \nabla_{a1},\ldots,\nabla_{ak},\ldots,\nabla_{an}\rangle \vdash_{\vec{I}} \nabla_{ak}}$$

$$\text{V-RULE}\quad \frac{\begin{array}{c}\forall i \in 1..n : \vec{\nabla}_a \vdash_{\vec{I}} \nabla_i \\ (\langle J_{1..n}\rangle \multimap^N J_0) \in \vec{I} \\ [\sigma]\langle J_{1..n}, J_0\rangle = \langle concl\langle \nabla_{1..n}\rangle, J\rangle\end{array}}{\vec{\nabla}_a \vdash_{\vec{I}} \langle \nabla_{1..n}\rangle \Rightarrow^N J}$$

Figure 5. Valid derivations $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$.

An extension X is a 4-tuple that introduces new constructor names $\vec{N}$, inference rules $\vec{I}$, guarded desugarings $\vec{G}$, and universal desugarings $\vec{U}$. We explicitly distinguish an extension's inference rules $\vec{I}$, which will be used by the inference engine, from the patterns of guarded desugarings, which will be used for desugaring but not for inferring derivations. An extension $X = (\vec{N}, \vec{I}, \vec{G}, \vec{U})$ is well-formed ((B, X) ok) with respect to a base-language definition $B = (\vec{N}_B, \vec{I}_B)$ if $\vec{N}_B \not\cap \vec{N}$, the inference rules $\langle \vec{I}_B, \vec{I}\rangle$ have unique names, and $cons(\langle \vec{I}, \vec{G}, \vec{U}\rangle) \subseteq \langle \vec{N}_B, \vec{N}\rangle$.

We write ∇ to denote derivations. Instead of the space consuming two-dimensional display of derivations we write $\langle \nabla_1, \ldots, \nabla_n\rangle \Rightarrow^N J$ for the derivation

$$\frac{\nabla_1 \;\cdots\; \nabla_n}{J}N.$$

Later, in Section 4.4, we also need assumptions $!J$, which are judgements that we take for granted without any further evidence. We use the function *concl* to select the conclusion of a derivation $concl(\vec{\nabla} \Rightarrow^N J) = J$ and $concl(!J) = J$, and function *rule* to select the rule name N from a derivation $(\vec{\nabla} \Rightarrow^N J)$.

A substitution σ is a finite partial function from variables to expressions $\{N_1 \mapsto E_1, \ldots, N_n \mapsto E_n\}$. We write $[\sigma]a$ for the application of σ to some abstract syntax a. Since our abstract syntax has no binding structure, $[\sigma]a$ is direct replacement of variables (names that occur as atoms in s-expressions) by expressions.

4.2 Valid Derivations

The notion of a valid derivation is central to SoundX. The abstract syntax of Figure 4 permits the formation of arbitrary derivation trees ∇. Of course, only a subset of these derivations is actually valid with respect to some given inference rules $\vec{I}$. We formalise the validity of derivations in Figure 5 through judgement $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$.

Judgement $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ expresses that the derivation ∇ is valid with respect to the inference rules $\vec{I}$ and assumptions $\vec{\nabla}_a$. The derivations $\vec{\nabla}_a$ are assumed to be valid and, in particular, they may contain assumption judgements $!J$. The first rule V-ASSUMPTION states that ∇ is valid if it is an assumption. The second rule V-RULE specifies the conditions under which an instantiation of an inference rule is valid. Namely, it is valid if all subderivations are valid, there is an inference rule named N with n premises, and there exists a substitution σ that instantiates the inference rule such that it matches the conclusion J and premises $concl\langle \nabla_{1..n}\rangle$ in the derivation.

Lemma 1. *Judgement $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ satisfies the following structural properties:*

1. Substitution: *If $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$, then $[\sigma]\vec{\nabla}_a \vdash_{\vec{I}} [\sigma]\nabla$.*
2. Assumption exchange: *If $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ and $concl(\vec{\nabla}_a) = concl(\vec{\nabla}'_a)$, then there exists ∇' such that $\vec{\nabla}'_a \vdash_{\vec{I}} \nabla'$ and $concl(\nabla) = concl(\nabla')$.*
3. Transitivity: *If $\langle \nabla_{1..n}\rangle \vdash_{\vec{I}} \nabla$ and $\forall i \in 1..n : \langle\rangle \vdash_{\vec{I}} \nabla_i$, then $\langle\rangle \vdash_{\vec{I}} \nabla$.*

Proof. By induction on a derivation of $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ and a case analysis on the last rule applied in the derivation. The full proof is included in the supplement of this paper. □

$$\begin{array}{l} C_E ::= \bullet \mid N\langle \vec{E}, C_E, \vec{E}\rangle \\ C_J ::= \vdash_N \langle \vec{E}, C_E, \vec{E}\rangle \\ C_I ::= \langle \vec{J}, C_J, \vec{J}\rangle \multimap^N J \mid \vec{J} \multimap^N C_J \end{array}$$

$$\text{R-UNIVERSAL}\quad \frac{(E_0 \rightsquigarrow E'_0) \in \vec{U} \qquad E = [\sigma]E_0 \qquad E' = [\sigma]E'_0}{C_a[E] \longmapsto_{\vec{G};\vec{U}} C_a[E']}\, a \in \{E, J, I\}$$

$$\text{R-GUARDED}\quad \frac{(I_0[E] \rightsquigarrow E') \in \vec{G} \qquad I = [\sigma](I_0[E]) \qquad I' = [\sigma](I_0[E'])}{I \longmapsto_{\vec{G};\vec{U}} I'}$$

Figure 6. Definition of the small-step rewriting $a \longmapsto_{\vec{G};\vec{U}} a'$.

$$\text{RECONSTRUCT}\quad \frac{\vec{\nabla} \vdash_{\langle \vec{I}, \vec{I}_x\rangle} \nabla \qquad concl(\nabla) = J \qquad rule(\nabla) \in \vec{I}}{reconstruct_{\vec{I};\vec{I}_x}(\vec{\nabla}, J) \rhd \nabla}$$

Figure 7. Derivation reconstruction.

$$\text{S-BASE}\quad \frac{\begin{array}{c} rewrite_{\vec{G}_x;\vec{U}_x}(\langle J_{1..n}\rangle \multimap^N J) = \langle J'_{1..n}\rangle \multimap^N J' \\ cons\langle J'_{1..n}\rangle \subseteq \vec{N} \\ reconstruct_{\vec{I};\langle\rangle}(\langle !J'_{1..n}\rangle, J') \rhd \nabla \end{array}}{(\vec{N}, \vec{I}); (\vec{N}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x) \ltimes (\langle J_{1..n}\rangle \multimap^N J) : \mathcal{B}}$$

$$\text{S-EXT}\quad \frac{\begin{array}{c} rewrite_{\vec{G}_x;\vec{U}_x}(\langle J_{1..n}\rangle \multimap^N J) = \langle J'_{1..n}\rangle \multimap^N J' \\ reconstruct_{\vec{I};\vec{I}_x}(\langle !J_{1..n}\rangle, J') \rhd \nabla \end{array}}{(\vec{N}, \vec{I}); (\vec{N}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x) \ltimes (\langle J_{1..n}\rangle \multimap^N J) : \mathcal{X}}$$

Figure 8. Extension verification and classification $B; X \ltimes I : \{\mathcal{B}, \mathcal{X}\}$.

4.3 Rewriting

In Figure 6, we define the small-step rewriting $a \longmapsto_{\vec{G};\vec{U}} a'$ that applies universal and guarded desugarings to some abstract syntax a. A universal desugaring applies to the expressions within expressions, judgements, and inference rules. We use reduction contexts $C_a[E]$ to navigate to an expression E in the abstract syntax. A universal desugaring $U = E_0 \rightsquigarrow E'_0$ rewrites an expression E if there is a substitution σ such that σ instantiates the pattern E_0 of U to E. If the pattern matches, the desugaring replaces E by the right-hand side E'_0 instantiated through σ.

The small-step rewriting for guarded desugarings only applies to inference rules I (we describe the desugaring of typing derivations ∇ in Section 4.5). A guarded desugaring $I_0[E] \rightsquigarrow E'$ rewrites an inference rule I if there is a substitution σ such that σ instantiates the rule to I. The result then is the same inference rule I where E is replaced by $[\sigma]E'$.

SoundX does not depend on the details of the rewrite engine. We assume there is a function $rewrite_{\vec{G};\vec{U}}(a)$ that exhaustively applies the small-step rewriting $a \longmapsto_{\vec{G};\vec{U}} a'$ to $a \in \{E, J, I\}$. That is, we make the following assumption about *rewrite*:

Assumption 2.
If $rewrite_{\vec{G};\vec{U}}(a) = a'$, then $a \longmapsto^*_{\vec{G};\vec{U}} a'$ and $\nexists a''.\ a' \longmapsto_{\vec{G};\vec{U}} a''$.

Definition 3. *Terminating rewrite system*
A list of universal desugarings $\vec{U}$ forms a terminating rewrite system if for all $a \in \{E, J, I\}$ there exists a' such that $rewrite_{\langle\rangle;\vec{U}}(a) = a'$.

4.4 Extension Verification

With function *rewrite* in place, we can develop the SoundX verification procedure for extensions. The purpose of this procedure is to verify that an extension is sound, which means that the code generated by its desugaring is well-typed in the base language. To this end, extension verification has to guarantee that it is always possible to reconstruct a valid derivation for the desugared conclusion. We do so by symbolically desugaring the conclusion of inference rules and verifying that the premises entail the desugared conclusion. As a consequence, desugaring yields a valid derivation for any instance of such inference rule.

In our overview in Section 3, we illustrated that different extensions require different desugaring strategies. The decisive factor is whether a guarded desugaring relies on its premises in desugared form or in non-desugared form. For example, we saw that pair projection `Fst` and `Snd` rely on its premise in desugared form so that the generated function application is valid. Conversely, we saw that let binding `Let2` relies on its second premise in non-desugared form because it generates partially non-desugared code. We classify the inference rules of an extension according to what kind of guarded desugarings apply to them. We call an inference rule a $\mathcal{B}$-rule if the guarded desugarings rely on their premises to be in desugared form, that is, if the premises are part of the base language B. We call an inference rule an $\mathcal{X}$-rule if the guarded desugarings rely on their premises to be in non-desugared form, that is, unchanged.

Our verification procedure needs to ensure that it is possible to reconstruct a derivation for the desugared conclusion of an inference rule. To this end, we define relation $\mathit{reconstruct}_{\vec{I};\vec{I}_x}(\vec{\nabla}, J) \rhd \nabla$ in Figure 7. Given the assumptions $\vec{\nabla}$ and desugared conclusion J, a reconstructed derivation ∇ exists if ∇ is a valid derivation under the assumptions, ∇ shows J, and the last rule applied in ∇ is part of the base-language inference rules $\vec{I}$. The fact that reconstruction requires the final rule instantiated in ∇ to be from the base language ensures that the application of guarded desugarings always terminates. However, an efficient proof-search algorithm for finding the derivation ∇ is not in scope of this paper; in our implementation we use backward chaining with backtracking.

We describe our verification procedure in Figure 8. Given a base language definition $B = (\vec{N}, \vec{I})$ and an extension $X = (\vec{N}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$, an inference rule $I \in \vec{I}_x$ is a sound $\mathcal{B}$-rule if statement $B; X \ltimes I : \mathcal{B}$ can be derived by rule S-Base. We first symbolically desugar the inference rule using the universal and guarded desugarings of X. An inference rule can only be a $\mathcal{B}$-rule if all constructors occurring in the desugared premises are from the base language. An inference rule is a $\mathcal{B}$-rule we can reconstruct a derivation for the symbolically desugared conclusion J' given the symbolically desugared premises $\langle J'_{1..n} \rangle$ as assumptions, using only the inference rules of the base language $\vec{I}$. This allows the desugared conclusion to rely on the desugared premises.

The soundness criterion for $\mathcal{X}$-rules is similar. An inference rule I is a sound $\mathcal{X}$-rule if statement $B; X \ltimes I : \mathcal{X}$ can be derived by rule S-Ext. This rule differs from S-Base in two ways. First, S-Ext does not require the desugared premises to be from the base language; they may refer to constructors from the base language and from the extension. Second, S-Ext reconstructs a derivation of the symbolically desugared conclusion J' based on the non-desugared premises $\langle J_{1..n} \rangle$, using the inference rules of the base language $\vec{I}$ and extension $\vec{I}_x$. This allows the desugared conclusion to rely on the original, non-desugared premises.

Finally, an extension $(\vec{N}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ is sound $B \ltimes X$ with respect to a base language B if $B; X \ltimes I : \mathcal{B}$ or $B; X \ltimes I : \mathcal{X}$ holds for each inference rule $I \in \vec{I}_x$. Note that some inference rules simultaneously satisfy the conditions for $\mathcal{B}$-rules and $\mathcal{X}$-rules. For example, rules `Let1` and `Pair` from Section 3 satisfy both criteria. Our implementation chooses to use the inference rule as an $\mathcal{X}$-rule, which does not require forwarding. An extension with an inference rule that simultaneously relies on desugared and original premises is

FWD-Ok
$$\frac{\mathit{concl}(\vec{\nabla}) = [\sigma]\vec{J}}{\mathit{forward}(\vec{\nabla}, \vec{J} \multimap^{N} J) \rhd [\sigma]J}$$

FWD-Fail
$$\frac{\nexists \sigma.\ \mathit{concl}(\vec{\nabla}) = [\sigma]\vec{J}}{\mathit{forward}(\vec{\nabla}, \vec{J} \multimap^{N} J) \rhd \lightning}$$

Figure 9. Forwarding.

not supported by SoundX. However, in our experience, extensions like that can be represented as the composition (cf. Section 4.8) of two separately verified SoundX extensions, one that desugars top-down and one that desugars bottom-up.

4.5 Derivation Desugaring

We formalise the process of desugaring a typing derivation as outlined in Section 3. We first introduce forwarding and derivation reconstruction before combining them into a full desugaring traversal over derivations.

Forwarding. Forwarding propagates desugared forms from a subderivation toward the root of the derivation. We define forwarding as a relation $\mathit{forward}(\vec{\nabla}, I) \rhd J$ in Figure 9. Forwarding tries to find a refinement of inference rule I such that the premises of I match the conclusions of the (desugared) subderivations $\vec{\nabla}$. If such a refinement exists, rule FWD-Ok yields the refined conclusion of I.

If no matching substitution for the conclusions of the subderivations and the premises of the current rule exists, forwarding fails as indicated by $\lightning$ in rule FWD-Fail. We have not yet precisely formalised the conditions under which forwarding can fail but Section 4.7 provides a characterisation and an example of such a situation. As we later show, forwarding is the only part of our derivation desugaring that can fail.

Derivation Reconstruction. Our desugarings rewrite conclusions in a derivation into a desugared form. Therefore, the result of a rewriting is typically not a valid derivation. To reconstruct a valid derivation after a rewriting, we use *reconstruct* that we already used for extension verification in the previous subsection. In fact, derivation reconstruction after a rewriting always succeeds in SoundX, because the extension verification only admits extensions for which this property can be guaranteed. To verify that all usages of an extension permit derivation reconstruction, our verification procedure uses *reconstruct* to construct a derivation that generically verifies that the desugared conclusion is always derivable. In a practical implementation, it is sufficient to reuse this generic derivation and to instantiate it for usages of the extension; no further proof search is necessary for user programs. In our formalisation, we separately invoke *reconstruct* for every desugaring step to keep the presentation straight.

Down-up Desugaring Traversal. We desugar typing derivations through a down-up traversal [30]. Conceptually, a down-up traversal consists of a top-down traversal followed by bottom-up traversal. Technically, the two traversals are interleaved into a single pass over the derivation. The down-up traversal invokes a small-step downward desugaring while going down to the leaves and a small-step upward desugaring while back going up to the root. Specifically, we apply $\mathcal{X}$-rule desugarings downward (because they require the original premises) and we apply $\mathcal{B}$-rule desugarings upward (because they require the desugared premises).

We define the small-step downward desugaring of the root of a derivation by relation $\nabla \downarrow_{B;X} \nabla'$ in Figure 10(a). If the last rule used in a derivation is from the base language, no downward desugaring occurs and the derivation remains unchanged (rule TD-Base). Similarly, if the last rule is an $\mathcal{B}$-rule from the extension, we

$$\text{TD-Base}\quad \frac{(\vec{J_0} \rightarrowtriangle^N J_0) \in \vec{I}}{(\vec{\nabla} \Rightarrow^N J) \mapsdown_{(\vec{N},\vec{I});X} (\vec{\nabla} \Rightarrow^N J)}$$

$$\text{TD-ExtB}\quad \frac{\begin{array}{c}(\vec{J_0} \rightarrowtriangle^N J_0) \in \vec{I_x} \\ B; (\vec{N}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x) \ltimes (\vec{J_0} \rightarrowtriangle^N J_0) : \mathcal{B}\end{array}}{(\vec{\nabla} \Rightarrow^N J) \mapsdown_{B;(\vec{N}_x,\vec{I}_x,\vec{G}_x,\vec{U}_x)} (\vec{\nabla} \Rightarrow^N J)}$$

$$\text{TD-ExtX}\quad \frac{\begin{array}{c}(\vec{J_0} \rightarrowtriangle^N J_0) \in \vec{I_x} \\ (\vec{N}, \vec{I}); (\vec{N}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x) \ltimes (\vec{J_0} \rightarrowtriangle^N J_0) : \mathcal{X} \\ \mathit{rewrite}_{\vec{G}_x;\vec{U}_x}(\vec{J_0} \rightarrowtriangle^N J_0) = \vec{J'}_0 \rightarrowtriangle^N J'_0 \\ [\sigma]\langle \vec{J_0}, J_0 \rangle = \langle \mathit{concl}(\vec{\nabla}), J \rangle \\ \mathit{reconstruct}_{\vec{I};\vec{I}_x}(\vec{\nabla}, [\sigma]J'_0) \rhd \nabla'\end{array}}{(\vec{\nabla} \Rightarrow^N J) \mapsdown_{(\vec{N},\vec{I});(\vec{N}_x,\vec{I}_x,\vec{G}_x,\vec{U}_x)} \nabla'}$$

(a) Small-step downward desugaring $\nabla \mapsdown_{B;X} \nabla'$.

$$\text{BU-Base}\quad \frac{\begin{array}{c}(\vec{J_0} \rightarrowtriangle^N J_0) \in \vec{I} \\ \mathit{forward}(\vec{\nabla}, \vec{J_0} \rightarrowtriangle^N J_0) \rhd J' \\ [\sigma]J_0 = J \\ \sigma' = \{N \mapsto \mathit{rewrite}_{\langle\rangle;\vec{U}_x}(\sigma(N)) \mid N \in \mathit{dom}(\sigma) \setminus \mathit{vars}(\vec{J_0})\}\end{array}}{(\vec{\nabla} \Rightarrow^N J) \mapsup_{(\vec{N},\vec{I});(\vec{N}_x,\vec{I}_x,\vec{G}_x,\vec{U}_x)} (\vec{\nabla} \Rightarrow^N [\sigma']J')}$$

$$\text{BU-Ext}\quad \frac{\begin{array}{c}(\vec{J_0} \rightarrowtriangle^N J_0) \in \vec{I_x} \\ \mathit{rewrite}_{\vec{G}_x;\vec{U}_x}(\vec{J_0} \rightarrowtriangle^N J_0) = \vec{J'}_0 \rightarrowtriangle^N J'_0 \\ \mathit{forward}(\vec{\nabla}, \vec{J'}_0 \rightarrowtriangle^N J'_0) \rhd J' \\ [\sigma]J_0 = J \\ \sigma' = \{N \mapsto \mathit{rewrite}_{\langle\rangle;\vec{U}_x}(\sigma(N)) \mid N \in \mathit{dom}(\sigma) \setminus \mathit{vars}(\vec{J'_0})\} \\ \mathit{reconstruct}_{\vec{I};\langle\rangle}(\vec{\nabla}, [\sigma']J') \rhd \nabla'\end{array}}{(\vec{\nabla} \Rightarrow^N J) \mapsup_{(\vec{N},\vec{I});(\vec{N}_x,\vec{I}_x,\vec{G}_x,\vec{U}_x)} \nabla'}$$

(b) Small-step upward desugaring $\nabla \mapsup_{B;X} \nabla'$.

Figure 10. Small-step derivation desugarings.

$$\text{DU-Desugar}\quad \frac{\begin{array}{c}\nabla \mapsdown_{B;X} (\langle \nabla'_{1..n} \rangle \Rightarrow^{N'} J') \\ \forall i \in 1..n : \nabla'_i \Updownarrow_{B;X} \nabla''_i \\ (\langle \nabla''_{1..n} \rangle \Rightarrow^{N'} J') \mapsup_{B;X} \nabla'''\end{array}}{\nabla \Updownarrow_{B;X} \nabla'''}$$

Figure 11. Down-up derivation desugaring.

can leave derivation unchanged and perform a upward desugaring later on (rule TD-ExtB). Finally, rule TD-ExtX performs the downward desugaring given that the last rule of the derivation is an $\mathcal{X}$-rule from the extension. We obtain the desugared conclusion of the derivation by instantiating the desugared conclusion of the inference rule J'_0 using the original substitution σ from the derivation. We obtain a valid derivation ∇' for the desugared conclusion $[\sigma]J'_0$ through reconstruction based on the original subderivations $\vec{\nabla}$.

We define the small-step upward desugaring of the root of a derivation by relation $\nabla \mapsup_{B;X} \nabla'$ in Figure 10(b). If the last rule used in a derivation is from the base language, no guarded upward desugaring occurs. However, there may have been desugarings in subderivations, which need to propagate toward the root. To this end, rule BU-Base applies forwarding from the subderivations to the last rule of the derivation, yielding the new conclusion J'. Moreover, the small-step upward desugaring is also responsible for applying universal desugarings to the concrete terms occurring in the conclusion. To this end, we construct substitution σ' that binds and universally desugars the values of those metavariables N of the conclusion that do not occur in any premise. For metavariables that occur in some premise, the universal desugaring was already conducted in the corresponding subderivation. Rule BU-Ext handles instantiations of $\mathcal{B}$-rules of the extension. It desugars the instantiated rule, forwards the rewrites from the subderivations into the new conclusion, universally desugars the concrete terms of the conclusion, and reconstructs a valid derivation for the new conclusion.

The relations $\nabla \mapsdown_{B;X} \nabla'$ and $\nabla \mapsup_{B;X} \nabla'$ only desugar the root of the derivation. The desugaring of an entire derivation is a down-up traversal that applies these rules. We define the down-up desugaring traversal $\nabla \Updownarrow_{B;X} \nabla'$ in Figure 11. The traversal first applies a downward desugaring on derivation ∇, followed by a recursive down-up desugaring of the resulting subderivations ∇'_i. Finally, the traversal applies an upward desugaring that performs the necessary forwarding and yields the final derivation ∇'''.

As mentioned earlier, forwarding can fail. With the definitions of derivation desugaring presented above, a failure of the forward step entails that the desugaring becomes stuck, that is, there exists no ∇' such that we can derive $\nabla \Updownarrow_{B;X} \nabla'$. In order to prove that desugaring can only get stuck due to a failure in the forward step, we make this failure explicit and write $\nabla \Updownarrow_{B;X} \lightning$ whenever forwarding yields $\lightning$. We augment the small-step downward and upward desugarings with additional rules to propagate a forwarding failure. The rules are straightforward and appear as part of the supplementary material.

4.6 Soundness

We present our key metatheoretical result, namely that the down-up desugaring of sound extensions satisfies preservation and weak progress. This soundness result is fundamental for SoundX and ensures that desugaring after type checking yields a valid base-language derivation. All proofs are included in the supplementary material of this paper.

Preservation. Given a well-formed base language B, a well-formed extension X relative to B, and a valid derivation ∇ in the extended language, down-up desugaring preserves the validity of the typing derivations and yields a desugared derivation ∇' that is valid in the base language:

Theorem 4. *(Preservation)*
Let $B = (\vec{N}, \vec{I})$ *and* $X = (\vec{N}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$.
If B ok, (B, X) ok, $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$, *and* $\nabla \Updownarrow_{B;X} \nabla'$, *then* $\langle\rangle \vdash_{\vec{I}} \nabla'$.

It may come as a surprise that preservation does not require a sound extension but only a well-formed one. Extension soundness is not needed in the preservation theorem because the desugaring only succeeds if the derivation reconstruction in BU-Ext and TD-ExtX are successful. Thus, the progress theorem has to demonstrate that derivation reconstruction does not get stuck.

To prove the preservation theorem, we need the following preservation properties for the small-step downward and upward desugaring statements:

Lemma 5.
Let $B = (\vec{N}, \vec{I})$ *and* $X = (\vec{N}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ *with* B ok *and* (B, X) ok.

1. *If* $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ *and* $\nabla \mapsdown_{B;X} \nabla'$, *then* $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla'$.
2. *If* $\langle\rangle \vdash_{\vec{I}} \nabla_i$ *for all* $\nabla_i \in \vec{\nabla}$ *and* $(\vec{\nabla} \Rightarrow^N J) \mapsup_{B;X} \nabla'$, *then* $\langle\rangle \vdash_{\vec{I}} \nabla'$.

Proof. By a case analysis on the last rule applied in the derivations of $\langle\rangle \vdash_{\langle\vec{I},\vec{I}_x\rangle} \nabla$ (part 1) and $(\vec{\nabla} \Rightarrow^{N} J) \overline{\uparrow}_{B;X} \nabla'$ (part 2). □

Using these properties we can prove the preservation theorem:

Proof of Theorem 4 (Preservation). By induction on derivation $\nabla \Updownarrow_{B;X} \nabla'$ using Lemma 5. □

Progress. Given a well-formed base language B, a well-formed and sound extension X relative to B with terminating universal desugarings, and a valid typing derivation ∇, down-up desugaring either fails at a forwarding step or yields a desugared derivation ∇'.

Theorem 6. *(Progress)*
Let $B = (\vec{N}, \vec{I})$ and $X = (\vec{N}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$.
If B ok, (B, X) ok, $B \ltimes X$, $\vec{U}_x$ forms a terminating rewrite system, and $\langle\rangle \vdash_{\langle\vec{I},\vec{I}_x\rangle} \nabla$, then either $\nabla \Updownarrow_{B;X} \lightning$ or $\nabla \Updownarrow_{B;X} \nabla'$ for some ∇'.

In some sense, the progress theorem is more substantial than the preservation theorem for SoundX, because we have to show that the soundness of an extension indeed implies that the reconstruction of derivations always succeeds. Moreover, the statement $\nabla \Updownarrow_{B;X} \lightning$ is only derivable via the FWD-Fail rule which means that failed forwarding is the only possible cause of a stuck down-up desugaring. In particular, it is impossible that the desugaring of an ill-specified extension leads to this statement since this extension would be rejected upfront during verification.

The assumption of the progress theorem that $\vec{U}_x$ forms a terminating rewrite system is, of course, not decidable. A particular $\vec{U}_x$ which does not terminate for some input may cause the verification of the desugaring or the execution of the desugaring to diverge. However, this does not compromise soundness since no desugared code is produced. If the desugaring terminates, our theorems guarantee the result is well-typed.

To prove the progress theorem, we need the following progress properties for the small-step downward and upward desugaring statements:

Lemma 7.
Let $B = (\vec{N}, \vec{I})$ and $X = (\vec{N}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ with B ok, (B, X) ok, $B \ltimes X$, and $\vec{U}_x$ forms a terminating rewrite system.

1. *If $\langle\rangle \vdash_{\langle\vec{I},\vec{I}_x\rangle} \nabla$ then $\nabla \overline{\downarrow}_{B;X} \nabla'$ for some ∇'.*
2. *If $\nabla \overline{\downarrow}_{B;X} (\vec{\nabla}' \Rightarrow^{N'} J')$ then either $(\vec{J}_0 \rightarrow^{N'} J_0) \in \vec{I}$ or $(\vec{J}_0 \rightarrow^{N'} J_0) \in \vec{I}_x$ with $B; X \ltimes (\vec{J}_0 \rightarrow^{N'} J_0) : \mathcal{B}$.*

Proof. By a case analysis on the last rule applied in the derivations of $\langle\rangle \vdash_{\langle\vec{I},\vec{I}_x\rangle} \nabla$ (part 1) and $\nabla \overline{\downarrow}_{B;X} (\vec{\nabla}' \Rightarrow^{N'} J')$ (part 2) using the extension soundness of X in the TD-ExtX case. □

Using these properties we can prove the progress theorem.

Proof of Theorem 6 (Progress). By induction on derivation $\langle\rangle \vdash_{\langle\vec{I},\vec{I}_x\rangle} \nabla$ and a case analysis on the last rule applied in the derivation using Lemmas 5 and 7 and Theorem 4. □

4.7 Failure of Forwarding

As we describe at the beginning of Section 4.5 forwarding can fail if there exists no matching substitution for the conclusions of the subderivations and the current premises. Intuitively, forwarding can only fail if a metavariable occurs multiple times in the current premises and the respective instantiations are inconsistently desugared in the subderivations. Despite the fact that metavariables regularly occur multiple times in the premises of a rule, we are not aware of any *meaningful* example where forwarding failed.

We can construct an *artificial* example where forwarding fails. Consider base language $\lambda^{\lightning}_{\rightarrow}$ that contains the following typing rule:

```
Bogus:  (Γ,a:Nat ⊢ t : T)  (Γ,a:Bool ⊢ t : T)
        -------------------------------------
                     Γ ⊢ t : Nat
```

Here `a` and `b` are concrete object-level identifiers and `t` is a metavariable that appears in both premises. Using the let-expression extension of Figure 2, we can construct a derivation for term `bogus (let c=a in 1)`, abbreviating `Nat` as `N` and `Bool` as `B`:

```
-----------   ----------------        -----------   ----------------
a:N ⊢ a : N   a:N,c:N ⊢ 1 : N         a:B ⊢ a : B   a:B,c:B ⊢ 1 : N
-----------------------------Let1  ------------------------------Let1
a:N ⊢ let c=a in 1 : N                     a:B ⊢ let c=a in 1 : N
--------------------------------------------------------------------Bogus
                    ⊢ bogus (let c=a in 1) : N
```

The desugaring of `Let1` transforms the conclusion of the left subderivation into `a:N ⊢ (λc:N.1) a : N` and the conclusion of the right subderivation into `a:B ⊢ (λc:B.1) a : N`. Forwarding into the `Bogus`-rule now tries to solve the following two equations with a single substitution:

```
Γ,a:N ⊢ t : T = a:N ⊢ (λc:N.1) a : N
Γ,a:B ⊢ t : T = a:B ⊢ (λc:B.1) a : N
```

Since the type annotations in the lambda-abstractions differ, no substitution exists for metavariable `t` and forwarding fails.

The failure is related to the double occurence of the metavariable `t` in different contexts in `Bogus` and the dependence of the let-desugaring on the type of `t1` which is copied into the generated code. Neither of the two is problematic as such but only their combination leads to the failure. In the example, `Bogus` is a base-language rule. All our attempts to construct an extension, even an artificial one, containing a similar typing rule were either rejected during verification or classified as $\mathcal{X}$-rule, which is not subject to the forwarding step. None of our case studies shows problems with failed forwarding, thus providing practical evidence that SoundX can be successfully applied.

4.8 Extension Composition

SoundX extensions can be composed in two manners. First, SoundX supports *incremental extension* [7] where one extension desugars into an already extended base language, thus stacking extensions on top of each other. Second, SoundX supports *extension unification* [7] where two extensions are combined into a new extension. In both cases, a modular verification of each extension provides a sound system and it is not necessary to reverify the composition of extensions. It suffices to check that the names of constructors and inference rules of different extensions do not overlap.

For incremental extensions, let us consider an extension X_1 that is sound relative to base language B and another extension X_2 that is sound relative to $B \cup X_1$. SoundX sequentially desugars extensions X_1 and X_2. Specifically, SoundX desugars a valid derivation ∇ in $B \cup X_1 \cup X_2$ into a valid derivation ∇' in $B \cup X_1$, which it subsequently desugars into a valid derivation ∇'' in B. Preservation ensures that ∇'' is valid and progress ensures that the desugaring indeed yields a derivation ∇'' (or forwarding fails).

For extension unification, let us consider extensions X_1 and X_2 that are sound relative to base language B and that do not have overlapping constructors or inference-rule names. SoundX can desugar X_1 and X_2 in any order. For example, we can first desugar X_1 considering $B \cup X_2$ the base language. This is sound since desugarings defined in X_1 are never applied to rule instantiations of X_2 because inference rule names do not overlap and reconstruction is stable under the addition of the inference rules from X_2 to B's inference rules. Thus, SoundX supports the composition of extensions that have been independently verified sound.

5. Case Study: Extensible Java

To demonstrate the applicability of SoundX in a more realistic language than $\lambda_{\rightarrow}$, we implemented a subset of Java as a SoundX base language. We call this subset JavaLight, which comprises inheritance, generic classes, primitive types `boolean` and `int`, packages and imports, public methods and constructors, private fields, statements like assignment, `while`, `if`, and method invocation, and

```
package javalight.lang;

import javalight.util.Iterator;

public class Iterable<A> {
  // must be overridden by subclasses
  public Iterator<A> iterator() { return null; }
}
```

Figure 12. The `Iterable<A>` class.

```
lexical syntax
  [a-zA-Z] ALPHANUMS -> ID  // metavariables for ID: x, y, z
  { ID "." }+        -> PID // metavariable for PID: pkg
  PID "." ID         -> QID // metavariable for QID: q

context-free syntax
  "int"              -> AType  // metavariables for AType: t, s
  "boolean"          -> AType
  QID ATypes         -> AType
  ID                 -> AType
  "Object"           -> AType

  "[]"               -> ATypes // metavariable for ATypes: t*, s*
  AType "::" ATypes  -> ATypes
```

Figure 13. Abstract syntax of JavaLight types.

expressions. Compared to Java, the most notable omissions are interfaces, anonymous, local, and nested classes, and modifiers like `private`, `protected`, or `static`.

Based on JavaLight we define Scala-like for-comprehensions with inferred types for the bound variables as a syntactic extension. For-comprehensions can be used to iterate over elements of a collection class that extends class `Iterable<A>` shown in Figure 12. Our classes `Iterable<A>` and `Iterator<A>` take the role of the corresponding Java interfaces, where `Iterator<A>` declares methods `hasNext` and `next`. For-comprehensions are an interesting case study for SoundX because (i) the desugaring is type-dependent, (ii) the desugaring spans multiple syntactic sorts, (iii) and the desugaring builds on another extension for enhanced for-loops of the form `for(T x : e) stm`.[1]

In the remainder of this section, we abbreviate package `javalight.lang` with `jl.l` and `javalight.util` with `jl.u`.

5.1 The Base Language JavaLight

Due to its volume, we cannot describe all details of the JavaLight base-language definition. Thus, we only sketch those parts that are relevant for the extension with for-comprehensions.

Abstract Syntax for Types and Class Tables. In the definition of the JavaLight type system, we use the abstract syntax of Figure 13 for the representation of types. We provide judgements that translate the concrete syntax for types that appears as part of the user program into the abstract syntax for types. An abstract syntax is necessary since the concrete syntax of types does not distinguish between type variables and class names. For example, in the field declaration `private Foo x;`, `Foo` could be a type variable or a class name. They can only be distinguished by looking at the enclosing class declaration. If the field is declared in a generic class with the type parameter `Foo`, the declaration refers to that parameter, otherwise it is a class reference. In the abstract syntax, we represent type variables as identifiers `x` and class types as fully qualified identifiers `q` followed by a list of type arguments `ts`. The type of a non-generic class is written `q []` with an empty list of type arguments. We provide judgements `norm(CT, X*, T) = t` and `normPrim(CT, X*, T) = t` to translate user-written types `T` into their abstract representation `t` where `X*` are the type parameters of the enclosing class and `CT` is the class table. `norm` only succeeds for class types whereas `normPrim` also succeeds for primitive types.

[1] Source code available at `http://github.com/florenzen/soundx`.

A class table `CT` associates class names with class signatures. A class signature `ct` contains the types of the constructor's arguments, the types of the public methods, the superclass, and the type parameters of the class. For example, the signature of class `Iterable<A>` is
```
{[],                                          // constructor args
 iterator:([]->jl.u.Iterator A::[]) :: [],    // method types
 Object,                                      // superclass
 A::[]}                                       // type parameters
```
Note that list syntax `a :: as` and `[]` is overloaded and used at multiple places. Similar to the abstract syntax of types, class signatures are internal to the type system; they are not part of the surface syntax of JavaLight.

Judgements. The main judgements of JavaLight are the typing judgements for statements and expressions as well as the subtyping relation. The typing judgement for statements

$$\texttt{CT; Ef; El} \vdash \texttt{stm*} \leadsto \texttt{rt}$$

assigns a return type `rt` to a sequence of statement `stm*`. A return type is either `void` or an `AType` from Figure 13. Statements are typed under a class table `CT` and two local environments `Ef` and `El`. The environment `Ef` contains the names and types of the fields of the enclosing class whereas `El` contains the names and types of the parameters of the current method and the locally defined variables. According to Java's scoping rules, a local variable can shadow a field but not another local variable. The lookup judgement $\texttt{x:t} \in \texttt{Ef;El}$ reflects this by searching for `x` in `El` prior to `Ef`.

The typing judgement for expressions

$$\texttt{CT; Ef; El} \vdash \texttt{e : t}$$

assigns a type `t` to expression `e` using the class table `CT` and the two typing environments `Ef` and `El`. The type of an expression is an `AType` that excludes `void`.

The subtyping judgement $\texttt{CT} \vdash \texttt{s <: t}$ asserts that type `s` is a subtype of type `t`. A class type `(q s*)` is a subtype of another type `t` if `t` corresponds to the declared superclass of the class named `q`. Our definition of the subtyping judgement is reflexive and transitive.

In addition to the regular type rules of JavaLight, our formulation includes structural type rules for weakening and permutation of the typing context as well as subsumption. These type rules are not needed for type checking, but our verification procedure requires them as lemmas to proof the soundness of certain extensions. Since weakening, permutation, and subsumption are properties of the JavaLight type system, it is safe to add them explicitly.

Apart from the main judgement forms, JavaLight requires several other judgements to support substitution, well-formedness checking, class-table handling, class-name qualification, method lookup, and type normalisation to name a few. Many of these judgements are defined over multiple syntactic sorts, such as expressions, statements, methods, types, and classes, and many judgements are lifted to operate on lists of elements. In total, our definition of JavaLight uses 48 judgement forms that are implemented by 150 inference rules. A significant portion of these inference rules only implement congruence rules and could be considered boilerplate.

5.2 Enhanced For-Statement

We base the implementation of for-comprehensions on Java's enhanced for-loop `for(T x : e) stm`. Since this statement is not part of JavaLight, we define it as an extension which is desugared into a while-loop over an iterator:
```
{ Iterator<T> it = e.iterator();
  while(it.hasNext()) {
    T x = it.next();
    stm } }
```
Note that the result of the desugaring is wrapped into a code block to limit the scope of the generated variable `it`. Moreover, the variable `it` may not shadow any variable from the enclosing local environment. SoundX features a simple facility to generate fresh

```
S-EnhancedFor:
  CT; Ef; El ⊢ e : s            ①
  CT ⊢ s <: jl.l.Iterable t::[] ②
  x ∉ dom(El)                   ③
  CT; Ef; El,x:t ⊢ stm ~ void   ④
  CT; Ef; El ⊢ stm* ~ rt        ⑤
  typevars(CT) = X*             ⑥
  norm(CT, X*, T) = t           ⑦
  CT.jl.l.Iterable =
    {[],iterator:[]->jl.u.Iterator A::[]::[],Object,A::[]}   ⑧
  CT.jl.u.Iterator =
    {[],hasNext:[]->boolean::next:[]->A::[],Object,A::[]}    ⑨
  norm(CT, X*, Iterator<T>) = jl.u.Iterator t::[]            ⑩
  ----------------------------------------------------------------
           CT; Ef; El ⊢ [ for(T x : e) stm stm* ]  ~ rt
  ~~~> { Iterator<T> y = e.iterator();
         while(y.hasNext()) {
           T x = y.next();
           stm
       } }
       stm*
       where y = fresh(Ef; El,x:t)
```

Figure 14. Guarded desugaring for enhanced for-loops.

names based on information from the typing derivation. Specifically, we can generate fresh names that are not bound in the environments `Ef` and `El`. This way, our desugaring for enhanced for-statements obtains a fresh variable name for the iterator. SoundX rejects any definition of enhanced for-loops that fails to ensure proper scoping.

Figure 14 shows the guarded desugaring of an enhanced for-statement with typing rule `S-EnhancedFor` that captures all necessary prerequisites. An enhanced for statement is well-typed if the right-hand side `e` is a subtype of class `Iterable<T>` and the statement `stm` has type `void` in the environment extended by the bound variable `x` of type `T`. The premises ①, ②, and ④ implement these conditions. Since in Java, and hence in JavaLight, local variables may only shadow fields but not other local variables, we need premise ③, which prevents that variable `x` shadows a variable from the surrounding scope. Without this premise SoundX rejects the desugaring rule because the variable declaration of `x` in the desugared code is illegal given the context `El`. Since the statement typing judgement acts on a sequence of statements, premise ⑤ requires that the statements `stm*` following the enhanced for-statement are well-typed. The scope of the bound variable `x` is limited to the for-statement and does not extend to the subsequent statements.

The next two premises ⑥ and ⑦ check that the declared type `T` is a valid type in the current class table and can be translated into its abstract representation. Premise ⑥ `typevars(CT) = X*` simply extracts the type paramaters `X*` of the enclosing class to normalise the type `T`. The last three premises ⑧ to ⑩ are concerned with the fact that an enhanced for-statement can only be desugared if the classes `jl.l.Iterable` and `jl.u.Iterator` are defined in the class table. Premises ⑧ and ⑨ do a class-table lookup and assert the classes have the right signature, as the desugared code would otherwise be illegal. Due to these premises, the type checker rejects any enhanced for-statement in user code where `jl.l.Iterable` or `jl.u.Iterator` are unavailable or provide the wrong methods.

Fresh Name Generation. To desugar an enhanced for-statement, we have to generate a fresh name `y` for binding the iterator as shown at the bottom of Figure 14. This name must be fresh with respect to `x` and the current local and field environments `El` and `Ef` such that SoundX can deduce for the desugared code

```
CT; Ef; El,y:(jl.u.Iterator t::[]),x:t ⊢ stm ~ void
```

from the original premise

```
CT; Ef; El,x:t ⊢ stm ~ void
```

using the weakening and permutation rules. The application of these rules requires `x≠y`, `y ∉ dom(Ef)`, and `y ∉ dom(El)`. The call to the built-in function `fresh` with argument `(Ef;El,x:t)` yields a name that satisfies these conditions. Function `fresh` is a small extension to the desugaring procedure of Section 4 provided by the SoundX implementation. During desugaring, a call `fresh(Ef;El)` is replaced by an identifier `y` which satisfies `y ∉ dom(Ef)` and `y ∉ dom(El)`. The syntax and freshness condition of `fresh` are declared as part of the base-language definition; the fresh-name generation is generically handled by SoundX.

In our verification procedure from Section 4, instead of generating a fresh name, we use the call `fresh(Ef;El)` to represent the fresh name symbolically. Since `fresh(Ef;El) ∉ dom(Ef;El)` by definition, our symbolic representation indeed mimics a fresh name. This way, the SoundX verification procedure succeeds to apply weakening and permutation and manages to verify the extension for enhanced for-loops sound with respect to JavaLight.

An alternative solution for obtaining a fresh name `y` is to include the freshness condition `y ∉ dom(Ef;El,x:t)` as an additional premise of `S-EnhancedFor`. For the verification procedure, this condition is exactly what is needed to apply weakening and permutation and to prove the extension sound. When using `S-EnhancedFor` to construct a typing derivation, the type checker instantiates `y` with a fresh name, because `y` does neither occur in any other premise nor in the non-desugared conclusion. In contrast, function `fresh` only constructs a fresh name during desugaring, not during type checking. More importantly, in SoundX, a typing rule also acts as the interface of its extension. Therefore, the typing rule should not expose implementation details of fresh-name generation to clients. We added function `fresh` to hide name generation from clients.

SoundX uses a first-order approach for name binding and more sophisticated approaches for handling names exist in the literature [2, 3, 27, 28]. However, as SoundX verifies extensions against the type system of the base language, which necessarily also codifies variable scoping, extensions with unintended variable captures are rightfully rejected by SoundX. For example, suppose we used a fixed name `it` instead of `y` on the right-hand side of `S-EnhancedFor`. Under these circumstances, `CT;Ef;El,x:t ⊢ stm ~ void` does not imply `CT;Ef;El,it:(jl.u.Iterator t:[]),x:t ⊢ stm ~ void`, because `it` might shadow a field in `Ef` or a local variable in `El`. SoundX rejects such ill-scoped extensions.

```
"for" "(" Enumerators ")" Statement -> Statement

Enumerator                          -> Enumerators
Enumerator ";" Enumerators          -> Enumerators

ID "<-" Expr                        -> Enumerator
"if" Expr                           -> Enumerator
```

Figure 15. Syntax of for-comprehensions.

5.3 Scala-like For-Comprehensions

With all the hard work with respect to fresh names and while-loops already handled in the extension for enhanced for-loops, the definition of Scala-like for-comprehensions with inferred types, generators, and guards is relatively simple. We desugar for-comprehension into JavaLight extended by enhanced for-loops.

Figure 15 shows the syntax of for-comprehensions, which introduces the new sort `Enumerator`. An enumerator is either a generator `x <- e` iterating over elements provided by `e` or a guard `if e` to skip an iteration if `e` is false. The variables bound by generators are visible from left to right in the enumerators.

We desugar enumerators of a for-comprehension in a stepwise fashion similar to the bindings of let-expressions in Section 3. Figure 16 shows the two recursive cases of the guarded desugarings that implement for-comprehensions. As defined by the desugaring of the rule `S-ForCompGenEnums`, a generator is directly translated into an enhanced for-loop with a residual for-comprehension in the body. The type annotation `T` that is required by the enhanced for-

```
S-ForCompGenEnums:
  CT; Ef; El ⊢ e : s
  CT ⊢ s <: jl.l.Iterable t::[]
  typevars(CT) = X*
  norm(CT, X*, T) = t
  x ∉ dom(El)
  CT; Ef; El,x:t ⊢ for(enums) stm ~ void
  CT; Ef; El ⊢ stm* ~ rt
  // plus premises ⑧ to ⑩ from S-EnhancedFor (Figure 14)
  --------------------------------------------------------
     CT; Ef; El ⊢ [ for(x <- e; enums) stm stm* ] ~ rt
  ~~~> (for(T x : e) for(enums) stm) stm*

S-ForCompIfEnums:
  CT; Ef; El ⊢ e : boolean
  CT; Ef; El ⊢ for(enums) stm ~ void
  CT; Ef; El ⊢ stm* ~ rt
  ---------------------------------------------
  CT; Ef; El ⊢ [ for(if e; enums) stm stm* ] ~ rt
  ~~~>  (if(e) for(enums) stm) stm*
```

Figure 16. Excerpt of the typing rules and desugarings of for-comprehensions (recursive cases).

loop is copied from the typing derivation (fourth premise). Similarly, a guard `if e` directly desugars into an `if`-statement. The rules for desugaring the non-recursive cases of for-comprehensions with only one generator or guard are similar, except they have no residual for-comprehension.

SoundX automatically verifies the soundness of the implementation of for-comprehensions relative to base language JavaLight extended with enhanced for-loops. As explained in Section 4.8, during the desugaring of a program that uses for-comprehensions, the SoundX desugaring procedure performs two passes of down-up desugaring to first eliminate for-comprehensions and then eliminate enhanced for-loops.

6. Related Work

We provide an overview of related work in Figure 17. First, we distinguish systems that provide context information to a desugaring from systems that do not provide context information. Second, we distinguish systems that check the correctness of generated code dynamically after the desugaring has run from systems that statically guarantee that a desugaring only generates well-typed code. We discuss the four resulting categories of systems in turn.

First, we have systems that check the soundness of generated code dynamically and do not provide any context information to a desugaring transformation. Systems of this category include syntactic preprocessors such as TemplateHaskell or Camlp4 that only run the type checker after desugaring, but also include systems like MetaHaskell or Ziggurat that perform type checking before desugaring but do not expose this information to the desugaring.

Second, there are a number of systems that provide context information to the desugaring transformation but only check the well-typedness of generated code after the desugaring has run. To provide context information to the desugaring, systems of this category type check part of the non-desugared program prior to desugaring. The desugaring then operates on a syntax object that is enriched with typing information. For example, Scala type checks the arguments of a macro before expanding the macro and the macro can inspect the type of its arguments during expansion. Since the systems of this category do not statically verify that a desugaring only generates well-typed code, an explicit type check after desugaring is necessary. If this type check fails, users of an extension will see error messages in terms of the desugared code.

Third, a few systems guarantee static soundness, that is, they verify that a desugaring can only generate well-typed code. However, these systems do not provide context information to the desugaring, which simplifies the verification but also limits the expressiveness of desugarings. MacroML extends ML with type-safe macros based on

	no context info.	context info.
dynamic soundness	TemplateHaskell [26], Camlp4 [5], MetaHaskell [20], Ziggurat [9]	Xoc [4], Scala macros [1], TSLs in Wyvern [23]
static soundness	MacroML[11], λ_m [17], SoundExt [18]	CPS for DML [31], **SoundX**

Figure 17. Overview of related work.

a multi-stage type-system. Its expressiveness is limited to generative macros and the binding structure of macros is limited to predefined patterns like lambda-abstraction or let-expressions. Herman's λ_m-calculus adds a signature system to a Scheme-like macro system to declare the binding structure of macros. In this system, signature-correct macros generate well-scoped code but there is no further guarantee with respect to static typing. The approach to type-sound language extension of SoundExt is similar to ours with respect to extension verification, but solely relies on context-free term-level desugarings.

Finally, we have the category of systems that guarantee static soundness of desugarings while allowing desugarings to reflect on typing information of the program. Besides our own work, we are only aware of one other work in this category. Namely, Xi and Schürmann present a type-preserving CPS transformation for a core of dependent ML that is type-dependent and verified to be sound [31]. Like in our system, the transformation operates on typing derivations. However, while SoundX is a *framework* for type-preserving transformations, a generalisation of the technique used by Xi and Schürmann to develop the CPS transformation has not been explored; the work presented by Xi and Schürmann is language-specific and transformation-specific.

The reasoning techniques employed by SoundX for verifying extension is limited to backward chaining of type rules. Other systems such as Twelf [16, 25] or Veritas [14] employ more sophisticated reasoning techniques that, for example, support inductive proofs. While we did not require such reasoning techniques in our experiments so far, they would increase the expressiveness of SoundX and enable, for example, modularly specified typing rules that govern a class of desugarings.

7. Conclusion

We have presented SoundX, a system for sound syntactic language extensibility where desugarings can depend on typing information from the original program. To this end, we developed a new desugaring technique that transforms typing derivations rather than abstract syntax. By verifying that desugarings can only produce valid typing derivations, SoundX guarantees that the desugared code is well-typed. SoundX rejects any extension that fails this soundness criterion. We have formalised our techniques for derivation desugaring and extension verification and proved that the desugaring of a sound extension indeed yields well-typed code. In future work, we will integrate more features of Java into JavaLight and try to encode richer extensions in order to better understand the expressiveness and limitations of SoundX.

Acknowledgments

We thank Sylvia Grewe and the reviewers for their helpful feedback.

References

[1] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of*

the 4th Workshop on Scala, SCALA '13, pages 3:1–3:10. ACM, 2013.

[2] A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012.

[3] J. Cheney and C. Urban. Nominal logic programming. *ACM Transactions on Programming Languages and Systems*, 30(5):26:1–26:47, Sept. 2008.

[4] R. Cox, T. Bergan, A. T. Clements, F. Kaashoek, and E. Kohler. Xoc, an extension-oriented compiler for systems programming. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 244–254. ACM, 2008.

[5] D. de Rauglaudre. Camlp4 reference manual, 2003.

[6] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: library-based syntactic language extensibility. In *Proceedings of the 2011 ACM International Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '11, pages 391–406. ACM, 2011.

[7] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 7:1–7:8. ACM, 2012.

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

[9] D. Fisher and O. Shivers. Building language towers with Ziggurat. *Journal of Functional Programming*, 18(5–6):707–780, Sept. 2008.

[10] M. Flatt. Creating languages in Racket. *Communications of the ACM*, 55(1):48–56, Jan. 2012.

[11] S. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 74–85. ACM, 2001.

[12] G. Giorgidze and H. Nilsson. Embedding a functional hybrid modelling language in Haskell. In *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*, IFL '08, pages 138–155. Springer, 2008.

[13] J. Gosling, B. Joy, G. L. Steele, Jr., G. Bracha, and A. Buckley. *The Java® Language Specification – Java SE 8 Edition*. Oracle America, Inc., 2014.

[14] S. Grewe, S. Erdweg, P. Wittmann, and M. Mezini. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *Proceedings of Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2015. ACM, 2015. to appear.

[15] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2013.

[16] R. Harper and D. R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4-5):613–673, July 2007.

[17] D. Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, Massachusetts, 2010.

[18] F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. In *Proceedings of 18th International Conference on Functional Programming*, ICFP '13, pages 331–342. ACM, 2013.

[19] G. Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, Haskell '07, pages 73–82. ACM, 2007.

[20] G. Mainland. Explicitly heterogeneous metaprogramming with metahaskell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 311–322. ACM, 2012.

[21] Marlow, Simon. Haskell 2010 – language report, June 2010.

[22] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, 2008.

[23] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 105–130. Springer, 2014.

[24] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[25] C. Schürmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In *Proceedings of International Conference on Automated Deduction*, volume 1421 of *LNCS*, pages 286–300. Springer, 1998.

[26] T. Sheard and S. L. Peyton Jones. Template metaprogramming for Haskell. In *Proceedings of the ACM Workshop on Haskell*, Haskell '02, pages 1–16. ACM, 2002.

[27] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 263–274. ACM, Aug. 2003.

[28] P. Stansifer and M. Wand. Romeo: A system for more flexible binding-safe programming. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 53–65. ACM, 2014.

[29] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[30] E. Visser, Z.-E.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 13–26. ACM, 1998.

[31] H. Xi and C. Schürmann. CPS transform for Dependent ML (abstract). *Logic Journal of IGPL*, 9(5):739–754, Sept. 2001.