# Type-sound Syntactic Language Extension

vorgelegt von

Dipl.-Ing.

Florian Lorenzen

aus Bassum

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
– Dr. rer. nat. –

genehmigte Dissertation

Promotionsausschuss

| | |
|---|---|
| Vorsitzender: | Prof. Dr. rer. nat. Rolf Niedermeier |
| Berichtende: | Prof. Dr. rer. nat. Peter Pepper |
| | Prof. Dr.-Ing. Mira Mezini |
| | Prof. Dr.-Ing. Uwe Nestmann |

Tag der wissenschaftlichen Aussprache: 25. September 2015

Berlin 2015

D 83

## ZUSAMMENFASSUNG

Syntaktische Erweiterungen von Programmiersprachen ermöglichen einen hohen Grad an Abstraktion, verringern die Notwendigkeit sich wiederholender oder gleichender Code-Fragmente, passen Sprachen an spezifische Anwendungsfelder an und sind ein essenzielles Werkzeug um die Software-Komplexität zu beherrschen und den Entwicklungsprozess produktiver zu gestalten. Syntaktische Spracherweiterungen werden üblicherweise durch eine Transformation *(desugaring)* zur Compile-Zeit in die Basissprache implementiert. Damit sie eine tragfähige Abstraktion bilden, darf der generierte Code keine Typfehler enthalten. Typfehler dieser Art bilden eine starke Einschränkung der Benutzbarkeit syntaktischer Erweiterungen: Sie werden relativ zum generierten Code gemeldet, was zu einer unvollständigen Abstraktion führt, sie sind häufig schwer zu finden und zu beheben und es ist oft unklar, ob die Definitions- oder die Anwendungsstelle einer Erweiterung fehlerhaft ist.

In dieser Arbeit entwicklen wir das System und die Methode SoundX (Sound eXtensions) zur syntaktischen Spracherweiterung statisch typisierter Programmiersprachen. SoundX ist sprachunabhängig und lehnt sich an den Sugar*-Ansatz „Erweiterungen als Bibliotheken“ an [ER13]. SoundX ermöglicht, einen syntaktisch erweiterbaren Dialekt aus der formalen Definition einer Basissprache abzuleiten. Die Definition einer Basissprache umfasst die kontextfreie Syntax und das Typsystem dieser Basissprache. Dabei wird das Typsystem durch induktiv definierte Relationen mittels Inferenzregeln spezifiziert. Die Definition einer Erweiterung besteht aus zusätzlichen Syntaxdeklarationen, Transformationsregeln, die die Erweiterung in die Basissprache übersetzen, sowie neuen Typregeln, die als Schnittstelle der Erweiterung fungieren. Erweiterungen können jede syntaktische Sorte der Basissprache erweitern, beispielsweise Terme, Typen oder Deklarationen. SoundX verwendet die Typregeln der Basissprache und der Erweiterungen zur Typanalyse bevor das Programm transformiert wird. Auf diese Art und Weise werden Typfehler relativ zum Originalprogramm gemeldet und die Transformation hat Typinformation für die Codegenerierung zur Verfügung. Dadurch werden Erweiterungen ermöglicht, die Typinferenz benötigen, und die Ausdrucksstärke wird im Vergleich zu rein syntaktischen Transformationen signifikant verbessert. Zur Nutzung von Typinformationen im Codegenerator bildet SoundX Typableitung statt reinen Ausdrücken der erweiterten Sprache in Typableitungen der Basissprache ab, wobei ein neuartiger ableitungsbasierter Algorithmus eingesetzt wird.

SoundX garantiert, dass das Resultat der Transformation einer Erweiterung wohltypisiert ist. Zu diesem Zweck wird automatisch verifiziert, dass die Transformationsregeln und die Typregeln im Hinblick aufeinander korrekt sind. Wir beweisen ein *Preservation*-Theorem, das ausdrückt, dass die SoundX Verifikationsprozedur hinreichend ist, um sicherzustellen, dass

eine gültige Typableitung in der erweiterten Sprache in eine gültige Typableitung der Basissprache übersetzt wird. Weiterhin charakterisieren wir mittels eines *Progress*-Theorems, unter welchen Umständen solch eine Ableitung als Resultat existiert. Nach dem „Erweiterungen als Bibliotheken“ Paradigma wird die Sichtbarkeit von Erweiterungen über das Modulsystem kontrolliert und Erweiterungen werden per Import aktiviert. Unabhängig voneinander definierte Erweiterungen werden beim Importieren komponiert und es ist möglich, dass eine Erweiterung von einer anderen Erweiterung inkrementell erweitert [EGR12] wird, indem erstere als Zielsprache genutzt wird. Die SoundX Verifikationsprozedur verifiziert Erweiterungen modular, so dass es nicht nötig ist, die Komposition von Erweiterungen erneut zu verifizieren.

SoundX ist als ein Plugin für das Sugar* Framework implementiert, das eine Integration in die Eclipse-Entwicklungsumgebung bereitstellt. Die Implementierung enthält eine pragmatisch inspirierte aber fundierte Lösung zum praktisch relevanten Problem der Erzeugung frischer Namen während der Codegenerierung sowie eine Variation des Resolutionsalgorithmus, um einen Ableitungsbaum explizit aufzubauen. Zusätzlich enthält sie ein allgemeines und typsystemunabhängiges Verfahren, um Typfehler zu lokalisieren und zu melden, das in der Praxis angemessene und verständliche Fehlermeldungen liefert. Wir demonstrieren die Anwendbarkeit und die Ausdrucksstärke von SoundX durch mehrere Fallstudien, bei denen wir eine auf dem einfach typisierten $\lambda$-Kalkül basierende Basissprache nutzen.

## ABSTRACT

Syntactic programming language extension provides a high level of abstraction, considerably reduces boilerplate code, adapts a programming language to a specific application area, and is an essential tool to manage complexity and to increase the productivity of the software development process. Syntactic language extensions are commonly implemented by rewriting or *desugaring* them into code of the base language at compile time. To form a sustainable abstraction, the desugaring of extended code must not introduce type errors in the generated code. These type errors are a serious threat to the usability of syntactic abstraction: they are reported in terms of the generated code leading to a leaky abstraction, they are often hard to find, and it is often unclear if the use site or the definition site of the extension is defective.

In this thesis, we develop the language extension system and method SoundX (Sound eXtensions) for statically typed programming languages. SoundX is language independent and follows the "extensions as libraries" approach of Sugar* [ER13]. It derives a syntactically extensible language dialect from a formal base language definition. A base language definition comprises the context-free syntax and the type system of the base language. The type system is specified by inductively defined judgements using inference rule schemata. Extensions consist of additional syntax, the desugaring into the base language, and new typing rules for the extended code that act as an interface of the extension. Extensions are free to add new phrases to all syntactic sorts of the base language like terms, types, or declarations. SoundX uses the typing rules of the base language and the extension to type check the extended program prior to desugaring. In this way, type errors are reported relative to the original sugared program and the desugaring can employ type information during the code generation. This allows the implementation of extensions that require type inference and significantly increases the expressiveness of SoundX compared to purely syntactical transformations. To process the type information in the code generator, SoundX maps typing derivations instead of plain expressions of the extended language to typing derivations of the base language using a novel derivation-centred desugaring procedure.

SoundX guarantees that the desugaring of an extension results in well-typed code. To this end, it automatically verifies that the desugarings and the typing rules are sound with respect to each other. We formally prove that the SoundX verification procedure is sufficient to ensure that a valid derivation in the extended language is mapped to a valid derivation in the base language by establishing a Preservation theorem and characterise under which conditions such a resulting derivation exists using a Progress theorem. According to the "extensions as libraries" paradigm, extensions are scoped by the module system and activated by importing them. Inde-

pendently defined extension are composed on import and it is possible to use one extension as the target of another extension enabling *incremental* extension [EGR12]. The SoundX verification procedure verifies an extension modularly and it is not necessary to re-verify the composition of extensions.

SoundX is implemented as a plugin for the Sugar* framework providing an integration into the Eclipse IDE. The implementation contains a pragmatically inspired but principled solution to the practically relevant problem of fresh name generation during code generation and a variation of the resolution algorithm to obtain an explicit representation of a derivation tree. Moreover, it includes a general and type system independent procedure to locate and report type errors, which returns adequate and understandable error message in practice. We demonstrate the applicability and expressiveness of SoundX by several case studies using a base language built on the Simply-Typed $\lambda$-Calculus.

## ACKNOWLEDGEMENTS

I thank my supervisor Peter Pepper for his warm and friendly encouragement throughout the ups and downs that finally lead to this thesis and for providing a good working environment. I am deeply indebted to Peter for regularly challenging me to think unconventionally and his unconditional support in following my own ideas.

I have met Sebastian Erdweg in October 2012 in Marburg and our collaboration has been very fruitful ever since. Sebastian taught me many things, especially about scientific presentation, and constantly pushed me to squeeze out better results.

Christoph Höger has been sitting in the office next door since 2010 and we lead many interesting, inspiring, and also entertaining discussions about programming languages, life, and the rest. Christoph always had time to provide help and many solutions and results in this thesis emerged from brainstorming with him.

Finally, I am deeply grateful to my partner Judith Rohloff, for her endless support, both technically but especially mentally. Judith regularly forced me to clear my mind, provided the time necessary for this work, in particular since our son Jakob was born, and never lost confidence that all would end well. Thank you!

A precursor of this work is already published:

| | |
|---|---|
| Principal author | Florian Lorenzen |
| Authors | Florian Lorenzen and Sebastian Erdweg |
| Title | Modular and automated type-soundness verification for language extensions |
| Book title | Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming |
| Series | ICFP '13 |
| Pages | 331–342 |
| Year | 2013 |
| Publisher | ACM |

# CONTENTS

# 1 INTRODUCTION

In 1964, Peter J. Landin coined the term "syntactic sugar" to characterise certain forms of a programming language as "syntactic variants" of others [Lan64]. Two years later, he rephrased this as

> "Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things." [Lan66]

Other names for syntactic sugar are "paraphrase" [Sta75] or "derived form" [Pie02], which both underline that it expresses things in terms of other things and that we can, in theory, dispense with it. But in the practice of software development syntactic sugar is an essential tool to manage complexity, to abstract from low-level implementation details, to reduce boilerplate code, and to improve the readability and maintainability of a code base. Syntactic sugar ranges from very simple syntactic variations to full-blown embedded languages, for example where-clauses [Lan64], Java's enhanced for-statement [GJS+14], Haskell's do-notation [Mar10], PLT Redex [FFF09], embedded XML in Scala [OSV08], or the functional-hybrid modelling language Hydra [GN08].

The syntax of a programming language constitutes the interface to the computer that we as programmers use on a day to day basis. Syntactic sugar enriches this interface with concise and application-specific notations that can be used as easily as predefined notations, especially with respect to binding structures and control flow. It is an essential answer to Steele's call that "Parts of the language must be designed to help the task of growth" and that "new words defined by a library should look just like primitives of the language" [Ste99].

Syntactic sugar is implemented by translating or *desugaring* it to more basic code at compile time. The desugaring process can be hard-wired into the compiler limiting syntactic sugar to some predefined set. Only the authors of the compiler can add new or modify existing syntactic sugar. But the idea of *extensible languages*, which empower the regular programmer to add new syntactic sugar, is around since about 50 years [CCJ69] and many programming languages are equipped with extension facilities like macros or templates.

In general, the implementation of an extension consists of a syntax description and the desugaring into primitive code. The compiler executes all the desugarings at compile time to obtain a desugared program ready for machine code generation or interpretation.

In this thesis, we explore the syntactic extension of statically typed programming languages. In this setting type errors in the desugared code represent a serious threat to the usability of extensions. These type errors can either arise due to an incorrect use or due to an incorrect definition of an extension. We propose the extension system SoundX (Sound eXtensions) to develop type-sound extensions which helps the extension implementor to define correct extensions and the client programmer to apply them correctly. SoundX derives an extensible language from a base language and provides a compiler for the extensible dialect. The extensible language is equipped with features to define syntactic extensions to all elements of the base language like terms, types, or declarations. The compiler provided by SoundX checks the definition of an extension and verifies that the desugaring always generates well-typed code according to the typing rules of the base language. Consequently, there will never be any type errors in the generated code, but ill-typed input is detected upfront in the original (sugared) program. An extensible language obtained by SoundX is a language with a static typing discipline in which extension definitions are also subject to a static verification discipline. Like static typing which only permits the execution of well-typed programs, SoundX only permits the desugaring of type-sound extensions. In reminiscence of Milner's famous slogan "Well-Typed Expressions Do Not Go Wrong" [Mil78] we summarise the essence of SoundX as *Type-sound extensions do not get ill-typed*.

Other approaches to syntactic extensions or macro systems for statically typed languages, most notably MacroML [GST01] and SafeGen [HZS05], have been proposed since Steele's and Gabriel's remark that

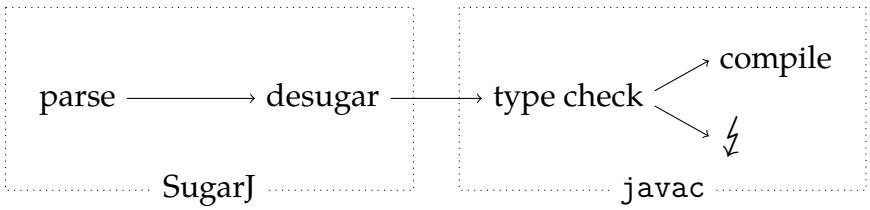
> "Nearly everyone agreed that macro facilities were invaluable in principle and in practice but looked down upon each particular instance as a sort of shameful family secret. If only The Right Thing could be found!" [SG93]

but none of them is language-independent nor do they allow a flexible syntactic extension of all elements of the base language. We therefore believe that SoundX is a significant step towards "The Right Thing" for extensible statically typed languages.

## 1.1 MOTIVATION

A typical approach to implement extensible statically typed languages is to parse the program, apply all active desugarings, and type check the desugared code. Since only desugared programs that pass the type checker are ever executed, such a system is type-sound. TypedRacket macros, SugarJ, Haskell quasiquotations, Scala macros, or OCaml extension points are

practical examples of this approach. The following picture sketches the compilation pipeline for the example of SugarJ [ERKO11], a modularly extensible dialect of Java:




The SugarJ frontend provides an extensible parser that recognises Java's syntax plus the syntax of the imported extensions. The resulting abstract syntax tree is transformed into plain Java using the desugarings of the imported extensions. The output of the SugarJ frontend is passed to the Java compiler which performs the type checking. If the desugared program is well-typed the program is compiled, otherwise the Java compiler stops and emits some error messages. The problems that we address in this thesis arise in the second, erroneous case:

P1 The error messages are relative to the generated code and not relative to the original code. This exposes implementation details of the syntactic extension and violates the abstraction that it is intended to establish.

P2 To resolve the type error a manual inspection of the generated code may be necessary which is a tedious and time consuming task in many cases, especially if the extension is provided by a third-party library.

P3 A type error in the generated code may be either due to mistakes in the desugarings (the definition site of the extension) or due to mistakes in the code employing the extension (the use site). To clarify the situation, it is necessary to inspect the desugarings, that is, the implementation of the extension, to extract the implicit contract and to check if the client code satisfies that contract.

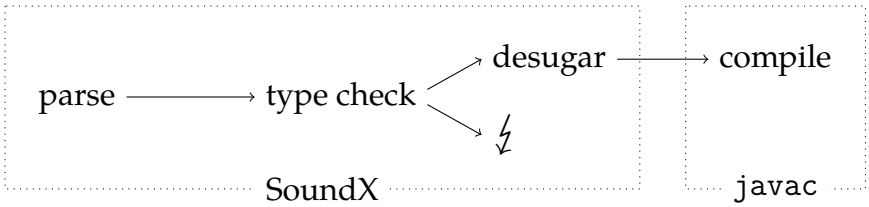
Rather than experiencing these problems, we prefer error messages referring to the sugared code and a static check detecting mistakes in the definition of an extension. Such a static check would provide machine assistance for the definition of extensions much like we are accustomed to with respect to the definition of functions. A function (or a method or a procedure) definition is type-checked independent of its call sites. Call sites of a function can be checked independently of the implementation of the function only using signature (or interface) information.

We tackle the issues P1 and P2 by changing the order of desugaring and type checking. If the code is type-checked before it is desugared, type errors are reported relative to the original program and it is not necessary to inspect the generated code. To resolve P3 we have to check that the generated code

is well-typed if the use site employs the extension in the intended way. To realise these changes we have to add another central idea:

Extension definitions include their own typing rules.

Typing rules declare how an extension should be used. They act as an interface explicitly describing the extension's contract. Using the analogy of function and extension definitions, typing rules correspond to function signatures. We change the front-end such that not only the parser is extensible but also the type checker. This extensible type checker analyses extended code according to its declared typing rules. A sugared program is only desugared if it is well-typed and we end up with the following SoundX compilation pipeline:




This reversal of the type checking and the desugaring pass is only valid if the desugaring does not produce type errors. More precisely, for a base language $B$, an extension $X$, and a program $p$ in the extended language, the following proposition must hold:

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

Fortunately, using the typing rules of an extension we can automatically verify that the desugaring and the typing match each other in the following sense: if the client code satisfies the typing rules, the desugaring generates well-typed code. Extensions satisfying this criterion are *sound* and, as we will see later, soundness is sufficient to establish the proposition given above.

SoundX only permits the activation of sound extensions. Consequently, we resolve P3 by accounting the use site responsible for an error. If the defect was due to the definition of the extension it could not have been activated in the first place.

A very important additional consequence of the rearrangement is that the desugaring has all the information available that the type checker extracts from the program. This enables more expressive extensions since the desugaring can employ this context information. The extensions obtained by exploiting context information in the desugaring are exactly those that Pierce describes as "a little less derived": their "evaluation behaviour" can be derived by desugaring but their "typing behaviour" has to be built in [Pie02, 11.5]. We claim that desugarings with access to context information are crucial for the practical usefulness of our approach.

## 1.2 EXAMPLE: FOR-COMPREHENSIONS

We illustrate the problematic situations and our approach to resolve them with a small example: we extend Java with Scala-like for-comprehensions. For-comprehensions are similar to the enhanced for-statement but more expressive since they allow several bindings and also conditions. Here is an example of a possible adaption of for-comprehensions to Java:

```
for(Integer n ← new IntegerRange(1,3);                    [1.1]
    Integer m ← new IntegerRange(1,n); if m < n)
    System.out.println(n + "," + m);
```

Output:

```
2,1
3,1
3,2
```

The right-hand side of a generator, for example `new IntegerRange(1,3)`, must implement the `Iterable<T>` interface. We may also include a guard like `if m < n` in the second line which skips the corresponding iteration if the condition is not satisfied. In contrast to Scala, we annotate the left-hand side variable of a generator with a type as usual in Java.

To implement Scala-style for-comprehensions in SugarJ (and also in SoundX), we declare new syntactic productions using SDF2 [Vis97]. SDF2 is similar to EBNF but the left- and right-hand sides of productions are exchanged and terminal symbols are enclosed in ".

```
"for" "(" Gen Enums ")" Stm -> Stm                        [1.2]

Type ID "←" Expr -> Gen

                 -> Enums
";" Enum Enums -> Enums

"if" Expr -> Enum
Gen       -> Enum
```

We define the desugaring into Java by a set of rewrite rules. We use the SoundX notation, which is slightly different from SugarJ, especially with respect to meta-variables and concrete syntax. To make the translation as simple as possible we convert a for-comprehension into an enhanced for-statement:

```
for(T x ← e) s          ~~~> for(T x : e) s               [1.3]
for(T x ← e; enums) s ~~~> for(T x : e) for(enums) s
for(if e) s             ~~~> if(e) s
for(if e; enums) s      ~~~> if(e) for(enums) s
```

A desugaring consists of a left-hand pattern and a right-hand expression. During rewriting, an expression of the input that matches a left-hand pattern

is replaced by the right-hand expression. Desugarings are applied iteratively until a fixed point is reached.

For each possible form of a comprehension, we implement one desugaring. The first one simply rewrites a single generator comprehension into an enhanced for-statement. The second desugaring rewrites a generator followed by one or more enumerators into an enhanced for-statement with a residual comprehension as its body. This residual comprehension implements a recursive desugaring and it is desugared in the following iterations. The third and fourth desugarings are similar: they rewrite a condition into an if-statement. Here is the desugared code for Example [1.1]:

```
for(Integer n : new IntegerRange(1,3))                    [1.4]
  for(Integer m : new IntegerRange(1,n))
    if(m < n)
      System.out.print(n + "," + m);
```

We now examine the problematic situations. Suppose, we are mistaken and neglect that the class `Integer` does not implement the `Iterable<Integer>` interface. Hence, we pass this input code to SugarJ:

```
for(Integer n ← new IntegerRange(1,3);                    [1.5]
    Integer m ← new Integer(n); if m < n)
    System.out.println(n + "," + m);
```

SugarJ translates this input to plain Java and the Java compiler responds:

```
ForComp.java:30: error: for-each not applicable to        [1.6]
                        expression type
          for(Integer m : new Integer(n))
                          ^
required: array or java.lang.Iterable
found:    Integer
```

This error is an instance of P1 and P2. The error message contains the generated code, an enhanced for-statement, which is not part of the original sugared program. The abstraction leaks, since its implementation appears in the error message. To understand and fix the error, we have to mentally reverse the desugaring to identify that the code `new Integer(n)` must be changed. Of course, this is not really a challenge for small extensions like for-comprehensions but for more involved extension it quickly gets intractable.

To illustrate P3 we change the first desugaring into the following line:

```
for(T x ← e) s ~~~> for(T x : e.iterator()) s             [1.7]
```

This implementation is defective because the implementation of the enhanced for-statement itself calls the `iterator` method. Feeding the correct code of Example [1.1] with the wrong desugarings into SugarJ provokes `javac` to reply

```
ForComp.java:29: error: for-each not applicable to          [1.8]
                        expression type
          for(Integer m : new IntegerRange(1,n).iterator())
                                                          ^
required: array or java.lang.Iterable
found:    Iterator<Integer>
```

To fix this error, we first have to understand that it is not caused by the application but by the faulty definition of the extension. We have to inspect the desugarings to find the problematic invocation of the `iterator` method.

To apply our solution, we have to declare the typing rules of for-comprehensions. To this end, we assume that the type system of the base language Java has been defined by two judgements: `Γ ⊢ e : T` asserts that the expression `e` has type `T` in the environment `Γ` and `Γ ⊢ s` asserts that the statement `s` is a well-formed statement in the environment `Γ`. We extend the definition of the latter judgement by four inference rules, one for each form of a comprehension:

```
CompGen:                                                    [1.9]
  (Γ ⊢ e : Iterable<T>) (Γ,x:T ⊢ s)
  ---------------------------------
         Γ ⊢ for(T x ← e) s
CompGenEnums:
  (Γ ⊢ e : Iterable<T>) (Γ,x:T ⊢ for(enums) s)
  -------------------------------------------
           Γ ⊢ for(T x ← e; enums) s
CompIf:
  (Γ ⊢ e : boolean) (Γ ⊢ s)
  -------------------------
      Γ ⊢ for(if e) s
CompIfEnums:
  (Γ ⊢ e : boolean) (Γ ⊢ for(enums) s)
  -----------------------------------
        Γ ⊢ for(if e; enums) s
```

These rules clearly state the intended typing and scoping behaviour of for-comprehensions. Particularly, the first premises of the rules `CompGen` and `CompGenEnums` guarantee that the right-hand side of a generator implements the `Iterable<T>` interface. Consequently, the extended type checker will report that `new Integer(n)` does not implement this interface for Example [1.5].

With these rules, the soundness verification procedure of SoundX is also able to detect that the wrong desugaring of [1.7] does not match the typing rule `CompGen`. The details of the verification are thoroughly explained in Section 3.4 but, basically, the reason is that it is impossible to establish `Γ ⊢ for(T x : e.iterator()) s`: By the typing rules of the enhanced for-statement the expression must implement `Iterable<T>` but the result type of the `iterator` method is `Iterator<T>`, which does not implement this interface.

Since with SoundX, the type check is performed prior to desugaring,

we can make for-comprehensions "a little less derived." We omit the type annotations of the left-hand sides as they are inferred by the type checker anyway. That is, we switch from desugarings that are applied to expressions or statements to desugarings that are applied to the result of the type checker. The result of the type checker is a derivation tree constructed according to the typing rules of the base language Java and the additional rules from the extensions. At ICFP'13, Philip Wadler suggested to define such a kind of desugaring with a typing rule as its left-hand side instead of a plain statement or expression. We slightly generalise this idea and allow to match a fragment of a derivation tree.

With desugarings that act on typing derivations instead of plain statements we design a new version of for-comprehensions where the type annotation of the bound variables is dropped. This leads to the following syntax for a generator:

```
ID "←" Expr -> Gen                                              [1.10]
```

The new versions of the typing rules are identical to the old versions of Example [1.9] except for the missing type annotations. Here is the new rule `CompGen`, which handles a single generator; `CompGenEnums` is adapted in a similar fashion:

```
CompGen:                                                        [1.11]
  (Γ ⊢ e : Iterable<T>) (Γ,x:T ⊢ s)
---------------------------------
        Γ ⊢ for(x ← e) s
```

We define the desugaring of a single generator with a left-hand pattern that matches the node of a derivation tree. The derivation tree contains the necessary information to fill in the type annotation in the code of the right-hand side: the type parameter `T` of `Iterable` is inferred by the type checker and used as the type of the bound variable `x` in the resulting enhanced for-statement.

```
(Γ ⊢ e : Iterable<T>) (Γ,x:T ⊢ s)                               [1.12]
---------------------------------
      Γ ⊢ [ for(x ← e) s ]
~~~> for(T x : e) s
```

Such a desugaring rewrites the code enclosed in `[, ]` if it appears in the surrounding derivation tree described by the pattern. This tree pattern guards the applicability of the desugaring and we call it a *guarded desugaring*. The simpler desugarings of Example [1.3] are called *universal* since they can be applied unconditionally.

## 1.3 SOLUTION

In order to reach our goal of statically provable type-sound extensions with access to context information we develop an automatic verification and a desugaring procedure that transforms derivation trees rather than plain

expressions. Moreover, we need an extensible type checker and facilities to define the base language and the extensions. We give a sketch of the techniques and concepts involved in our realisation of these demands.

SoundX is based on the Sugar* framework [ER13] which is a plugin-based system that implements the "language extensions as libraries" [Erd13] approach for language extensions. With Sugar*, a modularly extensible language can be derived from a non-extensible base language by implementing a language plugin. Such a plugin mainly consists of the base language's syntax definition and a description of its modular structure. The aforementioned SugarJ is the Java plugin for Sugar*. An extension or *sugar library* consists of additional syntax definitions and desugarings written as Stratego rewrite rules [VBT98] and may also contain static analyses in that are run prior to the desugaring phase.

SoundX is implemented as a Sugar* plugin which, given a base language definition, derives a type-sound syntactically extensible language. Such a SoundX base language definition is an amendment of a Sugar* language plugin: it comprises the syntax, modular structure, *and typing rules* of the base language. Similar, a SoundX extension definition is an amendment of a Sugar* extension with typing rules. The SoundX plugin uses the typing rules to perform a type check and the extension verification in the Sugar* static analysis phase. The derivation desugaring procedure is implemented as Sugar* rewrite rules and executed in the desugaring phase.

In the remainder of this section we use the Simply-Typed $\lambda$-Calculus with natural numbers $\lambda_{\rightarrow}$ as an example for a base language and a pair type as a representative of possible extensions.

### 1.3.1 *Base language definition*

A base language definition in SoundX specifies the context-free syntax using SDF2 and the context conditions by inductively defined judgements. We neglect the aspects of modularity here. They are discussed thoroughly in Chapter 4.

The syntax descriptions of many programming languages, for example Java, C, C#, or SQL, is available as an SDF2 definition on the WWW at `https://strategoxt.org/Sdf/SdfGrammars` and, with small adaptions, they can be reused in a SoundX base language definition. For our toy language $\lambda_{\rightarrow}$, we use the following short SDF2 definition:

```
base language definition SimpleTypes                [1.13]

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

```
context-free syntax                                    [1.13 cont'd]
  ID                          -> Term
  "λ" ID ":" Type "." Term -> Term
  Term Term                   -> Term
  NUM                         -> Term
  Term "+" Term               -> Term
  "(" Term ")"                -> Term

  "Nat"          -> Type
  Type "→" Type -> Type
  "(" Type ")"   -> Type
```

In the `lexical syntax` section, we define the lexical sorts of identifiers `ID` and natural numbers `NUM`, respectively. The following `context-free syntax` section defines the syntax of terms including variables, abstractions, applications, number constants, additions, and parenthesised terms. The third `context-free syntax` section defines the type of numbers `Nat` and the function type constructor `→`.

In this section, we ignore certain aspects of concrete syntax like precedence and associativity. SDF2 offers various mechanisms, for example associativity annotations and priorities, to resolve these ambiguities, which can be included in SoundX definitions.

For the inductive definition of the typing judgements, we need meta-variables that range over the different syntactic sorts like `ID` or `Term`. It is common practice in the programming language community to have a convention for the naming of meta-variables. In SoundX, such a convention is enforced by meta-variable declarations:

```
variables                                                  [1.14]
  "x" [a-zA-Z0-9]* -> ID
  "y" [a-zA-Z0-9]* -> ID
  "n" [a-zA-Z0-9]* -> NUM
  "t" [a-zA-Z0-9]* -> Term
  "S" [a-zA-Z0-9]* -> Type
  "T" [a-zA-Z0-9]* -> Type
```

According to this declaration, meta-variables starting with `x` or `y` range over identifiers, meta-variables starting with `n` range over number constants, meta-variables starting with `t` range over terms, and meta-variables starting with `S` or `T` range over types.

For $\lambda_{\rightarrow}$, we need the typing judgement `Γ ⊢ t : T` which establishes that a term `t` is well-typed with respect to the environment `Γ` and has type `T`. In order to define this judgement we need the syntax of environments and meta-variables ranging over environments:

```
context-free syntax                                    [1.15]
  "∅"                    -> Env
  Env "," ID ":" Type -> Env

variables
  "Γ" [a-zA-Z0-9]* -> Env
```

Meta-variables starting with the letter Γ range over environments. We write the empty environment as ∅ and the phrase `Γ,x:T` adds the binding "`x` has type `T`" to an existing environment Γ.

SoundX requires a declaration of the different judgement forms similar to the context-free productions for expressions. The next declaration states the arity and syntax of the typing judgement and the variable lookup judgement:

```
judgement forms                                        [1.16]
  { Env "⊢" Term ":" Type }
  { ID ":" Type "∈" Env }
```

Judgements are defined inductively by a set of inference rule schemata. We also use the phrases "inference rule" or just "rule" as synonyms for inference rule scheme, although this is technically not quite correct.

As usual, an inference rule consists of a list of premises and a conclusion. Premises and conclusions are instances of the judgement forms previously defined and in general contain meta-variables. SoundX imitates the pencil-and-paper dividing-bar notation. We transfer the textbook definition of the typing judgement of $\lambda_{\rightarrow}$, for example from [Pie02, Har13], into SoundX definitions:

```
inductive definitions                                  [1.17]
  Lookup:
    ------------
    x:T ∈ Γ,x:T
  LookupSkip:
    (x≠y) (x:T ∈ Γ)
    ---------------
      x:T ∈ Γ,y:S
  Var:
     x:T ∈ Γ
    ---------
    Γ ⊢ x : T
  Abs:
       Γ,x:T1 ⊢ t2 : T2
    -----------------------
    Γ ⊢ λx:T1. t2 : T1 → T2
  App:
    (Γ ⊢ t1 : T11 → T12) (Γ ⊢ t2 : T11)
    -----------------------------------
              Γ ⊢ t1 t2 : T12
```

```
Nat:                                                    [1.17 cont'd]
  ----------
  Γ ⊢ n : Nat
Add:
  (Γ ⊢ t1 : Nat) (Γ ⊢ t2 : Nat)
  ------------------------------
       Γ ⊢ t1 + t2 : Nat
```

An inference rule may have zero premises, like `Lookup`, to form an axiom scheme. If several premises are given on the same line, they have to be individually parenthesised to avoid ambiguity like in `App` or `Add`. If a single premise spans several lines the indentation-sensitive offside rule [Lan66] is applied for disambiguation.

### 1.3.2 *Extension definition*

Similar to base language definitions, an extension consists of context-free productions and inference rules to specify the syntax and context conditions of the extension. But they do not declare additional judgement forms. That is, the context conditions of the extension must be defined in terms of the judgement forms of the base language. The dynamic semantics is expressed by desugarings into the base language or other imported extensions.

We consider the addition of pairs to $\lambda_{\to}$ as an example where we extend the syntax of terms and types. Since $\lambda_{\to}$ does not suffice to encode pairs with components of different types, we only add a type `Pair T` to account for pairs of values of the same type:

```
context-free syntax                                     [1.18]
  "Pair" Type -> Type

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

Pair creation is written in the usual tuple notation like `(1,2)` and the individual components of a pair are selected by a `.1` or `.2` suffix.

Typing rules are written similarly to typing rules in the base language definition and we begin with the rules for selection:

```
inductive definitions                                   [1.19]
  Fst:
    Γ ⊢ t : Pair T
    --------------
     Γ ⊢ t.1 : T
  Snd:
    Γ ⊢ t : Pair T
    --------------
     Γ ⊢ t.2 : T
```

In the desugaring, we encode a pair as a function expecting a selector function that either returns the first or the second component:

```
desugarings                                                    [1.20]
  { Pair T ~~~> (T → T → T) → T }
```

A selection from a term `t` of type `Pair T` is translated into an application of that term to a selector function `λa:T.λb:T.a` for the first and `λa:T.λb:T.b` for the second component. The variables `a` and `b` are concrete term-level variables of $\lambda_{\to}$, not meta-variables. Here is a first attempt for the desugaring of `t.1`:

```
{ t.1 ~~~> t (λa:T.λb:T.a) }                                   [1.21]
```

The question is how we obtain the type annotation `T` for the bound variables `a` and `b`. The right-hand side of a desugaring may only use the variables introduced on its left-hand side, where `T` is not mentioned.

However, the type is available in a typing derivation of `Γ ⊢ t.1 : T`. For example, in the following derivation of `∅,p:Pair Nat ⊢ p.1 : Nat`, `T` is assigned `Nat`:

```
                                                               [1.22]
  -------------------------Lookup
  p:Pair Nat ∈ ∅,p:Pair Nat
  ---------------------------Var
    ∅,p:Pair Nat ⊢ p : Pair Nat
    ---------------------------Fst
       ∅,p:Pair Nat ⊢ p.1 : Nat
```

SoundX allows to define desugarings whose left-hand side matches a derivation node. These desugarings have access to all the information this derivation node provides. This leads to the following desugarings for selection:

```
desugarings                                                    [1.23]
  { Γ ⊢ t : Pair T
    --------------
    Γ ⊢ [ t.1 ] : T ~~~> t (λa:T.λb:T.a) }
  { Γ ⊢ t : Pair T
    --------------
    Γ ⊢ [ t.2 ] : T ~~~> t (λa:T.λb:T.b) }
```

The right-hand side of such a desugaring may use all meta-variables of the left-hand side. Thus, the necessary type `T` is readily available in this way since the type checker will have inferred it from the type of `t`. The brackets `[` and `]` on the left-hand side indicate which expression is actually rewritten. The brackets must appear in the conclusion and fully enclose one argument of the conclusion's judgement.

We call desugarings that are defined on the typing rule *guarded* desugarings. The desugaring of the expression in square brackets is guarded by the additional context provided by the derivation node it appears in. In contrast, desugarings that are defined solely on an expression are called *universal*. They are applicable everywhere, provided the pattern of their left-hand side matches.

Since the left-hand node of a guarded desugaring is often identical to a typing rule, a desugaring and an inference rule definition can be combined. We use this style for the typing rule and desugaring of pair construction:

```
inductive definitions and desugarings                    [1.24]
  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
```

The typing rule part of the previous definition states that we can form a pair of two terms of the same type `T`.

The desugaring part gives the translation of a pair `(t1,t2)` into a function that expects a selector function `s` as its argument. We have used the $\eta$-expanded form on the right-hand side instead of `λs:T→T→T. s t1 t2`. The reason is that the latter form might capture a free occurrence of the concrete object-level variable `s` in `t1` or `t2`. The $\eta$-expanded form avoids this problem since `s` is neither a free variable of the concrete object-level variables `a` nor `b`. In Section 5.1.3 we describe a pragmatic approach to generate fresh names to replace such ad-hoc solutions.

### 1.3.3 *Desugaring a derivation*

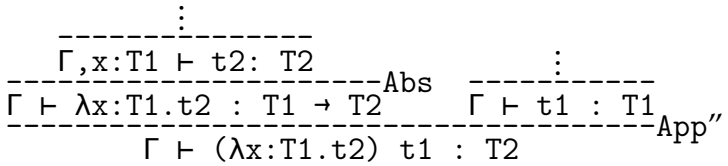
Since guarded desugarings employ information from the typing derivation, the SoundX desugaring procedure transforms derivation trees rather than plain expressions. Pierce describes the general idea in [Pie02, 11.5] using the example of let-expressions, which we quote here. In SoundX notation the syntax, typing rule, and desugaring of let-expressions reads as follows:

```
context-free syntax                                      [1.25]
  "let" ID "=" Term "in" Term -> Term

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

desugarings
  { (Γ ⊢ t1: T1) (Γ,x:T1 ⊢ t2 : T2)
    -------------------------------
    Γ ⊢ [ let x = t1 in t2 ] : T2
    ~~~> (λx:T1. t2) t1 }
```

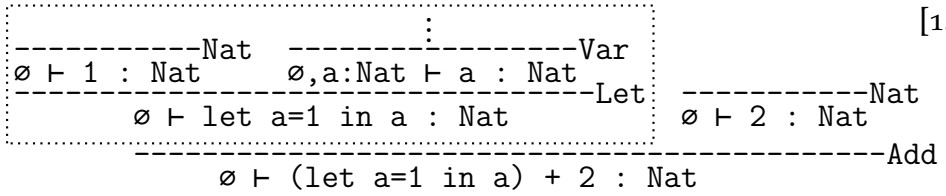
The crucial point in this example is that the desugaring uses the type `T1` from the typing derivation as annotation for the $\lambda$-abstraction. Pierce concludes that

> "we should regard it [the `let` constructor] as a transformation on *typing derivations* […] that maps a derivation involving `let`
>
> ```
>      ⋮           ⋮
> ----------  ----------
> Γ ⊢ t1 : T1  Γ ⊢ t2 : T2
> ------------------------Let
> Γ ⊢ let x=t1 in t2 : T2
> ```

to one using abstraction and application:



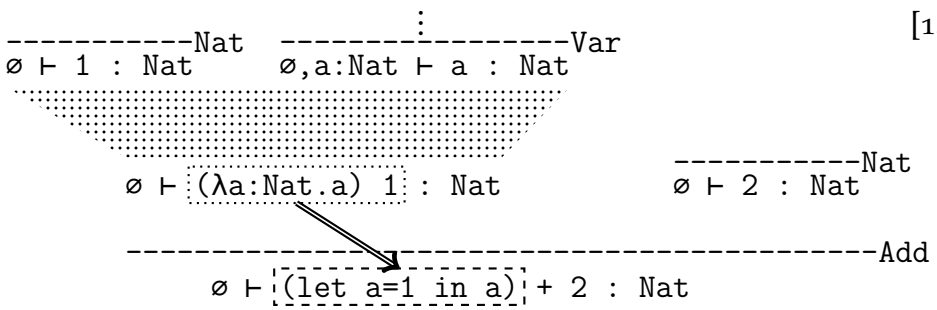

Most of the technical development in this thesis is concerned with developing a procedure that, given a set of universal and guarded desugarings, performs exactly that transformation. We give an outline of the desugaring procedure using Pierce's example. Consider the following derivation tree for the judgement `ø ⊢ (let a=1 in a) + 2 : Nat`:




[1.26]

The important difference to the transformation quoted above is that let appears as a subexpression here. Intuitively, it is clear that the desugaring of the let-expression must be applied to the root of the subtree enclosed in the dotted box: This node is the only one that provides the required context. We cannot apply the let desugaring directly to the root of the derivation as in the previous sketch.

Application of this desugaring replaces `let a=1 in a` by `(λa:Nat. a) 1` and we obtain the following situation:




[1.27]

The result of desugaring the let-expression has to be forwarded to the root of the derivation as suggested by the double arrow. That is, the content of the dashed box must be replaced by the content of the dotted box. At this point, we observe that the derivation desugaring is essentially a transformation proceeding from the leaves to the root of the derivation. Technically, this is a bottom-up transformation; it looks like top-down, though, since we sketch derivation trees with the root at the bottom. Moreover, we have to fill the dotted gap with valid instantiations of typing rules, App and Abs from [1.17] in this case. In summary, derivation desugaring has to master two challenges:

C1 Results of subexpression desugaring have to be forwarded towards the root of the derivation.

C2 The resulting gap in the derivation has to be filled with valid instantiations of typing rules.

We develop the derivation desugaring procedure in detail in Chapter 3 where we also see that certain extensions must be handled by a top-down pass prior to the bottom-up pass.

### 1.3.4 *Verification*

SoundX statically guarantees that desugaring results in a valid derivation. To this end, it verifies the soundness of an extension. The basic principle of the verification procedure is to perform a "dry-run" of the desugaring procedure for each typing rule. A successful dry-run is sufficient to ensure that the challenge C2 can always be met.

To give an impression of the verification procedure we check that our implementation of let-expressions is sound. Verification proceeds in two steps:

1. The typing rule is desugared using all the desugarings of the extension until none of them is applicable anymore. For `Let`, we obtain the desugared rule

$$\frac{\texttt{(Γ ⊢ t1 : T1) (Γ,x:T1 ⊢ t2 : T2)}}{\texttt{Γ ⊢ (λx:T1.t2) t1 : T2}} \qquad [1.28]$$

2. It is checked if the desugared conclusion `Γ ⊢ (λx:T1.t2) t1 : T2` is derivable from the premises `Γ ⊢ t1 : T1` and `Γ,x:T1 ⊢ t2 : T2` by the typing rules of the extension and the base language. Of course, here this is possible by exactly the derivation using `App` and `Abs` that we have quoted at the beginning of the previous section on page 26.

Unsound extensions are rejected in the second step. For example, consider the defective desugaring `let x=t1 in t2 ~~~> t2 t1`. This desugaring leads to the conclusion `Γ ⊢ t2 t1 : T2` which cannot be derived from the premises using the typing rules of $\lambda_{\rightarrow}$. A typing judgement `Γ ⊢ t2 : T1→T2` would be required for this. Conversely, if we forget to add `x` to the typing environment in the second premise of `Let`, the required judgement `Γ ⊢ λx:T1.t2 : T1→T2` cannot be derived. We observe that SoundX checks that the desugaring and the typing rule are sound with respect to each other.

## 1.4 CONTRIBUTIONS AND OUTLINE

The main contribution of this thesis is the design of SoundX, a language-independent mechanism for modular and type-sound syntactic language extension, and its soundness proof. Alongside this core we make various technical and practical contributions necessary towards a usable realisation.

In Chapter 2 we introduce the SoundX meta-language to specify base languages and extensions. The SoundX meta-language is a simple but powerful and principled language definition formalism for the static aspects of a formal language. It is based on context-free grammars for the syntax and inductively defined judgements for the type system. Extensions are defined by desugarings into the base language, and they may also depend on contextual information from the type analysis. In this chapter, we also define the central notion of a *valid derivation* and prove several important properties about it.

Chapter 3 contains our main contribution. To enable desugarings that employ type information we develop a desugaring procedure for derivations. We also present the verification procedure for extensions. We prove that the successful verification of an extension guarantees that the desugaring procedure maps a valid derivation in the extended language into a valid derivation in the base language (Preservation). Technically, the key to make this work is the *forward step*, which propagates transformations from subderivations into the enclosing context. Finally, we analyse under which conditions a successful desugaring is possible and prove a Progress theorem.

We lift the verification and desugaring procedures of Chapter 3 onto the modular level in Chapter 4. In this chapter, we add the definition of module systems to the SoundX meta-language. Based on this definition, we develop the context analysis of a module including verification and composition of extensions, import processing, and type-checking. We refine our desugaring procedure for a composition of extensions to desugar a module. Most importantly, we prove that the individual verification of an extension as developed in the previous chapter is sufficient to ensure a sound desugaring for a composition of extensions.

In the first part of Chapter 5, we briefly describe the technical realisation of our SoundX implementation as an Eclipse plugin using the Sugar* framework. As a prerequisite for the implementation, we present a variation of the resolution algorithm that builds a derivation for a given judgement and we prove that it generates valid derivations. We also describe our solution for two practical problems: the generation of fresh names during desugaring and a general algorithm to identify the source of a type error and to generate an understandable and usable error message. In the second part of this chapter, we present several case studies of sound and unsound extensions to demonstrate the expressiveness of SoundX and how different erroneous,

especially unhygienic, situations are handled.

In the last Chapter 6, we investigate the limits of SoundX, discuss related work, and conclude this thesis. With respect to limitations, we address the aspects of possible nontermination, failures of the forward step, fresh name generation, and $\alpha$-equivalence. Especially for the latter two, we sketch possible directions for future improvements of SoundX.

# 2 BASICS

Type systems are commonly specified by a set of judgements which assert that a program or program fragment is valid in some form depending on the concrete language. For example, the judgement `∅,a:Nat ⊢ 1+a : Nat` of $\lambda_{\rightarrow}$ asserts that the term `1+a` is well-typed under the environment `∅,a:Nat` and has the type `Nat`. These kinds of judgements are usually defined inductively by a set of inference rule schemata and a judgement is valid if and only if it is derivable using these schemata.

The design of SoundX closely follows Christensen's vision of a "programming system" which "includes a *base language* and a *meta-language* [...] to produce a *derived language*" [CCJ69]. The SoundX language to define the syntax, typing rules, and desugarings of extensions is the meta-language and the resulting extended language is the derived language. With respect to the base language, SoundX even exceeds Christensen's vision since the base language is not fixed but can be specified using the SoundX base language definition facilities. That is, the SoundX language includes a meta-level language specification formalism to describe object languages.

To study the type-sound extension of arbitrary base languages and to develop the necessary meta-theory we have to define the SoundX language as an object language itself, which embodies elements from the meta-level of the base language. In this chapter, we recast syntax and inference rule definitions into the SoundX language in an abstract and general style. We review the concept of deriving a judgement by inference rules where we make the derivation tree an explicit object. We prove several properties of valid derivations to prepare for the technical development of the Chapters 3 and 4.

We also use this chapter to introduce most of the notational devices which we employ in this thesis. They are summarised in Appendix A which also contains an overview of the naming conventions and all inductively defined statements. We assign a unique name to each definition and Appendix B provides an index with page references.

## 2.1 SOUNDX LANGUAGE

During the formal study of SoundX we work with an abstract syntax to express all possible base language definitions, extensions, and extended programs. The abstract syntax neglects the concrete syntax of expressions of the base language and the separation into lexical and context-free sorts of SDF2 that we have seen in Section 1.3.1. Especially longer phrases written

in abstract syntax are hard to read but it provides us with an unambiguous notation for all base language definitions and extensions that can be formulated in SoundX.

The abstract syntax assumes a countably infinite set $N$ of atomic names, which are used to identify sorts, constructors, judgements, and inference rules. We use the set of nonempty alphanumeric strings printed in `monospaced` font whenever we need concrete elements of $N$.

We use the following shorthands to write lists: the variable $\vec{\phi}$ stands for a list of elements of $\phi$, $\langle \phi_1, \ldots, \phi_n \rangle$ is an enumeration of the elements $\phi_1$ to $\phi_n$, which we often abbreviate as $\langle \phi_{1..n} \rangle$. We write $\langle \vec{\phi}_1, \vec{\phi}_2 \rangle$ for a concatenation of two lists $\vec{\phi}_1$ and $\vec{\phi}_2$ and also use the comma to add an element $\phi$ to the head $\langle \phi, \vec{\phi} \rangle$ or to the tail $\langle \vec{\phi}, \phi \rangle$ of a list $\vec{\phi}$. As a slight abuse of notation, we write $\phi \in \vec{\phi}$ for list containment where the position of $\phi$ in $\vec{\phi}$ does not matter, and $\phi \notin \vec{\phi}$ for $\phi$ is not contained in $\vec{\phi}$. For two lists $\vec{\phi}_1$ and $\vec{\phi}_2$, we write list inclusion as $\vec{\phi}_1 \sqsubseteq \vec{\phi}_2$. It is an abbreviation for $\phi \in \vec{\phi}_1$ implies $\phi \in \vec{\phi}_2$, that is, the order of elements in $\vec{\phi}_1$ and $\vec{\phi}_2$ is irrelevant.

### 2.1.1 *Syntax definitions*

Instead of a set of context-free productions, the abstract syntax contains a list of constructor arities that assign a name to each syntactic expression. A constructor arity $A = (N : \langle N_{1..n} \rangle \to N_0)$ introduces the constructor name $N$ that builds expressions of sort $N_0$. It takes $n$ arguments where the $i$-th argument must be an expression of sort $N_i$.

For example, we capture the syntax of types by the definition of two constructors `Nat` and `Fun` corresponding to `Nat` and `→` in the concrete syntax:

$$\begin{array}{l} \texttt{"Nat"} \quad\quad\quad\quad \texttt{->}\ \texttt{Type} \mathrel{\hat{=}} \texttt{Nat} : \langle\rangle \quad\quad\quad\quad \to \texttt{Type} \\ \texttt{Type}\ \texttt{"→"}\ \texttt{Type}\ \texttt{->}\ \texttt{Type} \mathrel{\hat{=}} \texttt{Fun} : \langle \texttt{Type}, \texttt{Type} \rangle \to \texttt{Type} \end{array} \qquad [2.1]$$

The correspondence *concrete syntax* $\mathrel{\hat{=}}$ *abstract syntax* in the previous example can also be read as "*concrete syntax* parses to *abstract syntax*."

A list of constructor arities is similar to an algebraic signature [EM85] and the correspondence outlined in the foregoing example is an example of the relationship between a context-free grammar and an algebraic signature [Rus72, HR76].

A list of constructor arities $\vec{A}$ is well-formed, written as $\vec{A}$ ⊱ $\diamond$, if the names of all constructors are different, where $m..n$ is a short-hand for the enumeration $\{m, \ldots, n\}$:

W-Arities
$$\frac{\text{for each } i, j \in 1..n, i \neq j : N_i \neq N_j}{\langle N_1 : \vec{N}_{11} \to N_{12}, \ldots, N_n : \vec{N}_{n1} \to N_{n2} \rangle \text{ ⊱ } \diamond}$$

In the abstract syntax, all syntactic phrases like terms, types, or contexts are subsumed by expressions $E$ following the nomenclature Pierce employs in his textbook [Pie02, 3.1]. Similar to Lisp S-expressions [McC60], an expression is either a meta-variable $V$ or an application of a constructor with the name $N$ to a list of subexpressions $\vec{E}$. In the abstract syntax meta-variables are annotated with their sort, for example `(T : Type)`. During translation from the concrete base language definition to the abstract form, the meta-variable convention is used to complete the sort annotations.

Here are a few examples of expressions from $\lambda_{\to}$, where we introduce the constructors `abs` and `app` for abstraction and application and null-ary constructors like $\texttt{id}_\texttt{a}$ or $\texttt{num}_2$ for identifiers and numbers:

$$\begin{array}{lcl} \texttt{λa:Nat.a} & \hat{=} & \texttt{abs}\langle \texttt{id}_\texttt{a}\langle\rangle, \texttt{Nat}\langle\rangle, \texttt{var}\langle \texttt{id}_\texttt{a}\langle\rangle\rangle\rangle \\ \texttt{f 2} & \hat{=} & \texttt{app}\langle \texttt{var}\langle \texttt{id}_\texttt{f}\langle\rangle\rangle, \texttt{num}\langle \texttt{num}_2\langle\rangle\rangle\rangle \\ \texttt{t n} & \hat{=} & \texttt{app}\langle (\texttt{t : Term}), \texttt{num}\langle (\texttt{n : NUM})\rangle\rangle \end{array} \qquad [2.2]$$

Note that the identifiers `a` and `f` are object level variables, that is, expressions of type `ID`, but `t` and `n` are meta-variables by the convention declared on page 22 and appear as `(t : Term)` and `(n : NUM)` in the abstract syntax.

An expression $E$ is of sort $N$ with respect to a list of constructor arities $\vec{A}$, written $\vec{A} \vdash E : N$, if all constructors used in $E$ are applied to arguments of the sort declared in $\vec{A}$.

WE-Var
$$\frac{}{\vec{A} \vdash (N : N_0) : N_0}$$

WE-Con
$$\frac{(N : \langle N_{1..n}\rangle \to N_0) \in \vec{A} \qquad \text{for each } i \in 1..n : \vec{A} \vdash E_i : N_i}{\vec{A} \vdash N\langle E_{1..n}\rangle : N_0}$$

These well-formedness conditions are the pendant to the set of terms over a signature from universal algebra written as $T_{SIG}(X)$ for a signature $SIG$ and a set of variables $X$ in [EM85].

The following function *vars* returns all the meta-variables of an expression or a list of expressions:

Vars-Expr
$$\begin{aligned} vars(V) &= \{V\} \\ vars(N\ \vec{E}) &= vars(\vec{E}) \\ vars\langle E_{1..n}\rangle &= vars(E_1) \cup \cdots \cup vars(E_n) \end{aligned}$$

### 2.1.2 *Judgements*

Judgements express relationships between expressions. Many different forms of judgements are commonly used in the formalisation of programming languages, like typing, kinding, subtyping, or type equivalence.

A judgement form $F = (N : \vec{N})$ defines the syntax of a judgement and directly corresponds to an entry in the `judgement forms` section, for example [1.16], of a base language definition . It specifies the sorts of the arguments for the judgement $N$ to be $\vec{N}$ and is comparable to the predicate part of a logical signature [EMC+01, BM07]. For $\lambda_{\to}$, we have the following correspondence, where `TJ` abbreviates "typing judgement":

$$\texttt{\{ Env "⊢" Term ":" Type \}} \mathrel{\hat{=}} \texttt{TJ} : \langle \texttt{Env}, \texttt{Term}, \texttt{Type} \rangle \qquad [2.3]$$

Similar to constructor arities, a list of judgement forms is well-formed, written $\vec{F}$ ⊱ ⋄, if the names of all judgements are different:

W-Forms
$$\frac{\text{for each } i, j \in 1..n, i \neq j : N_i \neq N_j}{\langle N_1 : \vec{N}_1, ..., N_n : \vec{N}_n \rangle \text{ ⊱ } \diamond}$$

A concrete judgement $J$ is written as a list of expressions $\vec{E}$ forming the arguments of the judgement followed by the judgements name $N$. This "postfix" notation is borrowed from [Har13] and makes it simple to distinguish between judgements and expressions. Here is a possible typing judgement from $\lambda_{\to}$, both in its concrete and its abstract representation:

$$\texttt{⌀ ⊢ 1 + 2 : Nat} \mathrel{\hat{=}} \left\{ \begin{array}{l} \langle \texttt{empty}\langle\rangle, \texttt{add}\langle \texttt{num}\langle \texttt{num}_1\langle\rangle\rangle, \\ \qquad\qquad \texttt{num}\langle \texttt{num}_2\langle\rangle\rangle, \texttt{Nat}\langle\rangle \rangle\ \texttt{TJ} \end{array} \right\} \qquad [2.4]$$

A judgement must be well-formed with respect to a list of constructor arities and judgement forms, written $\vec{A}; \vec{F}$ ⊱ $J$, similar to the set of well-formed atomic formulas over a logical signature. Since judgements cannot be nested, well-formedness boils down to the requirement that the argument expressions are of the declared sort:

W-Judgement
$$\frac{(N : \langle N_{1..n} \rangle) \in \vec{F} \qquad \text{for each } i \in 1..n : \vec{A} \text{ ⊱ } E_i : N_i}{\vec{A}; \vec{F} \text{ ⊱ } \langle E_{1..n} \rangle\ N}$$

Like for expressions, we also define the function *vars* for judgements and lists of judgements:

Vars-Judg
$$\begin{aligned} vars(\vec{E}\ N) &= vars(\vec{E}) \\ vars\langle J_{1..n} \rangle &= vars(J_1) \cup \cdots \cup vars(J_n) \end{aligned}$$

### 2.1.3 *Inference rules*

Judgements are defined inductively by inference rule schemata. We also often write "inference rule" or just "rule" for short, though this is technically not quite correct. Inference rules consist of a list of premise judgements

and a conclusion judgement. It is common practice to write inference rules in a "dividing-bar" notation with the premises above and the conclusion below the bar. This is also the concrete notation used in SoundX and we have already seen several examples in Section 1.3. In the abstract syntax we write an inference rule $I$ as $(\vec{J} \twoheadrightarrow^{N} J)$ where $\vec{J}$ are the premises, $N$ is the name of the inference rule, and $J$ is its conclusion. Here are two rules from the definition of the typing judgement in their abstract representation. We use $\Gamma = (\Gamma : \texttt{Env})$, $x = (\texttt{x} : \texttt{ID})$, $T = (\texttt{T} : \texttt{Type})$, and so on, as abbreviations for meta-variables.

$$\left\{\begin{array}{l}\texttt{Var:}\\ \texttt{\quad ----------}\\ \texttt{\quad Γ,x:T ⊢ x : T}\end{array}\right\} \quad \widehat{=} \quad \langle\rangle \twoheadrightarrow^{\texttt{Var}} \langle \texttt{cons}\langle \Gamma, x, T\rangle, x, T\rangle\ \texttt{TJ} \qquad [2.5]$$

$$\left\{\begin{array}{l}\texttt{App:}\\ \texttt{\quad Γ ⊢ t1 : T11 → T12}\\ \texttt{\quad Γ ⊢ t2 : T11}\\ \texttt{\quad ----------}\\ \texttt{\quad Γ ⊢ t1 t2 : T12}\end{array}\right\} \quad \widehat{=} \quad \left\{\begin{array}{l}\langle\langle \Gamma, t_1, \texttt{Fun}\langle T_{11}, T_{12}\rangle\rangle\ \texttt{TJ},\\ \langle \Gamma, t_2, T_{11}\rangle\ \texttt{TJ}\rangle\\ \twoheadrightarrow^{\texttt{App}} \langle \Gamma, \texttt{app}\langle t_1, t_2\rangle, T_{12}\rangle\ \texttt{TJ}\end{array}\right\}$$

In general, judgements are defined by a list of inference rules. The inference rules defining a specific judgement may include other judgements in their premises. These dependencies are allowed to be bidirectional leading to mutually dependent judgements.

Inference rules are well-formed with respect to a list of constructor arities and judgement forms if they all have different names and are composed from well-formed judgements:

W-InfRules
$$\frac{\begin{array}{c}\text{for each } i \in 1..n, J \in \langle \vec{J}_{i1}, J_{i2}\rangle : \vec{A}; \vec{F} \ \&\vdash J\\ \text{for each } i, j \in 1..n, i \neq j : N_i \neq N_j\end{array}}{\vec{A}; \vec{F} \ \&\vdash \langle \vec{J}_{11} \twoheadrightarrow^{N_1} J_{12}, \ldots, \vec{J}_{n1} \twoheadrightarrow^{N_n} J_{n2}\rangle}$$

We also augment the *vars* function with a case for inference rules:

Vars-Inf-Rules
$$vars(\vec{J} \twoheadrightarrow^{N} J) = vars(\vec{J}) \cup vars(J)$$

### 2.1.4 *Desugarings*

Apart from syntax definitions and inference rules, extensions also contain desugarings. Desugarings are comparable to rewrite rules like in Stratego [VBT98], for example. But since SoundX provides two kinds of desugarings, namely guarded and universal ones, and the desugaring procedure applying them is very specific, we also introduce specific abstract syntax for them.

A universal desugaring $U$ consists of a left-hand side pattern $E$ and a right-hand side expression $E'$ separated by $\rightsquigarrow$. Here is the pair type desugaring in concrete and abstract syntax:

$$\texttt{Pair T ~~~> (T→T→T) → T} \mathrel{\hat{=}} \left\{\begin{array}{l}\texttt{Pair}\langle T\rangle \rightsquigarrow \\ \texttt{Fun}\langle\texttt{Fun}\langle T, \texttt{Fun}\langle T, T\rangle\rangle, T\rangle\end{array}\right\} \quad [2.6]$$

A guarded desugaring $G$ has a derivation node as its left-hand side which is written $(\vec{J} \Rightarrow (\vec{E}_1, E, \vec{E}_2)\ N)$. As we have seen previously in the examples in Sections 1.2 and 1.3, one argument of the conclusion is marked by `[` and `]` in the concrete syntax. In the abstract syntax, this argument $E$ is given explicitly as illustrated by the following example:

$$\left\{\begin{array}{l}\texttt{Γ ⊢ t : Pair T} \\ \texttt{---------------} \\ \texttt{Γ ⊢ [ t.1 ] : T} \\ \texttt{~~~> t (λa:T.λb:T.a)}\end{array}\right\} \mathrel{\hat{=}} \left\{\begin{array}{l}(\langle\langle\Gamma, t, \texttt{Pair}\langle T\rangle\rangle\ \texttt{TJ}\rangle \Rightarrow \\ \quad (\langle\Gamma\rangle, \texttt{fst}\langle t\rangle, \langle T\rangle)\ \texttt{TJ}) \rightsquigarrow \\ \texttt{app}\langle t, \texttt{abs}\langle \texttt{id}_\texttt{a}\langle\rangle, T, \\ \qquad \texttt{abs}\langle \texttt{id}_\texttt{b}\langle\rangle, T, \texttt{var}\langle \texttt{id}_\texttt{a}\langle\rangle\rangle\rangle\rangle\rangle\rangle\end{array}\right\} \quad [2.7]$$

The arguments of the conclusion on the left-hand side pattern are separated into three parts: a list of arguments preceding the rewritten expressions ($\langle\Gamma\rangle$ in the example), the marked expression ($\texttt{fst}\langle t\rangle$ in the example), and a list of arguments following the rewritten expression ($\langle t\rangle$ in the example).

It might be tempting to try to consider universal desugarings as a special case of guarded desugarings. But this is not possible since there may be syntactic sorts in the base language or in extensions that do not have any judgement form associated with them. An example are the types of $\lambda_{\rightarrow}$. Any syntactically correct type expression is a well-formed type and there is no judgement that all type expressions are the subject of. For terms, this is different since a term is only well-typed if a typing judgement is derivable. For this reason, SoundX features both kinds of desugarings.

A universal desugaring $U = (E \rightsquigarrow E')$ is well-formed if its left-hand and right-hand side are well-formed expressions of identical sort:

W-Universal
$$\frac{\vec{A} \vdash E : N \qquad \vec{A} \vdash E' : N}{\vec{A} \vdash E \rightsquigarrow E'}$$

For a guarded desugaring $G = ((\vec{J} \Rightarrow (\vec{E}_1, E, \vec{E}_2)\ N) \rightsquigarrow E')$ to be well-formed we demand that all premises $\vec{J}$ and the conclusion $(\vec{E}_1, E, \vec{E}_2)\ N$ are well-formed. Moreover, the left-hand side expression $E$ and the right-hand side expression $E'$ must be well-formed and of identical sort:

W-Guarded
$$\frac{\begin{array}{c}\text{for each } J \in \vec{J} : \vec{A}; \vec{F} \vdash J \\ \vec{A}; \vec{F} \vdash \langle \vec{E}_1, E, \vec{E}_2\rangle\ N \\ \vec{A} \vdash E : N_0 \\ \vec{A} \vdash E' : N_0\end{array}}{\vec{A}; \vec{F} \vdash (\vec{J} \Rightarrow (\vec{E}_1, E, \vec{E}_2)\ N) \rightsquigarrow E'}$$

It might be a surprise that the well-formedness conditions for desugarings do allow that a right-hand side uses variables which are not mentioned on

the left-hand side. The reason is, intuitively speaking, that these desugarings do not survive the SoundX verification procedure since the verifier does not know enough about such a variable to deduce anything useful.

### 2.1.5 *Base language and extension definitions*

A base language definition $B$ consists of a list of constructor arities $\vec{A}$, judgement forms $\vec{F}$, and inference rules $\vec{I}$ written as a triple $(\vec{A}, \vec{F}, \vec{I})$. Such a base language definition $B$ is well-formed, written $\mathfrak{E} B$, if all its components are well-formed:

W-Base
$$\frac{\vec{A} \mathfrak{E} \diamond \qquad \vec{F} \mathfrak{E} \diamond \qquad \vec{A}; \vec{F} \mathfrak{E} \vec{I}}{\mathfrak{E} (\vec{A}, \vec{F}, \vec{I})}$$

An extension $X$ consists of a list of constructor arities $\vec{A}_x$, inference rules $\vec{I}_x$, guarded desugarings $\vec{G}_x$, and universal desugarings $\vec{U}_x$ written as a quadruple $(\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$. An extension $X$ is well-formed with respect to a base language definition $B$, written $B \mathfrak{E} X$, if all its components are well-formed:

W-Ext
$$\frac{\begin{array}{c}\langle \vec{A}, \vec{A}_x \rangle \mathfrak{E} \diamond \\ \langle \vec{A}, \vec{A}_x \rangle; \vec{F} \mathfrak{E} \langle \vec{I}, \vec{I}_x \rangle \\ \text{for each } G \in \vec{G}_x : \langle \vec{A}_x, \vec{A} \rangle; \vec{F} \mathfrak{E} G \\ \text{for each } U \in \vec{U}_x : \langle \vec{A}_x; \vec{A} \rangle \mathfrak{E} U\end{array}}{(\vec{A}, \vec{F}, \vec{I}) \mathfrak{E} (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)}$$

We summarise the abstract syntax of base language definitions and extensions in the following grammar:

Abs-Syn

| | | |
|---|---|---|
| $A$ | $::= N : \vec{N} \to N$ | Constructor arities |
| $F$ | $::= N : \vec{N}$ | Judgement forms |
| $V$ | $::= N : N$ | Meta-variables |
| $E$ | $::= V$ | Expressions |
| | $\quad N\ \vec{E}$ | |
| $J$ | $::= \vec{E}\ N$ | Judgements |
| $I$ | $::= \vec{J} \to^N J$ | Inference rules |
| $G$ | $::= (\vec{J} \Rightarrow (\vec{E}, E, \vec{E})\ N) \rightsquigarrow E$ | Guarded desugarings |
| $U$ | $::= E \rightsquigarrow E$ | Universal desugarings |
| $B$ | $::= (\vec{A}, \vec{F}, \vec{I})$ | Base language definitions |
| $X$ | $::= (\vec{A}, \vec{I}, \vec{G}, \vec{U})$ | Extensions |

## 2.2 DERIVATIONS

A judgement like `⌀ ⊢ 3 + (1,2).1 : Nat` is valid if and only if it can be derived or inferred using the inference rule schemata that define the judgement, in this case the typing judgement `Γ ⊢ t : T`. For a judgement to hold, it must be a substitution instance of the conclusion of an inference rule and the instantiated premises of that rule must also hold. We usually visualise such a derivation as a tree where each node is an instantiation of an inference rule and the subtrees are the derivations of the premises. For example, here is the derivation of the previous typing statement using the rules of $\lambda_{\to}$ and the pair extension:

```
                        -----------Nat   -----------Nat
                        ⌀ ⊢ 1 : Nat       ⌀ ⊢ 2 : Nat
                        ----------------------------Pair
                             ⌀ ⊢ (1,2) : Pair Nat
-----------Nat             ----------------------Fst
⌀ ⊢ 3 : Nat                    ⌀ ⊢ (1,2).1 : Nat
-----------------------------------------------Add
                  ⌀ ⊢ 3 + (1,2).1 : Nat
```
[2.8]

Type checking a program according to a collection of typing rules means to find such a derivation. Since SoundX provides guarded desugarings that may utilise information contained in the typing derivation we have to make these derivations explicit objects in our system.

Derivations $\nabla$ are either built by using a judgement $J$ as an assumption or by the instantiation of an inference rule. The following abstract syntax is a linear representation of the two-dimensional structure of a derivation tree.

DERIV $\quad \nabla ::= !J$ Assumptions

$\vec{\nabla} \Rrightarrow^N J$ Instantiation of inference rules

We also augment the function *vars* with cases for derivations and provide a function *concl* that returns the conclusion of a derivation.

VARS-DERIV
$$\begin{aligned} vars(!J) &= vars(J) \\ vars(\vec{\nabla} \Rrightarrow^N J) &= vars(\vec{\nabla}) \cup vars(J) \\ vars\langle \nabla_{1..n} \rangle &= vars(\nabla_1) \cup \cdots \cup vars(\nabla_n) \end{aligned}$$

CONCL
$$\begin{aligned} concl(!J) &= J \\ concl(\vec{\nabla} \Rrightarrow^N J) &= J \\ concl\langle \nabla_{1..n} \rangle &= \langle concl(\nabla_1), \ldots, concl(\nabla_n) \rangle \end{aligned}$$

An assumption $!J$ states that we take the judgement $J$ for granted without any further evidence. They can only appear as leafs of a derivation $\nabla$. An instantiation node $(\vec{\nabla} \Rrightarrow^N J)$ is formed by a substitution instance of an inference rule named $N$. Of course, only those substitution instances are

valid whose premises are in turn conclusions of valid instances. Since the syntax of derivations permits arbitrary derivation trees we formally codify which derivations are valid derivations. To this end, we introduce substitutions of expressions for meta-variables as the set of finite partial functions from meta-variables to expressions:

SUBST $$\sigma := V \twoheadrightarrow E$$

Since substitutions are finite functions, we often enumerate them explicitly. For a set of pairwise distinct meta-variables $\{V_1, ..., V_n\}$, the enumeration $\{V_1 \mapsto E_1, ..., V_n \mapsto E_n\}$ denotes the substitution $\sigma : \{V_1, ..., V_n\} \to \{E_1, ..., E_n\}$ with $\sigma(V_i) = E_i$ for $i \in 1..n$. Given a substitution $\sigma : \{V_1, ..., V_n\} \to \{E_1, ..., E_n\}$, we write $dom(\sigma)$ for its domain $\{V_1, ..., V_n\}$. The application of a substitution $\sigma$ to a phrase $\phi$ is written $[\sigma](\phi)$ and defined as follows:

SUBST-APPLY

$$[\sigma](V) = \begin{cases} \sigma(V), & \text{if } V \in dom(\sigma) \\ V, & \text{if } V \notin dom(\sigma) \end{cases} \qquad \text{Expressions}$$

$$[\sigma](N\ \vec{E}) = N\ [\sigma](\vec{E})$$
$$[\sigma]\langle E_{1..n}\rangle = \langle [\sigma](E_1), ..., [\sigma](E_n)\rangle$$

$$[\sigma](\vec{E}\ N) = [\sigma](\vec{E})\ N \qquad \text{Judgements}$$
$$[\sigma]\langle J_{1..n}\rangle = \langle [\sigma](J_1), ..., [\sigma](J_n)\rangle$$

$$[\sigma](\vec{\nabla} \Rightarrow^N J) = ([\sigma](\vec{\nabla}) \Rightarrow^N [\sigma](J)) \qquad \text{Derivations}$$
$$[\sigma](!\,J) = (!\,[\sigma](J))$$
$$[\sigma](\nabla_{1..n}) = \langle [\sigma](\nabla_1), ..., [\sigma](\nabla_n)\rangle$$

$$[\sigma](\langle J_{1..n}\rangle \twoheadrightarrow^N J_0) = [\sigma]\langle J_{1..n}\rangle \twoheadrightarrow^N [\sigma](J_0) \qquad \text{Inference rules}$$

The statement $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ asserts that a derivation $\nabla$ is valid with respect to a list of inference rules $\vec{I}$ and a list of derivations $\vec{\nabla}_a$ which are assumed to be valid. Usually, we call the list $\vec{\nabla}_a$ *assumptions*. The statement $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ is inductively defined by the following two rules:

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

V-RULE
$$\frac{\begin{array}{c}(\langle J_{01..0n}\rangle \twoheadrightarrow^N J_0) \in \vec{I} \\ [\sigma]\langle J_{01..0n}, J_0\rangle = \langle concl\langle \nabla_{1..n}\rangle, J\rangle \\ \text{for each } i \in 1..n : \vec{\nabla}_a \vdash_{\vec{I}} \nabla_i\end{array}}{\vec{\nabla}_a \vdash_{\vec{I}} (\langle \nabla_{1..n}\rangle \Rightarrow^N J)}$$

The axiom V-ASSUMPTION states that a derivation is valid if it is mentioned in the assumptions.

For an instantiation of a rule $\langle \nabla_{1..n} \rangle \Rightarrow^N J$ to be valid, V-Rule requires three conditions to hold:

According to the first premise, the list of inference rules $\vec{I}$ must contain a rule $I_0 = (\langle J_{01..0n} \rangle \rightarrow^N J_0)$ that has the same name $N$ as the derivation node.

The second premise demands that the conclusions $concl(\nabla_1)$ to $concl(\nabla_n)$ of the subderivations and the conclusion $J$ must be substitution instances of the premises and the conclusion of $I_0$.

The last premise requires the subderivations $\nabla_1$ to $\nabla_n$ to be valid as well.

The definition of $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ is a formalisation of the process of inference that is described in many textbooks, for example [Rey98, Har13, Pie02], which we adapt to our syntax of rules and judgements. It is inspired by the provability relation $\vec{J} \vdash_{\vec{I}} J$ from logic which states that $J$ can be proven from the assumptions $\vec{J}$ using the rules $\vec{I}$ [EMC+01]. In these fsormulations, the order of appearance of the subderivations and the order of appearance of the premises of an instantiated rule is irrelevant. We restrict these two orders to match in the second premise of V-Rule. Since the order is irrelevant, we can fix a particular one without loss of generality, which simplifies the technical development considerably. In contrast to the order of the premises and the subderivations, the order of the inference rules $\vec{I}$ of a statement $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ is irrelevant because the list containment premise $(\langle J_{01..0n} \rangle \rightarrow^N J_0) \in \vec{I}$ in V-Rule makes no assumption about the position of the particular inference rule in $\vec{I}$.

The validity statement $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ has the following important structural properties:

Lemma 2.1 Structural properties of $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$

(i) Substitution: *If* $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ *then* $[\sigma](\vec{\nabla}_a) \vdash_{\vec{I}} [\sigma](\nabla)$.

(ii) Assumption exchange: *If* $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ *and* $concl(\vec{\nabla}_a) = concl(\vec{\nabla}'_a)$ *then there exists* $\nabla'$ *such that* $\vec{\nabla}'_a \vdash_{\vec{I}} \nabla'$ *and* $concl(\nabla) = concl(\nabla')$.

(iii) Transitivity: *If* $\langle \nabla_{1..n} \rangle \vdash_{\vec{I}} \nabla$ *and* $\langle \rangle \vdash_{\vec{I}} \nabla_i$, *for each* $i \in 1..n$, *then* $\langle \rangle \vdash_{\vec{I}} \nabla$.

(iv) Stability: *If* $\vec{I} \sqsubseteq \vec{I}_1$ *and* $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ *then* $\vec{\nabla}_a \vdash_{\vec{I}_1} \nabla$.

*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 details are listed in Appendix C.1. □

# 3 VERIFICATION AND DESUGARING

In [LE13] we present a type-sound language extension mechanism that is based solely on universal desugarings. Moreover, it only supports a fixed typing judgement of the form `Γ ⊢ t : T` and syntactic extensions can only be added at the term level. This impedes base languages that require additional judgement forms like kinding or subtyping, type-level extensions similar to the `Pair` type, or extensions that require type inference like let-expressions. The solution for type-sound language extensibility that are presented in this thesis overcomes these limitations and subsumes the simpler system. However, these advancements require a derivation-centred desugaring which cannot be presented technically as an extension of the former work. Therefore, we skip a description of the previous approach and directly concentrate on its successor.

The SoundX desugaring procedure translates a valid derivation $\nabla$ in the extended language into a valid derivation $\nabla'$ in the base language. The conclusion of the resulting derivation $\nabla'$ contains the desugared program. In principle, it would suffice to translate $\nabla$ into a judgement $J'$ which is derivable in the base language. But, as we see later in Chapter 4, we have to do several desugaring passes to rewrite extensions that import other extensions. Each of these passes requires a valid derivation as input. If the outcome of the rewrite procedure was just the judgement $J'$ its derivation would have to be reconstructed in an additional pass.

## 3.1 EXAMPLE: BOTTOM-UP DESUGARING

The underlying idea of the derivation desugaring procedure is best explained by an example. We begin with the derivation of the typing judgement for the term `((λp:Pair Nat.p) (1,2)).1` and conduct a step-by-step transformation into a derivation in the base language $\lambda_{\rightarrow}$. To reduce page turning, we repeat the necessary typing rules and desugarings of the pair extension from Section 1.3.2:

```
desugarings                                                    [3.1]
  { Pair T ~~~> (T → T → T) → T }

inductive definitions and desugarings
  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
```

```
Fst:                                                    [3.1 cont'd]
  Γ ⊢ t : Pair T
  ---------------
  Γ ⊢ [ t.1 ] : T
  ~~~> t (λa:T.λb:T.a)
```

To save space and reduce clutter in the following examples we abbreviate the type `Nat` with `N` and the type `Pair` with `P`. Here is the input derivation for the desugaring:

```
          ⋮                                                    [3.2]
  --------------------Var       ---------Nat  ---------Nat
  ø,p:P N ⊢ p : P N          ø ⊢ 1 : N     ø ⊢ 2 : N
-------------------------Abs ----------------------Pair
ø ⊢ λp:P N. p : P N → P N          ø ⊢ (1,2) : P N
-------------------------------------------------------App
                ø ⊢ (λp:P N. p) (1,2) : P N
               ---------------------------Fst
               ø ⊢ ((λp:P N. p) (1,2)).1 : N
```

As we have already indicated in Section 1.3.3, the desugaring of a derivation must be performed in bottom-up direction, that is, from the leafs of the derivation towards the root. Confusingly, bottom-up rewriting looks like top-down rewriting on a concrete derivation because we are used to sketch it with the root at the bottom.

To understand why the desugaring must proceed from the bottom up consider the pair construction `(1,2)` in the derivation of Example [3.2]. It appears in the conclusions of the `Pair`, `App`, and `Fst` node. But the left-hand side of the guarded desugaring of pair construction [3.1] only matches at the `Pair` node. It does not match the other two nodes because `(1,2)` appears as a subterm, not as a direct argument to the typing judgement. But since we must desugar the entire derivation into a derivation of the base language, these occurrences of `(1,2)` must also be rewritten. For this reason, we perform a bottom-up desugaring which rewrites the pair construction at the `Pair` node and then propagates the result to the `App` and `Fst` nodes.

For an intuitive understanding of the derivation desugaring, it is helpful to capture the notion of bottom-up transformation of a derivation more precisely. The following function *bottomup* applies a one-step transformation $t : \nabla \to \nabla$ to a derivation using a bottom-up strategy.

$$\mathit{bottomup}(t, \langle \nabla_{1..n} \rangle \Rightarrow^N J) = t(\langle \mathit{bottomup}(t, \nabla_1), \ldots, \mathit{bottomup}(t, \nabla_n) \rangle \Rightarrow^N J)$$

*bottomup* recursively transforms the subderivations $\nabla_1$ to $\nabla_n$. It then forms a new derivation consisting of the transformed subderivations, the original rule name $N$, and the original conclusion $J$. Finally, it applies $t$ to this derivation.

For the derivation desugaring, we need a transformation $t$ that carries out two tasks: it applies the desugarings of the extension and it propagates the results of rewriting the subderivation into the current conclusion. The latter turns out to be the main challenge and we focus on it first.

Pretending that we already knew how to desugar a derivation, calling upon our intuition for the expected outcome, we assume that *bottomup* has

transformed the subderivations of the App node into the base language derivations $\nabla'_{\texttt{Abs}}$ and $\nabla'_{\texttt{Pair}}$ with the following conclusions

$$\begin{aligned} \mathit{concl}(\nabla'_{\texttt{Abs}}) &= (\varnothing \vdash \lambda \texttt{p:(N→N→N)→N. p} : \texttt{((N→N→N)→N) → ((N→N→N)→N)}) \\ \mathit{concl}(\nabla'_{\texttt{Pair}}) &= (\varnothing \vdash \texttt{(}\lambda \texttt{a:N.}\lambda \texttt{b:N.}\lambda \texttt{s:N→N→N. s a b) 1 2 : (N→N→N)→N}), \end{aligned}$$

where the `Pair` type and the pair construction have been fully desugared. As we see later in Section 3.6, derivation desugaring generates valid derivations in the base language. This means, in particular, that $\nabla'_{\texttt{Abs}}$ and $\nabla'_{\texttt{Pair}}$ are valid derivations in $\lambda_{\to}$.

The next step is the application of $t$ to

$$\langle \nabla'_{\texttt{Abs}}, \nabla'_{\texttt{Pair}} \rangle \Rightarrow^{\texttt{App}} (\varnothing \vdash \texttt{(}\lambda \texttt{p:P N. p) (1,2) : P N})$$

which propagates the results of desugaring from the subderivations $\nabla'_{\texttt{Abs}}$ and $\nabla'_{\texttt{Pair}}$ into the conclusion $\varnothing \vdash$ `(λp:P N. p) (1,2) : P N` to replace `P N` and `(1,2)`. The basic idea is that $t$ applies `App` in forward direction to the conclusions of the desugared subderivations yielding the new conclusion.

This *forward step* matches the conclusions of the subderivations and the premises of `App` and applies the resulting substitution to the conclusion of `App`. In our example, this leads to the following substitution:

$$\sigma = \left\{ \begin{array}{l} \Gamma \mapsto \varnothing, \\ \texttt{t1} \mapsto \lambda \texttt{p:(N→N→N)→N. p}, \\ \texttt{t2} \mapsto \texttt{(}\lambda \texttt{a:N.}\lambda \texttt{b:N.}\lambda \texttt{s:N→N→N. s a b) 1 2}, \\ \texttt{T11} \mapsto \texttt{(N→N→N)→N}, \\ \texttt{T12} \mapsto \texttt{(N→N→N)→N} \end{array} \right\}$$

$t$ applies $\sigma$ to the conclusion $\Gamma \vdash$ `t1 t2 : T12` of `App` and yields the following desugared form:

$$\begin{aligned} J'_{\texttt{App}} = (\varnothing \vdash\ & \texttt{(}\lambda \texttt{p:(N→N→N)→N. p)} \\ & \texttt{((}\lambda \texttt{a:N.}\lambda \texttt{b:N.}\lambda \texttt{s:N→N→N. s a b) 1 2) : (N→N→N)→N}) \end{aligned}$$

Finally, $t$ forms the desugared derivation ending with the `App` rule by taking the desugared subderivations and the new conclusion:

$$\nabla'_{\texttt{App}} = (\langle \nabla'_{\texttt{Abs}}, \nabla'_{\texttt{Pair}} \rangle \Rightarrow^{\texttt{App}} J'_{\texttt{App}})$$

Since $\nabla'_{\texttt{Abs}}$ and $\nabla'_{\texttt{Pair}}$ are valid derivations in the base language and we obtained $J'_{\texttt{App}}$ by a forward instantiation of the rule `App`, this is certainly also a valid derivation in the base language.

Equipped with the desugared subderivation $\nabla'_{\texttt{App}}$, *bottomup* continues by the application of $t$ to

$$\langle \nabla'_{\texttt{App}} \rangle \Rightarrow^{\texttt{Fst}} (\varnothing \vdash \texttt{((}\lambda \texttt{p:P N. p) (1,2)).1 : N}).$$

Since `Fst` is a rule from the extension, the current conclusion in general contains extended expressions that have not already been desugared in the subderivations. In this case, this is the selection `((λp:P N. p) (1,2)).1`. $t$ must apply the desugarings from the extension to transform this selection into base language expressions. Nevertheless, it also has to perform the forward step to replace `P N` and `(1,2)` by their desugared forms, which emerge from the desugaring of the subderivations.

Rules from the extension may, in general, mention extended expressions in their premises. For `Fst`, this is `P N`. So simply executing the forward step with the premise of `Fst` and the conclusion of $\nabla'_{\texttt{App}}$ will not work because the latter is a base language derivation that cannot contain `P N` anymore. So, prior to performing the forward step, we completely desugar the rule `Fst` by exhaustively applying universal and guarded desugarings. We obtain the desugared premise

$$J'_{01\texttt{Fst}} = (\texttt{Γ ⊢ t : (T→T→T)→T})$$

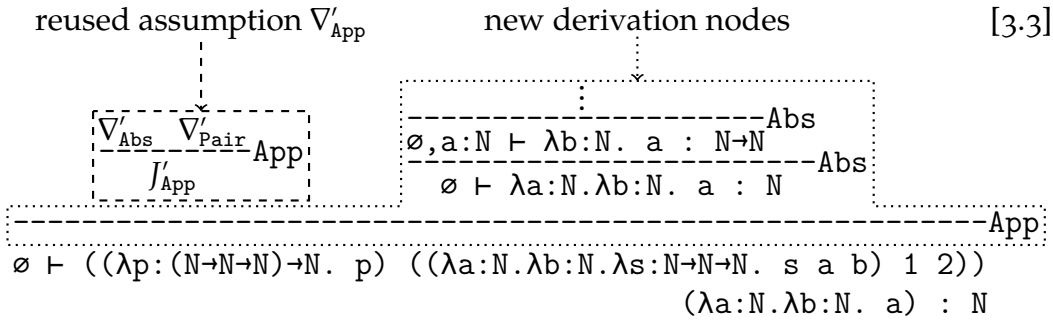
and conclusion

$$J'_{0\texttt{Fst}} = (\texttt{Γ ⊢ t (λa:T.λb:T. a) : T}).$$

We proceed as before but use the desugared premise and the desugared conclusion instead of the original forms. That is, $t$ matches the conclusion of the subderivation $\nabla'_{\texttt{App}}$ with $J'_{01\texttt{Fst}}$ and applies the resulting substitution to $J'_{0\texttt{Fst}}$. The last step produces the following judgement:

$$J'_{\texttt{Fst}} = (\texttt{∅ ⊢ ((λp:(N→N→N)→N. p)}\\ \texttt{((λa:N.λb:N.λs:N→N→N. s a b) 1 2)) (λa:N.λb:N. a) : N})$$

We observe that the forward step has already desugared the selection. This is due to the fact that we applied the matching substitution to the desugared conclusion of `Fst`. At this point, $t$ has finished the two tasks that we initially required. But it still has to build a valid derivation in the base language. It cannot simply return $(\langle \nabla'_{\texttt{App}} \rangle \Rightarrow^{\texttt{Fst}} J'_{\texttt{Fst}})$: this is not a valid derivation in the base language since the rule `Fst` does not belong to the base language. It is not even a valid derivation in the extended language since, for example, the term argument of $J'_{\texttt{Fst}}$ does not have the form `t.1`. Instead, $t$ bridges the gap between the subderivation $\nabla'_{\texttt{App}}$ and the conclusion $J'_{\texttt{Fst}}$ by suitable instantiations of base language rules. Formally, it calculates a derivation $\nabla'_{\texttt{Fst}}$ such that $\langle \nabla'_{\texttt{App}} \rangle \vdash_{\vec{I}_{\lambda_\to}} \nabla'_{\texttt{Fst}}$ and $\mathit{concl}(\nabla'_{\texttt{Fst}}) = J'_{\texttt{Fst}}$, where $\vec{I}_{\lambda_\to}$ are the typing rules of our base language $\lambda_\to$. To build the derivation $\nabla'_{\texttt{Fst}}$ the assumption $\nabla'_{\texttt{App}}$ is used without rederiving it. The final result $\nabla'_{\texttt{Fst}}$ is too big to fit it on a single page using concrete syntax. Therefore, we instead give a sketch using some of the previous definitions where we label the reused assumption and the newly created nodes which bridge the gap:




Having seen this sketch of the SoundX desugaring procedure two questions arise naturally:

1. Is it always possible to rebuild the derivation with base language rules when desugaring the instantiation of an extension rule?

2. Is the matching in the forward step always successful?

In Section 3.6.2 we prove that the SoundX verification procedure only accepts extensions such that the first question can be answered with "yes."

The answer to the second question is less satisfactory. Currently, we have no sufficient condition to decide for a given base language and extension if the forward step is always successful and it is an open problem if such a decidable criterion exists at all. In practical terms, this means that the desugaring may get stuck at this point under certain circumstances. But as we also show in Section 3.6.2, this is at least the only reason that can cause the procedure to get stuck.

Since the forward step is performed after the subderivations have been desugared into base language derivations, a necessary condition for its success is that the premises of the respective rule are base language judgements. For base language rules, this is certainly the case. For rules from the extension we check this statically during verification (see Section 3.4). But, as we discuss in Section 6.2, this condition is not sufficient and also has another downside: it excludes a whole class of useful extensions. We explore the latter issue in the next section.

## 3.2 EXAMPLE: TOP-DOWN DESUGARING

The necessary condition of the forward step excludes all typing rules whose desugared forms have premises with extended expressions. These typing rules naturally arise in many cases, especially for recursive desugarings. To illustrate the situation we define a let-expression with multiple bindings, for example `let a = 1; b = a in a+b`.

```
context-free syntax                                          [3.4]
  "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 ⊢ let bs in t2 : T2)
    ------------------------------------------
        Γ ⊢ [ let x = t1; bs in t2 ] : T2
    ~~~> (λx:T1. let bs in t2) t1
  Let2:
    (Γ ⊢ t1 : T1) (Γ,x:T1 ⊢ t2 : T2)
    --------------------------------
     Γ ⊢ [ let x = t1 in t2 ] : T2
    ~~~> (λx:T1. t2) t1
```

The list of bindings is desugared step-by-step. The necessary recursion is implemented by a residuum of extended code in the right-hand side of the `Let1` desugaring

Desugaring the rule `Let1` results in the following premises

$$J'_{01\texttt{Let1}} = (\texttt{Γ ⊢ t1 : T1})$$
$$J'_{02\texttt{Let1}} = (\texttt{Γ,x:T1 ⊢ let bs in t2 : T2})$$

and conclusion

$$J'_{0\texttt{Let1}} = (\texttt{Γ ⊢ (λx:T1. let bs in t2) t1 : T2})$$

and it is obvious that the second premise $J'_{02\texttt{Let1}}$ can never be successfully matched with a pure base language judgement since it contains the extended expression `let bs in t2`. We call rules like `Let1` whose desugared form has extended premises $\mathcal{X}$-rules and rules like `Fst` whose desugared form has only base language premises $\mathcal{B}$-rules.

As extensions with $\mathcal{X}$-rules are of huge practical relevance it is worth to enhance the desugaring procedure to handle them as well. The idea is to defer the forward step until all instantiations of $\mathcal{X}$-rules have been replaced by instantiations of base language or $\mathcal{B}$-rules, for which we assume that the forward step is successful. To this end we change the desugaring from a bottom-up transformation into a top-down/bottom-up transformation. The top-down pass is responsible for the desugaring of $\mathcal{X}$-rule instantiations into base language or $\mathcal{B}$-rule instantiations such that the bottom-up pass can carry out the forward-steps. We base the explanation of the top-down pass on the following function for the top-down transformation of a derivation,

which is the counterpart of *bottomup*:

$$\begin{aligned} \mathit{topdown}(u,\nabla) = \text{let } (\langle \nabla_{1..n} \rangle \Rightarrow^N J) &= u(\nabla) \\ \text{in } (\langle \mathit{topdown}(u,\nabla_1), \ldots, \mathit{topdown}(u,\nabla_n) \rangle &\Rightarrow^N J) \end{aligned}$$

*topdown* applies a transformation $u : \nabla \to \nabla$ to the derivation and recursively transforms the resulting subderivations. Finally, it builds a new derivation from the transformed subderivations, the rule name $N$, and the conclusion $J$ returned by the initial call to $u$.

We now describe a one-step transformation $u$ that desugars instantiations of $\mathcal{X}$-rules from the extension into instantiations of base language rules.

Consider the typing derivation for the term `let a = 1; b = a in b`:

```
                  ⋮                          ⋮
          ------------------Var  ---------------------Var
          ø,a:N ⊢ a : N          ø,a:N,b:N ⊢ b : N
---------Nat  ------------------------------------Let2
ø ⊢ 1 : N     ø,a:N ⊢ let b = a in b : N
--------------------------------------------Let1
        ø ⊢ let a = 1; b = a in b : N
```
[3.5]

*topdown* applies $u$ to the entire derivation. The rule instantiated at the root of the derivation is `Let1` which stems from the extension. In this case, $u$ works as follows:

1. It desugars the rule `Let1` completely. This yields the premises $J'_{01\texttt{Let1}}$, $J'_{02\texttt{Let1}}$, and the conclusion $J'_{0\texttt{Let1}}$.

2. It retrieves the substitution $\sigma$ used to instantiate `Let1`:

$$\sigma = \left\{ \begin{array}{ll} \texttt{Γ} \mapsto \texttt{ø}, & \texttt{t2} \mapsto \texttt{b}, \\ \texttt{x} \mapsto \texttt{a}, & \texttt{T1} \mapsto \texttt{N}, \\ \texttt{t1} \mapsto \texttt{1}, & \texttt{T2} \mapsto \texttt{N}, \\ \texttt{bs} \mapsto \texttt{(b = 1)} & \end{array} \right\}$$

   In contrast to the forward step this step is always possible since we have a valid instantiation of `Let1`.

3. It generates the new conclusion by applying $\sigma$ to the desugared conclusion of `Let1`, that is, $u$ calculates $J'_{\texttt{Let}} = [\sigma](J'_{0\texttt{Let1}})$.

4. It builds a derivation of the new conclusion $J'_{\texttt{Let}}$ with the original subderivations (ending with `Nat` and `Let2` in this case) as assumptions using the typing rules of the base language and the typing rules from the extension.

Here is the result of $u$ applied to the derivation of Example [3.5]:

```
      ⋮                          ⋮
------------------Var  ---------------------Var
ø,a:N ⊢ a : N          ø,a:N,b:N ⊢ b : N
-------------------------------------Let2
     ø,a:N ⊢ let b = a in b : N
  ---------------------------------Abs  ---------Nat
  ø ⊢ (λa:N. let b = a in b) : N→N      ø ⊢ 1 : N
  ---------------------------------------------App
        ø ⊢ (λa:N. let b = a in b) 1 : N
```
[3.6]

The fourth step of $u$ has inserted the instantiations of the base language rules `App` and `Abs`.

The second line in the definition of *topdown* now recursively transforms the subderivations of the `App` node. Both subderivations end with a base language rule for which $u$ is the identity function. Hence, the top-down transformation is applied to the subderivation ending with `Let2`. `Let2` is a rule from the extension but it is a $\mathcal{B}$-rule. Here are its desugared premises and its desugared conclusion:

$$\begin{aligned} J'_{01\texttt{Let2}} &= (\Gamma \vdash \texttt{t1} : \texttt{T1}) \\ J'_{02\texttt{Let2}} &= (\Gamma, \texttt{x:T1} \vdash \texttt{t2} : \texttt{T2}) \\ J'_{0\texttt{Let2}} &= (\Gamma \vdash (\lambda \texttt{x:T1.\ t2})\ \texttt{t1} : \texttt{T2}) \end{aligned}$$

None of the premises contains extended expressions. In this case, $u$ is also the identity function. That is, the derivation of Example [3.6] is the result of the top-down pass. The instantiation of the recursive rule `Let1` has been replaced by instantiations of base language rules. Now, the bottom-up pass desugars the remaining extensions and the forward step propagates the result of desugaring towards the root. In the example, this means, in particular, that `let b = a in b` is replaced by `(λb:N.b) a`. Here is the result of the bottom-up pass:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\varnothing, a{:}N, b{:}N \vdash b : N}\,\textsc{Var}}{\varnothing, a{:}N \vdash \lambda b{:}N.b : N \to N}\,\textsc{Abs} \quad \dfrac{\vdots}{\varnothing, a{:}N \vdash a : N}\,\textsc{Var}}{\varnothing, a{:}N \vdash (\lambda b{:}N.b)\ a : N}\,\textsc{App}}{\varnothing \vdash \lambda a{:}N.(\lambda b{:}N.b)\ a : N \to N}\,\textsc{Abs} \qquad \dfrac{}{\varnothing \vdash 1 : N}\,\textsc{Nat}}{\varnothing \vdash (\lambda a{:}N.\ (\lambda b{:}N.b)\ a)\ 1 : N}\,\textsc{App} \tag{3.7}$$

Similar to the bottom-up pass, the top-down pass raises the question if it is always possible to build a derivation for the new conclusion in the fourth step of $u$. Again, the answer is yes, since the verification procedure accepts only extensions where this step is possible (see Section 3.6.2).

Since bottom-up also embodies a descend into the derivation and top-down also embodies an ascend towards the root, we can combine both passes into a *downup* strategy

$$\begin{aligned} \mathit{downup}(u, t, \nabla) = \text{let } (\langle \nabla_{1..n} \rangle \Rightarrow^N J) &= u(\nabla) \\ \text{in } t(\langle \mathit{downup}(u, t, \nabla_1), \ldots, \mathit{downup}(u, t, \nabla_n) \rangle &\Rightarrow^N J)\,, \end{aligned}$$

which is equivalent to $\mathit{bottomup}(t, \mathit{topdown}(u, \nabla))$. The precise formulation of the top-down/bottom-up derivation desugaring is presented in Section 3.5. We have seen that this desugaring procedure is based on the more basic rewriting of inference rules and requires the distinction between $\mathcal{X}$- and $\mathcal{B}$-rules. The classification into $\mathcal{X}$- and $\mathcal{B}$-rules is part of the verification procedure which also depends on the rewriting of inference rules.

## 3.3 BASIC REWRITING

The precise details of derivation desugaring are formulated by inductive definitions. We start with the basic operations which are the applications of universal and guarded desugarings. They are captured by two one-step rewrite statements: $E \mapsto_{\vec{U}} E'$ for universal and $(\vec{J}, J) \mapsto_{\vec{G}} J'$ for guarded desugarings.

One-step rewriting of expressions using the desugarings $\vec{U}$ is inductively defined by two rules:

RE-APPLY
$$\frac{(E_0 \rightsquigarrow E_0') \in \vec{U} \qquad [\sigma](E_0) = E}{E \mapsto_{\vec{U}} [\sigma](E_0')}$$

RE-STRUCT
$$\frac{E \mapsto_{\vec{U}} E'}{N\langle \vec{E}_1, E, \vec{E}_2 \rangle \mapsto_{\vec{U}} N\langle \vec{E}_1, E', \vec{E}_2 \rangle}$$

The first rule RE-APPLY covers the case that the left-hand side $E_0$ of some universal desugaring in $\vec{U}$ directly matches an expression $E$. RE-APPLY requires that $E$ is a substitution instance of the left-hand side $E_0$ witnessed by $\sigma$. Consequently, the original expression $E$ is rewritten to the right-hand side $E_0'$ with the subterms of $\sigma$ plugged in.

The second structural rule RE-STRUCT covers nondeterministic rewriting of subterms.

Later, we also have to apply universal desugarings inside judgements and substitutions. Therefore, we lift the one-step rewriting of expressions to the level of judgments and substitutions by two structural rules:

RJ-STRUCT
$$\frac{E \mapsto_{\vec{U}} E'}{\langle \vec{E}_1, E, \vec{E}_2 \rangle\, N \mapsto_{\vec{U}} \langle \vec{E}_1, E', \vec{E}_2 \rangle\, N}$$

RS-STRUCT
$$\frac{E_i \mapsto_{\vec{U}} E_i'}{\{V_1 \mapsto E_1, \ldots, V_i \mapsto E_i, \ldots, V_n \mapsto E_n\} \mapsto_{\vec{U}} \{V_1 \mapsto E_1, \ldots, V_i \mapsto E_i', \ldots, V_n \mapsto E_n\}}$$

On the basis of judgement rewriting, we define the one-step rewriting of inference rules using universal and guarded desugarings. Since the result of rewriting an inference rule $(\vec{J} \twoheadrightarrow^N J)$ is a different rule we cannot also name it $N$. To avoid the unnecessary invention of names, rule rewriting is defined on pairs $(\vec{J}, J)$ instead of inference rules $I$. The statement $(\vec{J}, J) \mapsto_{\vec{G};\vec{U}} (\vec{J}', J')$ means that a rule with the premises $\vec{J}$ and the conclusion $J$ is rewritten to a rule with the premises $\vec{J}'$ and the conclusion $J'$:

RI-Apply
$$\frac{(\vec{J}_0 \Rightarrow (\langle E_{01..0i-1}\rangle, E_{0i}, \langle E_{0i+1..0n}\rangle)\ N) \rightsquigarrow E'_{0i}) \in \vec{G} \qquad [\sigma]\langle \vec{J}_0, \langle E_{01..0i-1}, E_{0i}, E_{0i+1..0n}\rangle\ N\rangle = \langle \vec{J}, \langle E_{1..i-1}, E_i, E_{i+1..n}\rangle\ N\rangle}{(\vec{J}, \langle E_{1..i-1}, E_i, E_{i+1..n}\rangle\ N) \mapsto_{\vec{G};\vec{U}} (\vec{J}, \langle E_{1..i-1}, [\sigma](E'_{0i}), E_{i+1..n}\rangle\ N)}$$

RI-UnivConcl
$$\frac{J \mapsto_{\vec{U}} J'}{(\vec{J}, J) \mapsto_{\vec{G};\vec{U}} (\vec{J}, J')}$$

RI-UnivPrem
$$\frac{J \mapsto_{\vec{U}} J'}{(\langle \vec{J}_1, J, \vec{J}_2\rangle, J) \mapsto_{\vec{G};\vec{U}} (\langle \vec{J}_1, J', \vec{J}_2\rangle, J)}$$

A one-step rewriting of an inference rule consists either of the application of a guarded desugaring as defined by RI-Apply or the application of a universal desugaring to the conclusion (RI-UnivConcl) or to one of the premises (RI-UnivPrem). The latter two rules are a lifting of judgement rewriting to inference rules. The first rule RI-Apply is similar to RE-Apply: it picks a guarded desugaring whose left-hand side is a substitution instance of the rule $(\vec{J}, J)$ and replaces one argument in the conclusion by the instantiation of the right-hand side.

During the desugaring of a derivation we apply desugarings exhaustively to substitutions and inference rules. Exhaustively means that desugarings are applied repeatedly until none is applicable anymore. We complement the rewriting of substitutions and of inference rules by the two statements $\sigma \Longmapsto_{\vec{U}} \sigma'$ and $(\vec{J}, J) \Longmapsto_{\vec{G};\vec{U}} (\vec{J}', J')$ which are used by the top-down/bottom-up derivation desugaring (Section 3.5) and the latter is also necessary for the verification of an extension (Section 3.4). Both statements define the necessary repetition similar to the `repeat` strategy in [VBT98]:

DS-Iter
$$\frac{\sigma \mapsto_{\vec{U}} \sigma' \qquad \sigma' \Longmapsto_{\vec{U}} \sigma''}{\sigma \Longmapsto_{\vec{U}} \sigma''}$$

DS-Stop
$$\frac{\sigma \not\mapsto_{\vec{U}}}{\sigma \Longmapsto_{\vec{U}} \sigma}$$

DI-Iter
$$\frac{(\vec{J}, J) \mapsto_{\vec{G};\vec{U}} (\vec{J}', J') \qquad (\vec{J}', J') \Longmapsto_{\vec{G};\vec{U}} (\vec{J}'', J'')}{(\vec{J}, J) \Longmapsto_{\vec{G};\vec{U}} (\vec{J}'', J'')}$$

DI-Stop
$$\frac{(\vec{J}, J) \not\mapsto_{\vec{G};\vec{U}}}{(\vec{J}, J) \Longmapsto_{\vec{G};\vec{U}} (\vec{J}, J)}$$

In the rule DS-Stop, we use the negated arrow $\sigma \not\mapsto$ as a shorthand for

$$\text{there exists no } \sigma' \text{ such that } \sigma \mapsto_{\vec{U}} \sigma'$$

and similar for $(\vec{J}, J) \not\mapsto_{\vec{G};\vec{U}}$ in DI-Stop.

The statements $\sigma \Longmapsto_{\vec{U}} \sigma'$ and $(\vec{J}, J) \Longmapsto_{\vec{G};\vec{U}} (\vec{J}', J')$ are similar to the reflexive-transitive closure of $\sigma \mapsto_{\vec{U}} \sigma'$ and $(\vec{J}, J) \mapsto_{\vec{G};\vec{U}} (\vec{J}', J')$, respectively, but not quite the same. For example, $\sigma \Longmapsto_{\vec{U}} \sigma$ is only valid if none of the desugarings of $\vec{U}$ is applicable anymore which is enforced by the premise of DS-Stop. In contrast, the conclusion $\sigma \Longmapsto_{\vec{U}} \sigma$ of DS-Stop would be an axiom for the reflexive-transitive closure. That is, the premise $\sigma \not\mapsto_{\vec{U}}$ is responsible for the exhaustive application of desugarings.

To reduce the notational overhead with respect to inference rule desugaring, we also write $\vec{I} \Longmapsto_{\vec{G};\vec{U}} (\vec{J}', J')$ and define it as the following abbreviation

Inf-Desugar $$(\vec{J} \twoheadrightarrow^{N} J) \Longmapsto_{\vec{G};\vec{U}} (\vec{J}', J') := (\vec{J}, J) \Longmapsto_{\vec{G};\vec{U}} (\vec{J}', J')\,.$$

Depending on the desugarings $\vec{G}$ and $\vec{U}$ it is of course possible that rewriting of substitutions or inference rules diverges. Since it is undecidable if some given desugarings lead to non-termination or not, this phenomenon is usually accepted and handled pragmatically, for example using timeouts. We discuss termination issues in more detail in Section 6.1.

## 3.4 Verification procedure

With the definition of inference rule desugaring 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 during the desugaring is well-typed in the base language.

In the description of the desugaring procedure in the Sections 3.1 and 3.2 we distinguished two classes of inference rules: $\mathcal{X}$-rules, whose desugared premises contain extended expressions, and $\mathcal{B}$-rules, whose desugared premises are base language expressions. We capture the distinction of inference rules slightly more generally here. A rule can either be sound relative to the extended language ($\mathcal{X}$-rule) or relative to the base language ($\mathcal{B}$-rule). Rules with extended premises belong to the former class, rules with base language premises belong to the latter class. The verification procedure assigns a class to each inference rule of an extension or rejects it as unsound.

Given an extension $X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ and a base system $B = (\vec{A}, \vec{F}, \vec{I})$, an inference rule $I \in \vec{I}_x$ is sound relative to the extended language if the

statement $B; X \ltimes I : \mathcal{X}$ can be derived by the following rule:

$$\text{S-Ext}\quad \frac{\begin{array}{c}(\langle J_{01..0n}\rangle \twoheadrightarrow^{N} J_0) \Longmapsto_{\vec{G}_x;\vec{U}_x} (\langle J'_{01..0n}\rangle, J'_0) \\ \langle !\, J_{01}, \ldots, !\, J_{0n}\rangle \vdash_{\langle \vec{I}, \vec{I}_x\rangle} (\vec{\nabla}' \Rightarrow^{N'} J'_0) \\ (\vec{J}' \twoheadrightarrow^{N'} J') \in \vec{I}\end{array}}{(\vec{A}, \vec{F}, \vec{I}); (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x) \ltimes (\langle J_{01..0n}\rangle \twoheadrightarrow^{N} J_0) : \mathcal{X}}$$

The first premise of this rule fully desugars $I$. The second premise of S-Ext derives the desugared conclusion $J'_0$ of $I$ from the original premises as assumptions using all the inference rules of the base language $\vec{I}$ and the extension $\vec{I}_x$. This premise is the reason that we call $I$ sound relative to the extended language.

The last premise requires that the valid derivation of the second premise ends with a base language rule. With this requirement we ensure that the application of guarded desugarings always terminates. In particular, it implies that the conclusion $J'_0$ of the second premise is not derived from any of the assumptions $!\, J_{01}$ to $!\, J_{0n}$. Intuitively, this prevents the generation of a possibly infinite derivation only involving inference rules from the extension. We return to this issue in the discussion of the Progress theorem in Section 3.6.2.

The soundness criterion for $\mathcal{X}$-rules imitates top-down rewriting. The subderivations are desugared after the root has been rewritten, that is, they may contain extended expressions. This corresponds to the fact that S-Ext takes the original premises of $I$, which possibly contain extended expressions, as assumptions and uses the inference rules of the base language and the extension to derive the desugared conclusion.

Similarly, the soundness criterion for $\mathcal{B}$-rules imitates bottom-up rewriting. An inference rule $I \in \vec{I}_x$ is sound relative to the base language if the statement $B, X \ltimes I : \mathcal{B}$ can be derived by the following rule:

$$\text{S-Base}\quad \frac{\begin{array}{c}(\langle J_{01..0n}\rangle \twoheadrightarrow^{N} J_0) \Longmapsto_{\vec{G}_x;\vec{U}_x} (\langle J'_{01..0n}\rangle, J'_0) \\ \text{for each } i \in 1..n : \vec{A}; \vec{F} \mathrel{\mathfrak{S}\!\!-} J'_{0i} \\ \langle !\, J'_{01}, \ldots, !\, J'_{0n}\rangle \vdash_{\vec{I}} (\vec{\nabla}' \Rightarrow^{N'} J'_0)\end{array}}{(\vec{A}, \vec{F}, \vec{I}); (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x) \ltimes (\langle J_{01..0n}\rangle \twoheadrightarrow^{N} J_0) : \mathcal{B}}$$

In the first premise, the inference rule $I$ is fully desugared. Since a $\mathcal{B}$-rule is desugared on the bottom-up pass we require that their desugared premises $J'_{01}$ to $J'_{0n}$ are well-formed in the base language (second premise of S-Base), that is, they must not contain extended expressions, since this is a necessary condition for the forward step to be successful. Finally, S-Base derives the desugared conclusion $J'_0$ from the desugared premises but using only the rules from the base language, that is, relative to the base language.

Finally, an extension $X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ is sound with respect to a base system $B = (\vec{A}, \vec{F}, \vec{I})$ if the statement $B; X \ltimes I : \mathcal{B}$ or $B; X \ltimes I : \mathcal{X}$ can be derived for each inference rule $I \in \vec{I}_x$, which we write as $B \ltimes X$.

We close this section by the following observations:

- Certain rules can be classified as $\mathcal{X}$ as well as $\mathcal{B}$ since the conditions of S-Ext and S-Base are not mutually exclusive. Rules with base language premises whose conclusion is desugared into a base language judgement can be classified either way. Their premises are desugared into themselves, that is, $\langle J_{01..0n} \rangle = \langle J'_{01..0n} \rangle$. Since the desugared conclusion $J'_0$ is a base language judgement it is derivable from $\vec{I}$ and from $\langle \vec{I}, \vec{I}_x \rangle$, if at all. Obviously, the derivation of $J'_0$ als ends with a base language rule in that case. We see that such an inference rule satsfies the requirement of both classification conditions. The rule `Let2` on page 46 is an example. Its premises are base language judgements and its conclusion is also desugared into a pure base language judgement.

- If an extension only contains guarded desugarings, any inference rule can be classified as $\mathcal{X}$, if it can be classified at all. In other words, this means $B; X \ltimes I : \mathcal{B}$ implies $B; X \ltimes I : \mathcal{X}$. The reason is that we always have $\langle J_{01..0n} \rangle = \langle J'_{01..0n} \rangle$ as premises are only transformed by universal desugarings. Under these conditions, the requirements of S-Base imply the requirements of S-Ext. Consequently, the forward step can only get stuck while desugaring an instantiation of a base language rule.

- An inference rule whose premises can be desugared by universal desugarings but still contain extended expressions can only be classified as an $\mathcal{X}$-rule. Classification as $\mathcal{B}$ is prevented by the second premise of S-Base.

  Usually, it is not possible to classify it as $\mathcal{X}$ either, since this means that the desugarings in the premises are ignored. Consequently, such a rule is rejected by the SoundX verifier. This is due to the fact that the desugaring procedure cannot handle such a rule. Neither of our two running examples nor any of the case studies of Section 5.2 embody an inference rule with these properties. Hence, we claim that this restriction is of minor practical relevance.

## 3.5 Derivation desugaring

We formulate the details of top-down/bottom-up desugaring of derivations for a given base system $B$ and an extension $X$.

A one-step top-down desugaring at the root of a derivation is expressed by the statement $\nabla \downarrow_{B;X} \nabla'$ and defined by three rules:

TD-Base
$$\frac{(\vec{J}_0 \twoheadrightarrow^N J_0) \in \vec{I}}{(\vec{\nabla} \Rightarrow^N J) \downarrow_{(\vec{A},\vec{F},\vec{I});X} (\vec{\nabla} \Rightarrow^N J)}$$

TD-ExtBase
$$\frac{\begin{array}{c}(\vec{J}_0 \twoheadrightarrow^N J_0) \in \vec{I}_x \\ B; (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x) \ltimes (\vec{J}_0 \twoheadrightarrow^N J_0) : \mathcal{B}\end{array}}{(\vec{\nabla} \Rightarrow^N J) \downarrow_{B;(\vec{A}_x,\vec{I}_x,\vec{G}_x,\vec{U}_x)} (\vec{\nabla} \Rightarrow^N J)}$$

TD-ExtExt
$$\frac{\begin{array}{c}(\vec{J}_0 \twoheadrightarrow^N J_0) \in \vec{I}_x \\ (\vec{A},\vec{F},\vec{I}); (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x) \ltimes (\vec{J}_0 \twoheadrightarrow^N J_0) : \mathcal{X} \\ (\vec{J}_0 \twoheadrightarrow^N J_0) \Longmapsto_{\vec{G}_x;\vec{U}_x} (\vec{J}'_0, J'_0) \\ [\sigma_1]\langle \vec{J}_0, J_0 \rangle = \langle concl(\vec{\nabla}), J \rangle \\ \vec{\nabla} \vdash_{\langle \vec{I},\vec{I}_x \rangle} (\vec{\nabla}' \Rightarrow^{N'} [\sigma_1](J'_0)) \\ (\vec{J}' \twoheadrightarrow^{N'} J') \in \vec{I}\end{array}}{(\vec{\nabla} \Rightarrow^N J) \downarrow_{(\vec{A},\vec{F},\vec{I});(\vec{A}_x,\vec{I}_x,\vec{G}_x,\vec{U}_x)} (\vec{\nabla}' \Rightarrow^{N'} [\sigma_1](J'_0))}$$

The first two rules implement the identity desugaring for instantiations of base language and $\mathcal{B}$-rules.

The third rule implements desugaring of instantiations of $\mathcal{X}$-rules. The desugared conclusion of the derivation is obtained by instantiating the desugared conclusion of the inference rule using the original substitution $\sigma_1$ from the derivation. The desugared derivation is obtained by deriving the desugared conclusion using the original subderivations as assumptions, enforcing the rule instantiated at the root node to be from the base system.

For the definition of one-step bottom-up desugaring we need the restriction function for substitutions that removes a given set of variables from the substitution's domain:

Subst-Restrict
$$\sigma \setminus \{V_1, ..., V_n\} = \{V \mapsto E \mid \sigma(V) = E \text{ and } V \notin \{V_1, ..., V_n\}\}$$

A one-step bottom-up desugaring at the root of a derivation is expressed by the statement $\nabla \updownarrow_{B;X} \nabla'$ and defined by two rules:

$$\text{BU-Base}\quad \frac{\begin{array}{c}(\vec{J}_0 \twoheadrightarrow^N J_0) \in \vec{I} \\ [\sigma_1](\vec{J}_0) = \mathit{concl}(\vec{\nabla}) \\ [\sigma_2](J_0) = J \\ \sigma_2 \setminus \mathit{vars}(\vec{J}_0) \Longmapsto_{\vec{U}_x} \sigma_2'\end{array}}{(\vec{\nabla} \hookrightarrow^N J) \uparrow_{(\vec{A},\vec{F},\vec{I});(\vec{A}_x,\vec{I}_x,\vec{G}_x,\vec{U}_x)} (\vec{\nabla} \hookrightarrow^N [\sigma_1 \circ \sigma_2'](J_0))}$$

$$\text{BU-Ext}\quad \frac{\begin{array}{c}(\vec{J}_0 \twoheadrightarrow^N J_0) \in \vec{I}_x \\ (\vec{J}_0 \twoheadrightarrow^N J_0) \Longmapsto_{\vec{G}_x;\vec{U}_x} (\vec{J}_0', J_0') \\ [\sigma_1](\vec{J}_0') = \mathit{concl}(\vec{\nabla}) \\ [\sigma_2](J_0) = J \\ \sigma_2 \setminus \mathit{vars}(\vec{J}_0') \Longmapsto_{\vec{U}_x} \sigma_2' \\ \vec{\nabla} \vdash_{\vec{I}} (\vec{\nabla}' \hookrightarrow^{N'} [\sigma_1 \circ \sigma_2'](J_0'))\end{array}}{(\vec{\nabla} \hookrightarrow^N J) \uparrow_{(\vec{A},\vec{F},\vec{I});(\vec{A}_x,\vec{I}_x,\vec{G}_x,\vec{U}_x)} (\vec{\nabla}' \hookrightarrow^{N'} [\sigma_1 \circ \sigma_2'](J_0'))}$$

The first rule BU-Base handles instantiations of base language rules. Its second premise implements the forward step. The third and fourth premise are concerned with an aspect we have neglected so far: an inference rule may have variables in the conclusion that are not mentioned in its premises. The axiom `Lookup` of $\lambda_{\rightarrow}$

```
Lookup:
  ----------
  x:T ∈ Γ,x:T
```
[3.8]

is an example. Since it is an axiom scheme none of the variables `Γ`, `x`, `T` appears above the dividing bar. But `T` may be bound to a type expression containing extended syntax, for example `Pair Nat`. The substitution $\sigma_1$ cannot take care of these variables. Therefore, BU-Base matches the original conclusion $J$ and the conclusion $J_0$ of the inference rule. The right-hand sides of the resulting substitution $\sigma_2$ of variables that are not mentioned in the premises $\vec{J}_0$ are rewritten to $\sigma_2'$. The desugared derivation is then assembled from the desugared subderivations and the substitution instance $[\sigma_1 \circ \sigma_2'](J_0)$.

The rule BU-Ext handles instantiations of $\mathcal{B}$-rules of the extension. It desugars the instantiated rule and performs the forward step. Similar to BU-Base it handles variables that only appear in the rule's conclusion by a second matching substitution $\sigma_2$. The desugared conclusion is formed from the conclusion of the desugared rule using the matching substitutions. The result of desugaring is the derivation of that conclusion using the subderivations of the input as assumptions.

The statements $\nabla \downarrow_{B;X} \nabla'$ and $\nabla \uparrow_{B;X} \nabla'$ only desugar the root of the derivation. The desugaring of an entire derivation is a top-down, bottom-up

application of these rules. This is implemented by the big-step desugaring statement $\nabla \Updownarrow_{B;X} \nabla'$ with this definition:

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

It is a specialisation of the `downup` strategy from [VBT98] for our setting. It first applies one-step top-down desugaring to the root of the derivation. Then the resulting subderivations are desugared using the top-down, bottom-up strategy. Finally, one-step bottom-up desugaring is applied to the derivation resulting from the first two steps.

It might come unexpected that we do not use the function *downup* introduced in Section 3.2 but the $\Updownarrow$ statement. The reason is that *downup* expects two functions $t$ and $u$ as arguments and additional technical development would be necessary to turn $\uparrow$ and $\downarrow$ into functions. Functions like *downup* are useful to give an intuition for the desugaring procedure but for the formalities, inductive definitions are easier to handle.

## 3.6 SOUNDNESS

The type-soundness of a programming language is usually established by two theorems, namely Preservation and Progress, that relate the typing judgement with a small-step evaluation relation. Preservation states that if a program is well-typed and performs a step of evaluation the result of this evaluation is also well-typed. Progress states that if a program is well-typed it is either able to perform a step of evaluation or it is a value. This formulation crucially relies on the fact that a small-step evaluation relation is employed to model the dynamics of the program.

With a big-step evaluation relation we can give a formulation of Preservation: if a program is well-typed and it evaluates to a value this value is also well-typed. But we get into trouble with the formulation of the Progress theorem: if a program is well-typed it evaluates to a value. Phrased like this it implies that all programs can be evaluated into a value. This can only be true for languages that allow only terminating programs and is not appropriate for most programming languages. There are ways to obtain a Progress theorem also for a big-step evaluation but they are not entirely satisfactory and rather cumbersome. See [Har13, 7.3] for a discussion of this issue.

For SoundX we prove two theorems similar to Preservation and Progress where we essentially replace "evaluation" by "desugaring." However, the SoundX desugaring procedure is a big-step relation and it is possible to

formulate desugarings that diverge. That is, we faced the same difficulties related to the Progress theorem as outlined above. But we observe that we obtain a desugaring that always terminates by requiring that the universal desugarings form a terminating rewrite system. With this restriction, we are able to prove an adequate Progress theorem for the big-step desugaring.

### 3.6.1 *Preservation*

It is probably a surprise that well-formedness of extensions is sufficient to guarantee that desugaring always produces valid derivations, provided the original derivation was valid. That is, we have the following theorem:

THEOREM 3.1 Preservation
*Let* $B = (\vec{A}, \vec{F}, \vec{I})$ *be a well-formed base system and* $X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ *be a well-formed extension with respect to* $B$.
*If* $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ *and* $\nabla \Updownarrow_{B;X} \nabla'$ *then* $\langle\rangle \vdash_{\vec{I}} \nabla'$.

Intuitively, Preservation holds because the derivation of its second requirement $\nabla \Updownarrow_{B;X} \nabla'$ includes validation statements due to the validation premises in BU-EXT and TD-EXTEXT.

For the proof of the Preservation theorem, we need the following basic preservation properties of the one-step desugaring statements:

LEMMA 3.2
*Let* $B = (\vec{A}, \vec{F}, \vec{I})$ *be a well-formed base system and* $X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ *be a well-formed extension with respect to* $B$.

(i) *If* $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ *and* $\nabla \downarrow_{B;X} \nabla'$ *then* $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla'$.

(ii) *If for each* $\nabla_1 \in \vec{\nabla} : \langle\rangle \vdash_{\vec{I}} \nabla_1$ *and* $(\vec{\nabla} \Rightarrow^N J) \updownarrow_{B;X} \nabla'$ *then* $\langle\rangle \vdash_{\vec{I}} \nabla'$.

*Proof.*

(i) By a case analysis on the last rule applied in the derivation of $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ using Lemma 2.1.

(ii) By a case analysis on the last rule applied in the derivation of $(\vec{\nabla} \Rightarrow^N J) \updownarrow_{B;X} \nabla'$ using Lemma 2.1.

The details are listed in Appendix C.2. □

Using these properties we can prove the Preservation theorem:

*Proof of Theorem 3.1 (Preservation).* By induction on a derivation of $\nabla \Updownarrow_{B;X} \nabla'$ using Lemma 3.2. The details are listed in Appendix C.3. □

### 3.6.2 *Progress*

The Preservation theorem guarantees that the result of desugaring is a valid derivation. But it does not make any statement under which conditions this result exists. Such a statement is provided by a Progress theorem.

In our context, we would like to formulate Progress for a given base system $B = (\vec{A}, \vec{F}, \vec{I})$ and extension $X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ in the following way:

*If* $\vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ *then there exists* $\nabla'$ *such that* $\nabla \Updownarrow_{B;X} \nabla'$.

However, this statement remains wishful thinking since it implies that

1. any extension $X$ can be desugared,
2. derivation desugaring always terminates, and
3. the forward step is always successful.

None of these implications is true and the real SoundX Progress theorem differs from the initial (false) attempt in the following aspects: To deal with the first issue, it is sufficient to demand that the extension $X$ is sound with respect the base system $B$. The second implication actually is true under the additional requirement that the universal desugarings form a terminating rewrite system. The last issue is resolved by explicitly including the possible failure of the forward step in the conclusion of the theorem.

We write $\nabla \Updownarrow_{B;X} \lightning$ to indicate that derivation desugaring is stuck in a forward step. The rules under which this statement is derivable are presented shortly. The requirement for the universal desugarings to form a terminating rewrite system is made precise by the following definition:

Definition 3.3 Terminating rewrite system

*A system of universal desugaring* $\vec{U}$ *forms a terminating rewrite system if for all substitutions* $\sigma$ *there exists a* $\sigma'$ *such that* $\sigma \Longmapsto_{\vec{U}} \sigma'$.

Since the statement $\sigma \Longmapsto_{\vec{U}} \sigma'$ is defined inductively, the existence of $\sigma'$ is sufficient to ensure that $\sigma$ is rewritten to $\sigma'$ in a finite number of steps. As a simple example of a nonterminating system of universal desugarings, which would be excluded by the previous definition, consider the following universal desugarings $\vec{U}$:

```
{ Pair T  ~~~> Tuple T }
{ Tuple T ~~~> Pair T  }
```
[3.9]

For the substitution $\sigma = \{\texttt{T} \mapsto \texttt{Pair Nat}\}$ the rewriting sequence is infinite:

$$\{\texttt{T} \mapsto \texttt{Pair Nat}\} \Longmapsto_{\vec{U}} \{\texttt{T} \mapsto \texttt{Tuple Nat}\} \Longmapsto_{\vec{U}} \{\texttt{T} \mapsto \texttt{Pair Nat}\} \Longmapsto_{\vec{U}} \cdots$$

Since the statement $\Longmapsto_{\vec{U}}$ implements exhaustive rewriting, no right-hand side for `T` exists that terminates the rewrite sequence.

Taking the previous considerations into account leads to the following SoundX Progress theorem:

THEOREM 3.4 Progress

*Let $B = (\vec{A}, \vec{F}, \vec{I})$ be a well-formed base system and $X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ be a well-formed and sound extension with respect to B.*

*If $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ and the universal desugarings $\vec{U}_x$ form a terminating rewrite system then either $\nabla \Updownarrow_{B;X} \lightning$ or there exists a $\nabla'$ such that $\nabla \Updownarrow_{B;X} \nabla'$.*

This theorem guarantees that desugaring cannot get stuck except due to a failure in the forward step. Especially, it also means that it is always possible to find a valid derivation for the desugared conclusion, that is, we can always "bridge the gap" in the derivation. In more practical terms, this can be rephrased by the slogan "no type errors in generated code."

This theorem also guarantees that desugaring always terminates, if the input derivation $\nabla$ is valid and the universal desugarings form a terminating rewrite system. In particular, this means that the desugaring of a sound extension containing guarded desugarings only is guaranteed to terminate. We do not exclude any extension by the requirement that the universal desugarings terminate because this is necessary in practical applications anyway. But of course we cannot prove this property automatically in general. This has to be done on a case-to-case basis externally to SoundX.

Before we dedicate ourselves to the proof of the Progress theorem we augment the definitions of desugaring with the explicit failure value $\lightning$ which is returned if the forward step fails. We give two additional rules for $\nabla \uparrow_{B;X} \lightning$ that cover the case that there exists no matching substitution:

$$\text{BU-BaseStuck} \qquad \frac{(\vec{J}_0 \rightarrowtail^N J_0) \in \vec{I} \quad \text{for all } \sigma : [\sigma](\vec{J}_0) \neq \mathit{concl}(\vec{\nabla})}{(\vec{\nabla} \Rightarrow^N J) \uparrow_{(\vec{A},\vec{F},\vec{I});X} \lightning}$$

$$\text{BU-ExtStuck} \qquad \frac{(\vec{J}_0 \rightarrowtail^N J_0) \in \vec{I}_x \quad (\vec{J}_0 \rightarrowtail^N J_0) \Longmapsto_{\vec{G}_x;\vec{U}_x} (\vec{J}'_0, J'_0) \quad \text{for all } \sigma_1 : [\sigma_1](\vec{J}'_0) \neq \mathit{concl}(\vec{\nabla})}{(\vec{\nabla} \Rightarrow^N J) \uparrow_{B;(\vec{A}_x,\vec{I}_x,\vec{G}_x,\vec{U}_x)} \lightning}$$

With these additional stuckness rules, it is possible to derive a statement $\nabla \uparrow_{B;X} \lightning$ in situations where no derivation was possible before.

We reflect the changes of the one-step bottom-up desugaring in the derivation desugaring by two corresponding stuckness rules:

$$\text{DU-StuckSub}\quad \frac{\nabla \downarrow_{B;X} (\vec{\nabla}' \Rightarrow^{N'} J') \quad \text{for some } \nabla'_1 \in \vec{\nabla}' : \nabla'_1 \Updownarrow_{B;X} \lightning}{\nabla \Updownarrow_{B;X} \lightning}$$

$$\text{DU-StuckBU}\quad \frac{\nabla \downarrow_{B;X} (\langle \nabla'_{1..n} \rangle \Rightarrow^{N'} J') \quad \text{for each } i \in 1..n : \nabla'_i \Updownarrow_{B;X} \nabla''_i \quad (\langle \nabla''_{1..n} \rangle \Rightarrow^{N'} J') \uparrow_{B;X} \lightning}{\nabla \Updownarrow_{B;X} \lightning}$$

An important consequence of these definitions is that they do not invalidate Lemma 3.2 and Theorem 3.1 (Preservation) since their requirements contain a successful desugaring. For example, Preservation requires $\nabla \Updownarrow_{B;X} \nabla'$ and is immediately true for $\nabla \Updownarrow_{B;X} \lightning$.

The proof of Progress requires the following lemma about one-step top-down rewriting. This lemma highly depends on the last premise of the rule S-Ext (see the $\mathcal{X}$-case in the proof of 3.5(i) on page 124).

Lemma 3.5
*Let $B = (\vec{A}, \vec{F}, \vec{I})$ be a well-formed base system and $X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ be a well-formed and sound extension with respect to $B$.*

(i) *If $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ then there exists a $\nabla'$ such that $\nabla \downarrow_{B;X} \nabla'$.*

(ii) *If $\nabla \downarrow_{B;X} (\vec{\nabla}' \Rightarrow^{N'} J')$ then either $(\vec{J}_0 \twoheadrightarrow^{N'} J_0) \in \vec{I}$ or $(\vec{J}_0 \twoheadrightarrow^{N'} J_0) \in \vec{I}_x$ with $B; X \ltimes (\vec{J}_0 \twoheadrightarrow^{N'} J_0) : \mathcal{B}$.*

*Proof.*

(i) By a case analysis on the last rule applied in the derivation of $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ using Lemma 2.1.

(ii) By a case analysis on the last rule applied in the derivation of $\nabla \downarrow_{B;X} (\vec{\nabla}' \Rightarrow^{N'} J')$ using the soundness criterion in the TD-ExtExt case.

The details are listed in Appendix C.5. □

Finally, we have all the bits and pieces collected to complete the proof of Progress:

*Proof of Theorem 3.4 (Progress).* By induction on a derivation of $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ and a case analysis on the last rule applied in the derivation using Lemmas 2.1, 3.2 and 3.5 and Theorem 3.1. The details are listed in Appendix C.4. □

# 4 MODULAR EXTENSIONS

SoundX inherits the "language extensions as libraries" paradigm from Sugar*. In this this approach, extensions are defined in libraries and activated by importing them. The Sugar* compiler makes a few basic assumptions on the module structure of the base language. It considers a module as a sequence of toplevel declarations and distinguishes between namespace, import, body, and extension declarations. A namespace declaration contains the name of the module, an import declaration imports another module, a body declaration is a piece of code of the possibly extended base language like a function or a class definition, and an extension declaration defines a syntactic extension. Whenever the Sugar* compiler processes an import declaration and the imported module contains extension declarations, these extensions are activated for the subsequent definitions. That is, extensions which are defined independently of each other are composed by importing them.

For the type analysis, we assume that the result of checking a module is an interface that describes the entities which are exported by the module. The form and content of this interface depends on the base language. When SoundX processes an import declaration it makes this interface available to the type checking process. At the same time it also activates the typing rules of any extension defined in the imported module. When two modules, each defining an extension, are imported, these two are composed into a new extension, which is the union of the two. Moreover, when a module imports an extension $X_1$ and itself defines an extension $X_2$, $X_2$ may extend $X_1$ by desugaring into code of $X_1$. In the terminology of [EGR12], SoundX supports *extension unification* and *incremental extension*.

In this chapter, we extend the base language definition by a description of the module structure of the base language. Based on the verification and the desugaring procedure of the previous chapter, we develop an analysis including type checking and verification and a desugaring procedure for modules. The analysis verifies extensions modularly: an extension is verified when the containing module is analysed, and it is not necessary to reverify it when it is composed with other imported extension.

## 4.1 A MODULE SYSTEM FOR SIMPLE TYPES

Unlike for expression languages where the literature offers a vast variety of well-studied calculi like $\lambda_{\rightarrow}$, Featherweight Java [IPW01], or System F [Rey74], module systems are far less "settled." For this reason, we design our

own very simple module system around the expression language $\lambda_{\rightarrow}$. This module system fits into the framework prescribed by Sugar* and SoundX and we use it to illustrate how module systems are described in a SoundX base language definition.

A module starts with a header declaring its name followed by zero or more import declarations and ends with a list of function or value definitions. Here is an example module with the name `Foo` that defines two functions `inc` and `twice`:

```
module Foo;

inc = λn:Nat. n+1;
twice = λf:Nat→Nat.λn:Nat. f (f n);
```
[4.1]

In this simple module system all definitions of a module are exported. The following module `Bar` imports `Foo` and uses the two exported functions of `Foo` to define a function `add2`:

```
module Bar;

import Foo;

add2 = twice inc;
```
[4.2]

### 4.1.1 *Describing the module system*

In order to derive a modularly extensible language for $\lambda_{\rightarrow}$, we have to add a definition of the module system to the base language definition. This includes the following elements:

- We label the lexical sort that identifies the namespace (the module's name) by the `sx-namespace-flat` attribute:

  ```
  [a-zA-Z][a-zA-Z0-9]* -> MID {sx-namespace-flat}
  ```
  [4.3]

  SoundX also supports nested namespaces as used in, for example, Haskell by the `sx-namespace-nested` attribute. This attribute takes a character as argument which denotes the namespace separator like `.` in Haskell.

- We indicate which context-free production defines the namespace declaration:

  ```
  "module" mid:MID ";" -> Header {sx-namespace-dec(mid)}
  ```
  [4.4]

  The attribute `sx-namespace-dec(mid)` denotes that `module mid` declares the namespace and that `mid` is the identifier of that namespace.

- We indicate which context-free production defines the import declaration:

```
"import" mid:MID ";" -> Import {sx-import-dec(mid)}
```
[4.5]

Similar to the namespace declaration, we label the identifier `mid` of the imported module in the attribute `sx-import-dec`. Of course, it is possible to have different kinds of import declarations, for example, we could add

```
"import" "qualified" mid:MID ";"
  -> Import {sx-import-dec(mid)}
```
[4.6]

for an import that qualifies the imported definitions by the module name.

- We indicate which context-free production defines a body declaration by `sx-body-dec`:

  ```
  ID "=" Term ";" -> Def {sx-body-dec}
  ```
  [4.7]

  The sorts `ID` and `Term` are from the syntax definitions of Section 1.3.1.

### 4.1.2 *Interfaces*

As mentioned at the beginning of this chapter Sugar* and SoundX expect a module to be a sequence of toplevel declarations. From the previous attributes like `sx-import-dec` or `sx-body-dec`, SoundX knows that the possible set of toplevel declarations consists of `Header`, `Import`, and `Def`. However, SoundX cannot know the conditions under which such a sequence of declarations constituting the module is well-typed and how to obtain the module's interface. These conditions have to be provided in the base language definition by a judgement. For $\lambda_{\to}$, we define a judgement `Γ ⊢ tlds ⇒ Γi` which states that the module comprised by the toplevel declarations `tlds` is well-typed under the environment `Γ` and has the interface `Γi`:

```
judgement forms
  { Env "⊢" ToplevelDecs "⇒" Env }
```
[4.8]

For the entire module `tlds` the judgement `∅ ⊢ tlds ⇒ Γi` must be derivable, and `Γi` constitutes the interface of the module. We encode this requirement in the base language definition by these two lines:

```
interface Env
interface for tlds is Γi derived by ∅ ⊢ tlds ⇒ Γi
```
[4.9]

The first line declares the sort of the interface to be `Env`. The second line declares the judgement which must be derivable for a well-typed module and how the interface can be extracted from a derivation of that judgement. If the declared interface judgement is not derivable, the module contains type errors and cannot be compiled further.

To define the judgement `Γ ⊢ tlds ⇒ Γi` it is necessary to match on the sequence of toplevel declarations. SoundX automatically adds the following

SDF2 productions to the syntax definition:

```
Header -> ToplevelDec                                    [4.10]
Import -> ToplevelDec
Def    -> ToplevelDec

                            -> ToplevelDecs
ToplevelDec ToplevelDecs -> ToplevelDecs
```

The first three lines are generated from the attributes described in the previous section, the last two lines are simply an encoding of a possibly empty sequence. The names of the sequence and the element nonterminal are not predefined but can be configured in the base language definition:

```
toplevel declaration ToplevelDec                         [4.11]
toplevel declarations ToplevelDecs
```

To define the interface judgement we declare that meta-variables with the prefix `tlds` range over toplevel declarations:

```
variables                                                [4.12]
  "tlds" [a-zA-Z0-9]* -> ToplevelDecs
```

The judgement `Γ ⊢ tlds ⇒ Γi` is defined by the following inference rules:

```
Header:                                                  [4.13]
          Γ ⊢ tlds ⇒ Γi
  ------------------------
  Γ ⊢ module mid; tlds ⇒ Γi
Import:
  (Γimp = interface(mid)) (Γ1 = Γ + Γimp) (Γ1 ⊢ tlds ⇒ Γi)
  ------------------------------------------------------
                Γ ⊢ import mid; tlds ⇒ Γi
Def:
  (x∉dom(Γ)) (Γ ⊢ t : T) (Γ,x:T ⊢ tlds ⇒ Γ1) (Γi = ∅,x:T+Γ1)
  ----------------------------------------------------------
                   Γ ⊢ x = t; tlds ⇒ Γi
Empty:
  -------
  Γ ⊢ ⇒ ∅
```

The first rule `Header` simply skips the module header since the name of the module is irrelevant for the interface.

The second rule `Import` employs the predefined judgement `Γimp = interface(mid)` that retrieves the interface of the imported module. Later in Section 4.2.3, we discuss how this judgement is handled formally and in the implementation. The interface `Γimp` of the imported module is added to the current environment in the second premise. To make the imported definitions visible in the body of the module, the remaining toplevel declarations are checked in the enriched environment `Γ1`. The import does not contribute to the interface `Γi` of the module. Therefore, `Γi` is only made up by the subsequent declarations `tlds`.

The third rule `Def` checks that the name of the definition is not yet used (first premise) and that the right-hand side is well-typed according to the typing judgement of Section 1.3.1 (second premise). The third premise brings the current definition into the scope of the remaining toplevel declaration. Our simple module system implements only a top-to-bottom visibility without recursion. The last premise adds the the name and type of the current definition to the interface.

The fourth rule terminates module checking by assigning the empty interface to the empty module.

The definition of the interface judgement uses two auxiliary judgements: `Γ = Γ1 + Γ2` and `x ∉ dom(Γ)`. Both of these judgements are also defined by inference rules, but their definitions are rather technical and we skip them. They are included in the implementation of the case studies in Chapter 5.

## 4.2 MODULE ANALYSIS

The analysis of a module includes a type check according to the interface judgement as declared in the base language definition and the verification of an extension possibly defined in that module. On the abstract level, a module $m$ is a pair of some expression and an extension:

MODULE $\quad m ::= (E, X)$ Modules

If a module does not define an extension this is captured by the empty extension $X = (\langle\rangle, \langle\rangle, \langle\rangle, \langle\rangle)$. The expression $E$ contains the entire code of the module consisting of namespace, import, and body declarations.

The analysis assigns a module binding $\rho$ to a module. A binding associates the identifier of the module (an expression) with the module's exports $\Sigma$. The exports comprise the interface (also an expression) and a list of extensions:

BINDING-EXPORT $\quad \rho ::= E : \Sigma$ Module bindings

$\Sigma ::= (E, \vec{X})$ Exports

A module exports its own extension as well as all imported extensions. Therefore, $\Sigma$ includes a list of extensions. It is easy to see that it is necessary to re-export imported extensions. Consider the following example where a module `A` imports the pair extension:

```
module A;                                                    [4.14]
import PairExtension;
a = (1,2);
```

The interface of this module is `∅,a:Pair Nat`.

Now suppose a module `B` importing `A`:

```
module B;                                                    [4.15]
import A;
b = a;
```

The typing derivation of the right-hand side of `b` ends as follows:

```
-------------------------Lookup
a:Pair Nat ∈ ∅,a:Pair Nat
-----------------------------Var                            [4.16]
  ∅,a:Pair Nat ⊢ a : Pair Nat
```

This derivation is only a valid derivation in the extended language and it is only possible to desugar it with the pair extension activated. Therefore, the import of `A` also brings the extension defined in the module `PairExtension` into scope.

The analysis of a module takes place in the context of other modules which are available for import. We call this context a *module repository* which consists of a list of module bindings $\vec{\rho}$. In the remainder of this section we define the statement $\vec{\rho} \Vdash m : (\rho, \nabla)$, which expresses that a module $m$ is valid under the module repository $\vec{\rho}$, that $m$ is assigned the module binding $\rho$, and that $\nabla$ is a valid derivation of the interface judgement.

In order to analyse the module, we need various technical aspects of the module system definition of the base language. We extend a base language definition by a fourth component $M$ which defines the module system:

$$B ::= (\vec{A}, \vec{F}, \vec{I}, M)$$

To streamline the description of the analysis, we take this module system definition $M$ as an opaque object and retrieve information from $M$ on demand. At the end, in Section 4.2.4, we give a concrete definition of $M$, conditions for its well-formedness, and relate it to the concrete representation of Section 4.2.4. We think it is easier to understand the design of $M$ if we first explain how it is used instead of beginning with a vacuous enumeration of its content.

### 4.2.1 *Structure analysis*

For a given module $(E, X)$, we first analyse its structure. We have to extract the identifier of the module and the list of imported modules. As we described at the beginning of this chapter, SoundX requires that a module is a sequence of toplevel declarations. To encode this on the abstract level, $M$ contains two constructor names $DCons(M)$ and $DNil(M)$ that form this sequence.

Corresponding to the `sx-namespace-dec` attribute, $M$ contains the constructor name $Head(M)$ declaring the name of the namespace constructor. The argument to $Head(M)$ at list position $midH(M)$ is the identifier of the module.

Similarly, corresponding to the `sx-import-dec` attribute, $M$ contains a list of import declarations $Imp(M)$. This list consists of pairs $N/i$ naming the import constructor and the argument position of the imported module.

All these constructors are included in the syntax definition of the base language, that is, they have declared arities. To illustrate the structure, we show the abstract syntax of the module `Bar` from Example [4.2] with this definition of $M$:

$$\begin{aligned} DCons(M) &= \texttt{dcons} & DNil(M) &= \texttt{dnil} \\ Head(M) &= \texttt{head} & midH(M) &= 1 \\ Imp(M) &= \langle \texttt{import}/1 \rangle \end{aligned}$$

$$\begin{array}{lcl} & & \texttt{dcons}\langle \\ \texttt{module Bar;} & \widehat{=} & \quad \texttt{head}\langle \texttt{mid}_{\texttt{Bar}}\langle\rangle\rangle, \\ & & \quad \texttt{dcons}\langle \\ \texttt{import Foo;} & \widehat{=} & \quad\quad \texttt{import}\langle \texttt{mid}_{\texttt{Foo}}\langle\rangle\rangle, \\ & & \quad\quad \texttt{dcons}\langle \\ \texttt{add2 = ...;} & \widehat{=} & \quad\quad\quad \texttt{def}\langle \texttt{id}_{\texttt{add2}}\langle\rangle, \ldots\rangle, \\ & & \quad\quad\quad \texttt{dnil}\langle\rangle\rangle\rangle \end{array} \qquad [4.17]$$

Based on this structure we define two simple selection statements to extract the identifier of the module and the list of imported modules.

The statement $E \; {-}{:}_M \; E_{mid}$ extracts the module identifier $E_{mid}$ from the module $E$. It traverses the list formed by $DCons(M)$ and returns the identifier mentioned in the first occurrence of $Head(M)$. We use the exclamation mark to select the $k$-th element of a list: $\langle \phi_1, \ldots, \phi_k, \ldots, \varphi_m \rangle ! \, k = \phi_k$.

EH-Header
$$\frac{N = Head(M)}{DCons(M)\langle N \; \vec{E}, E\rangle \; {-}{:}_M \; \vec{E} ! \, midH(M)}$$

EH-Skip
$$\frac{N \neq Head(M) \qquad E \; {-}{:}_M \; E_{mid}}{DCons(M)\langle N \; \vec{E}, E\rangle \; {-}{:}_M \; E_{mid}}$$

Similarly, the statement $E \; {-}{:}_M \; \vec{E}_{imp}$ extracts the identifiers of all imported modules by traversing the list of toplevel declarations:

EI-Import
$$\frac{(N/i) \in Imp(M) \qquad E \; {-}{:}_M \; \vec{E}_{imp}}{DCons(M)\langle N \; \vec{E}, E\rangle \; {-}{:}_M \; \langle \vec{E} ! \, i, \vec{E}_{imp} \rangle}$$

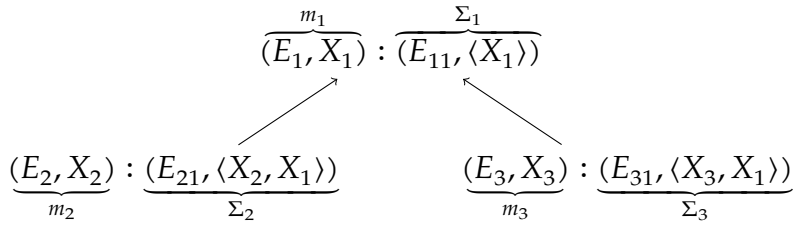
EI-Skip
$$\frac{\begin{array}{c} \text{for each } (N_1/i_1) \in Imp(M) : N_1 \neq N \\ E \; {-}{:}_M \; \vec{E}_{imp} \end{array}}{DCons(M)\langle N \; \vec{E}, E\rangle \; {-}{:}_M \; \vec{E}_{imp}}$$

In rule EI-Import, the $i$-th argument of the import declaration is added to the list of imported modules if the current constructor $N$ is an import declaration, that is, it is contained in $Imp(M)$. Conversely, if the current constructor $N$ is not found in $Imp(M)$ (rule EI-Skip) the traversal proceeds with the next toplevel declaration.

### 4.2.2 *Extension analysis*

After the structure analysis, SoundX composes the imported extensions and verifies the extension implemented in the module itself.

To illustrate the composition of extensions, we consider an example. Suppose, the module repository contains the module bindings of three modules $m_1$, $m_2$, and $m_3$. The module $m_1$ has no imports but the other two modules both import $m_1$. We depict this situation by the following diagram where the arrow $m_2 \to m_1$ is to be understood as "$m_2$ imports $m_1$" and we annotate the modules with their exports:




The module $m_1$ exports one extension $X_1$. Both modules $m_2$ and $m_3$ define an extension $X_2$ and $X_3$, respectively, and import $X_1$. As we already mentioned at the beginning of this section, imported extension are re-exported. Therefore, $m_2$ exports $X_2$ and $X_1$ and $m_3$ exports $X_3$ and $X_1$. The exported extension are sorted in dependency order from left to right: $X_2$ possibly depends on $X_1$ and $X_3$ possibly depends on $X_1$.

We consider a module $m_4 = (E_4, X_4)$ which first imports $m_2$ and then $m_3$. The exports of $m_4$ contain $X_4$ and the extensions imported from $m_2$ and $m_3$. Naively, this would be the list $\langle X_4, X_2, X_1, X_3, X_1 \rangle$ which is in dependency order but contains $X_1$ twice. As we see later in Section 4.3, SoundX performs one pass of desugaring per imported extension. To avoid unnecessary passes, we would like to drop one occurrence of $X_1$. With Lemma 4.2(iii), which we develop in Section 4.4, we justify that it is safe to drop the left occurrence of $X_1$ resulting in the list $\langle X_4, X_2, X_3, X_1 \rangle$. This list respects the dependencies $X_2 \to X_1$ and $X_3 \to X_1$. The following "nub" concatenation function $\vec{X}_1 \twoheadrightarrow \vec{X}_2$ joins two lists of extensions $\vec{X}_1$ and $\vec{X}_2$ but includes only those extension of $\vec{X}_1$ which are not contained in $\vec{X}_2$:

ExtNub
$$\langle\rangle \twoheadrightarrow \vec{X} = \vec{X}$$
$$\langle X, \vec{X}_1 \rangle \twoheadrightarrow \vec{X}_2 = \begin{cases} \langle X, \vec{X}_1 \twoheadrightarrow \vec{X}_2 \rangle, & \text{if } X \notin \vec{X}_2 \\ \vec{X}_1 \twoheadrightarrow \vec{X}_2, & \text{if } X \in \vec{X}_2 \end{cases}$$

Using the list of imported modules $\vec{E}_{imp}$ from the structure analysis, a module repository $\vec{\rho}$, a base language definition $B$, and the previously defined nub concatenation function we define a statement $\vec{E}_{imp} \; -:_{B;\vec{\rho}} \; \vec{X}$ which composes an extension list from the list of imports:

CX-Empty
$$\frac{}{\langle\rangle \; -:_{B;\vec{\rho}} \; \langle\rangle}$$

CX-Exts
$$\frac{E : (E_{intf}, \vec{X}_1) \in \vec{\rho} \qquad \vec{E} \; -:_{B;\vec{\rho}} \; \vec{X}_2 \qquad B \; \& \vdash \; (\vec{X}_1 \twoheadrightarrow \vec{X}_2)}{\langle E, \vec{E}\rangle \; -:_{B;\vec{\rho}} \; (\vec{X}_1 \twoheadrightarrow \vec{X}_2)}$$

The rule CX-Exts picks the exports of an imported module from the repository (first premise) and then composes the remaining imports $\vec{E}$ (second premise). Finally, it checks that the nub-concatenation of all the imported extension is well-formed with respect to $B$ (third premise). Essentially, this condition checks that there are no constructor and inference rule name clashes.

A successful composition of extensions provides the statement $\vec{E}_{imp} \; -:_{B;\vec{\rho}} \; \vec{X}_{imp}$ which is the precondition for the next step: the verification of the extension of the current module. In our example this amounts to the verification of $X_4$.

The extension of the current module is verified with respect to the base language merged with the imported extensions. That is, relative to the current module, these extensions are considered to be part of the base language. SoundX first merges $B$ and $\vec{X}_{imp}$ into $B_x$ with the following function:

BXMerge
$$B \uplus \langle\rangle = B$$
$$(\vec{A}, \vec{F}, \vec{I}, M) \uplus \langle(\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x), \vec{X}\rangle = (\langle\vec{A}, \vec{A}_x\rangle, \vec{F}, \langle\vec{I}, \vec{I}_x\rangle, M) \uplus \vec{X}$$

The merge function $B \uplus \vec{X}$ adds the constructor arities and the inference rules of the extension $\vec{X}$ to the respective components of the base language definition $B$. To assert the soundness of the extension $X$ of the current module, SoundX tries to establish the two statements

$$B_x \; \& \vdash \; X \qquad \text{and} \qquad B_x \ltimes X$$

requiring $X$ to be well-formed with respect to the extended base language and to be sound with respect to the extended base language. In our example, $X_4$ has to satisfy the two statements

$$B \uplus \langle X_2, X_3, X_1\rangle \; \& \vdash \; X_4 \qquad \text{and} \qquad B \uplus \langle X_2, X_3, X_1\rangle \ltimes X_4 \, .$$

This constitutes the extension analysis of a module. The most important aspect of this analysis is that it is sufficient to ensure the well-formedness of the imported extension (in rule CX-Exts) to retain soundness. A re-verification of the composed extensions is not necessary. Moreover, it is also possible to verify the soundness of an extension like $X_4$ separately from the soundness of the imported extensions.

The well-formedness condition of the extension composition in rule CX-Exts and of extension verification implies that no syntactic overlap or ambiguity can arise from the merging of the base language with the imported and the defined extensions. This is because well-formedness requires that the names of all declared constructors are pairwise different and that only declared constructors may be applied in expressions. The issue of well-formedness only arises on the level of the abstract syntax. In the SoundX implementation, a concrete syntax is declared which is automatically mapped to a well-formed abstract syntax. However, there may be ambiguities on the concrete level, since two independently developed extensions might define conflicting syntax. In this case, the SDF2 parser implementation returns all possible parse trees and marks the result as ambiguous. At this point, Sugar* stops processing with an error since it does not handle ambiguous outcomes. Hence, syntactic ambiguity on the level of concrete syntax is not an issue for the SoundX implementation. It solely works with unambiguous abstract syntax which is enforced to be well-formed.

### 4.2.3 *Type checking*

The type correctness of a module is solely determined by the interface judgement as declared in the base language definition. In the type checking phase, SoundX establishes a derivation of this interface judgement for the current module. The interface judgement $\mathit{intfJ}(M)$ contains meta-variables as placeholders for the code of the current module, the interface of the current module, and the module repository. To establish a derivation of this interface judgement, SoundX performs the following steps:

1. Plug in the code of the module into $\mathit{intfJ}(M)$.
2. Plug in the module repository into $\mathit{intfJ}(M)$.
3. Establish a derivation $\nabla$ for the judgement obtained by the previous steps that instantiates the placeholder for the module's interface.
4. Extract the module's interface from the conclusion of the derivation $\nabla$.

The placeholders in the judgement $\mathit{intfJ}(M)$ are expressions which are matched with the concrete inputs. The placeholders for the code of the module and the interface are included in the base language definition for $\lambda_{\rightarrow}$ by the line

```
interface for tlds is Γi derived by ∅ ⊢ tlds ⇒ Γi   [4.18]
```

The expression `tlds` is the placeholder for the entire module and `Γi` is the placeholder for the interface. In this example, these expressions are single meta-variables but this is not a requirement.

In the first step, SoundX matches the module expression $E$ with its placeholder $tldsE(M)$ to obtain a substitution $\sigma_1$:

$$E = [\sigma_1](tldsE(M))$$

The second step is a bit more involved and requires some elaboration. In the example of the $\lambda_{\rightarrow}$ module system, we used the predefined judgement `Γimp = interface(mid)` to retrieve an interface. In the formal development we do not consider this judgement as predefined but as inductively defined like the typing or interface judgement. This simplification makes the meta-theory more uniform and direct. Without the predefined judgement we lift the module repository onto the level of the base language. We define a syntactic sort `Rep` that encodes the association list of module identifiers and interfaces and a meta-variable `R` ranging over repositories:

```
context-free syntax                                  [4.19]
                        -> Rep
  MID ":" Env "," Rep -> Rep

variables
  "R" [a-zA-Z0-9] -> Rep
```

The retrieval of an interface is defined by a lookup judgement `R | Γi = interface(mid)` that traverses the repository `R`:

```
judgement forms                                      [4.20]
  { Rep "|" Env "=" "interface" "(" MID ")" }

inductive definitions
  Intf:
    ---------------------------------
    mid : Γi, R | Γi = interface(mid)
  IntfSkip:
    (mid1 ≠ mid) (R | Γi = interface(mid))
    ---------------------------------------
     mid1 : Γi1, R | Γi = interface(mid)
```

We augment the interface judgement with an additional argument for the repository and adapt its rules accordingly. Here is the adaption of the `Import` rule:

```
ImportRep:                                           [4.21]
  (R | Γimp = interface(mid)) (Γ1 = Γ+Γimp) (Γ1 ⊢ tlds ⇒ Γi)
  ----------------------------------------------------------
              R | Γ ⊢ import mid tlds ⇒ Γi
```

Finally, we add a placeholder `R` for the repository similar to the placeholder for the interface:

```
repository Rep                                                    [4.22]
interface for tlds from R is Γi derived by R|∅ ⊢ tlds ⇒ Γi
```

This lifting of the module repository is systematic and identical for every base language: it always consists of the definition of an association list, a judgement form, and an augmentation of all judgements with an additional argument. For practical reasons, the SoundX implementation provides the predefined judgement `Γ = interface(mid)`, which behaves like the inductively defined judgement outlined above. The additional argument `R` is implicit to reduce boilerplate.

We return to the explanation of the second step of the type checking phase. Similar to the module code we plug the module repository into the interface judgement. That is, SoundX matches the repository placeholder $repE(M)$ with the lifted repository $repexpr(M, \vec{\rho})$ to obtain another substitution $\sigma_2$:

$$[\sigma_2](repE(M)) = repexpr(M, \vec{\rho})$$

The function *repexpr* lifts $\vec{\rho}$ onto the level of the base language using the association list constructors $RNil(M)$ and $RCons(M)$:

Rep-Expr
$$repexpr(M, \langle\rangle) = RNil(M)$$
$$repexpr(M, \langle E_{mid} : (E, \vec{X}), \vec{\rho}\rangle) = RCons(M)(E_{mid}, E, repexpr(M, \vec{\rho}))$$

In the previous concrete encoding, the constructor $RNil(M)$ is provided by the production `-> Rep` and the constructor $RCons(M)$ is provided by the production `MID ":" Env "," Rep -> Rep`.

For the third step of the type checking phase, SoundX replaces the placeholders for the module code and the repository in $intfJ(M)$ by the substitutions $\sigma_1$ and $\sigma_2$ to obtain a goal judgement $J = [\sigma_1 \circ \sigma_2](intfJ(M))$. The proper type check is performed by deriving a substitution instance of $J$. The type check uses the inference rules of the base language plus the imported inference rules, that is, all inference rules $I_x$ of $B_x$ from the extension analysis phase. This implies that an extension defined in a module cannot yet be applied inside that module. It must be explicitly enabled by importing it into another module. This behaviour is consistent with the "language extensions as libraries" paradigm as provided by the underlying Sugar* implementation. Precisely formulated, SoundX derives the following statement:

$$\langle\rangle \vdash_{\vec{I}_x} \nabla \qquad \text{with} \qquad concl(\nabla) = [\sigma](J)$$

Finally, the fourth step extracts the interface from the result of type checking by applying the substitution $\sigma$ from the third step to the interface

placeholder $intfE(M)$. For a successfully analysed module $m = (E, X)$ we obtain the statement

$$\vec{\rho} \Vdash \underbrace{(E, X)}_{m} : \underbrace{(E_{mid} : \underbrace{([\sigma](intfE(M)), \langle X, \vec{X}_{imp} \rangle)}_{\Sigma}, \nabla)}_{\rho}$$

with $E_{mid}$ and $\vec{X}_{imp}$ from the first and second phase.

The following definition summarises the three phases of the module analysis:

A-Module

$$\left.\begin{array}{c} E \ -:_M \vec{E}_{imp} \\ E \ -:_M E_{mid} \end{array}\right\} \text{structure analysis}$$

$$\left.\begin{array}{c} \vec{E}_{imp} \ -:_{(\vec{A},\vec{F},\vec{I},M);\vec{\rho}} \vec{X}_{imp} \\ (\vec{A}_x, \vec{F}_x, \vec{I}_x, M_x) = (\vec{A}, \vec{F}, \vec{I}, M) \Cup \vec{X}_{imp} \\ (\vec{A}_x, \vec{F}_x, \vec{I}_x, M_x) \Subset X \\ (\vec{A}_x, \vec{F}_x, \vec{I}_x, M_x) \ltimes X \end{array}\right\} \text{extension analysis}$$

$$\left.\begin{array}{c} [\sigma_1](tldsE(M)) = E \\ [\sigma_2](repE(M)) = repexpr(M, \vec{\rho}) \\ [\sigma_1 \circ \sigma_2](intfJ(M)) = J \\ \langle\rangle \vdash_{\vec{I}_x} (\vec{\nabla} \Rightarrow^N [\sigma](J)) \end{array}\right\} \text{type checking}$$

$$\vec{\rho} \Vdash_{(\vec{A},\vec{F},\vec{I},M)} (E, X) : (E_{mid} : ([\sigma](intfE(M)), \langle X, \vec{X}_{imp} \rangle), \vec{\nabla} \Rightarrow^N [\sigma](J))$$

The list of imported extensions $\vec{X}_{imp}$ and the derivation of the interface judgement $(\vec{\nabla} \Rightarrow^N [\sigma](J))$ are the input for the module desugaring (see Section 4.3).

### 4.2.4 *Module system definition*

We have introduced various components of the module system definition $M$ in the previous sections. We summarise these aspects by the following concrete definition of $M$:

Mod-Sys-Def

$$\begin{array}{lll} M ::= & (N_{intf}, J_{intfJ}, E_{tldsE}, E_{intfE}, E_{repE}, & \text{Module system definition} \\ & N_{rep}, N_{RNil}, N_{RCons}, & \\ & N_{tld}, N_{tlds}, N_{DNil}, N_{DCons}, & \\ & N_{mid}, N_{Head}, i_{midH}, & \\ & \vec{P}_{Imp}) & \\ P ::= & N/i & \text{Import declaration} \end{array}$$

To avoid lengthy function definitions we make use of the following convention: We take a phrase $\phi_{sel}$ of the right-hand side of $M$ as the implicit definition of the selection function $sel(..., \phi_{sel}, ...) = \phi_{sel}$. This way, all the selections that we already used are defined.

A module system definition must be well-formed with respect to the constructor arities and judgement forms of the base language definition it belongs to. We review the different components of $M$ and identify well-formedness conditions based on the usage in the module analysis. For a given $M$, $\vec{A}$, and $\vec{F}$ the following conditions establish well-formedness of $M$, written as $\vec{A}; \vec{F} \ \& \vdash M$:

1. The interface judgement $intfJ(M)$ is well-formed: $\vec{A}; \vec{F} \ \& \vdash intfJ(M)$.
2. The placeholder for the module code $tldsE(M)$ has the declared sort $tlds(M)$ of the toplevel declaration list: $\vec{A} \ \& \vdash tldsE(M) : tlds(M)$.
3. The placeholder for the interface $intfE(M)$ is of the declared interface sort $intf(M)$: $\vec{A} \ \& \vdash intfE(M) : intf(M)$.
4. The placeholder for the module repository $repE(M)$ is of the declared repository sort $rep(M)$: $\vec{A} \ \& \vdash repE(M) : rep(M)$.
5. The constructors for the lifting of the repository to the base language $RNil(M)$ and $RCons(M)$ have declared arities of the form $(RNil(M) : \langle\rangle \to rep(M))$ and $(RCons(M) : \langle mid(M), intf(M), rep(M)\rangle \to rep(M))$ in $\vec{A}$.
6. The constructors for the list of toplevel declarations $DNil(M)$ and $DCons(M)$ have declared arities in $\vec{A}$ of the forms $(DNil(M) : \langle\rangle \to tlds(M))$ and $(DCons(M) : \langle tld(M), tlds(M)\rangle \to tlds(M))$, where $tld(M)$ is the sort of a single toplevel declaration.
7. The constructor for a namespace declaration $Head(M)$ has a declared arity in $\vec{A}$ of the form $(Head(M) : \vec{N}_{Head} \to tld(M))$. The sort at position $midH(M)$ in $\vec{N}_{Head}$ is the declared sort of module identifiers $mid(M)$, that is, $\vec{N}_{Head}!\, midH(M) = mid(M)$.
8. A similar requirement is satisfied by the import declarations. For each $(N/i) \in Imp(M)$, $N$ is of declared arity in $\vec{A}$ of the form $N : \vec{N}_N \to tld(M)$ where $\vec{N}_N!\, i = mid(M)$.

## 4.3 MODULE DESUGARING

The module desugaring performs one pass of desugaring for each imported extension. It applies the desugaring procedure of Section 3.5 iteratively for each extension.

Module desugaring is implemented by the statement $\nabla \Updownarrow_{B;\vec{X}} \nabla'$ which differs from the top-down/bottom-up desugaring by acting on a list of extensions $\vec{X}$ instead of a single extension. It is defined by the following fourxs rules:

D-Empty
$$\frac{}{\nabla \Updownarrow_{B;\langle\rangle} \nabla}$$

D-Ext
$$\frac{\nabla \Updownarrow_{B \cup \vec{X};X} \nabla' \qquad \nabla' \Updownarrow_{B;\vec{X}} \nabla''}{\nabla \Updownarrow_{B;\langle X,\vec{X}\rangle} \nabla''}$$

D-Stuck1
$$\frac{\nabla \Updownarrow_{B \cup \vec{X};X} \lightning}{\nabla \Updownarrow_{B;\langle X,\vec{X}\rangle} \lightning}$$

D-Stuck2
$$\frac{\nabla \Updownarrow_{B \cup \vec{X};X} \nabla' \qquad \nabla' \Updownarrow_{B;\vec{X}} \lightning}{\nabla \Updownarrow_{B;\langle X,\vec{X}\rangle} \lightning}$$

Basically, the first two rules implement an iterated application of the desugaring procedure along the list of extensions $\vec{X}$ from left to right. But since an extension $X$ may depend on extensions $\vec{X}$ further to the right in the list, these extension are merged with the base language $B$ for the desugaring of $X$ (first premise of D-Ext). As defined in D-Stuck1 and D-Stuck2 module desugaring is stuck as soon as the desugaring of an individual extension is stuck. Subsequent extensions are not considered anymore.

With this definition of module desugaring it is clear why the desugaring procedure transforms derivations into derivations. The result of desugaring one extension is the input of the next pass of desugaring. We have already established that the desugaring of a single extension is a valid derivation with the proviso that this extension is sound. We transfer this result to modular desugaring in the next section.

## 4.4 SOUNDNESS

To prove the soundness of module desugaring we need a few definitions first. We call a list of extensions $\vec{X}$ well-formed with respect to a base language definition $B$ if the statement $B \vdash \vec{X}$ can be derived by the following two rules:

WX-Empty
$$\frac{}{B \vdash \langle\rangle}$$

WX-Exts
$$\frac{\vdash B \cup \vec{X} \qquad B \cup \vec{X} \vdash X \qquad B \vdash \vec{X}}{B \vdash \langle X, \vec{X}\rangle}$$

The tail of $\vec{X}$ merged with $B$ must be a well-formed base language definition (first premise of WX-Exts), the head of $\vec{X}$ must be well-formed with respect to that extended base language, and the tail itself must be well-formed with respect to $B$. The pattern of merging the extensions and the base system follows the definition of module desugaring in D-Ext.

Similarly, we call a list of extensions $\vec{X}$ sound with respect to a base language definition $B$ if the statement $B \ltimes \vec{X}$ can be derived by the following two rules:

SX-Empty
$$\frac{}{B \ltimes \langle\rangle}$$

SX-Exts
$$\frac{\wp\!- B \Cup \vec{X} \qquad B \Cup \vec{X} \ltimes X \qquad B \ltimes \vec{X}}{B \ltimes \langle X, \vec{X}\rangle}$$

This definition follows the same pattern as the well-formedness statement replacing $\wp\!-$ with $\ltimes$.

Using the previous definition we formulate the first lemma that provides Preservation and Progress for well-formed and sound composition of extensions:

Lemma 4.1 Preservation and Progress for composed extensions
*Let $B = (\vec{A}, \vec{F}, \vec{I}, M)$ be a well-formed base language definition, $\vec{X}$ a list of extensions with $B \wp\!- \vec{X}$ and $B \ltimes \vec{X}$ where $(\vec{A}_x, \vec{F}_x, \vec{I}_x, M_x) = B \Cup \vec{X}$.*

(i) *If $\langle\rangle \vdash_{\vec{I}_x} \nabla$, and $\nabla \Updownarrow_{B;\vec{X}} \nabla'$ then $\langle\rangle \vdash_{\vec{I}} \nabla'$.*

(ii) *If $\langle\rangle \vdash_{\vec{I}_x} \nabla$, and the universal desugarings of each $X \in \vec{X}$ form a terminating rewrite system then either $\nabla \Updownarrow_{B;\vec{X}} \lightning$ or there exists a $\nabla'$ such that $\nabla \Updownarrow_{B;\vec{X}} \nabla'$.*

*Proof.* By induction on the structure of $\vec{X}$ using Theorems 3.1 and 3.4. The details are listed in Appendix C.6. □

The previous lemma requires the well-formedness and soundness of the list of extensions $\vec{X}$. The module analysis of SoundX provides the statement $\vec{\rho} \Vdash m : (\rho, \nabla)$. In order to show that this statement implies the well-formedness and soundness of the exported extensions in $\rho$ we need some additional properties. They are stated in Lemma 4.2, where we use the notation $B \sqsubseteq B_1$ as an abbreviation for the following list inclusions:

Base-Incl $(\vec{A}, \vec{F}, \vec{I}, M) \sqsubseteq (\vec{A}_1, \vec{F}_1, \vec{I}_1, M) := \vec{A} \sqsubseteq \vec{A}_1$ and $\vec{F} \sqsubseteq \vec{F}_1$ and $\vec{I} \sqsubseteq \vec{I}_1$

The first part of Lemma 4.2 expresses that a well-formed extension $X$ can be merged with its base language $B$ to build a well-formed base language.

The second part states that a sound extension is also sound with respect to an enlarged base language. The third part is the key property to drop duplicate extensions during their composition. Two lists of well-formed and sound extensions which can be composed into a well-formed base language can be "nub" concatenated into a list of sound extensions.

Lemma 4.2 Properties of extensions

(i) *If* $\mathfrak{E} B$ *and* $B \mathfrak{E} \vec{X}$ *then* $\mathfrak{E} B \Cup \vec{X}$.

(ii) *If* $B \ltimes X$ *and* $B \sqsubseteq B_1$ *then* $B_1 \ltimes X$.

(iii) *If* $\mathfrak{E} B$, $B \mathfrak{E} \vec{X}_1$, $B \ltimes \vec{X}_1$, $B \mathfrak{E} \vec{X}_2$, $B \ltimes \vec{X}_2$, *and* $\mathfrak{E} B \Cup (\vec{X}_1 \twoheadrightarrow \vec{X}_2)$ *then* $B \ltimes (\vec{X}_1 \twoheadrightarrow \vec{X}_2)$.

*Proof.*

(i) Immediate by the definitions of the statements $\mathfrak{E} B$ and $B \mathfrak{E} X$.

(ii) By a case analysis on the derivation of $B \ltimes X$.

(iii) By induction on the structure of $\vec{X}_1$ using Lemma 4.2(ii).

The details are listed in Appendix C.7. □

An export $\Sigma = (E, \vec{X})$ is well-formed and sound with respect to a base language definition, written $B \ltimes \Sigma$, if $\vec{X}$ is well-formed and sound with respect to $B$:

WS-Exports
$$\frac{B \mathfrak{E} \vec{X} \qquad B \ltimes \vec{X}}{B \ltimes (E, \vec{X})}$$

A module repository $\vec{\rho}$ is well-formed and sound with respect to a base language definition $B$, written $B \ltimes \vec{\rho}$, if all its exports are well-formed and sound with respect to $B$:

WSR-Empty
$$\frac{}{B \ltimes \langle\rangle}$$

WSR-Bindings
$$\frac{\begin{array}{c}(\vec{A}, \vec{F}, \vec{I}, M) \ltimes \Sigma \\ \vec{A} \mathfrak{E} E : \mathit{mid}(M) \\ \text{for each } (E_1 : \Sigma_1) \in \vec{\rho} : E_1 \neq E \\ (\vec{A}, \vec{F}, \vec{I}, M) \ltimes \vec{\rho}\end{array}}{(\vec{A}, \vec{F}, \vec{I}, M) \ltimes \langle E : \Sigma, \vec{\rho}\rangle}$$

Moreover, as required by the third premise of WSR-Bindings, all module-identifiers contained in the repository must be unique.

The next lemma shows that imported extensions are well-formed and sound, provided they are imported from a sound repository.

Lemma 4.3 Soundness of imports
*If* $\mathfrak{E}\vdash B, B \ltimes \vec{\rho}$ *and* $\vec{E} -:_{B;\vec{\rho}} \vec{X}$ *then* $B\ \mathfrak{E}\vdash \vec{X}$ *and* $B \ltimes \vec{X}$.

*Proof.* The first part is immediate by the definition of $\vec{E} -:_{B;\vec{\rho}} \vec{X}$. We prove the second part by induction on the derivation of $\vec{E} -:_{B;\vec{\rho}} \vec{X}$ and a case analysis on the last rule applied. The details are listed in Appendix C.8. □

Now we prove the following theorems which lift Preservation and Progress onto the level of module analysis and desugaring which establish the soundness of modular extensions:

Theorem 4.4 Preservation of module desugaring
*Let* $B = (\vec{A}, \vec{F}, \vec{I}, M)$ *be a well-formed base language definition.*
*If* $B \ltimes \vec{\rho}, \vec{\rho} \Vdash_B m : (E_{mid} : (E_{intf}, \langle X, \vec{X}\rangle), \nabla)$, *and* $\nabla \Updownarrow_{B;\vec{X}} \nabla'$ *then* $\langle\rangle \vdash_{\vec{I}} \nabla'$.

*Proof.* By Lemmas 4.1(i) and 4.3. The details are listed in Appendix C.9. □

Theorem 4.5 Progress of module desugaring
*Let* $B = (\vec{A}, \vec{F}, \vec{I}, M)$ *be a well-formed base language definition.*
*If* $B \ltimes \vec{\rho}, \vec{\rho} \Vdash_B m : (E_{mid} : (E_{intf}, \langle X, \vec{X}\rangle), \nabla)$, *and the universal desugarings of each* $X \in \vec{X}$ *form a terminating rewrite system then either* $\nabla \Updownarrow_{B;\vec{X}} \not\downarrow$ *or there exists a* $\nabla'$ *such that* $\nabla \Updownarrow_{B;\vec{X}} \nabla'$.

*Proof.* By Lemmas 4.1(ii) and 4.3. The details are listed in Appendix C.10. □

Since the previous theorems require a well-formed and sound repository it is necessary that the SoundX module analysis provides well-formed and sound exports. This property is guaranteed by the following theorem:

Theorem 4.6 Soundness of module analysis
*Let B be a well-formed base language definition.*
*If* $B \ltimes \vec{\rho}$ *and* $\vec{\rho} \Vdash_B m : (E_{mid} : \Sigma, \nabla)$ *then* $B \ltimes \Sigma$.

*Proof.* The theorem is a direct consequence of the rule A-Module and Lemmas 4.2(i) and 4.3. The details are listed in Appendix C.11. □

The last theorem is of essential practical relevance. It allows to build a sound module repository starting from the empty repository by analysing a set of modules in dependency order. As of the time being, the dependency relation has to be acyclic. The underlying Sugar* provides SoundX with the necessary dependency tracking such that modules are analysed in dependency order.

# 5 REALISATION AND APPLICATION

To evaluate SoundX practically, we have implemented it in the Sugar* framework and conducted several case studies using $\lambda_{\rightarrow}$ as the base language. The Sugar* framework provides a batch compiler for the extensible language as well as a plugin for the Eclipse IDE that includes syntax highlighting, automatic builds, and error detection. The implementation and the case studies are available from the WWW at `http://www.user.tu-berlin.de/florenz/soundx`.

## 5.1 PRACTICAL REALISATION

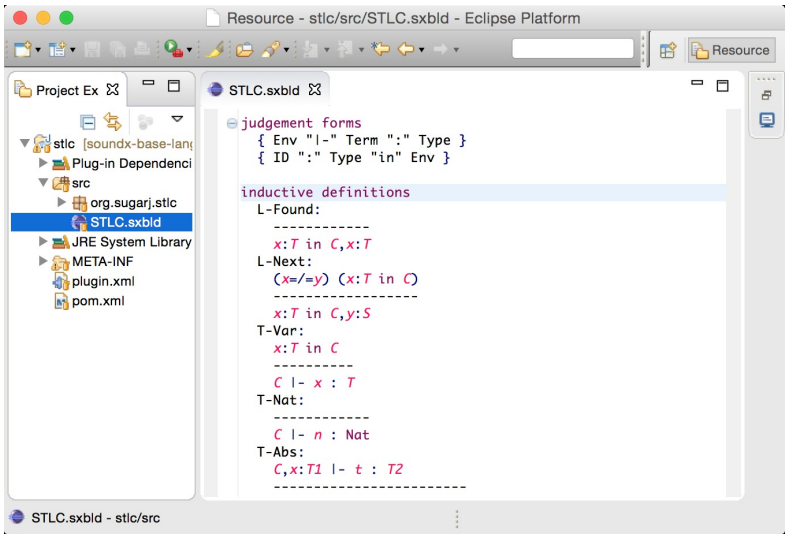
The SoundX implementation is a library for the implementation of Sugar* plugins which add type-sound syntactic extensibility to base languages. To support a new base language it is sufficient to provide a base language definition and a small Java class to load the base language definition using the SoundX library. SoundX analyses the base language definition and provides a description of the modular structure for Sugar* [ER13]. With this description the Sugar* compiler is able to process the modules of the base language in dependency order. For each input module, Sugar* generates three files: the code of the desugared module, an SDF2 file containing syntax definitions declared in the module, and a Stratego file containing desugarings and analyses declared in the module. As we already sketched in Chapter 4, Sugar* considers a module as a sequence of toplevel declarations which are either a namespace, an import, a body, or an extension declaration. When Sugar* processes an import declaration the extension defined in that module is activated. This means that the SDF2 file is loaded into the parser to recognise the extended syntax and that the analyses and desugarings of the Stratego file are activated for the analysis and the desugaring phase. When Sugar* processes an extension declaration it adds the SDF2 components to the current module's SDF2 output file and the Stratego components to the current modules's Stratego output. This way the extension is available for other modules by importing it. When Sugar* processes a body declaration it runs the currently activated analyses and desugarings on it and adds the result to the current module's code file.

The SoundX library hooks itself into the analysis and desugaring phase of Sugar*. In the analysis phase SoundX performs a type check of the entire module building the derivation for the interface judgement. In the desugaring phase this derivation is desugared and the resulting module is extracted and written into the code file. Moreover, the interface of the

module is stored in the Stratego file to be available using the predefined `interface` judgement.

Since a SoundX extension definition does not contain Stratego code for the analysis and desugaring phases but inference rules and guarded and universal desugarings, SoundX adapts the way how Sugar* processes extension declarations. First of all, the concrete syntax of guarded and universal desugarings as well as inference rules is added to the parser for extension declaration. For the offside rule parsing of inference rules, we employ the layout-sensitive extension of the SDF2 implementation, which provides a declarative specification of layout constraints [ERRO12, ERKO13]. After parsing, the guarded and universal desugarings are translated into Stratego rewrite rules, which are stored in the Stratego file and can be activated by the module desugaring procedure. The inference rules are converted into abstract syntax and are also stored in the Stratego file. During the type check all inference rules of the base language definition and the activated extension are collected and made available to build the derivation of the interface judgement. SoundX also runs the verification procedure for the extension definition using the Stratego code that represents the inference rules and desugarings.

The loading of a base language definition in order to provide a Sugar* plugin generates SDF2 code for the syntax description and Stratego code for the inference rules and several other declarations like the interface judgement. This is implemented by another Sugar* plugin for the SoundX meta-language. Since this language has no import declaration, it is not extensible. But we obtain Eclipse support for base language definitions and can reuse the entire parsing and desugaring infrastructure of Sugar* by this technique. The following screen shot shows an extract of the the base language definition for $\lambda_{\rightarrow}$ opened in the Eclipse IDE.



### 5.1.1 *Building a derivation*

So far, we have only specified that a derivation $\nabla$ is valid if the statement $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ holds for some assumptions $\vec{\nabla}_a$ and inference rules $\vec{I}$. But given the list of assumptions and inference rules and a goal judgement $J$ this statement does not help to obtain a derivation $\nabla$ of $J$, that is, $J = \mathit{concl}(\nabla)$. As we have seen in Section 4.1.2 SoundX has to derive the judgement $\varnothing \vdash \texttt{tlds} \Rightarrow \Gamma\texttt{i}$ for a given `tlds` but unknown $\Gamma\texttt{i}$ to establish the type correctness of a $\lambda_{\to}$ module. The goal judgement may contain meta-variables. Therefore, a practical application must be able to calculate a derivation $\nabla$ and a substitution $\sigma$ for a goal judgement $J$ such that $[\sigma](J) = \mathit{concl}(\nabla)$.

The SoundX implementation uses an adaption of SLD-resolution [NM95, 3.3] that actually builds the derivation instead of only returning a substitution. Our adaption also supports assumptions, which is required for the verification of an extension. In the following, we describe the algorithm and prove it sound with respect to the validity statement of Section 2.2.

We use the variable $\tilde{V}$ for a set of meta-variables and $\tilde{V}_1 \not\frown \tilde{V}_2$ denotes that $\tilde{V}_1$ and $\tilde{V}_2$ are disjoint. It is a shorthand for $\tilde{V}_1 \cap \tilde{V}_2 = \emptyset$.

The resolution statement $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \vec{J} \rhd (\sigma, \vec{\nabla})$ asserts that the judgements $\vec{J}$ can be derived under the assumptions $\vec{\nabla}_a$ using the rules $\vec{I}$ where the variables $\tilde{V}$ of $\vec{J}$ may be substituted. This assertion is witnessed by the derivations $\vec{\nabla}$ and the substitution $\sigma$ contains the assignments for the meta-variables in $\vec{J}$. To define the resolution statement, we first need the notion of a fresh inference rule. An inference rule $I$ is a variant of an inference rule $I_0 \in \vec{I}$ if $I$ is equal to $I_0$ but the meta-variables may be consistently renamed. We call $I$ a fresh variant with respect to a set of variables $\tilde{V}$ if $\mathit{vars}(I)$ is disjoint from $\tilde{V}$. We write $I \;\tilde{\in}_{\tilde{V}}\; \vec{I}$ to express that $I$ is a fresh variant with respect to $\tilde{V}$ of some rule $I_0 \in \vec{I}$ as defined by this rule:

$$\textsc{F-Rule}\quad \frac{I_0 \in \vec{I} \qquad I = [\sigma_f](I_0) \qquad \sigma_f : \mathit{vars}(I_0) \leftrightarrow \tilde{V}_f \qquad \tilde{V}_f \not\frown \tilde{V}}{I \;\tilde{\in}_{\tilde{V}}\; \vec{I}}$$

We capture the renaming by an explicit freshening substitution $\sigma_f$. The freshening substitution is a bijection, indicated by $\leftrightarrow$, between the variables of the original rule $I_0$ and the set $\tilde{V}_f$ of fresh variables which is kept disjoint from $\tilde{V}$.

The resolution statement $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \vec{J} \rhd (\sigma, \vec{\nabla})$ is defined inductively by three rules. It derives the judgements $\vec{J}$ from left to right and augments the substitution $\sigma$ in each step. The set $\tilde{V}$ contains those variables of the judgements $\vec{J}$ that may be substituted. In other words, $\tilde{V}$ keeps track of the existentially quantified variables of $\vec{J}$. For example, the statement $\vec{\nabla}_a \vdash_{\vec{I}}^{\{V_1,\ldots,V_n\}} \langle J \rangle \rhd (\sigma, \langle \nabla \rangle)$ can intuitively be understood as: the formula

$\exists V_1, ..., V_n.J$ is derivable from $\vec{\nabla}_a$ using the rules $\vec{I}$ and $\sigma$ contains the witnesses for $V_1$ to $V_n$.

The simplest rule R-Empty covers the empty list of judgements and gives the empty substitution $\emptyset$:

R-Empty
$$\frac{}{\vec{\nabla}_a \vdash^{\tilde{V}}_{\vec{I}} \langle\rangle \rhd (\emptyset, \langle\rangle)}$$

The second rule R-Assumption is concerned with the case that a judgement can be derived using an assumption:

R-Assumption
$$\frac{\begin{array}{c}\tilde{V} \not\frown vars\langle\nabla_{a1..am}\rangle \\ [\sigma_0](J) = concl(\nabla_{ak}) \\ dom(\sigma_0) \subseteq \tilde{V} \\ \langle\nabla_{a1..am}\rangle \vdash^{\tilde{V}}_{\vec{I}} [\sigma_0](\vec{J}) \rhd (\sigma_1, \vec{\nabla})\end{array}}{\langle\nabla_{a1}, ..., \nabla_{ak}, ..., \nabla_{am}\rangle \vdash^{\tilde{V}}_{\vec{I}} \langle J, \vec{J}\rangle \rhd (\sigma_1 \circ \sigma_0, \langle\nabla_{ak}, \vec{\nabla}\rangle)}$$

According to the second premise, the first judgement $J$ of the list must be a substitution instance of the conclusion of an assumption $\nabla_{ak}$. Consequently, this assumption $\nabla_{ak}$ is the corresponding derivation of $J$ (on the right-hand side of the conclusion). As we explained earlier, only members of $\tilde{V}$ are subject to substitution (the third premise).

Moreover, it is important that the variables substituted by the resolution procedure are disjoint from all variable that appear in the assumptions (first premise). Otherwise, we could, for example, derive the statement $\langle !\,(\langle V_1, V_2\rangle\ \mathtt{R})\rangle \vdash^{\{V_1,V_2\}}_{\langle\rangle} \langle\langle V_2, V_1\rangle\ \mathtt{R}\rangle \rhd (\{V_1 \mapsto V_2, V_2 \mapsto V_1\}, \langle !\,(\langle V_1, V_2\rangle\ \mathtt{R})\rangle)$ for two arbitrary but different meta-variables $V_1$ and $V_2$. But this is obviously wrong since it claims that $(!\,(\langle V_1, V_2\rangle\ \mathtt{R}))$ is a derivation for $(\langle V_2, V_1\rangle\ \mathtt{R})$.

Finally, the substitution $\sigma_0$ is applied to the remaining judgements $\vec{J}$. That is, the condition under which $J$ is derivable from $\nabla_{ak}$ is propagated to $\vec{J}$. The resulting judgements $[\sigma_0](\vec{J})$ must be derivable (fourth premise) yielding a substitution $\sigma_1$. The final substitution for the judgements $\langle J, \vec{J}\rangle$ is formed by the composition of $\sigma_1$ and $\sigma_0$.

The third and last rule R-Rule covers the instantiation of an inference rule of $\vec{I}$:

$$\text{R-Rule}\quad \frac{\begin{array}{c}\tilde{V} \not\frown vars(\vec{\nabla}_a) \\ (\langle J_{01..0m}\rangle \rightarrowtail^N J_0) \;\tilde{\in}_{\tilde{V}\cup vars(\vec{\nabla}_a)\cup vars\langle J,\vec{J}\rangle}\; \vec{I} \\ [\sigma_0](J) = [\sigma_0](J_0) \\ dom(\sigma_0) \subseteq \tilde{V} \cup vars(J_0) \\ \vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}\cup vars\langle J_{01..0m}, J_0\rangle} [\sigma_0]\langle J_{01..0m}, \vec{J}\rangle \rhd (\sigma_1, \langle \nabla_{01..0m}, \vec{\nabla}\rangle)\end{array}}{\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J, \vec{J}\rangle \rhd (\sigma_1 \circ \sigma_0, \langle \langle \nabla_{01..0m}\rangle \Rightarrow^N [\sigma_1 \circ \sigma_0](J), \vec{\nabla}\rangle)}$$

Similar to R-Assumption, the first premise requires that variables mentioned in the assumptions are not substituted. The second premise demands that there is a fresh variant of an inference rule $\vec{I}$ whose conclusion $J_0$ can be unified with the first judgement $J$ (third premise). The addition of the fourth premise restricts the domain of the unifier $\sigma$ to variables contained in $\tilde{V}$ and to those mentioned in $J_0$. For $J$ to be derivable by that particular inference rule $N$, the fifth premise requires that the judgements $J_{01}$ to $J_{0m}$ and the remaining judgements $\vec{J}$, subject to the substitution $\sigma_0$, must be derivable. The derivation of $J$ is constructed from the subderivations $\nabla_{01}$ to $\nabla_{0m}$ and the resulting substitution $\sigma_1 \circ \sigma_0$ applied to $J$ (right-hand side of the conclusion).

Resolution constructs valid derivations, that is, resolution is sound with respect to validity. In order to prove this, we need some simple properties of resolution, namely, that the variables of assumptions are never substituted and that the resulting conclusions $concl\langle\nabla_{1..n}\rangle$ are substitution instances of the goal judgements $\langle J_{1..n}\rangle$:

Lemma 5.1 Properties of resolution

(i) *If* $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J_{1..n}\rangle \rhd (\sigma, \langle\nabla_{1..n}\rangle)$ *then* $dom(\sigma) \not\frown vars(\vec{\nabla}_a)$.

(ii) *If* $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J_{1..n}\rangle \rhd (\sigma, \langle\nabla_{1..n}\rangle)$ *then* $[\sigma]\langle J_{1..n}\rangle = concl\langle\nabla_{1..n}\rangle$.

*Proof.* By induction on a derivation of $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J_{1..n}\rangle \rhd (\sigma, \langle\nabla_{1..n}\rangle)$ and a case analysis on the last rule applied in the derivation. The details are listed in Appendix C.12. □

With the previous lemma we are able to prove the soundness of resolution as formulated in the next theorem.

Theorem 5.2 Soundness of resolution

*If* $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J_{1..n}\rangle \rhd (\sigma, \langle\nabla_{1..n}\rangle)$ *then for each* $i \in 1..n : \vec{\nabla}_a \vdash_{\vec{I}} \nabla_i$.

*Proof.* By induction on a derivation of $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J_{1..n}\rangle \rhd (\sigma, \langle\nabla_{1..n}\rangle)$ and a case analysis on the last rule applied in the derivation using Lemma 5.1 in the R-Rule case. The details are listed in Appendix C.13. □

The rules R-Assumption and R-Rule do not specify the search strategy to apply in the case that the goal judgement $J$ is derivable by more than one assumption in R-Assumption or can be unified with several conclusions of the inference rules $\vec{I}$ in R-Rule. Like in Prolog implementations our implementation chooses a depth-first search strategy because it is more efficient and easier to implement. It is a well-known fact that depth-first search is incomplete in the sense that it may fail to find a derivation for a valid judgement. A breadth-first search is complete in this sense but either strategy may fail to terminate if a given judgement is not derivable [NM95, 3.6]. For the base languages and extensions that we study in this thesis these limitations are not problematic. We also expect this observation to hold for more complex programming languages since we are usually able to circumvent inference rules which are prone to nontermination. For example, a type equivalence like in System $F_\omega$ [Gir72] containing a symmetry rule

$$\frac{\mathtt{S} \equiv \mathtt{T}}{\mathtt{T} \equiv \mathtt{S}} \qquad [5.1]$$

is usually restructured into an equivalent relation without the symmetry rule [Pie02, 30].

### 5.1.2 *Type error messages*

One of the main motivations for the development of SoundX is that type checking should be performed on the original sugared code to avoid type error messages to be relative to the desugared code. We subsume all kinds of errors like scoping or kinding errors that a context analysis might detect by the phrase "type error." This goal poses the challenge to generate usable type error messages from the specification of the type system by inductively defined judgements. This especially includes an accurate location of the source of an error. But the SoundX inference procedure that builds a derivation simply fails if a program contains type errors. This plain failure gives no hints to the programmer about the nature and location of the errors in the program.

Even if a type checker is tied to a specific language and a specific type system it is often difficult to accurately locate and report the source of an error [Wan86, LFGC07, PKW14]. It is obvious that useful error detection in a language-independent setting like SoundX does not get simpler. There are several systems that derive type checkers from declarative specifications of programming languages, for example the Synthesiser Generator [RT89], the Programming System Generator [BS86], or Centaur [BCD+88]. The former two are based on attribute grammars and it is not clear if the strong soundness guarantee provided by the the SoundX verification procedure can be transferred to such a formalism. In contrast, the Centaur system employs the Typol language [Des84], an implementation of natural semantics

[Kah87], to describe a type system. Type error messages are attached to inference rules by actions which are executed if the rule is instantiated. For example, the action attached to the following rule emits an error message if the identifier X is already declared in the environment e [Des84, page 225]:

```
(e1, ok = declare(e, X, TYPE))                                    [5.2]
 (e var-decl(X, TYPE) : e1);
{if not ok
    then printerror('identifier already declared', X)}
```

This means that it must be possible to instantiate this rule also for an ill-typed program. Otherwise, the action is not executed. Hence, this approach fundamentally conflicts with the basic principle of SoundX that a program is well-typed if and only if a specific judgement can be derived. Consequently, we cannot adopt the idea of actions to report type errors.

Instead, we propose a different approach. We suggest a second inference procedure that is solely concerned with the detection and location of errors. This second procedure is only called if the initial attempt to build a derivation for a judgement fails. The separate error analysis disentangles the two aspects of generating a valid derivation and generating error messages with locations, which simplifies the implementation a lot.

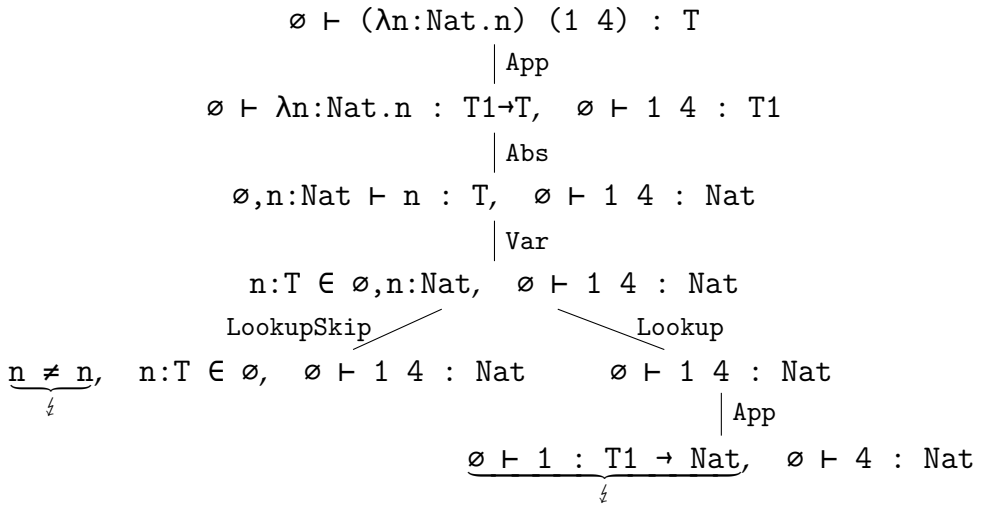
The idea of the error detection procedure is to extract error messages from the search tree that is implicitly traversed during the resolution process. We illustrate the idea by an example. The program

```
(λn:Nat.n) (1 4)                                                   [5.3]
```

is obviously ill-typed and it is impossible to derive the judgement

```
∅ ⊢ (λn:Nat.n) (1 4) : T                                           [5.4]
```

for any T. While trying to derive this judgement, the resolution procedure implicitly traverses the following search tree, where we mark the edges with the instantiated typing rules:



This search tree does not contain any successful path from the root to a leaf. Both leafs contain a judgement which cannot be derived, they are marked by the $\lightning$ symbol. We identify these underivable judgements as the reason for the type error. For example, SoundX converts the failed judgement `⌀ ⊢ 1 : T1→Nat` into the error message

```
Could not derive ⌀ ⊢ 1 : T1→Nat
```
[5.5]

Since we have two leafs in the tree, we also obtain the error message

```
Required something else than n
```
[5.6]

from the judgement `n≠n`, which does not adequately describe an error. To avoid such a spurious error, we mark each failed judgement with its predecessor judgements. The predecessor judgements are those judgements that lead to the necessity to show the failed judgement. The predecessors of `n≠n` are

```
n:T ∈ ⌀,n:Nat
⌀ ⊢ n : Nat
⌀ ⊢ (λn:Nat.n.) (1 4) : T.
```
[5.7]

If any of these predecessors is successfully derived in another path of the tree, the error is cancelled. In this case, the predecessor `n:T ∈ ⌀,n:Nat` is derived in the right path of the tree using the rule `Lookup`, which cancels the spurious error.

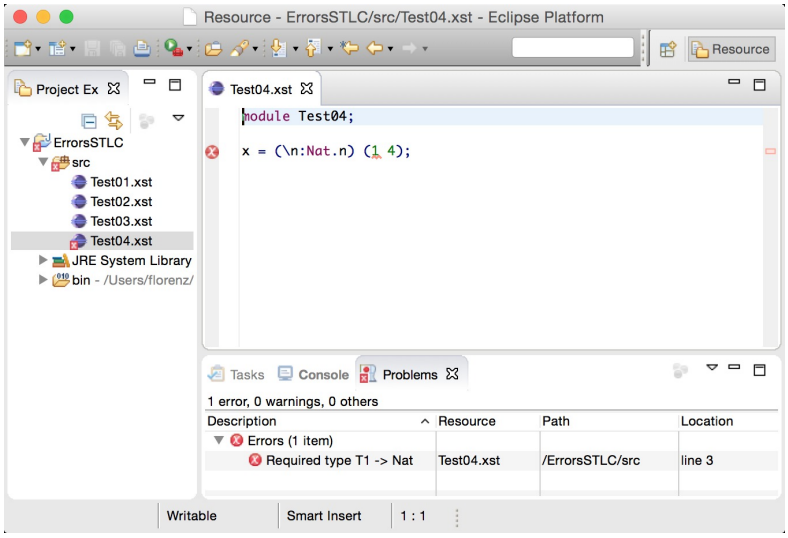
The general message that SoundX generates from a failed judgement does not indicate the location of the error in the source code. Since SoundX cannot guess which argument of such a failed judgement should be considered as erroneous, it is possible to add error annotation to judgement form declarations in the base language definition. These annotations define a template for the generated error message and indicate the location of the error. For example, for the typing judgement in $\lambda_{\rightarrow}$, we add the following annotations to the judgement form declaration:

```
judgement forms
  { Env "⊢" t:Term ":" T:Type
    error message "Required type %T%"
          location t }
```
[5.8]

The error message may contain meta-variables enclosed in % that refer to individual arguments of the judgement form. This annotation leads to the message

```
Required type T1→Nat
```
[5.9]

attached to the source fragment `1`. This information is used in the Eclipse plugin for the highlighting of errors with red squiggles:




The error detection procedure is not perfect:

- Usually, it only detects one error since a single failed judgement prevents any further analysis.
- Sometimes it emits several error messages for a single error.
- Even though it is possible to give templates for error messages, the resulting messages are difficult to understand in certain circumstances.

Nevertheless, we think this error detection procedure is useful. It gives acceptable messages and sufficiently precise locations in many cases. Since it is independent of the resolution procedure that is involved in the derivation building and desugaring, we can improve or experiment with it without affecting other parts of the implementation.

### 5.1.3 *Simple fresh name generation*

In the definition of the pair extension in Section 1.3.2, Example [1.24], we employ $\eta$-expansion to avoid an unintended name capture in the desugaring of pair construction:

```
{ (Γ ⊢ t1 : T) (Γ ⊢ t2 : T)                                    [5.10]
  ------------------------
  Γ ⊢ [ (t1,t2) ] : Pair T
  ~~~> (λa:T.λb:T.λs:T→T→T. s a b) t1 t2 }
```

Instead of this pedestrian encoding, we would rather write

```
(λy:T→T→T. y t1 t2)                                            [5.11]
```

on the right-hand side where the variable y must be sufficiently fresh. Of course, the notion "sufficiently fresh" depends on the base language. For $\lambda_{\rightarrow}$ we can demand that y is not bound in the context Γ. SoundX provides a

simple way to declare freshness conditions in the base language definition. The declaration

```
fresh x:ID in Γ:Env by x ∉ dom(Γ)                    [5.12]
```

states that whenever a name `x` of the sort `ID` is required that must be fresh relative to a given `Γ` this name must satisfy the judgement `x ∉ dom(Γ)`. We add the judgement form `ID "∉" "dom" "(" Env ")"` to the base language definition and define it by two rules:

```
F-Empty:                                             [5.13]
  ----------
  x ∉ dom(∅)
F-Var:
  (x≠y) (x ∉ dom(Γ))
  ------------------
   x ∉ dom(Γ,y:T)
```

The freshness condition in a freshness declaration can be any inductively defined judgement that takes two arguments. It is the responsibility of the base language designer to ensure that it is a useful and correct freshness condition.

Due to the freshness declaration SoundX provides a `fresh` function that can be used in desugarings. Using the `fresh` function we can implement the desugaring of the pair constructor like this:

```
{ (Γ ⊢ t1 : T) (Γ ⊢ t2 : T)                          [5.14]
  ------------------------
  Γ ⊢ [ (t1,t2) ] : Pair T
  ~~~> (λy:T→T→T. y t1 t2)
  where y = fresh(Γ) }
```

The call `fresh(Γ)` makes sure that the generated name fulfils the freshness condition. The implementation of the `fresh` function satisfies the following axiom:

```
-----------------                                    [5.15]
fresh(Γ) ∉ dom(Γ)
```

Currently, the SoundX implementation generates a name candidate and then checks the freshness condition. If it is not fulfilled it tries another name. The `where` clause in the previous desugaring is merely an abbreviation for

```
λfresh(Γ):T→T→T. fresh(Γ) t1 t2                      [5.16]
```

and can also be used for any expression other than a call to `fresh`. Here, we see that it is important that `fresh` is a function because the binding and the bound occurrence of `fresh(Γ)` must be the same.

The freshness declaration and the definition of the freshness condition is sufficient to generate fresh names during the desugaring phase. But in the current formulation of $\lambda_{\rightarrow}$, this is not enough to enable the verification of the `Pair` rule. The verification procedure has to deduce the two judgements

```
Γ,fresh(Γ):T→T→T ⊢ t1 : T                                    [5.17]
Γ,fresh(Γ):T→T→T ⊢ t2 : T
```

from the assumptions `Γ ⊢ t1 : T` and `Γ ⊢ t2 : T`. This is not possible with the available rules. But for $\lambda_{\_}$, the following Weakening Lemma holds [Pie02, 9.3] which expresses that typing is invariant under the addition of fresh variables to the environment:

*If* `Γ ⊢ t : T` *and* `x ∉ dom(Γ)` *then* `Γ,x:S ⊢ t : T`.

We can add Weakening as an inference rule to our base language:

```
Weak:                                                        [5.18]
  (Γ ⊢ t : T) (x ∉ dom(Γ))
  ------------------------
       Γ,x:S ⊢ t : T
```

With this additional rule in place the verification is able to deduce the two required judgements using the assumptions and the axiom for the `fresh` function.

This simple mechanism for fresh names requires several precautions by the designer of the base language. It is necessary to include a freshness declaration which contains a suitable freshness condition. This freshness condition must be sufficient to reason about expressions containing calls to `fresh` which may require additional inference rules in the base language which are not necessary for type checking. Due to these complications concerning the base language definition we consider this approach as not entirely satisfactory. It works quite well in practice for simple examples but it is easy to reach its limits. We discuss this aspect further in Section 6.3.

## 5.2 CASE STUDIES

We have implemented the pair type and the let-expression extensions in two modules `PairExtension` and `LetSeqExtension` and SoundX automatically verifies their soundness. For the desugaring of the pair constructor we use the fresh name generator as described in Section 5.1.3. To demonstrate how extensions in turn can be extended or be used in other extensions we present the implementation of two binding forms for pairs. In the module `LetPairExtension` we define a pattern matching `(x1,x2) = t` which decomposes a pair and binds its components to the variables `x1` and `x2`. The pattern match can be used as a binding in a let-expression. In the module `OpenPairExtension` we define a record-like opening expression `open t1 in t2` which binds the components of `t1` to the names `fst` and `snd` in the scope of `t2`. This extension is defined in terms of `LetPairExtension`. We also show two examples of how SoundX rules out unsound definitions of the let-extension. Finally, we examine how SoundX reacts to potentially unhygienic desugarings.

### 5.2.1 *Pattern matching for pairs*

The pattern matching extension for pairs extends the pair type and the let-expression extension. We activate these extensions by the following import declarations:

```
import PairExtension;                                    [5.19]
import LetSeqExtension;
```

With these imports the syntax and typing rules are available for the definition of the pattern matching. We first introduce a new `Binding` with the syntax declaration

```
context-free syntax                                      [5.20]
  "(" ID "," ID ")" "=" Term -> Binding
```

Since the syntax for let-expressions is available due to the import of `LetSeqExtension` expressions like `let (a,b) = (1,2) in a+b` ares now permissible. But without typing rules the type checker will reject this code because no rule matches the pattern match binding. Similar to the typing rules of let-expressions in Example [3.4] we add one typing rule for a single binding and one typing rule for a list of bindings.

```
LetPair1:                                                [5.21]
  (Γ ⊢ t1 : Pair T1) (Γ,x1:T1,x2:T1 ⊢ let bs in t2 : T2)
  ------------------------------------------------------
          Γ ⊢ let (x1,x2) = t1; bs in t2 : T2
LetPair2:
  (Γ ⊢ t1 : Pair T1) (Γ,x1:T1,x2:T1 ⊢ t2 : T2)
  -------------------------------------------
        Γ ⊢ let (x1,x2) = t1 in t2 : T2
```

Both rules clearly document the intended meaning: the right-hand side of a pattern match must be a pair and the components are brought into scope in the body of the let-expression.

The desugaring is performed by two straightforward guarded desugarings that apply the selectors to the right-hand side:

```
                                                         [5.22]
{ (Γ ⊢ t1 : Pair T1) (Γ,x1:T1,x2:T1 ⊢ let bs in t2 : T2)
  ------------------------------------------------------
        Γ ⊢ [ let (x1,x2) = t1; bs in t2 ] : T2
  ~~~> (λx1:T1.λx2:T1. let bs in t2) (t1.1) (t1.2) }

{ (Γ ⊢ t1 : Pair T1) (Γ,x1:T1,x2:T1 ⊢ t2 : T2)
  -------------------------------------------
      Γ ⊢ [ let (x1,x2) = t1 in t2 ] : T2
  ~~~> (λx1:T1.λx2:T1. t2) (t1.1) (t1.2) }
```

One obvious deficiency of these desugarings is the duplication of `t1`. Since $\lambda_{\rightarrow}$ does not have side-effects it does not corrupt the observable program behaviour. But depending on the $\lambda_{\rightarrow}$ interpreter or compiler it might evaluate `t1` twice. To avoid this duplicate evaluation we might be tempted to desugar into a let binding using a fresh variable:

```
{ (Γ ⊢ t1 : Pair T1) (Γ,x1:T1,x2:T1 ⊢ t2 : T2)                    [5.23]
    ------------------------------------------
        Γ ⊢ [ let (x1,x2) = t1 in t2 ] : T2
  ~~~> let y = t1 in (λx1:T1.λx2:T1. t2) (y.1) (y.2)
       where y = fresh(Γ,x1:T1,x2:T1) }
```

Unfortunately, SoundX is currently not able to verify this extension. We return to this issue in Section 6.3.1 but in short, the reason is that `Γ,y:Pair T1,x1:T1,x2:T1 ⊢ t2 : T2` cannot be deduced from `Γ,x1:T1,x2:T1 ⊢ t2 : T2` without permuting the context `Γ,y:Pair T1,x1:T1,x2:T1` to `Γ,x1:T1,x2:T2,y:Pair T1` to make the `Weak`-rule applicable.

### 5.2.2 *Opening a pair*

The opening expression for pairs considers a pair as a record with the selectors `fst` and `snd`. With this extension we can, for example, write a swap function for points like this:

```
swap = λp:Pair Nat. open p in (snd,fst);                          [5.24]
```

We implement the opening of pairs by a direct translation into a let expression using the following universal desugaring:

```
{ open t1 in t2 ~~~> let (fst,snd) = t1 in t2 }                   [5.25]
```

It is important to note that `fst` and `snd` are $\lambda_{\rightarrow}$-level identifiers and not meta-variables. This is also documented in the typing rule:

```
Open:                                                             [5.26]
  (Γ ⊢ t1 : Pair T1) (Γ,fst:T1,snd:T1 ⊢ t2 : T2)
  ----------------------------------------------
            Γ ⊢ open t1 in t2 : T2
```

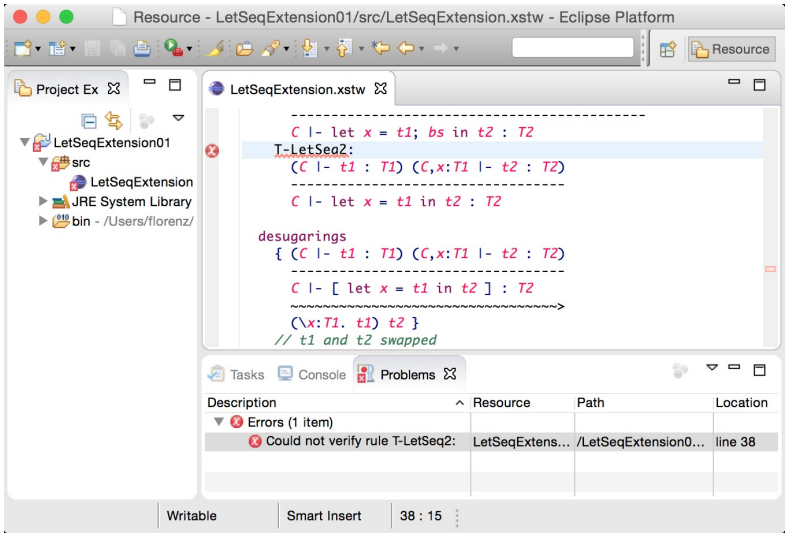
To enable the simple and direct implementation of `open` as an incremental extension (see Chapter 4) we also have to import `LetPairExtension`. It is not necessary to import the pair extension since it is re-exported by `LetPairExtension`.

### 5.2.3 *Unsound let-expressions*

Consider the following desugaring for a let-expression with one binding where the body `t2` and the argument `t1` of the $\lambda$-abstraction are accidentally swapped:

```
{ (Γ ⊢ t1 : T1) (Γ,x:T1 ⊢ t2 : T)                                 [5.27]
  ------------------------------
  Γ ⊢ [ let x = t1 in t2 ] : T2
  ~~~> (λx:T1. t1) t2 }
```

With this desugaring the extension is rejected as the following screen shot shows:




Similarly, if we forget to implement a desugaring for `let x = t1 in t2`, SoundX detects an unsound extension and replies with this message:

```
Could not verify rule Let2                                    [5.28]
  The last rule (Let2) in the derivation of
  Γ ⊢ let x = t1 in t2 : T2 is from the extension
  (forgotten desugaring?)
```

### 5.2.4 *Unhygienic extensions*

Like any other system that deals with open code, that is code with free variables, SoundX must be careful about unintended variable capture or unintended variable reference. An unintended variable capture happens if a desugaring introduces a variable that binds a free variable of one of its arguments. An unintended reference happens if a desugaring refers to some external variable which may be shadowed in the application site of the respective extension. A program transformation system that avoids unintended references and captures by an automatic renaming of variables during the translation is called *hygienic* [KFFD86]. We illustrate both problems by examples and describe if and how SoundX prevents these possibly dangerous situations.

We already touched the issue of unintended capture in Example [1.24] (pair construction) where we chose to avoid it by an $\eta$-expansion. Moreover, in Section 5.1.3 we discuss a pragmatic approach to generate a fresh variable name which is different from all possibly free variables such that capture cannot occur. But in both cases we were aware that unintended capture might be a problem and took counter measures. The important question is: what happens if we implement a possibly capturing desugaring? To be more precise, how does SoundX respond to this inference rule and desugaring?

```
Pair:                                                          [5.29]
  (Γ ⊢ t1 : T) (Γ ⊢ t2 : T)
  -------------------------
  Γ ⊢ [ (t1,t2) ] : Pair T
  ~~~> (λs:T→T→T. s t1 t2)
```

Note that `s` is a concrete $\lambda_{\rightarrow}$-level identifier. Fortunately, SoundX rejects this desugaring since it cannot deduce `Γ,s:T→T→T ⊢ t1 : T` from `Γ ⊢ t1 : T`. Since SoundX does not know if `s` is contained in `Γ` it cannot apply the weakening rule. This uncertainty about `s` and `Γ` expresses a possible dangerous capture if `s` is a free variable of `t1` or `t2`. That is, if we definitely want to desugar `t1,t2` using the concrete variable name `s` we have to change the typing rule and request that `t1` and `t2` must be well-typed in a context `Γ,s:T→T→T`:

```
Pair:                                                          [5.30]
  (Γ,s:T→T→T ⊢ t1 : T) (Γ,s:T→T→T ⊢ t2 : T)
  -----------------------------------------
          Γ ⊢ [ (t1,t2) ] : Pair T
  ~~~> (λs:T→T→T. s t1 t2)
```

With this formulation, the typing rule clearly documents that the variable `s` is brought into the scope of `t1` and `t2`. The bottom line is that SoundX does not avoid unintended capture automatically like a hygienic system but it detects it and a possibly dangerous extension is not verified. In such a case, we can make the capture explicit or resort to the fresh name generator or a different transformation like $\eta$-expansion. We emphasise that the latter alternative is much better with respect to robustness and usability.

To illustrate unintended reference we use the following increment extension that defines a postfix increment operator `++` for numbers.

```
module IncrementExtension;                                     [5.31]

inc = λn:Nat. n+1;

extension {
  context-free syntax
    Term "++" -> Term

  inductive definitions
    Increment:
       Γ ⊢ t : Nat
      -------------
      Γ ⊢ t++ : Nat

  desugarings
    { t++ ~~~> inc t }
}
```

The important point is that the desugaring of `t++` refers to the externally defined function `inc`. Of course, we could directly desugar into `t+1` but then the problem we want to illustrate does not appear. The increment extension is rejected by SoundX since it does not know the type of `inc`

during desugaring. This is because extensions are verified independently of any base language specific environment the module might provide. This is a consequence of the module analysis as defined by A-Module. But we can add the premise `Γ ⊢ inc : Nat→Nat` to the rule `Increment` to make verification succeed. This additional premise documents that the increment extension may only be used in a context where the `inc` function with the specified type is available. Since `inc` is defined in the module providing the extension we can assume that this requirement is satisfied and the following module is well-typed:

```
module IncrementExtensionTest;                                    [5.32]

import IncrementExtension;

two = 1++;
```

But what happens if we change the definition of `two` into

```
two = (λinc:Nat. 1++) 1;                                          [5.33]
```

This code is ill-typed since `Γ,inc:Nat ⊢ inc : Nat→Nat` cannot be derived for any `Γ` because `Nat ≠ Nat→Nat`. If we change `two` again into

```
two = (λinc:Nat→Nat. 1++) (λn:Nat. n+2)                           [5.34]
```

the code is well-typed but the result is wrong! The problem is that `inc` from the right-hand side of the desugaring unintentionally references the $\lambda$-bound variable `inc`. Since these two are of the same type this code is accepted. The only solution to this problem is to make the reference to `inc` in the desugaring a unique reference. But this is only possible if the base language allows such a unique qualification, for example by prefixing the name with the module identifier:

```
inductive definitions                                             [5.35]
  Increment:
    (Γ ⊢ t : Nat) (Γ ⊢ IncrementExtension.inc : Nat→Nat)
    ------------------------------------------------------------
                        Γ ⊢ t++ : Nat

desugarings
  { t++ ~~~> IncrementExtension.inc t }
```

Currently, our base language $\lambda_{\rightarrow}$ does not support such a qualification and we have no chance to avoid unintended reference. The bottom line is that SoundX is sometimes able to detect a possibly unintended reference, namely, if a premise regarding the reference is forgotten. Nevertheless, SoundX is dependent on the capabilities of the base language for the implementation of extensions to have safe references.

In summary, SoundX is not hygienic but it detects unhygienic situations due to unintended capture. With respect to unintended reference the system is not safe but gives hints in some situations.

# 6 DISCUSSION

In our presentation of SoundX, we have commented on several issues only very briefly to keep a straight line of argument. Nonetheless, the aspects termination, failure of the forward step, fresh names, and binding structures deserve further attention and we catch up with them in this chapter. We close this thesis with a discussion of related work, a summary of our results, and an outline of future research directions.

## 6.1 TERMINATION

The aspect of termination or nontermination has appeared at several points so far but only been touched on very briefly. In SoundX, it is relevant for several parts.

First of all, any inference engine that constructs a derivation using the inference rules of a base language and an extension is necessarily incomplete. Since the language of inference rules is a Turing-complete system there is no possibility to determine if the inference process terminates for a given set of rules and a given judgement to derive. It depends on the search strategy when and under which circumstances divergence arises. For example, using a breadth-first search, the inference engine will always find a valid derivation for a certain judgement; if such a derivation exists. In contrast, it may fail to terminate for a judgement which is not derivable. Using a depth-first search, as in our implementation, divergence may also arise for derivable judgements.

Since the verification procedure requires an inference engine it is also incomplete. Moreover, it is also dependent on the desugaring of inference rules. Due to this dependence on the desugarings, this process may also diverge. For example, the iterated application of the following guarded desugaring obviously leads to an infinite loop:

```
(Γ ⊢ t1 : T) (Γ ⊢ t2 : T)                                    [6.1]
------------------------
Γ ⊢ [ (t1,t2) ] : Pair T
~~~> (t2,t1)
```

Consequently, the remark that guarded desugarings do not lead to a divergence in the process of derivation desugaring on page 58 only holds for inference rules that have been successfully verified, that is, the verification procedure has terminated for them.

Finally, universal desugarings may cause the derivation desugaring to diverge, even for sound extensions. But if there are no universal desugarings

involved, derivation desugaring always terminates for sound extensions, either successfully or stuck.

## 6.2 FORWARD STEP

In Section 3.6.1, we explained that we currently do not have a sufficient criterion which guarantees that the forward step is always successful. The forward step is only executed during the bottom-up phase of derivation desugaring. Consequently, it can only fail for rules of the base language or extended $\mathcal{B}$-rules. The difficulty is that an awkward formulation of the base language may also lead to stuckness. The example we present here is completely artificial and without any practical relevance. We do not know of any useful base language definition which also causes the problem, but we do not have a criterion to forbid the awkward ones.

We add a `bogus` expression to $\lambda_{\rightarrow}$ with the following syntax and typing rule:

```
context-free syntax                                          [6.2]
  "bogus" Term -> Term

inductive definitions
  Bog:
    (Γ,a:Nat ⊢ t : T) (Γ,a:Nat→Nat ⊢ t : T)
    ---------------------------------------
              Γ ⊢ bogus t : Nat
```

This typing rule requires that the argument to `bogus` is well-typed under two different and fixed types for the concrete variable `a` in the environment `Γ`. We consider the following expression which also uses the let-expression extension:

```
bogus (let c=a in 1)                                         [6.3]
```

Here is the typing derivation for this expression in the empty environment:

```
                                                             [6.4]
          ⋮                                  ⋮
----------⋮-------------Let2 ----------⋮-------------Let2
ø,a:N ⊢ let c=a in 1 : N     ø,a:N→N ⊢ let c=a in 1 : N
-------------------------------------------------------Bog
               ø ⊢ bogus (let c=a in 1) : N
```

The desugared conclusions of the subderivations of the instantiation of the base language rule `Bog` are

```
ø,a:N ⊢ (λc:N. 1) a : N                                      [6.5]
ø,a:N→N ⊢ (λc:N→N. 1) a : N
```

The trouble is that the forward step has to find a substitution for the meta-variable `t` but `(λc:N. 1) a` $\neq$ `(λc:N→N. 1) a`. The forward step is stuck.

On an abstract level, the problem seems to be related to the fact that the meta-variable `t` appears in different contexts in the premises of the rule

`Bog`. Since extensions of terms can be desugared by guarded desugarings, the same instantiation of `t` may lead to different results depending on the context. Perhaps it is sufficient to separate the syntactic sorts into two disjoint categories: those sorts that may only be subject to universal desugarings and those sorts that may be subject to guarded desugarings. Moreover, all inference rules that mention a meta-variable ranging over the latter category in several premises are rejected. This criterion would forbid the problematic rule `Bog` but it requires further elaboration if this criterion is sufficient and, at least evenly important, an investigation if it disallows useful or even indispensable rules.

We have tried to find an extension which contains a $\mathcal{B}$-rule causing similar trouble but all the rules we could come up with were either $\mathcal{X}$-rules, which act well-behaved in that respect, or rejected by the verification procedure for other reasons. Due to these observations we conclude that, from a pragmatic point of view, the danger to get stuck is relatively low. Probably the most difficult part is to design the base language definition in a form that stuckness cannot arise. Since the design of a base language definition is an expert task in which various other aspects have to be considered as discussed, for example, in Section 5.1.3 we deem the situation acceptable.

## 6.3 Fresh names and binding structure

We concluded the section about our simple fresh name generation with the remark that it is easy to reach its limits. We discuss two such examples here and propose possible solutions.

### 6.3.1 *Weakening is not enough*

We added the `Weak` rule to $\lambda_{\to}$ to be able to verify desugarings that bring fresh names into the scope of their arguments (see Section 5.1.3). But there are situations where this rule is not enough. For example, the following desugaring for pair pattern matching (repeated from the end of Section 5.2.1)

```
{ (Γ ⊢ t1 : Pair T1) (Γ,x1:T1,x2:T1 ⊢ t2 : T2)
    ------------------------------------------
        Γ ⊢ [ let (x1,x2) = t1 in t2 ] : T2
  ~~~> let y = t1 in (λx1:T1.λx2:T1. t2) (y.1) (y.2)
       where y = fresh(Γ,x1:T1,x2:T1) }
```

[6.6]

cannot be verified because `Γ,y:Pair T1,x1:T1,x2:T1 ⊢ t2 : T2` cannot be deduced from `Γ,x1:T1,x2:T1 ⊢ t2 : T2`. The reason is that SoundX does not know that also the following Permutation Lemma is valid for $\lambda_{\to}$:

*If* `Γ1,x1:T1,x2:T2#Γ2 ⊢ t : T` *and* `x1 ≠ x2` *then*
`Γ1,x2:T2,x1:T1#Γ2 ⊢ t : T` *where*

$$\Gamma \# \varnothing = \Gamma$$
$$\Gamma \# \Gamma 1\texttt{,x:T} = (\Gamma \# \Gamma 1)\texttt{,x:T}\ .$$

Since our formulation of $\lambda_{\rightarrow}$ does not preclude identical names in a typing environment, we adapt the Permutation Lemma from [Pie02, 9.3] and only allow to swap adjacent and different names in the environment. Of course, we could add this lemma as a rule to the base language like `Weak`. So we have to formulate the concatenation function `#` as a judgement which is pretty cumbersome. Let us assume for a moment that we have such a suitable `Permutation` rule. Then SoundX could apply `Weak` to the second premise and obtain `Γ,x1:T1,x2:T1,y:Pair T1 ⊢ t2 : T2`. By a repeated application of `Permutation` SoundX then deduces `Γ,y:Pair T1,x1:T1,x2:T2 ⊢ t2 : T2` as required. But for this last step it is necessary to add rules such that SoundX is able to deduce `fresh(Γ,x1:T1,x2:T2) ≠ x1` and `fresh(Γ,x1:T2,x2:T2) ≠ x2`.

On a more abstract level, the problem is that desugarings generate fresh names and that the verification procedure has to deduce a judgement containing these names from assumptions that do not contain them. For example, in the pair construction, SoundX must deduce `Γ,y:T→T→T ⊢ t1 : T` from `Γ ⊢ t1 : T` for `y ∉ dom(Γ)`. Our preliminary solution is to add additional rules to the base language. These rules must be suited to the simple verification procedure. For example, we have to be careful to avoid rules that lead to nontermination, especially for ill-typed input programs. Moreover, they must not make the base language unsound. As this approach requires careful engineering and usefulness testing by the base language designer it is not really satisfactory. It is an open question if it is possible to add systematic support in the verification procedure to deduce judgements with fresh names from judgements without fresh names. One possibility would be to integrate support for environments in SoundX and hardwire Permutation and Weakening in the verification procedure. Therefore, it needs to be elaborated how a general support for environments and a general formulation of Permutation and Weakening can be used to define base languages of different flavour. Moreover, it has to be evaluated if Permutation and Weakening or a combination thereof is sufficient for different base languages, especially if environments keep different sorts of identifiers like term and type variables or consist of different components like one environment for methods and another one for variables.

### 6.3.2 *Type systems with substitution*

Our base language $\lambda_{\rightarrow}$ is very primitive. If we move on to more expressive type systems, the formulation of the typing rules becomes more complicated.

As a representative of such a system we give a quick sketch of a System F centred base language. For a good introduction to System F and several interesting encodings, for example pairs, lists, and existential types, we recommend [Pie02, 23, 24].

The most notable difference between $\lambda_{\rightarrow}$ and System F is that the latter has a universal type `∀x.T`, which is a type with binding structure. To instantiate such a type it is necessary to define capture-avoiding substitutions in the type system. This is, in principle, not a problem in SoundX since we can implement it by a judgement `[x↦S](T)=T1`. The problematic case is the universal type where we have to perform an $\alpha$-conversion in order to avoid a free-variable capture. We have extended our simple approach for fresh name generation from Section 5.1.3 to a predefined judgement `x = fresh(T)`. The implementation of this judgement satisfies the following inference rule where `x ∉ fv(T)` is the declared freshness condition for types:

$$\frac{\texttt{x = fresh(T)}}{\texttt{x} \notin \texttt{fv(T)}} \qquad [6.7]$$

With the freshness judgement, the crucial rule of the substitution judgement is

$$\texttt{Sub-Universal:} \qquad [6.8]$$
$$\frac{\texttt{(y1 = fresh(x,S,T))} \quad \texttt{([y↦y1](T) = T1)} \quad \texttt{([x↦S](T1) = S1)}}{\texttt{[x↦S](∀y.T) = ∀y1.S1}}$$

A possible extension based on System F are polymorphic pairs `T1 * T2`, which we encode by the following universal desugaring:

```
{ T1 * T2 ~~~> ∀y.(T1 → T2 → y) → y                    [6.9]
  where y = fresh(T1,T2) }
```

To make this syntactic sugar for types usable, we have to explain how substitution works for the pair type:

$$\texttt{Sub-Pair:} \qquad [6.10]$$
$$\frac{\texttt{([x↦S](T1) = S1)} \quad \texttt{([x↦S](T2) = S2)}}{\texttt{[x↦S](T1 * T2) = S1 * S2}}$$

The verification procedure now tries to deduce the desugared conclusion

```
[x↦S](∀fresh(T1,T2). (T1→T2→fresh(T1,T2)) → fresh(T1,T2)) =   [6.11]
  ∀fresh(S1,S2). (S1→S2→fresh(S1,S2)) → fresh(S1,S2))
```

from the assumptions `[x↦S](T1) = S1` and `[x↦S](T2) = S2`. In other words, it has to verify that substitution and desugaring commute. Unfortunately, these two only commute modulo $\alpha$-equivalence. Let $\delta$ be the desugaring, then we have

$$\texttt{[x↦S]}(\delta(\texttt{T})) \equiv_{\alpha} \delta(\texttt{[x↦S](T)})$$

instead of equality. For example, we take the type `A * B` and the substitution `[A↦Nat]`. We calculate and obtain the following identities:

$$\delta(\texttt{[A↦Nat](A * B)}) = \delta(\texttt{Nat * B}) = \texttt{∀A.(Nat→B→A) → A}$$
$$\texttt{[A↦Nat]}(\delta(\texttt{A * B})) = \texttt{[A↦Nat](∀C.(A→B→C) → C)} = \texttt{∀D.(Nat→B→D) → D}$$

The variable names `A`, `C`, `D` chosen by the `fresh` function and the freshness judgement are irrelevant to the result that the two right-hand sides are not equal in general but $\alpha$-equivalent.

We observe that there is a fundamental difficulty here: the SoundX verification procedure works with structural equality of expressions and not with $\alpha$-equivalence. It has no knowledge of binding structure. For this reason, there is currently no chance that our verification procedure is able to derive the desugared conclusion. Consequently, we can define System F as a base language but we cannot write interesting extensions involving the universal type.

To overcome these limitations we propose to switch to a system which is based on nominal logic [Pit03] and nominal unification [UPG04], similar to logic programming in $\alpha$Prolog [CU08]. In nominal logic programming, the binding structure of terms is encoded in their types by a binding signature and there is a predefined freshness predicate. The notion of freshness is internal to the system and derived from the binding signatures of the term constructors. Our initial experiments using $\alpha$Prolog as a test environment are quite promising: $\alpha$Prolog is able to derive the desugared conclusion of the substitution for pairs automatically.

Apart from the circumstance that the switch to expressions with binding structure requires a re-development of the entire meta-theory, there are also two other challenges related to a nominal logic based system. The binding structure of a term has to be described by a binding signature. Currently, nominal unification, which is the base of $\alpha$Prolog, only supports relatively simple binding structures in the form of nominal signatures. They are comparable to the abstract binding trees in [Har13]. For example, the following signature specifies the abstract syntax of a recursive let expression `let rec x = t1 in t2`:

```
LetRec: <ID>(Term × Term) → Term                    [6.12]
```

The prefix `<ID>` defines that the identifier `x` is bound in both terms `t1` and `t2` and the abstract form of the concrete let-expression is `LetRec(<x>(t1, t2))`. In this case, the order of arguments in the concrete syntax and the abstract syntax matches. But for a simple let-expression `let x = t1 in t2` this is not the case anymore:

```
Let: Term × <ID>Term → Term                         [6.13]
```

The abstract form of the let-expression is `Let(t1, <x>t2)`. Since `x` is bound in `t2` it has to be swapped with `t1`. These transformations become more

involved for complicated binding structures like, for example, mutually recursive function. As a consequence the concrete syntax description has to be accompanied by a description of the abstract syntax which includes the binding signatures and a translation from the concrete to the abstract syntax.

There exist other more expressive binding specification formalisms, for example Ott [SNO+10] or Romeo [SW14]. But it is currently unclear if it is possible to automatically convert such a specification into nominal signatures or if nominal signatures and nominal unification can be extended to support complex binding structures without rearranging the arguments.

Finally, the verification procedure for extensions also has to make sure that the binding specification given for syntactic sugar is compatible to the binding structure of the desugared code. This is the binding-related pendant to the requirement that well-typed extended code must lead to well-typed desugared code. The $\lambda_m$-calculus [HW08, Her10] and notational definitions [Gri88] are both concerned with this problem but they are not based on nominal signatures. We discuss both systems in the next section.

We conclude this discussion with the remark that a nominal approach to language extension might simplify reasoning about fresh names generated in desugarings, which we discussed in the previous section, but does not solve it. The circumstances under which it is permissible to add fresh names to an environment is still a property of the type system, not of the binding structure alone.

## 6.4 RELATED WORK

There is a tremendous amount of literature on the subject of extensible programming languages and extension mechanisms like templates, macros, or compile time meta-programming. We focus on languages and systems that have goals similar in spirit to the goals of SoundX. These are approaches that analyse the definition of extensions to statically prevent the generation of ill-formed code and that analyse the use sites of extensions prior to desugaring. That is, we do not discuss dynamically typed systems like hygienic macros in Scheme [KFFD86, DHB92], the Common Lisp macro system [Ste90], Syntax Macros [Lea66], nor systems that perform a static type check of the desugared code like Type Specific Languages [OKN+14], Typed Racket [THF10], Camlp4 [dR03], OCaml extension points [LDF+14], Template Haskell [SPJ02], Haskell Quasiquoting, [Mai07], Metamorphic Syntax Macros [BS02], the Java Syntactic Extender [BP01] or Cardelli's Extensible Syntax [CMA94]. Even though many of the latter approaches require the code generator to be a statically well-typed function these types are not strong enough to express typing properties of the generated code.

We divide our discussion of related work into two sections. In the first section, we describe principled approaches that provide a specific safety

or soundness guarantee with respect to the generated code. The second section is concerned with more pragmatic systems that focus on usefulness rather than soundness.

### 6.4.1 *Principled approaches*

Kleene [Kle52] introduces the concept of *definitional extension* for formal languages which is very similar to sound language extensions. A language $\mathcal{L}_x$ is a definitional extension of $\mathcal{L}$ if the expressions of $\mathcal{L}$ are a subset of the expressions of $\mathcal{L}_x$ and there is a mapping $\varphi$ from $\mathcal{L}_x$-expressions to $\mathcal{L}$-expressions. Among others $\varphi$ has to satisfy the condition

$$\text{if } t \in \mathit{Thm}(\mathcal{L}_x) \text{ then } \varphi(t) \in \mathit{Thm}(\mathcal{L}) \,.$$

That is, if $t$ is a theorem in $\mathcal{L}_x$ then its desugared form $\varphi(t)$ must be a theorem in $\mathcal{L}$. This requirement is on par with the SoundX goal (page 16):

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

The definitional extension $\mathcal{L}_x$ corresponds to the extended language $B \cup X$, $\varphi$ corresponds to the desugaring from $p$ to $p'$, and the statement $t \in \mathit{Thm}(\mathcal{L}_x)$ corresponds to $p$ being well-typed in $B \cup X$. In contrast to Kleene's purely mathematical work we present a procedure that automatically verifies if the extension satisfies the necessary condition.

One of the main benefits of syntactic extension facilities or macro systems is that they are able to manipulate binding structure or introduce new binding forms. In statically or lexically scoped languages, which subsume most dynamically and statically typed languages, it is desireable that an extension desugars into well-scoped code. This means, for example, that an identifier which appears in binding position in the extended code also appears in binding position in the desugared code. There are several approaches to guarantee the well-scopedness of the generated code.

Griffin's *notational definitions* [Gri88] provide syntactic extensions in the context of logical frameworks. Notational definitions are declared by $\Delta$-equations based on higher-order abstract syntax [PE88]. A $\Delta$-equation $d(\sigma) \stackrel{\Delta}{=} e$ introduces a new notation $d$ as a short-hand for the expression $e$ where the binding structure of $d$'s arguments is described by $\sigma$. For example, the common short-hand $\forall x \in a.b$ for $\forall x.(x \in a) \Rightarrow b$ is encoded in the $\Delta$-equation

$$\forall(a, \lambda x.b) \stackrel{\Delta}{=} \forall(\lambda x.(x \in a) \Rightarrow b) \,.$$

The $\lambda$-abstraction on the left-hand side declares that $x$ is visible in $b$ but not in $a$. Griffin identifies certain compatibility conditions for the left- and right-hand sides which ensure that a well-scoped application of the new

notation always expands into well-scoped code. Notational definitions can be declared in terms of previous definitions but they must not be recursive.

The $\lambda_m$-calculus [HW08, Her10] augments a Scheme-like macro system with binding signature types for macro definitions that describe their binding structure. It is similar in motivation to notational definitions but the specification of the binding structure is external to the macro and not implicit in its definition and the binding specification is based on tree addresses rather than higher-order abstract syntax. This is comparable to the typing rules in SoundX which are also given explicitly and are not integrated into the desugarings. In contrast to SoundX extensions, which are activated upon import, $\lambda_m$-macro definitions are expressions and macros are lexically scoped. Like in SoundX, macros are first-order only, that is, they cannot expand to other macro definitions. The $\lambda_m$-calculus solves the long-standing issue of $\alpha$-equivalence for unexpanded Scheme programs. However, guarantees are restricted to the static scoping aspects of macros and do not scale to the static typing of terms.

The language Smith [Lin14], based on ideas of the core-calculus $\lambda PB$ [Lor12], is a purely functional programming language that permits the definition of new binding structures. The scoping of a new binder may either be sequential, similar to our let-extension, or parallel, where all bindings are introduced simultaneously into the body and independent of each other. New binders are desugared after type checking by a fixed transformation which also employs type information. The definition of a new binder parameterises the desugaring with two term-level functions, which are a generalisation of the monadic `bindM` and `unitM` operators [Wad92], and determine the dynamic behaviour of the binder. Moreover, Smith also allows to assign flexible mixfix notations to type, term, and binder definitions. With this design, it is well suited for language embeddings avoiding certain inconveniences related to binding structures. Smith is similar in spirit to SoundX but the form of new binders is fairly limited due to the fixed transformation and it is tied to a System $F_\omega$ based type system.

The most advanced macro system which covers both scoping and typing aspects of the generated code is MacroML [GST01]. MacroML is an extension of the ML programming language with the possibility to define generative macros and new binding structures. The semantics of a MacroML program is defined by a translation into the multi-stage programming language MetaML [TS97] and a type preservation result of this translation has been proven. In contrast to MacroML, SoundX allows desugarings to inspect and decompose user programs using pattern matching. Furthermore, MacroML macros that introduce new binding constructs have to be syntactic variations of predefined binding structures like $\lambda$-abstraction to unambiguously identify binding and bound occurrences of variables. In particular, MacroML macros must receive the variables they bind as arguments. This restriction makes it impossible to implement the opening of

pairs from Section 5.2.2 in MacroML. In contrast to SoundX, where extensions are defined in the SoundX meta-language, the base and meta-language are identical in MacroML. This is only possible because MacroML inherits the specific type $\langle t \rangle$ from MetaML that describes code. Consequently, the MacroML approach to macros can only be transferred to other language that have a similarly expressive type system. Such a restriction does not apply to SoundX which only poses very modest requirements on the module system and no requirements at all on the type system of the base language.

The Bedrock Structured Programming System [Chl13] is an embedding of *certified low-level macros* in the Coq theorem prover. The base language is the Bedrock IL, an assembly-like language, which can easily be converted to real machine languages. With the macro definition facilities of Bedrock it is possible to implement higher-level control structures like conditionals, loops, or functions on top of the Bedrock IL. The crucial point is that each macro is accompanied by an interface based on the XCAP programming logic [NS06], a Hoare-style separation logic that supports embedded code pointers. Such an interface consists of a predicate transformer and a verification condition generator. The former answers the question "what does this macro do" and is comparable to the conclusion of a typing rule. The latter answers the question "which uses of this macro are safe and functionally correct" and is comparable to the premises of a typing rule. The Bedrock system requires a proof that the macro implementation is correct with respect to its interface. Correct macros are assigned a Hoare-style proof rule which is available for proving properties of programs involving calls to this macro. For each module of a Bedrock IL program, possibly with macro calls, a proof of memory safety (Preservation and Progress) is required. Most of the necessary proofs are carried out automatically by specific proof tactics available to the Bedrock programmer, which is essential from a pragmatic point of view. In a certain sense, the Bedrock Structured Programming System is very similar to SoundX. Macros and extensions, respectively, are accompanied by an interface and the two are proven sound with respect to each other. But both systems differ greatly on the abstraction level. The Bedrock system deals with low-level details of, for example, memory handling, and SoundX is concerned with the high degree of abstraction that a type system provides. Pictorially, the two systems sit at opposite ends of the spectrum. Nevertheless, the Bedrock system is more expressive since it is possible to also express and prove functional correctness and to raise the level of abstraction by building layers. On the other hand, it is a technically much more complicated system that requires the power of a full-blown Hoare-style logic and an interactive theorem prover at its back.

Safe static reflection mechanisms for Java like SafeGen [HZS05], MJ [HZS07], or MorphJ [HS08] enable the generation of code which depends on well-formed Java entities like classes or interfaces. These approaches rely on matching and iterating over substructures like fields or methods

of the respective input fragment. For example, the following MorphJ class `AddGetter` generates a subclass of its argument `X` which defines getter methods for all fields of `X`:

```
class AddGetter<X> extends X {                                    [6.14]
  <F>[f]for(F f : X.fields; no get#f() : X.methods)
  F get#f()  return super.f;
}
```

`AddGetter` emits a getter method for a field `f` only if the input class `X` does not define a method of that name. This is ensured by the condition `no get#f()` in the second line and necessary to avoid ill-formed code containing invalid method overrides. A safe static reflection system guarantees that the code generator produces well-typed results for all well-typed input fragments. MJ and its successor MorphJ establish this guarantee by their type system which have both been proven sound. SafeGen relies on a mixture of a conventional Java type checker and the automatic first-order theorem prover SPASS [Wei01] to validate typing requirements. For example, the MorphJ type checker would reject the `AddGetter` implementation without the `no get#f()` constraint since it detects that it emits ill-typed code for certain inputs. These static reflection mechanisms provide type-sound code generation but they do not provide syntactic language extensions like SoundX. Moreover, especially MJ and MorphJ are particularly tied to Java since they employ generics and subclassing to define code generators whereas SoundX is almost completely language independent.

The Ur/Web [Chl10] system is a statically typed language which targets heterogenous meta-programming for web applications. In particular, this include the generation of well-formed HTML/XML code and SQL expressions which are correct with respect to a specific database schema. The type system of Ur/Web is designed around the requirements of web programming. It is based on System $F_\omega$ and includes records and type level computation like folding or mapping over a type expression. It is heavily inspired by dependent-type systems but lacks the dependency on the value level avoiding many practical difficulties with these systems. The Ur/Web implementation has a hand-crafted type inference such that the degree of necessary type annotations is comparable to mainstream languages like Java or Scala. A Ur/Web program is converted into the Calculus of Inductive Constructions [BC04] using the type information from the inference algorithm and therefore Ur/Web is type-sound by construction. Similar to safe static reflection Ur/Web is not an extensible language but a heterogeneous compile time meta-programming system for a specific application domain. Like SoundX it focuses on static guarantees about the typing properties of the generated code. Since the approach is tied to a specific application domain, which enables a high degree of machine support, it is unclear if it is transferable to other areas and languages.

### 6.4.2 *Pragmatic approaches*

Heidenreich et al. [HJS+09] present a general mechanism to add a template language to a given object language. Templates are code generators that are executed at compile time. Similar to SoundX, it is necessary to specify the syntax and the static semantics of the object language where the latter is defined by an attribute grammar based formulation. The extended language obtained with this method contains a fixed set of template building blocks: loops, conditionals, and placeholders for template arguments. Based on the static semantics of the object-language, a semantic analysis for a template definition is generated. Like in SoundX, this analysis is intended to prevent the generation of ill-typed code at template definition time. The semantic analysis is conservative to avoid invalid results but may lead to false negatives. Therefore, it is possible to manually refine the analysis. However, neither the generated nor the manually refined analyses for templates are rigorously related to the template expansion process and the static semantics of the object language.

Marco [LGHM12] is a language-independent safe macro system that guarantees syntactic correctness and hygiene for the generated code. Type correctness is aimed at as future work. Marco provides a statically typed procedural macro definition language. To employ Marco for a specific target language, it is necessary to provide a lexer that recognises identifiers and three oracles to check syntactic well-formedness, free names, and captured names. The syntax oracle determines if a given input fragment is syntactically correct, the free names oracle returns the free names contained in a given input fragment, and the capture oracle checks if a specific name is captured by a given input fragment. One of the central ideas of Marco is to use the off-the-shelf target language compiler as an oracle. This is achieved by instantiating target language dependent completion and placeholder fragments, feeding them into the target language compiler, and interpreting the result. For example, if Marco needs the list of free names of the C expression `100 * (1.0 / (foo))`, it sends the completed fragment `int query_expr() { return (100 * (1.0 / (foo)); }` to the C compiler. The compiler returns the error message `name `foo' was not declared in this scope` from which Marco deduces that `foo` is free in the respective fragment. This method is similar in spirit to the way how GNU autoconf detects the capabilities of a build environment. The syntax oracle is used in the type analysis of a macro definition to ensure syntactic correctness of the expanded code. To detect accidental capture, Marco performs a static data-flow analysis on the macro definition which employs the free names and the capture oracles. Additionally, a dynamic data-flow analysis accompanies the macro expansion process to detect accidental name capture due to free names of a macro argument. Since Marco employs the target language compiler for the static checking of the macro definitions we can

expect that this analysis matches the semantics of the target language as provided by the compiler. But this connection is not worked out precisely nor is it proven sound in any sense.

Ziggurat [FS06, FS08] is a metalanguage to extend other programming languages through Scheme-like macros and to attach static analyses like type checking or termination analysis to the new syntax. In Ziggurat, nodes of the syntax tree are represented as objects with a parsing and a rewrite function in an object-oriented system called lazy delegation. Lazy delegation permits the implementation of static analyses explicitly for a node by implementing the corresponding method, or implicitly by delegating analysis to the objects generated during desugaring. However, static analyses themselves are not validated against desugarings and there is the danger that a malicious analysis of a subclass overrides a sound analysis of a super class.

MetaHaskell [Mai12] adds type-safe heterogeneous meta-programming facilities with different object languages to Haskell. To turn a language into a MetaHaskell object language, a quasiquoter for the surface syntax, a type checker, and a unification procedure for the object-language types have to be provided. The object language type checker is called by the MetaHaskell type checker to check embedded object programs. This technique only yields a type-safe type system if the object language's type checker is sound with respect to the dynamic semantics of the object language. In contrast to SoundX, this condition is not checked mechanically but has to be manually verified.

MOSTflexiPL (Modularly, Statically Typed, Flexibly Extensible Programming Language) [Hei12, Hei14] is an experimental programming language that strives for a minimum of syntactic restrictions. It is still under development but a working compiler targeting C++ is available. MOSTflexiPL follows the "one law for all" principle such that predefined and user-defined operations are indistinguishable. Every entity is defined as a mixfix operator including a syntax and type signature, an implementation, scope annotations, and exclude declarations for syntactic disambiguation (precedences and associativity). Although all operators have the same surface appearance, MOSTflexiPL divides them into dynamic operators corresponding to procedures or functions, static operators corresponding to a limited form of type operators, and virtual operators corresponding to type synonyms and simple macros. In MOSTflexiPL it is also possible to define new syntax for definition facilities but these possibilities are fairly limited especially with respect to scoping. Virtual and static operators are expanded at compile time but they are analysed by the type checker at their definition site. MOSTflexiPL is claimed to be type-safe but no evidence is presented, especially, a precise description of the dynamic semantics is missing.

Xoc (Extension-Oriented Compiler) [CBC+08] is an extensible compiler for C. Extensions are implemented in the zeta language and include syntax definitions and attribute definitions for the semantic analysis and the

desugaring. Similar to SDF2, Xoc uses GLR parsing and permits the composition of arbitrary context-free grammars. The semantic analysis and the desugaring are performed by lazy attribute calculations but like in the other pragmatic approach these two are not validated with respect to each other. In contrast to SoundX, which only permits the addition of syntactic sugar, an Xoc extension may also restrict the accepted language, for example, to implement style checkers.

Pluggable type systems [Bra04] extend an existing type system with additional constraints. Pluggable type systems may reject programs that, for example, do not comply to a certain design pattern or introduce new types, for example a non-null type in Java to check that `null` references are never dereferenced. Implementations of pluggable type systems like JavaCOP [MME+10] or the Checker Framework [PAC+08] provide infrastructure to implement type analyses and to integrate them into the semantic analysis phase of the compiler. The soundness of a pluggable type system is not verified mechanically, this is the implementer's responsibility. But JavaCOP has another interesting approach with respect to soundness: testing for soundness violations. Their test harness supports the instrumentation of Java byte code with runtime checks to detect "stuck expressions" that the pluggable type system is supposed to prevent statically.

## 6.5 Conclusion and Future Work

We have presented SoundX, a language-independent mechanism to derive a syntactically extensible dialect of a base language. A base language is specified by a SoundX base language definition which comprises the syntax and the static semantics (type system). Extensions are defined modularly and can be composed into larger extensions. An extension is verified individually to ensure that it generates well-typed code, also under composition with other extensions. To this end, extensions not only define new syntax and the transformation into the base language but also new typing rules that act as an interface of the extension. In contrast to many other language extension facilities, SoundX performs type checking prior to desugaring using the extended type system. This feature enables desugarings which employ type information deduced by the type checker. These *guarded* desugarings are far more expressive than purely syntactical *universal* desugarings and enable extensions that require type inference.

The SoundX verification procedure is strong enough to provide a Preservation and a Progress theorem. The Preservation theorem guarantees that a well-typed extended program is desugared into a well-typed base language program. The Progress theorem guarantees that such a desugaring is always possible, provided that the universal desugarings form a terminating rewrite system and all forward steps can be performed successfully.

SoundX is implemented as a plugin for the Sugar* framework which is integrated into the Eclipse IDE. The SoundX implementation provides pragmatic solutions to generate fresh names in desugarings and for the precise location of typing errors in extended programs. Several case studies based on the Simply Typed $\lambda$-Calculus augmented with a simple module system demonstrate the applicability of our approach. These extensions show how to extend the type level (pair type in Section 1.3.2), how to define new binding structures (let-expressions in Section 3.2), how to compose extensions (pattern matching for pairs in Section 5.2.1), and how to bring specific names into scope (open-expression in Section 5.2.2). We show that the SoundX verification procedure rejects unsound extensions and how it can help to handle unhygienic desugarings.

SoundX also fosters a change in language design and implementation, probably not as a tool but as a method. With SoundX, it is possible to incrementally build a language based on a very small but expressive base language only containing the essential core constructs. There exists a huge amount of research results describing how to encode advanced features like object-oriented programming [GM94, Bru02] or powerful module systems [RRD10] in core $\lambda$-calculi. However, these result have rarely been used as a basis of language implementations leaving a gap between theory and practice. With SoundX, these theoretical results can be made applicable in a systematic and principled way.

We discussed the limitations of SoundX, namely the forward step and certain deficiencies regarding fresh names and binding structures. These two subjects are the most prominent areas for future work to make the system more robust and to give it a wider applicability.

It should be investigated if it is possible to find a sufficient computable criterion which guarantees that the forward step is always possible for a given base language and its extensions. As we have outlined in Section 6.2, an awkward formulation of the base language may lead to failures of the forward step. In its current form, SoundX plainly accepts the base language definition as "god-given." Such a criterion would also pose requirements onto the base language definition in addition to a possibly extended verification procedure. If it should turn out that such a criterion is hard to find, is not computable, or does not exist, we should identify a property in the sense of a proof obligation which provides the necessary guarantee. This obligation must then be proven on an individual basis using pencil and paper for each base language definition. This would improve the situation considerably since this proof obligation would line in with properties like type-soundness of the base language, which we also do not establish automatically.

We already suggested that formally rebasing SoundX on nominal logic would also enable base languages that require substitution in the definition of the type system in Section 6.3.2. Such a foundation also enables a

principled and less pragmatically inspired treatment of generation of and reasoning about fresh names, an area where the current formulation of SoundX is also limited (see Sections 5.1.3, 5.2.1 and 6.3.1). The switch to nominal logic requires a rather drastic redevelopment of the notion of a valid derivation and of the desugaring procedure. Moreover, a specification of the binding structure and a binding compatibility check for extensions must be integrated. Nevertheless, we believe that this work pays off by a tremendous increase in expressiveness and usefulness, for which the results and insights of this thesis provide a firm basis.

# A NOTATIONS

## A.1 NOTATIONAL CONVENTIONS

| Notation | Meaning |
|---|---|
| $\vec{\phi}$ | List of elements from $\phi$ |
| $\langle \phi_1, ..., \phi_n \rangle$ | Enumeration of elements of a list |
| $\phi \in \vec{\phi}$ | $\phi$ is contained in list $\vec{\phi}$ |
| $\phi \notin \vec{\phi}$ | $\phi$ is not contained in $\vec{\phi}$ |
| $\vec{\phi}_1 \sqsubseteq \vec{\phi}$ | All elements of $\vec{\phi}_1$ are contained in $\vec{\phi}$ |
| $\langle \phi, \vec{\phi} \rangle$ | Add $\phi$ to the head of $\vec{\phi}$ |
| $\langle \vec{\phi}, \phi \rangle$ | Add $\phi$ to the tail of $\vec{\phi}$ |
| $\langle \vec{\phi}_1, \vec{\phi}_2 \rangle$ | Concatenation of $\vec{\phi}_1$ and $\vec{\phi}_2$ |
| $\langle \phi_{1..n} \rangle$ | Short form for $\langle \phi_1, ..., \phi_n \rangle$ |
| $\vec{\phi}!\, k$ | Select $k$-th element |
| $m..n$ | Short form for $\{m, m+1, ..., n\}$ |
| $\{V_1 \mapsto E_1, ..., V_n \mapsto E_n\}$ | Explicit enumeration of the substitution $\sigma : \{V_1, ..., V_n\} \to \{E_1, ..., E_n\}$ with $\sigma(V_i) = E_i$ for $i \in 1..n$ |
| $dom(\sigma)$ | Domain of the substitution $\sigma$ |
| $\tilde{V}$ | Set of meta-variables |
| $\tilde{V}_1 \leftrightarrow \tilde{V}_2$ | Bijection between $\tilde{V}_1$ and $\tilde{V}_2$ |
| $\tilde{V}_1 \between \tilde{V}_2$ | Short form for $\tilde{V}_1 \cap \tilde{V}_2 = \emptyset$ (disjointness) |

## A.2 VARIABLE NAMES

| Variable | Meaning |
|---|---|
| $A$ | Constructor arities |
| $B$ | Base language definitions |
| $E$ | Expressions |
| $F$ | Judgement forms |
| $G$ | Guarded desugarings |
| $I$ | Inference rules |

| Variable | Meaning |
|---|---|
| $J$ | Judgements |
| $m$ | Module |
| $M$ | Module system definitions |
| $N$ | Atomic names |
| $\nabla$ | Derivations |
| $P$ | Import declarations |
| $\rho$ | Module bindings |
| $\sigma$ | Substitutions |
| $\Sigma$ | Exports |
| $U$ | Universal desugarings |
| $V$ | Meta-variables |
| $X$ | Extensions |

## A.3 INDUCTIVELY DEFINED STATEMENTS

| Statement | Meaning | Page |
|---|---|---|
| $\vec{A} \ni \diamond$ | Well-formedness of constructor arities | 32 |
| $\vec{F} \ni \diamond$ | Well-formedness of judgement forms | 34 |
| $\vec{A} \ni E : N$ | Well-formedness of expressions | 33 |
| $\vec{A}; \vec{F} \ni J$ | Well-formedness of judgements | 34 |
| $\vec{A}; \vec{F} \ni \vec{I}$ | Well-formedness of inference rules | 35 |
| $\vec{A}; \vec{F} \ni M$ | Well-formedness of module system definition | 74 |
| $\ni B$ | Well-formedness of base language definitions | 37 |
| $\vec{A} \ni U$ | Well-formedness of universal desugaring | 36 |
| $\vec{A}; \vec{F} \ni G$ | Well-formedness of guarded desugaring | 36 |
| $B \ni X$ | Well-formedness of extensions | 37 |
| $\vec{\nabla} \vdash_{\vec{I}} \nabla$ | Validity of a derivation | 39 |
| $E \mapsto_{\vec{U}} E'$ | One-step rewriting of expressions | 49 |
| $J \mapsto_{\vec{U}} J'$ | One-step rewriting of judgements | 49 |
| $\sigma \mapsto_{\vec{U}} \sigma'$ | One-step rewriting of substitutions | 49 |
| $(\vec{J}, J) \mapsto_{\vec{G};\vec{U}} (\vec{J}', J')$ | One-step rewriting of inference rules | 50 |
| $\sigma \Longmapsto_{\vec{U}} \sigma'$ | Repeated rewriting of substitutions | 50 |
| $(\vec{J}, J) \Longmapsto_{\vec{G};\vec{U}} (\vec{J}', J')$ | Repeated rewriting of inference rules | 50 |

| Statement | Meaning | Page |
|---|---|---|
| $\nabla \updownarrow_{B;X} \nabla'$ | One-step bottom-up desugaring | 55 |
| $\nabla \downarrow_{B;X} \nabla'$ | One-step top-down desugaring | 54 |
| $\nabla \Updownarrow_{B;X} \nabla'$ | Derivation desugaring | 56 |
| $B; X \ltimes I : \mathcal{X}$ | Classification of inference rules as $\mathcal{X}$ | 52 |
| $B; X \ltimes I : \mathcal{B}$ | Classification of inference rules as $\mathcal{B}$ | 52 |
| $B \ltimes X$ | Soundness of extensions | 53 |
| $B \ltimes \Sigma$ | Well-formedness and soundness of exports | 77 |
| $B \ltimes \vec{\rho}$ | Well-formedness and soundness of repositories | 77 |
| $E -:_M \vec{E}$ | Extract imports | 67 |
| $E -:_M E_{mid}$ | Extract header | 67 |
| $\vec{E} -:_{\vec{\rho}} \vec{X}$ | Compose imported extensions | 69 |
| $B \ltimes \vec{X}$ | Well-formedness of composed extensions | 75 |
| $B \ltimes \vec{X}$ | Soundness of composed extensions | 76 |
| $\vec{\rho} \Vdash_B m : (\rho, \nabla)$ | Module analysis | 73 |
| $\nabla \Updownarrow_{B;\vec{X}} \nabla'$ | Iterated derivation desugaring | 75 |
| $I \tilde{\in}_{\tilde{V}} \vec{I}$ | Fresh inference rule | 81 |
| $\vec{\nabla}_a \vdash_{\vec{I}} \vec{J} \rhd (\sigma, \vec{\nabla})$ | Resolution | 82 |

# B INDICES

## B.1 DEFINITIONS



## B.2 RULES

# C DETAILED PROOFS

In this chapter, we use the symbols $*$, $\dagger$, and $\ddagger$ as superscripts to variables. Their mere purpose is to provide more variable names, there is no other meaning associated with them.

## C.1 LEMMA 2.1

LEMMA 2.1 Structural properties of $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$

(i) Substitution: *If* $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ *then* $[\sigma](\vec{\nabla}_a) \vdash_{\vec{I}} [\sigma](\nabla)$.

(ii) Assumption exchange: *If* $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ *and* $concl(\vec{\nabla}_a) = concl(\vec{\nabla}'_a)$ *then there exists* $\nabla'$ *such that* $\vec{\nabla}'_a \vdash_{\vec{I}} \nabla'$ *and* $concl(\nabla) = concl(\nabla')$.

(iii) Transitivity: *If* $\langle \nabla_{1..n} \rangle \vdash_{\vec{I}} \nabla$ *and* $\langle\rangle \vdash_{\vec{I}} \nabla_i$, *for each* $i \in 1..n$, *then* $\langle\rangle \vdash_{\vec{I}} \nabla$.

(iv) Stability: *If* $\vec{I} \sqsubseteq \vec{I}_1$ *and* $\vec{\nabla}_a \vdash_{\vec{I}} \nabla$ *then* $\vec{\nabla}_a \vdash_{\vec{I}_1} \nabla$.

*Proof.*

(i) *Substitution:* 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.

(a) Case V-ASSUMPTION: By the rule's conclusion we have the identities

(1) $\langle \nabla_{a1}, ..., \nabla_{ak}, ..., \nabla_{an} \rangle = \vec{\nabla}_a$

(2) $\nabla_{ak} = \nabla$

and directly obtain

(3) $\langle [\sigma](\nabla_{a1}), ..., [\sigma](\nabla_{ak}), ..., [\sigma](\nabla_{an}) \rangle \vdash_{\vec{I}} [\sigma](\nabla_{ak})$

using V-ASSUMPTION.

(b) Case V-RULE: The conclusions of the subderivations yield the statements

(4) $(\langle J_{01..0n} \rangle \twoheadrightarrow^N J_0) \in \vec{I}$

(5) $[\sigma^*]\langle J_{01..0n}, J_0 \rangle = \langle concl\langle \nabla_{1..n} \rangle, J \rangle$

(6) for each $i \in 1..n : \vec{\nabla}_a \vdash_{\vec{I}} \nabla_i$ .

Applying the induction hypothesis to (6) gives

(7) $\quad$ for each $i \in 1..n : [\sigma](\vec{\nabla}_a) \vdash_{\vec{I}} [\sigma](\nabla_i)$

and from Equation (5) we obtain

(8) $\quad [\sigma \circ \sigma^*]\langle J_{01..0n}, J_0\rangle = \langle concl\langle[\sigma](\nabla_{1..n})\rangle, [\sigma](J)\rangle$

such that the result $[\sigma](\vec{\nabla}_a) \vdash_{\vec{I}} [\sigma](\langle\nabla_{1..n}\rangle \Rightarrow^N J)$ follows by V-Rule where we instantiate the substitution of the second premise with $\sigma \circ \sigma^*$.

(ii) *Assumption exchange:* 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.

(a) Case V-Assumption: By the rule's conclusion we have the identities

(9) $\quad \langle\nabla_{a1}, ..., \nabla_{ak}, ..., \nabla_{an}\rangle = \vec{\nabla}_a$

(10) $\quad \nabla_{ak} = \nabla\,.$

We take $\nabla' = \nabla'_{ak}$. By requirement and (10), we have $concl(\nabla) = concl(\nabla_{ak}) = concl(\nabla'_{ak})$ and by V-Assumption we obtain $\vec{\nabla}'_a \vdash_{\vec{I}} \nabla'$.

(b) Case V-Rule: The conclusions of the subderivations yield the statements

(11) $\quad (\langle J_{01..0n}\rangle \twoheadrightarrow^N J_0) \in \vec{I}$

(12) $\quad [\sigma]\langle J_{01..0n}, J_0\rangle = \langle concl\langle\nabla_{1..n}\rangle, J\rangle$

(13) $\quad$ for each $i \in 1..n : \vec{\nabla}_a \vdash_{\vec{I}} \nabla_i\,.$

By the induction hypothesis applied to (13) we know there exist $\nabla'_1$ to $\nabla'_n$ such that for each $i \in 1..n : \vec{\nabla}'_a \vdash_{\vec{I}} \nabla'_i$ and $concl(\nabla_i) = concl(\nabla'_i)$. From Equation (12) we obtain

(14) $\quad [\sigma]\langle J_{01..0n}, J_0\rangle = \langle concl\langle\nabla'_{1..n}\rangle, J\rangle$

and derive $\vec{\nabla}'_a \vdash_{\vec{I}} (\langle\nabla'_{1..n}\rangle \Rightarrow^N J)$ by V-Rule. Since the conclusion of this derivation is the conclusion of the original derivation, it is the required witness for $\nabla'$.

(iii) *Transitivity:* By induction on a derivation of $\langle\nabla_{1..n}\rangle \vdash_{\vec{I}} \nabla$ and a case analysis on the last rule applied in the derivation.

(a) Case V-Assumption: By the rule's conclusion we have the identities

(15) $\quad \langle\nabla_{a1}, ..., \nabla_{ak}, ..., \nabla_{an}\rangle = \langle\nabla_1, ..., \nabla_k, ..., \nabla_n\rangle$

(16) $\quad \nabla_{ak} = \nabla\,.$

By the requirement of the lemma we have $\langle\rangle \vdash_{\vec{I}} \nabla_k$ and the result is immediate.

(b) Case V-Rule: The conclusions of the subderivations yield the statements

$$(\langle J_{01..0m}\rangle \twoheadrightarrow^N J_0) \in \vec{I} \tag{17}$$

$$[\sigma](\langle J_{01..0m}, J_0\rangle) = \langle concl\langle \nabla_{01..0m}\rangle, J\rangle \tag{18}$$

$$\text{for each } i \in 1..m : \langle \nabla_{1..n}\rangle \vdash_{\vec{I}} \nabla_{0i}\,. \tag{19}$$

By the induction hypothesis we have

$$\text{for each } i \in 1..m : \langle\rangle \vdash_{\vec{I}} \nabla_{0i}, \tag{20}$$

which permits an instantiation of V-Rule using (17) and (18) that produces the result $\langle\rangle \vdash_{\vec{I}} (\langle \nabla_{01..0m}\rangle \Rightarrow^N J)$.

(iv) *Stability:* 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.

(a) Case V-Assumption: By the rule's conclusion we have the identities

$$\langle \nabla_{a1}, ..., \nabla_{ak}, ..., \nabla_{an}\rangle = \vec{\nabla} \tag{21}$$

$$\nabla_{ak} = \nabla\,. \tag{22}$$

We obtain $\vec{\nabla} \vdash_{\vec{I}_1} \nabla$ by V-Assumption.

(b) The conclusions of the subderivations yield the statements

$$(\langle J_{01..0n}\rangle \twoheadrightarrow^N J_0) \in \vec{I} \tag{23}$$

$$[\sigma](\langle J_{01..0n}, J_0\rangle) = \langle concl\langle \nabla_{1..n}\rangle, J\rangle \tag{24}$$

$$\text{for each } i \in 1..n : \vec{\nabla} \vdash_{\vec{I}} \nabla_i\,. \tag{25}$$

Applying the induction hypothesis to (25) results in

$$\text{for each } i \in 1..n : \vec{\nabla} \vdash_{\vec{I}_1} \nabla_i. \tag{26}$$

Since $\vec{I} \sqsubseteq \vec{I}_1$, we have $(\langle J_{01..0n}\rangle \twoheadrightarrow J_0) \in \vec{I}_1$. Using this fact, (24) and (26) we get the required result by V-Rule. □

## C.2 Lemma 3.2

Lemma 3.2

*Let $B = (\vec{A}, \vec{F}, \vec{I})$ be a well-formed base system and $X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ be a well-formed extension with respect to $B$.*

(i) *If $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x\rangle} \nabla$ and $\nabla \downarrow_{B;X} \nabla'$ then $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x\rangle} \nabla'$.*

(ii) *If for each* $\nabla_1 \in \vec{\nabla} : \langle\rangle \vdash_{\vec{I}} \nabla_1$ *and* $(\vec{\nabla} \Rightarrow^N J) \uparrow_{B;X} \nabla'$ *then* $\langle\rangle \vdash_{\vec{I}} \nabla'$.

*Proof.*

(i) By a case analysis on the last rule used in the derivation of $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$.

(a) Case V-Assumption: Cannot happen since there are no assumptions.

(b) Case V-Rule: The conclusions of the subderivations yield the statements

$$(\vec{J}_0 \twoheadrightarrow^N J_0) \in \langle \vec{I}, \vec{I}_x \rangle \tag{1}$$

$$[\sigma]\langle \vec{J}_0, J_0 \rangle = \langle concl(\vec{\nabla}), J \rangle \tag{2}$$

$$\text{for each } \nabla_1 \in \vec{\nabla} : \langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla_1 \,. \tag{3}$$

Since $B$ is well-formed and $X$ is well-formed with respect to $B$ there exists only one inference rule scheme of the name $N$ in $\langle \vec{I}, \vec{I}_x \rangle$. Hence, $(\vec{J}_0 \twoheadrightarrow^N J_0)$ can either be from the base language or from the extension.

(A) Case $(\vec{J}_0 \twoheadrightarrow^N J_0) \in \vec{I}$ (base language): In this case, the second requirement $\nabla \downarrow_{B;X} \nabla'$ of the lemma can only be derived by TD-Base since the other two rules TD-ExtBase and TD-ExtExt demand $(\vec{J}_0 \twoheadrightarrow^N J_0)$ to be from the extension. According to the conclusion of TD-Base we have $\nabla' = \nabla$, and the result is immediate.

(B) Case $(\vec{J}_0 \twoheadrightarrow^N J_0) \in \vec{I}_x$ (extension): In this case, the second requirement $\nabla \downarrow_{B;X} \nabla'$ of the lemma can either be derived by TD-ExtBase or TD-ExtExt. We perform a case analysis:

(I) Case TD-ExtBase: According to the conclusion of this rule, we have $\nabla' = \nabla$, and the result is immediate.

(II) Case TD-ExtExt: The conclusions of the subderivations of the instantiation of TD-ExtExt yield the statement

$$\vec{\nabla} \vdash_{\langle \vec{I}, \vec{I}_x \rangle} (\vec{\nabla}' \Rightarrow^{N'} [\sigma_1](J_0')) \,. \tag{4}$$

We apply Lemma 2.1(iii) (Transitivity) to the last statement and (3) to obtain the required result

$$\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} (\vec{\nabla}' \Rightarrow^{N'} [\sigma_1](J_0')) \,. \tag{5}$$

(ii) The second requirement of the lemma $(\vec{\nabla} \Rightarrow^N J) \uparrow_{B;X} \nabla'$ can either be derived by BU-Base or BU-Ext. We perform a case analysis:

(a) Case BU-Base: The conclusions of the subderivations of the instantiation of BU-Base yield the statements

$$(\vec{J}_0 \twoheadrightarrow^N J_0) \in \vec{I} \tag{6}$$

$$[\sigma_1](\vec{J}_0) = concl(\vec{\nabla}) \tag{7}$$

$$[\sigma_2](J_0) = J \tag{8}$$

$$\sigma_2 \setminus vars(\vec{J}_0) \Longmapsto_{\vec{U}_x} \sigma_2' \,. \tag{9}$$

Since $dom(\sigma_2') = dom(\sigma_2 \setminus vars(\vec{J}_0)) = dom(\sigma_2) \setminus vars\langle\vec{J}_0\rangle$ by the definition of substitution desugaring and (9) we have $[\sigma_2'](\vec{J}_0) = \vec{J}_0$. Inserting this identity into (7) gives

$$[\sigma_1 \circ \sigma_2'](\vec{J}_0) = concl(\vec{\nabla}) \,. \tag{10}$$

So, we definitely have

$$[\sigma_1 \circ \sigma_2']\langle\vec{J}_0, J_0\rangle = \langle concl(\vec{\nabla}), [\sigma_1 \circ \sigma_2'](J_0)\rangle \,. \tag{11}$$

We derive the required result $\langle\rangle \vdash_{\vec{I}} (\vec{\nabla} \Rightarrow^N [\sigma_1 \circ \sigma_2'](J_0))$ by V-Rule applied to (6) and (11) and the requirement of the lemma.

(b) Case BU-Ext: By the conclusion of the subderivations of the instantiation of BU-Ext we have

$$\vec{\nabla} \vdash_{\vec{I}} (\vec{\nabla}' \Rightarrow^{N'} [\sigma_1 \circ \sigma_2'](J_0')) \,. \tag{12}$$

Applying Lemma 2.1(iii) (Transitivity) to this statement and the requirement of the lemma yields $\langle\rangle \vdash_{\vec{I}} (\vec{\nabla}' \Rightarrow^{N'} [\sigma_1 \circ \sigma_2'](J_0'))$ as necessary. □

## C.3 theorem 3.1

Theorem 3.1 Preservation

*Let $B = (\vec{A}, \vec{F}, \vec{I})$ be a well-formed base system and $X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ be a well-formed extension with respect to $B$.*
*If $\langle\rangle \vdash_{\langle\vec{I}, \vec{I}_x\rangle} \nabla$ and $\nabla \Updownarrow_{B;X} \nabla'$ then $\langle\rangle \vdash_{\vec{I}} \nabla'$.*

*Proof.* By induction on a derivation of $\nabla \Updownarrow_{B;X} \nabla'$. This statement must be derived by an instantiation of DU-Desugar. Consequently, the conclusions of the subderivations yield these statements:

$$\nabla \downarrow_{B;X} (\langle\nabla'_{1..n}\rangle \Rightarrow^{N'} J') \tag{1}$$

$$\text{for each } i \in 1..n : \nabla'_i \Updownarrow_{B;X} \nabla''_i \tag{2}$$

$$(\langle\nabla''_{1..n}\rangle \Rightarrow^{N'} J') \uparrow_{B;X} \nabla''' \tag{3}$$

By the requirement of the theorem and (1) together with Lemma 3.2(i) we get

(4) $$\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} (\langle \nabla'_{1..n} \rangle \Rightarrow^{N'} J') .$$

Since there are no assumptions, the previous statement must be derived by V-Rule such that we obtain

(5) $$\text{for each } i \in 1..n : \ \langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla'_i .$$

Using the induction hypothesis on these statements and (2) results in

(6) $$\text{for each } i \in 1..n : \ \langle\rangle \vdash_{\vec{I}} \nabla''_i .$$

We conclude $\langle\rangle \vdash_{\vec{I}} \nabla'''$ using Lemma 3.2(ii) applied to the previous statements and (3). □

## C.4 THEOREM 3.4

THEOREM 3.4 Progress

*Let $B = (\vec{A}, \vec{F}, \vec{I})$ be a well-formed base system and $X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ be a well-formed and sound extension with respect to B.*
*If $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ and the universal desugarings $\vec{U}_x$ form a terminating rewrite system then either $\nabla \Updownarrow_{B;X} \lightning$ or there exists a $\nabla'$ such that $\nabla \Updownarrow_{B;X} \nabla'$.*

*Proof.* By induction on a derivation of $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ and a case analysis on the last rule applied in the derivation.

(i) Case V-Assumption: Cannot happen since there are no assumptions.

(ii) Case V-Rule: The conclusions of the subderivations yield the statements

(1) $$(\vec{J}_0 \twoheadrightarrow^N J_0) \in \langle \vec{I}, \vec{I}_x \rangle$$

(2) $$[\sigma]\langle \vec{J}_0, J_0 \rangle = \langle concl(\vec{\nabla}), J \rangle$$

(3) $$\text{for each } \nabla_1 \in \vec{\nabla} : \langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla_1 .$$

Using Lemma 3.5(i), we obtain $\nabla^* = (\langle \nabla^*_{1..n} \rangle \Rightarrow^{N^*} J^*)$ with $\nabla \downarrow_{B;X} \nabla^*$. Applying Lemma 3.2(i) to this statement and the requirement of the theorem we get $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla^*$. Since there are no assumptions, the last statement must be derived by V-Rule and, by inversion, all subderivations $\nabla^*_1$ to $\nabla^*_n$ of $\nabla^*$ are valid.

Using the induction hypothesis on each assumption $\nabla^*_i$, $i \in 1..n$, gives

(4) $$\text{either } \nabla^*_i \Updownarrow_{B;X} \lightning \text{ or there exists } \nabla'_i \text{ such that } \nabla^*_i \Updownarrow_{B;X} \nabla'_i .$$

We have to consider two cases:

(a) Case for each $i \in 1..n$: there exists $\nabla'_i$ such that $\nabla^*_i \Updownarrow_{B;X} \nabla'_i$: Using Theorem 3.1 (Preservation) for each $i \in 1..n$ provides us with $\langle\rangle \vdash_{\vec{I}} \nabla'_i$.

By Lemma 3.5(ii) we know that either $(\vec{J}_0 \twoheadrightarrow^{N^*} J_0) \in \vec{I}$ or $(\vec{J}_0 \twoheadrightarrow^{N^*} J_0) \in \vec{I}_x$ with $B; X \ltimes (\vec{J}_0 \twoheadrightarrow^{N^*} J_0) : \mathcal{B}$. In both cases $J^*$ is a substitution instance of $J_0$, that is, $J^* = [\sigma](J_0)$ for some $\sigma$, since $\nabla^*$ is a valid derivation.

(A) Case $(\vec{J}_0 \twoheadrightarrow^{N^*} J_0) \in \vec{I}$: The conclusions of $\langle\nabla'_{1..n}\rangle$ are either a substitution instance of $\vec{J}_0$ or they are not.

(I) $[\sigma_1](\vec{J}_0) = \mathit{concl}\langle\nabla_{1..n}\rangle$: Since the universal desugarings form a terminating rewrite system, there exists a $\sigma'$ such that $\sigma \setminus \mathit{vars}(\vec{J}_0) \Longrightarrow_{\vec{U}_x} \sigma'$. We are in position to instantiate BU-Base to derive $(\langle\nabla'_{1..n}\rangle \Rightarrow^{N^*} J^*) \updownarrow_{B;X} (\langle\nabla'_{1..n}\rangle \Rightarrow^{N^*} [\sigma_1 \circ \sigma'](J_0))$. We get the required result $\nabla' = (\langle\nabla'_{1..n}\rangle \Rightarrow^{N^*} [\sigma_1 \circ \sigma'])$ using DU-Desugar.

(II) Case $[\sigma_1](\vec{J}_0) \neq \mathit{concl}\langle\nabla_{1..n}\rangle$ for all $\sigma_1$: By BU-BaseStuck we immediately have $(\langle\nabla'_{1..n}\rangle \Rightarrow^{N*} J^*) \updownarrow_{B;X} \lightning$ and hence, by DU-StuckBU, $\nabla \Updownarrow_{B;X} \lightning$.

(B) Case $(\vec{J}_0 \twoheadrightarrow^{N^*} J_0) \in \vec{I}_x$: Since the inference rule $(\vec{J}_0 \twoheadrightarrow^{N^*} J_0)$ is sound we have, by inversion of S-Base

$$(\langle J_{01..0n}\rangle \twoheadrightarrow^{N^*} J_0) \Longrightarrow_{\vec{G}_x;\vec{U}_x} (\langle J'_{01..0n}\rangle, J'_0) \tag{5}$$

$$\text{for each } i \in 1..n : \vec{A}; \vec{F} \mathrel{\S\!\!-} J'_{0i} \tag{6}$$

$$\langle !\, J'_{01}, \ldots, !\, J'_{0n}\rangle \vdash_{\vec{I}} (\vec{\nabla}^\dagger \Rightarrow^{N^\dagger} J'_0)\,. \tag{7}$$

The conclusions of $\langle\nabla'_{1..n}\rangle$ are either a substitution instance of $\vec{J}'_0 = \langle J'_{01..0n}\rangle$ or they are not.

(I) Case $[\sigma_1](\vec{J}'_0) = \mathit{concl}\langle\nabla'_{1..n}\rangle$: Since the universal desugarings form a terminating rewrite system, there exists a $\sigma'$ such that $\sigma \setminus \mathit{vars}(\vec{J}'_0) \Longrightarrow_{\vec{U}_x} \sigma'$. Applying Lemma 2.1(i) (Substitution) to the statement (7) results in

$$\langle !\, [\sigma_1 \circ \sigma'](J'_{01}), \ldots, !\, [\sigma_1 \circ \sigma'](J'_{0n})\rangle \vdash_{\vec{I}} ([\sigma_1 \circ \sigma'](\vec{\nabla}^\dagger) \Rightarrow^{N^\dagger} [\sigma_1 \circ \sigma'](J'_0))\,. \tag{8}$$

Since

$$\begin{aligned} \mathit{dom}(\sigma') &= \mathit{dom}(\sigma \setminus \mathit{vars}\langle J'_{01..0n}\rangle) \\ &= \mathit{dom}(\sigma) \setminus \mathit{vars}\langle J'_{01..0n}\rangle \end{aligned} \tag{9}$$

by the definition of substitution desugaring, we have $[\sigma'](J_{0i}) = J_{0i}$ for each $i \in 1..n$. Inserting these identities

into the previous statement yields

$$(10)\quad \langle !\,[\sigma_1](J'_{01}), \ldots, !\,[\sigma_1](J'_{0n})\rangle \vdash_{\vec{I}} ([\sigma_1 \circ \sigma'](\vec{\nabla}^{\dagger}) \Rightarrow^{N^{\dagger}} [\sigma_1 \circ \sigma'](J'_0))$$

to which we apply Lemma 2.1(ii) (Assumption exchange) and conclude

$$(11)\quad \langle \nabla'_{1..n}\rangle \vdash_{\vec{I}} (\vec{\nabla}^{\ddagger} \Rightarrow^{N^{\ddagger}} [\sigma_1 \circ \sigma'](J'_0))$$

for some $N^{\ddagger}$ and $\vec{\nabla}^{\ddagger}$. By an instantiation of BU-Ext we obtain

$$(12)\quad (\langle \nabla'_{1..n}\rangle \Rightarrow^{N^*} J^*) \uparrow_{B;X} (\vec{\nabla}^{\ddagger} \Rightarrow^{N^{\ddagger}} [\sigma_1 \circ \sigma'](J'_0))\,.$$

Finally, we derive the required result by DU-Desugar with $\nabla' = (\vec{\nabla}^{\ddagger} \Rightarrow^{N^{\ddagger}} [\sigma_1 \circ \sigma'](J'_0))$.

(II) Case $[\sigma](\vec{J}'_0) \neq \mathit{concl}\langle \nabla'_{1..n}\rangle$ for all $\sigma$: By BU-ExtStuck we immediately have $(\langle \nabla'_{1..n}\rangle \Rightarrow^{N^*} J^*) \uparrow_{B;X} \lightning$ and hence by DU-StuckBU $\nabla \Updownarrow_{B;X} \lightning$.

(b) Case for some $i \in 1..n : \nabla^*_i \Updownarrow_{B;X} \lightning$: We derive $\nabla \Updownarrow_{B;X} \lightning$ immediately by DU-StuckSub. □

## C.5 lemma 3.5

Lemma 3.5

*Let $B = (\vec{A}, \vec{F}, \vec{I})$ be a well-formed base system and $X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x)$ be a well-formed and sound extension with respect to $B$.*

(i) *If $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x\rangle} \nabla$ then there exists a $\nabla'$ such that $\nabla \downarrow_{B;X} \nabla'$.*

(ii) *If $\nabla \downarrow_{B;X} (\vec{\nabla}' \Rightarrow^{N'} J')$ then either $(\vec{J}_0 \twoheadrightarrow^{N'} J_0) \in \vec{I}$ or $(\vec{J}_0 \twoheadrightarrow^{N'} J_0) \in \vec{I}_x$ with $B; X \ltimes (\vec{J}_0 \twoheadrightarrow^{N'} J_0) : \mathcal{B}$.*

*Proof.*

(i) By a case analysis on the last rule used in the derivation of $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x\rangle} \nabla$.

(a) Case V-Assumption: Cannot happen since there are no assumptions.

(b) Case V-Rule: The conclusions of the subderivations yield the statements

(1) $$(\vec{J}_0 \twoheadrightarrow^N J_0) \in \langle \vec{I}, \vec{I}_x \rangle$$

(2) $$[\sigma]\langle \vec{J}_0, J_0 \rangle = \langle concl(\vec{\nabla}), J \rangle$$

(3) $$\text{for each } \nabla_1 \in \vec{\nabla} : \langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla_1 \,.$$

We perform a case analysis on the origin of the rule $(\vec{J}_0 \twoheadrightarrow^N J_0)$:

(A) Case $(\vec{J}_0 \twoheadrightarrow^N J_0) \in \vec{I}$ (base language): Immediate by TD-Base with $\nabla' = \nabla$.

(B) Case $(\vec{J}_0 \twoheadrightarrow^N J_0) \in \vec{I}_x$ (extension): Since the extension is sound, we either have

$$B; X \ltimes (\vec{J}_0 \twoheadrightarrow^N J_0) : \mathcal{B} \quad \text{or} \quad B; X \ltimes (\vec{J}_0 \twoheadrightarrow^N J_0) : \mathcal{X} \,.$$

(I) $\mathcal{B}$ case: The result is immediate by TD-ExtBase with $\nabla' = \nabla$.

(II) $\mathcal{X}$ case: We derive the conclusion of the lemma by an instantiation of TD-ExtExt. The first four premises are fulfilled immediately: the first premise is satisfied due to (1), the second premise is satisfied since the extension is sound, the third premise follows from the second one by inversion, and the fourth premise is satisfied due to (2). Moreover, by extension soundness, we have

(4) $$\langle !J_{01}, \ldots, !J_{0n} \rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} (\vec{\nabla}^* \Rightarrow^{N^*} J_0') \qquad \text{with} \quad \langle J_{01..0n} \rangle = \vec{J}_0$$

and $(\vec{J}^* \twoheadrightarrow^{N^*} J^*) \in \vec{I}$.
Applying Lemma 2.1(i) (Substitution) to this statement gives

(5) $$\langle ![\sigma](J_{01}), \ldots, ![\sigma](J_{0n}) \rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} ([\sigma](\vec{\nabla}^*) \Rightarrow^{N^*} [\sigma](J_0')) \,.$$

The last statement must be derived by V-Rule since

(6) $$\text{for each } i \in 1..n : \quad ([\sigma](\nabla^*) \Rightarrow^{N^*} [\sigma](J_0')) \neq ![\sigma](J_{0i}) \,.$$

By inversion of V-Rule we have

(7) $$\text{for each } \nabla_1^* \in [\sigma](\vec{\nabla}^*) : \quad \langle ![\sigma](J_{01}), \ldots, ![\sigma](J_{0n}) \rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla_1^*$$

(8) $[\sigma^*]\langle \vec{J}^*, J^* \rangle = \langle concl([\sigma](\vec{\nabla}^*)), [\sigma](J_0') \rangle$

According to Lemma 2.1(ii) (Assumption exchange) applied to (7) and (2) there exists $\vec{\nabla}^\dagger$ such that

(9) for each $\nabla_1^\dagger \in \vec{\nabla}^\dagger : \vec{\nabla} \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla_1^\dagger$

(10) $concl([\sigma](\vec{\nabla}^*)) = concl(\vec{\nabla}^\dagger)$ .

Inserting the identity (10) into (8) gives

(11) $[\sigma^*]\langle \vec{J}^*, J^* \rangle = \langle concl(\vec{\nabla}^\dagger), [\sigma](J_0') \rangle$ .

Since $(\vec{J}^* \twoheadrightarrow^{N^*} J^*) \in \vec{I}$ we also have $(\vec{J}^* \twoheadrightarrow^{N^*} J^*) \in \langle \vec{I}, \vec{I}_x \rangle$ and instantiate V-Rule using (9) and (11) to derive

(12) $\vec{\nabla} \vdash_{\langle \vec{I}, \vec{I}_x \rangle} (\vec{\nabla}^\dagger \Rightarrow^{N^*} [\sigma](J_0'))$ .

Finally, we are in the position to derive the required result by TD-ExtExt and obtain $\nabla = (\vec{\nabla}^\dagger \Rightarrow^{N^*} [\sigma](J_0'))$.

(ii) By a case analysis on the last rule applied in the derivation of $\nabla \downarrow_{B;X} (\vec{\nabla}' \Rightarrow^{N'} J')$.

(a) Case TD-Base: The result is immediate by the instantiations of the premise $(\vec{J}_0 \twoheadrightarrow^{N'} J_0) \in \vec{I}$ of this rule.

(b) Case TD-ExtBase: The result is immediate by the instantiations of the premises $B; X \ltimes (\vec{J}_0 \twoheadrightarrow^{N'} J_0) : \mathcal{B}$ of this rule.

(c) Case TD-ExtExt: The result is immediate by the instantiation of the last premise of this rule. □

## C.6 lemma 4.1

Lemma 4.1 Preservation and Progress for composed extensions
*Let $B = (\vec{A}, \vec{F}, \vec{I}, M)$ be a well-formed base language definition, $\vec{X}$ a list of extensions with $B \wr\!\!\!\leftarrow \vec{X}$ and $B \ltimes \vec{X}$ where $(\vec{A}_x, \vec{F}_x, \vec{I}_x, M_x) = B \Cup \vec{X}$.*

(i) *If $\langle\rangle \vdash_{\vec{I}_x} \nabla$, and $\nabla \Updownarrow_{B;\vec{X}} \nabla'$ then $\langle\rangle \vdash_{\vec{I}} \nabla'$.*

(ii) *If $\langle\rangle \vdash_{\vec{I}_x} \nabla$, and the universal desugarings of each $X \in \vec{X}$ form a terminating rewrite system then either $\nabla \Updownarrow_{B;\vec{X}} \not\downarrow$ or there exists a $\nabla'$ such that $\nabla \Updownarrow_{B;\vec{X}} \nabla'$.*

*Proof.*

(i) By induction on the structure of $\vec{X}$.

(a) Case $\vec{X} = \langle\rangle$: We have $\vec{I}_x = \vec{I}$ and $\nabla \Updownarrow_{B;\langle\rangle} \nabla'$ can only be derived by D-Empty such that $\nabla' = \nabla$. The result $\langle\rangle \vdash_{\vec{I}} \nabla'$ is immediate.

(b) Case $\vec{X} = \langle X_1, \vec{X}_2 \rangle$: Let $B_{x2} = B \Cup \vec{X}_2 = (\vec{A}_{x2}, \vec{F}_{x2}, \vec{I}_{x2}, M_{x2})$.
The statement $\nabla \Updownarrow_{B;\vec{X}} \nabla'$ can only be derived by D-Ext. That is, by inversion we obtain the statements

$$\nabla \Updownarrow_{B_{x2};X_1} \nabla_1 \tag{1}$$
$$\nabla_1 \Updownarrow_{B;\vec{X}_2} \nabla' . \tag{2}$$

Similarly, the statement $B \Subset \vec{X}$ can only be derived by WX-Exts such that we have

$$\Subset B_{x2} \tag{3}$$
$$B_{x2} \Subset X_1 \tag{4}$$
$$B \Subset \vec{X}_2 \tag{5}$$

by inversion. That is, $B_{x2}$ is a well-formed base language definition and $X_1$ is a well-formed extension with respect to that base language.
By Theorem 3.1 applied to $\langle\rangle \vdash_{\vec{I}_x} \nabla$ from the requirements of the lemma, (1), (3) and (4), we conclude

$$\langle\rangle \vdash_{\vec{I}_{x2}} \nabla_1 . \tag{6}$$

In this use of Theorem 3.1, we make use of the fact that the order of inference rules in its first requirement $\langle\rangle \vdash_{\langle \vec{I}, \vec{I}_x \rangle} \nabla$ is irrelevant. It is an immediate consequence of the definition of the validity statement that it is stable with respect to permutation of inference rules.
Using the induction hypothesis on (2), (5) and (6) provides the required result.

(ii) By induction on the structure of $\vec{X}$:

(a) Case $\vec{X} = \langle\rangle$: The result is immediate by D-Empty with $\nabla' = \nabla$.

(b) Case $\vec{X} = \langle X_1, \vec{X}_2 \rangle$: Let $B_{x2} = B \Cup \vec{X}_2 = (\vec{A}_{x2}, \vec{F}_{x2}, \vec{I}_{x2}, M_{x2})$. The statement $B \Subset \vec{X}$ can only be derived by WX-Exts such that we have

$$\Subset B_{x2} \tag{7}$$
$$B_{x2} \Subset X_1 \tag{8}$$
$$B \Subset \vec{X}_2 \tag{9}$$

by inversion and $B \ltimes \vec{X}$ can only by derived by SX-Exts such that we similarly have

(10) $$B_{x2} \ltimes X_1$$

(11) $$B \ltimes \vec{X}_2 \,.$$

By Theorem 3.4 we either have $\nabla \Updownarrow_{B_{x2};X_1} \lightning$ or there exists a $\nabla'$ such that $\nabla \Updownarrow_{B_{x2};X_1} \nabla'$. We perform a case analysis:

(A) Case $\nabla \Updownarrow_{B_{x2};X_1} \lightning$: The result is immediate by an application of D-Stuck1.

(B) Case $\nabla \Updownarrow_{B_{x2};X_1} \nabla'$: By Theorem 3.1 applied to (7) and (8) we obtain $\langle\rangle \vdash_{\vec{I}_{x2}} \nabla'$. Using the induction hypothesis with the requirements of the lemma, (8) and (10), and the previous statement yields either $\nabla' \Updownarrow_{B;\vec{X}_2} \lightning$ or there exists a $\nabla''$ such that $\nabla' \Updownarrow_{B;\vec{X}_2} \nabla''$. In the first case, the required result is immediate by D-Stuck2. In the second case, the result is obtained by an application of D-Ext. □

## C.7 lemma 4.2

Lemma 4.2 Properties of extensions

(i) *If* $\mathfrak{E} B$ *and* $B \mathfrak{E} \vec{X}$ *then* $\mathfrak{E} B \Cup \vec{X}$.

(ii) *If* $B \ltimes X$ *and* $B \sqsubseteq B_1$ *then* $B_1 \ltimes X$.

(iii) *If* $\mathfrak{E} B$, $B \mathfrak{E} \vec{X}_1$, $B \ltimes \vec{X}_1$, $B \mathfrak{E} \vec{X}_2$, $B \ltimes \vec{X}_2$, *and* $\mathfrak{E} B \Cup (\vec{X}_1 \twoheadrightarrow \vec{X}_2)$ *then* $B \ltimes (\vec{X}_1 \twoheadrightarrow \vec{X}_2)$.

*Proof.*

(i) Immediate by the definition of base language definition and extension well-formedness.

(ii) We perform the following decompositions:

$$B = (\vec{A}, \vec{F}, \vec{I}, M) \qquad B_1 = (\vec{A}_1, \vec{F}_1, \vec{I}_1, M_1) \qquad X = (\vec{A}_x, \vec{I}_x, \vec{G}_x, \vec{U}_x) \,.$$

Since $B \ltimes X$, we have

(1) $$\text{for each } I_x \in \vec{I}_x : \; B \ltimes I_x : \mathcal{X} \quad \text{or} \quad B \ltimes I_x : \mathcal{B} \,.$$

For each $I_x = (\langle J_{01..0n} \rangle \rightarrow^N J_0)$, we perform a case analysis.

(a) Case $B \ltimes I_x : \mathcal{X}$: By inversion of S-Ext we have

(2) $$(\langle J_{01..0n}\rangle \rightarrowtriangle^N J_0) \Longmapsto_{\vec{G}_x;\vec{U}_x} (\langle J'_{01..0n}\rangle, J'_0)$$

(3) $$\langle !J_{01}, \ldots, !J_{0n}\rangle \vdash_{\langle \vec{I},\vec{I}_x\rangle} (\vec{\nabla}' \Rightarrow^{N'} J'_0)$$

(4) $$(\vec{J}' \rightarrowtriangle^{N'} J') \in \vec{I}\,.$$

Since $B \sqsubseteq B_1$ we have $\vec{I} \sqsubseteq \vec{I}_1$ and hence

(5) $$\langle \vec{I},\vec{I}_x\rangle \sqsubseteq \langle \vec{I}_1,\vec{I}_x\rangle\,.$$

By Lemma 2.1(iv) (Stability) we get

(6) $$\langle !J_{01}, \ldots, !J_{0n}\rangle \vdash_{\langle \vec{I}_1,\vec{I}_x\rangle} (\vec{\nabla}' \Rightarrow^{N'} J'_0)$$

by (3) and (5). Clearly, $\vec{I} \sqsubseteq \vec{I}_1$ and (4) imply $(\vec{J}' \rightarrowtriangle^{N'} J') \in \vec{I}_1$ and we derive $B_1 \ltimes I_x : \mathcal{X}$ by the last statement, (2) and (6) using S-Ext.

(b) Case $B \ltimes I_x : \mathcal{B}$: By inversion of S-Base we have

(7) $$(\langle J_{01..0n}\rangle \rightarrowtriangle^N J_0) \Longmapsto_{\vec{G}_x;\vec{U}_x} (\langle J'_{01..0n}\rangle, J'_0)$$

(8) $$\text{for each } i \in 1..n : \vec{A};\vec{F} \mathrel{\&\!\!-} J'_{0i}$$

(9) $$\langle !J'_{01}, \ldots, !J'_{0n}\rangle \vdash_{\vec{I}} (\vec{\nabla}' \Rightarrow^{N'} J'_0)\,.$$

From $B \sqsubseteq B_1$ we have $\vec{A} \sqsubseteq \vec{A}_1$ and $\vec{F} \sqsubseteq \vec{F}_1$. By the definition of judgement well-formedness and (8) we obtain

(10) $$\text{for each } i \in 1..n : \vec{A}_1;\vec{F}_1 \mathrel{\&\!\!-} J'_{0i}\,.$$

From (9) and $\vec{I} \sqsubseteq \vec{I}_1$ using Lemma 2.1(iv) we obtain

(11) $$\langle !J'_{01}, \ldots, !J'_{0n}\rangle \vdash_{\vec{I}_1} (\vec{\nabla}' \Rightarrow^{N'} J'_0)\,.$$

From the last two statements and (7) we get $B_1 \ltimes I_x : \mathcal{B}$ using S-Base as required.

(iii) By induction on the structure of $\vec{X}_1$.

(a) Case $\vec{X}_1 = \langle\rangle$: Immediate, since by the definition ExtNub

(12) $$(\langle\rangle \twoheadrightarrow \vec{X}_2) = \vec{X}_2$$

and $B \ltimes \vec{X}_2$ by the second requirement of the lemma.

(b) Case $\vec{X}_1 = \langle X_{11}, \vec{X}_{12}\rangle$: Following the definition ExtNub we perform a case analysis:

(A) $X_{11} \in \vec{X}_2$: In this case, we have

$$(\vec{X}_1 \twoheadrightarrow \vec{X}_2) = (\vec{X}_{12} \twoheadrightarrow \vec{X}_2)\,. \tag{13}$$

Due to the structure of $\vec{X}_1$, the statement $B \ltimes \vec{X}_1$ must be derived by SX-Exts and the inversion of this rule yields $B \ltimes \vec{X}_{12}$. Similarly, we obtain $B \mathrel{\&\!\!-} \vec{X}_{12}$ by inversion of WX-Exts. Using (13), we obtain $\mathrel{\&\!\!-} B \Cup (\vec{X}_{12} \twoheadrightarrow \vec{X}_2)$ from the requirement of the lemma. We apply the induction hypothesis to $\vec{X}_{12}$ and conclude $B \ltimes (\vec{X}_{12} \twoheadrightarrow \vec{X}_2)$ which entails $B \ltimes (\vec{X}_1 \twoheadrightarrow \vec{X}_2)$ by (13).

(B) $X_{11} \notin \vec{X}_2$: In this case, we have

$$(\vec{X}_1 \twoheadrightarrow \vec{X}_2) = \langle X_{11}, \vec{X}_{12} \twoheadrightarrow \vec{X}_2\rangle\,. \tag{14}$$

The requirement $B \mathrel{\&\!\!-} \vec{X}_2$ implies $\mathrel{\&\!\!-} B \Cup \vec{X}_2$ by Lemma 4.2(i). By the inversion of SX-Exts and $B \ltimes \langle X_{11}, \vec{X}_{12}\rangle$ we have $\mathrel{\&\!\!-} B \Cup \vec{X}_{12}$. That is, neither the well-formedness of $B \Cup \vec{X}_2$ nor $B \Cup \vec{X}_{12}$ depends on the arities defined in $X_{11}$. Applying Equation (14) to the requirement $\mathrel{\&\!\!-} B \Cup (\vec{X}_1 \twoheadrightarrow \vec{X}_2)$ results in

$$\mathrel{\&\!\!-} B \Cup \langle X_{11}, \vec{X}_{12} \twoheadrightarrow \vec{X}_2\rangle\,. \tag{15}$$

Using the fact that both $B \Cup \vec{X}_{12}$ and $B \Cup \vec{X}_2$ do not depend on $X_{11}$ this implies

$$\mathrel{\&\!\!-} B \Cup (\vec{X}_{12} \twoheadrightarrow \vec{X}_2)\,. \tag{16}$$

Similar to the previous case, the inversion of SX-Exts provides us with $B \ltimes \vec{X}_{12}$ and the inversion of WX-Exts provides us with $B \mathrel{\&\!\!-} \vec{X}_{12}$ such that we obtain

$$B \ltimes (\vec{X}_{12} \twoheadrightarrow \vec{X}_2) \tag{17}$$

by the induction hypothesis using (16).
Also by the inversion of SX-Exts we have

$$B \Cup \vec{X}_{12} \ltimes X_{11}\,. \tag{18}$$

By the definitions ExtNub and BXMerge we clearly have

$$(B \Cup \vec{X}_{12}) \sqsubseteq (B \Cup (\vec{X}_{12} \twoheadrightarrow \vec{X}_2))\,. \tag{19}$$

Applying Lemma 4.2(ii) to (18) and (19) yields

$$B \Cup (\vec{X}_{12} \twoheadrightarrow \vec{X}_2) \ltimes X_{11}\,. \tag{20}$$

Instantiating SX-Exts with (16), (17) and (20) yields the required statement $B \ltimes \langle X_{11}, \vec{X}_{12} \twoheadrightarrow \vec{X}_2\rangle$. □

## C.8 LEMMA 4.3

LEMMA 4.3 Soundness of imports
*If* $\mathfrak{E}\vdash B$, $B \ltimes \vec{\rho}$ *and* $\vec{E} -:_{B;\vec{\rho}} \vec{X}$ *then* $B \mathfrak{E}\vdash \vec{X}$ *and* $B \ltimes \vec{X}$.

*Proof.* The part $B \mathfrak{E}\vdash \vec{X}$ is an immediate consequence of the definition of $\vec{E} -:_{B;\vec{\rho}} \vec{X}$.

We proof the statement $B \ltimes \vec{X}$ by induction on the derivation of $\vec{E} -:_{B;\vec{\rho}} \vec{X}$ and a case analysis on the last rule applied in the derivation:

(i) Case CX-EMPTY: In this case, $\vec{X} = \langle\rangle$ and the result is immediate by SX-EMPTY.

(ii) Case CX-EXTS: The conclusions of the subderivations provide these statements:

(1) $$E : (E_{intf}, \vec{X}_1) \in \vec{\rho}$$

(2) $$\vec{E} -:_{B;\vec{\rho}} \vec{X}_2$$

(3) $$B \mathfrak{E}\vdash (\vec{X}_1 \twoheadrightarrow \vec{X}_2)$$

With Lemma 4.2(i), (3) implies

(4) $$\mathfrak{E}\vdash B \Cup (\vec{X}_1 \twoheadrightarrow \vec{X}_2)\,.$$

We use the induction hypothesis and obtain

(5) $$B \mathfrak{E}\vdash \vec{X}_2 \quad \text{and} \quad B \ltimes \vec{X}_2$$

from the requirement of the theorem and (2). By the definition of soundness of module repositories and (1) we have $B \ltimes \vec{X}_1$ and $B \mathfrak{E}\vdash \vec{X}_1$ such that Lemma 4.2(iii) provides $B \ltimes (\vec{X}_1 \twoheadrightarrow \vec{X}_2)$ with (4) and (5). □

## C.9 THEOREM 4.4

THEOREM 4.4 Preservation of module desugaring
*Let* $B = (\vec{A}, \vec{F}, \vec{I}, M)$ *be a well-formed base language definition.*
*If* $B \ltimes \vec{\rho}$, $\vec{\rho} \Vdash_B m : (E_{mid} : (E_{intf}, \langle X, \vec{X}\rangle), \nabla)$, *and* $\nabla \Updownarrow_{B;\vec{X}} \nabla'$ *then* $\langle\rangle \vdash_{\vec{I}} \nabla'$.

*Proof.* By inversion of A-MODULE we have

(1) $$\vec{E}_{imp} -:_{(\vec{A},\vec{F},\vec{I},M);\vec{\rho}} \vec{X}$$

(2) $$(\vec{A}_x, \vec{F}_x, \vec{I}_x, M_x) = (\vec{A}, \vec{F}, \vec{I}, M) \Cup \vec{X}$$

(3) $$\langle\rangle \vdash_{\vec{I}_x} \nabla\,.$$

Applying Lemma 4.3 to the requirement of the theorem and (1) gives $B \mathfrak{E}\vdash \vec{X}$ and $B \ltimes \vec{X}$. The required statement $\langle\rangle \vdash_{\vec{I}} \nabla'$ follows from Lemma 4.1(i). □

## C.10 THEOREM 4.5

THEOREM 4.5 Progress of module desugaring
*Let $B = (\vec{A}, \vec{F}, \vec{I}, M)$ be a well-formed base language definition.*
*If $B \ltimes \vec{\rho}$, $\vec{\rho} \Vdash_B m : (E_{mid} : (E_{intf}, \langle X, \vec{X} \rangle), \nabla)$, and the universal desugarings of each $X \in \vec{X}$ form a terminating rewrite system then either $\nabla \Updownarrow_{B;\vec{X}} \lightning$ or there exists a $\nabla'$ such that $\nabla \Updownarrow_{B;\vec{X}} \nabla'$.*

*Proof.* By inversion of A-MODULE we have

$$\vec{E}_{imp} -:_{(\vec{A},\vec{F},\vec{I},M);\vec{\rho}} \vec{X} \tag{1}$$

$$(\vec{A}_x, \vec{F}_x, \vec{I}_x, M_x) = (\vec{A}, \vec{F}, \vec{I}, M) \Cup \vec{X} \tag{2}$$

$$\langle\rangle \vdash_{\vec{I}_x} \nabla\,. \tag{3}$$

Applying Lemma 4.3 to the requirement of the theorem and (1) gives $B \mathrel{\&\!\!-} \vec{X}$ and $B \ltimes \vec{X}$. The conclusion of the theorem follows by Lemma 4.1(ii). □

## C.11 THEOREM 4.6

THEOREM 4.6 Soundness of module analysis
*Let $B$ be a well-formed base language definition.*
*If $B \ltimes \vec{\rho}$ and $\vec{\rho} \Vdash_B m : (E_{mid} : \Sigma, \nabla)$ then $B \ltimes \Sigma$.*

*Proof.* By inversion of A-MODULE we have the statements

$$\Sigma = (E_{intf}, \langle X, \vec{X}_{imp} \rangle) \tag{1}$$

$$\vec{E}_{imp} -:_{B;\vec{\rho}} \vec{X}_{imp} \tag{2}$$

$$B \Cup \vec{X}_{imp} \mathrel{\&\!\!-} X \tag{3}$$

$$B \Cup \vec{X}_{imp} \ltimes X\,. \tag{4}$$

and by Lemma 4.3 we obtain

$$B \mathrel{\&\!\!-} \vec{X}_{imp} \tag{5}$$

$$B \ltimes \vec{X}_{imp} \tag{6}$$

and Lemma 4.2(i) with (6) implies

$$\mathrel{\&\!\!-} B \Cup \vec{X}_{imp}\,. \tag{7}$$

Using (4), (6) and (7) we obtain $B \ltimes \langle X, \vec{X}_{imp} \rangle$ by SX-EXTS and using (3), (5) and (7) we obtain $B \mathrel{\&\!\!-} \langle X, \vec{X}_{imp} \rangle$ by WX-EXTS. With the last two statements we establish $B \ltimes \Sigma$ with WS-EXPORTS as required. □

## C.12 LEMMA 5.1

LEMMA 5.1 Properties of resolution

(i) *If* $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J_{1..n}\rangle \rhd (\sigma, \langle \nabla_{1..n}\rangle)$ *then* $dom(\sigma) \# vars(\vec{\nabla}_a)$.

(ii) *If* $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J_{1..n}\rangle \rhd (\sigma, \langle \nabla_{1..n}\rangle)$ *then* $[\sigma]\langle J_{1..n}\rangle = concl\langle \nabla_{1..n}\rangle$.

*Proof.*

(i) By induction on a derivation of $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J_{1..n}\rangle \rhd (\sigma, \langle \nabla_{1..n}\rangle)$ and a case analysis on the last rule applied in the derivation.

(a) Case R-EMPTY: Since $\sigma = \emptyset$ and $dom(\sigma) = \emptyset$ the result is immediate.

(b) Case R-ASSUMPTION: By the rule's conclusion we have the identities

$$\langle \nabla_{a1}, ..., \nabla_{ak}, ..., \nabla_{am}\rangle = \vec{\nabla}_a \tag{1}$$
$$\langle J, \vec{J}\rangle = \langle J_1, J_{2..n}\rangle \tag{2}$$
$$\sigma_1 \circ \sigma_0 = \sigma \tag{3}$$
$$\langle \nabla_{ak}, \vec{\nabla}\rangle = \langle \nabla_1, \nabla_{2..n}\rangle\,. \tag{4}$$

The conclusions of the subderivations yield the statements

$$\tilde{V} \# vars\langle \nabla_{a1..am}\rangle \tag{5}$$
$$[\sigma_0](J) = concl(\nabla_{ak}) \tag{6}$$
$$dom(\sigma_0) \subseteq \tilde{V} \tag{7}$$
$$\langle \nabla_{a1..am}\rangle \vdash_{\vec{I}}^{\tilde{V}} [\sigma_0](\vec{J}) \rhd (\sigma_1, \vec{\nabla})\,. \tag{8}$$

$dom(\sigma_0) \# vars(\vec{\nabla}_a)$ is a direct consequence of (1), (5) and (7). By the induction hypothesis and the conclusion of subderivation (8) we obtain $dom(\sigma_1) \# vars(\vec{\nabla}_a)$. Since $dom(\sigma_1 \circ \sigma_0) = dom(\sigma_1) \cup dom(\sigma)$ we conclude $dom(\sigma_1 \circ \sigma) \# vars(\vec{\nabla}_a)$ as required.

(c) Case R-RULE: By the rule's conclusion we have the identities

$$\langle J, \vec{J}\rangle = \langle J_1, J_{2..n}\rangle \tag{9}$$
$$\sigma_1 \circ \sigma_0 = \sigma \tag{10}$$
$$\langle\langle \nabla_{01..0m}\rangle \Rightarrow^N [\sigma_1 \circ \sigma_0](J), \vec{\nabla}\rangle = \langle \nabla_1, \nabla_{2..n}\rangle\,. \tag{11}$$

The conclusions of the subderivations yield the statements

$$\tilde{V} \mathrel{\#} vars(\vec{\nabla}_a) \tag{12}$$

$$(\langle J_{01..0m}\rangle \twoheadrightarrow^N J_0) \mathrel{\tilde{\in}}_{\tilde{V} \cup vars(\vec{\nabla}_a) \cup vars\langle J, \vec{J}\rangle} \vec{I} \tag{13}$$

$$[\sigma_0](J) = [\sigma_0](J_0) \tag{14}$$

$$dom(\sigma_0) \subseteq \tilde{V} \cup vars(J_0) \tag{15}$$

$$\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V} \cup vars\langle J_{01..0m}, J_0\rangle} [\sigma_0]\langle J_{01..0m}, \vec{J}\rangle \rhd (\sigma_1, \langle \nabla_{01..0m}, \vec{\nabla}\rangle)\,. \tag{16}$$

By the freshness premise (13) we have $vars(J_0) \mathrel{\#} vars(\vec{\nabla}_a)$. We also have $dom(\sigma_0) \subseteq \tilde{V} \cup vars(J_0)$ by (15) and since $\tilde{V} \mathrel{\#} vars(\vec{\nabla}_a)$ by (12) we obtain $dom(\sigma_0) \mathrel{\#} vars(\vec{\nabla}_a)$. Applying the induction hypothesis to (16) provides us with $dom(\sigma_1) \mathrel{\#} vars(\vec{\nabla}_a)$ and we conclude $dom(\sigma_1 \circ \sigma_0) \mathrel{\#} vars(\vec{\nabla}_a)$ by the same argument as in the previous case.

(ii) By induction on a derivation of $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J_{1..n}\rangle \rhd (\sigma, \langle \nabla_{1..n}\rangle)$ and a case analysis on the last rule applied in the derivation.

(a) Case R-Empty: Vacuously true since $n = 0$.

(b) Case R-Assumption: By the rule's conclusion we have the identities

$$\langle \nabla_{a1}, ..., \nabla_{ak}, ..., \nabla_{am}\rangle = \vec{\nabla}_a \tag{17}$$

$$\langle J, \vec{J}\rangle = \langle J_1, J_{2..n}\rangle \tag{18}$$

$$\sigma_1 \circ \sigma_0 = \sigma \tag{19}$$

$$\langle \nabla_{ak}, \vec{\nabla}\rangle = \langle \nabla_1, \nabla_{2..n}\rangle\,. \tag{20}$$

The conclusions of the subderivations yield the statements

$$\tilde{V} \mathrel{\#} vars\langle \nabla_{a1..am}\rangle \tag{21}$$

$$[\sigma_0](J) = concl(\nabla_{ak}) \tag{22}$$

$$dom(\sigma_0) \subseteq \tilde{V} \tag{23}$$

$$\langle \nabla_{a1..am}\rangle \vdash_{\vec{I}}^{\tilde{V}} [\sigma_0](\vec{J}) \rhd (\sigma_1, \vec{\nabla})\,. \tag{24}$$

Applying the induction hypothesis to the (24) and using the equations from the conclusion we obtain $[\sigma_1 \circ \sigma_0]\langle J_{2..n}\rangle = \langle \nabla_{2..n}\rangle$. By the first part of the lemma we have $dom(\sigma_1) \mathrel{\#} vars(\vec{\nabla}_a)$, that is, $[\sigma_1](concl(\nabla_{ak})) = concl(\nabla_{ak})$. We insert (22) into this equation and obtain $[\sigma_1 \circ \sigma_0](J) = concl(\nabla_{ak})$ as required.

(c) Case R-Rule: By the rule's conclusion we have the identities

$$\langle J, \vec{J} \rangle = \langle J_1, J_{2..n} \rangle \tag{25}$$

$$\sigma_1 \circ \sigma_0 = \sigma \tag{26}$$

$$\langle \langle \nabla_{01..0m} \rangle \Rightarrow^N [\sigma_1 \circ \sigma_0](J), \vec{\nabla} \rangle = \langle \nabla_1, \nabla_{2..n} \rangle \,. \tag{27}$$

The conclusions of the subderivations yield the statements

$$\tilde{V} \not\frown vars(\vec{\nabla}_a) \tag{28}$$

$$(\langle J_{01..0m} \rangle \rightarrowtriangle^N J_0) \;\tilde{\in}_{\tilde{V} \cup vars(\vec{\nabla}_a) \cup vars\langle J, \vec{J} \rangle} \vec{I} \tag{29}$$

$$[\sigma_0](J) = [\sigma_0](J_0) \tag{30}$$

$$dom(\sigma_0) \subseteq \tilde{V} \cup vars(J_0) \tag{31}$$

$$\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V} \cup vars\langle J_{01..0m}, J_0 \rangle} [\sigma_0]\langle J_{01..0m}, \vec{J} \rangle \rhd (\sigma_1, \langle \nabla_{01..0m}, \vec{\nabla} \rangle) \,. \tag{32}$$

Applying the induction hypothesis to the conclusion of subderivation (32) provides us with

$$[\sigma_1 \circ \sigma_0]\langle J_{01..0m}, J_{2..n} \rangle = \langle concl\langle \nabla_{01..0m} \rangle, concl\langle \nabla_{2..n} \rangle \rangle \,, \tag{33}$$

hence $[\sigma_1 \circ \sigma_0]\langle J_{2..n} \rangle = concl\langle \nabla_{2..n} \rangle$. Equation (27) gives the missing bit since $concl(\langle \nabla_{01..0m} \rangle \Rightarrow^N [\sigma_1 \circ \sigma_0](J_1)) = [\sigma_1 \circ \sigma_0](J_1)$. □

## C.13 theorem 5.2

Theorem 5.2 Soundness of resolution

*If* $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J_{1..n} \rangle \rhd (\sigma, \langle \nabla_{1..n} \rangle)$ *then for each* $i \in 1..n : \vec{\nabla}_a \vdash_{\vec{I}} \nabla_i$.

*Proof.* By induction on a derivation of $\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V}} \langle J_{1..n} \rangle \rhd (\sigma, \langle \nabla_{1..n} \rangle)$ and a case analysis on the last rule applied in the derivation.

(i) Case R-Empty: Vacuously true since $n = 0$.

(ii) Case R-Assumption: By the rule's conclusion we have the identities

$$\langle \nabla_{a1}, ..., \nabla_{ak}, ..., \nabla_{am} \rangle = \vec{\nabla}_a \tag{1}$$

$$\langle J, \vec{J} \rangle = \langle J_1, J_{2..n} \rangle \tag{2}$$

$$\sigma_1 \circ \sigma_0 = \sigma \tag{3}$$

$$\langle \nabla_{ak}, \vec{\nabla} \rangle = \langle \nabla_1, \nabla_{2..n} \rangle \,. \tag{4}$$

The conclusions of the subderivations yield the statements

$$\tilde{V} \not\frown vars\langle \nabla_{a1..am} \rangle \tag{5}$$

$$[\sigma_0](J) = concl(\nabla_{ak}) \tag{6}$$

$$dom(\sigma_0) \subseteq \tilde{V} \tag{7}$$

$$\langle \nabla_{a1..am} \rangle \vdash_{\vec{I}}^{\tilde{V}} [\sigma_0](\vec{J}) \rhd (\sigma_1, \vec{\nabla}) \,. \tag{8}$$

Applying the induction hypothesis to (8), we have for each $i \in 2..n$ : $\vec{\nabla}_a \vdash_{\vec{I}} \nabla_i$. Using (4) we obtain $\vec{\nabla}_a \vdash_{\vec{I}} \nabla_1$ by V-Assumption as required.

(iii) Case R-Rule: By the rule's conclusion we have the identities

$$\langle J, \vec{J} \rangle = \langle J_1, J_{2..n} \rangle \tag{9}$$

$$\sigma_1 \circ \sigma_0 = \sigma \tag{10}$$

$$\langle \langle \nabla_{01..0m} \rangle \Rightarrow^N [\sigma_1 \circ \sigma_0](J), \vec{\nabla} \rangle = \langle \nabla_1, \nabla_{2..n} \rangle \,. \tag{11}$$

The conclusions of the subderivations yield the statements

$$\tilde{V} \mathrel{\#} vars(\vec{\nabla}_a) \tag{12}$$

$$(\langle J_{01..0m} \rangle \rightarrowtriangle^N J_0) \mathrel{\tilde{\in}}_{\tilde{V} \cup vars(\vec{\nabla}_a) \cup vars\langle J, \vec{J} \rangle} \vec{I} \tag{13}$$

$$[\sigma_0](J) = [\sigma_0](J_0) \tag{14}$$

$$dom(\sigma_0) \subseteq \tilde{V} \cup vars(J_0) \tag{15}$$

$$\vec{\nabla}_a \vdash_{\vec{I}}^{\tilde{V} \cup vars\langle J_{01..0m}, J_0 \rangle} [\sigma_0]\langle J_{01..0m}, \vec{J} \rangle \rhd (\sigma_1, \langle \nabla_{01..0m}, \vec{\nabla} \rangle) \,. \tag{16}$$

Applying the induction hypothesis to (16) we obtain

$$\text{for each } i \in 1..m : \vec{\nabla}_a \vdash_{\vec{I}} \nabla_{0i} \tag{17}$$

and

$$\text{for each } i \in 2..n : \vec{\nabla}_a \vdash_{\vec{I}} \nabla_i \tag{18}$$

such that $\vec{\nabla}_a \vdash_{\vec{I}} \nabla_1$ remains to be shown.

Inversion of F-Rule and (13) entail that there exists a substitution $\sigma_f$ such that $[\sigma_f](\langle J^*_{01..0n} \rangle \rightarrowtriangle^N J^*_0) = (\langle J_{01..0n} \rangle \rightarrowtriangle^N J_0)$ and $(\langle J^*_{01..0n} \rangle \rightarrowtriangle^N J^*_0) \in \vec{I}$.

By Theorem 5.1(ii) applied to (16) we get

$$[\sigma_1 \circ \sigma_0]\langle J_{01..0m} \rangle = concl\langle \nabla_{01..0m} \rangle = [\sigma_1 \circ \sigma_0 \circ \sigma_f]\langle J^*_{01..0m} \rangle \,. \tag{19}$$

By (14) we have $[\sigma_1 \circ \sigma_0](J_1) = [\sigma_1 \circ \sigma_0](J_0) = [\sigma_1 \circ \sigma_0 \circ \sigma_f](J^*_0)$. Using (19) we obtain $[\sigma_1 \circ \sigma_0 \circ \sigma_f]\langle J^*_{01..0m}, J^*_0 \rangle = \langle concl\langle \nabla_{01..0m} \rangle, [\sigma_1 \circ \sigma_0](J) \rangle$ such that we derive $\vec{\nabla}_a \vdash_{\vec{I}} \nabla_1$ by application of V-Rule to the previous identity and (17). □

# BIBLIOGRAPHY


[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[BCD+88] P. Borras, Dominique Clement, Thierry Despeyroux, Janet M. Incerpi, Gilles Kahn, Bernard Lang, and V. Pascual. Centaur: The system. In *Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 3, pages 14–24. ACM, 1988.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 2007.

[BP01] Jonathan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 31–42. ACM, 2001.

[Bra04] Gilad Bracha. Pluggable type systems. In *OOPSLA04 Workshop on Revival of Dynamic Languages*, 2004.

[Bru02] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[BS86] Rolf Bahlke and Gregor Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):547–576, October 1986.

[BS02] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '02, pages 31–40. ACM, 2002.

[CBC+08] Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie 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.

[CCJ69] Carlos Christensen and Shaw Christopher J., editors. *Proceedings of the Extensible Languages Symposium*, SIGPLAN Notices. ACM, August 1969.

[Chl10] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 122–133. ACM, 2010.

[Chl13] Adam Chlipala. The bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 391–402. ACM, 2013.

[CMA94] Luca Cardelli, Florian Matthes, and Martin Abadi. Extensible syntax with lexical scoping. Technical Report SRC-RR-121, Systems Research Center, 1994.

[CU08] James Cheney and Christian Urban. Nominal logic programming. *ACM Transactions on Programming Languages and Systems*, 30(5):26:1–26:47, September 2008.

[Des84] Thierry Despeyroux. Executable specification of static semantics. In Gilles Kahn, David B. MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 215–233. Springer, 1984.

[DHB92] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1992.

[dR03] Daniel de Rauglaudre. Camlp4 reference manual, 2003.

[EGR12] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann 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.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification I*. Springer, 1985.

[EMC+01] Hartmut Ehrig, Bernd Mahr, Felix Cornelius, Martin Große-Rohde, and Philip Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer, 2001.

[ER13] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 3–12. ACM, 2013.

[Erd13] Sebastian Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universität Marburg, 2013.

[ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus 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.

[ERKO13] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Layout-sensitive generalized parsing. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 244–263. Springer, 2013.

[ERRO12] Sebastian Erdweg, Felix Rieger, Tillmann Rendel, and Klaus Ostermann. Layout-sensitive language extensibility with sugarhaskell. In *Proceedings of the 2012 symposium on Haskell symposium*, Haskell '12, pages 149–160. ACM, 2012.

[FFF09] Matthias Felleisen, Robert B. Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[FS06] David Fisher and Olin Shivers. Static analysis for syntax objects. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 111–121. ACM, 2006.

[FS08] David Fisher and Olin Shivers. Building language towers with Ziggurat. *Journal of Functional Programming*, 18(5–6):707–780, September 2008.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[GJS+14] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java® Language Specification – Java SE 8 Edition*. Oracle America, Inc., 2014.

[GM94] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.

[GN08] George Giorgidze and Henrik 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.

[Gri88] Timothy G. Griffin. Notational definition – a formal account. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, LICS' 88, pages 372–383. IEEE, 1988.

[GST01] Steve Ganz, Amr Sabry, and Walid 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*, pages 74–85. ACM, 2001.

[Har13] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2013.

[Hei12] Christian Heinlein. MOSTflexiPL: Modular, statically typed, flexibly extensible programming language. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 159–178. ACM, 2012.

[Hei14] Christian Heinlein. Fortgeschrittene Syntaxerweiterungen durch virtuelle Operatoren in MOSTflexiPL. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2014, 25.-26. Februar 2014 in Kiel, Deutschland*, pages 193–212, 2014.

[Her10] David Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, Massachusetts, 2010.

[HJS+09] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, Christian Wende, and Marcel Böhme. Generating safe template languages. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09, pages 99–108. ACM, 2009.

[HR76] William S. Hatcher and Teodor Rus. Context-free algebras. *Journal of Cybernetics*, 6:65–77, 1976.

[HS08] Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with MorphJ. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 79–89. ACM, 2008.

[HW08] David Herman and Mitchell Wand. A theory of hygienic macros. In Sophia Drossopoulou, editor, *Programming Languages and*

*Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2008.

[HZS05] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically safe program generation with SafeGen. In Robert Glück and Michael Lowry, editors, *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 309–326. Springer, 2005.

[HZS07] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 399–424. Springer, 2007.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, May 2001.

[Kah87] Gilles Kahn. Natural semantics. In Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceeding 4th Annual Symposium on Theoretical Aspects of Computer Sciences (STACS 87)*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161. ACM, 1986.

[Kle52] Stephen C. Kleene. *Introduction to metamathematics*. Van Nostrond, 1952.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January 1964.

[Lan66] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[LDF+14] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.02*. INRIA, 2014.

[LE13] Florian Lorenzen and Sebastian 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.

[Lea66] B. M. Leavenworth. Syntax macros and extended translation. *Commun. ACM*, 9(11):790–793, November 1966.

[LFGC07] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 425–434. ACM, 2007.

[LGHM12] Byeongcheol Lee, Robert Grimm, Martin Hirzel, and Kathryn S. McKinley. Marco: Safe, expressive macros for any language. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 589–613. Springer, 2012.

[Lin14] Fabian Linges. Eine funktionale Sprache mit erweiterbaren Bindungsstrukturen und syntaktischen Formen. Master's thesis, TU Berlin, 2014.

[Lor12] Florian Lorenzen. A foundation for programmable binders in statically typed functional languages. In *Pre-proceedings of 2012 Symposium on Trends in Functional Programming*, 2012.

[Mai07] Geoffrey 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.

[Mai12] Geoffrey 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. Submitted to ICFP'2012.

[Mar10] Haskell 2010 – language report, June 2010.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communication of the ACM*, 3(4):184–195, April 1960.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MME+10] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. JavaCOP: Declarative pluggable types for Java. *ACM Transactions on Programming Languages and Systems*, 32(2):4:1–4:37, January 2010.

[NM95] Ulf Nilsson and Jan Maluszinski. *Logic, Programming and Prolog*. John Wiley & Sons Ltd, 2nd edition, 1995.

[NS06] Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 320–333. ACM, 2006.

[OKN+14] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 105–130. Springer, 2014.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, 2008.

[PAC+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212. ACM, 2008.

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208. ACM, 1988.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pit03] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, November 2003.

[PKW14] Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object-oriented Programming, Systems, Languages, And Applications*, OOPSLA '14, pages 525–542. ACM, 2014.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer, 1974.

[Rey98] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[RRD10] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *Proceedings of the 5th ACM SIGPLAN Workshop on*

*Types in Language Design and Implementation*, TLDI '10, pages 89–102. ACM, 2010.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-based Editors*. Springer, 1989.

[Rus72] Teodor Rus. $\sigma s$-algebra of a formal language. *Bulletin Mathématique de la Société de Science, Bucharest*, 15(63–2):227–235, 1972.

[SG93] Guy L. Steele, Jr. and Richard P. Gabriel. The evolution of Lisp. *SIGPLAN Notices*, 28(3):231–270, March 1993.

[SNO+10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, January 2010.

[SPJ02] Tim Sheard and Simon L. Peyton Jones. Template metaprogramming for Haskell. In *Proceedings of the ACM Workshop on Haskell*, Haskell '02, pages 1–16. ACM, 2002.

[Sta75] Thomas A. Standish. Extensibility in programming language design. *SIGPLAN Notices*, 10(7):18–21, July 1975.

[Ste90] Guy L. Steele, Jr. *Common Lisp – the Language*. Digital Press, 1990.

[Ste99] Guy L. Steele, Jr. Growing a language. *Higher Order Symbolic Computation*, 12(3):221–236, October 1999.

[SW14] Paul Stansifer and Mitchell 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.

[THF10] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128. ACM, 2010.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '97, pages 203–217. ACM, 1997.

[UPG04] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1–3):473–497, September 2004.

[VBT98] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew 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.

[Vis97] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14. ACM, 1992.

[Wan86] Mitchell Wand. Finding the source of type errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 38–43. ACM, 1986.

[Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 1965–2013. Elsevier, 2001.