INBOUND: Simple yet powerful Specification of Syntax with Binders

Steven Keuchel
Ghent University
steven.keuchel@ugent.be

Tom Schrijvers
KU Leuven
tom.schrijvers@cs.kuleuven.be

Abstract

Nearly all meta-programming tools need to deal with binders in abstract syntax trees. Unfortunately, ubiquitous boilerplate functions for computing free variables and variable substitution are remarkably intricate and hard to get right. Moreover, the complexity quickly increases with larger languages and non-trivial binding forms. On top of that, the underlying scoping rules are only implicitly encoded in these function definitions. Because their logic has to be repeated across functions, this leaves ample space for inconsistencies. In short, programming with binders is a pain!

Our new specification language INBOUND brings much needed relief: it allows programmers to explicitly state the scoping rules of syntax with binders in a concise and intuitive manner. From such a specification the INBOUND compiler automatically generates the boilerplate functions, and takes care of all the intricacies as well as mutual consistency.

We illustrate INBOUND with a number of examples and two larger case studies, including a simple type checker for Haskell. These show that INBOUND easily scales to larger languages and complex binding situations.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory

Keywords variable binders, code generation, attribute grammars, datatype-generic programming

1. Introduction

It is well-recognized in the programming language community that handling binders is intricate and tedious. This recognition has led to a steady stream of approaches and proposals that tackle the problem from different angles.

For instance, much effort has gone into developing different variable representations like de Bruijn indices\cite{6}, nested datatypes\cite{4}, the locally nameless representation\cite{3} and various forms of higher-order syntax\cite{11,13,14}. Many of these results are targeted towards reasoning and mechanization of meta-theory in proof assistants.

A second important line of work aims to provide binding-safe programming by means of native support for binders (e.g., FreshML, Caml\cite{10}, Beluga\cite{14} and Romeo\cite{20}) or through libraries (e.g., FreshLook\cite{18} and NaPa\cite{17}). Yet another approach is datatype generic programming support for the boilerplate operations that comes in the form of libraries like UNBOUND\cite{22} and Nameplate\cite{5}.

While most of these approaches are highly sophisticated, they are also far removed from the current software development practice which typically uses the most naive representation for abstract syntax with identifiers for variables. This approach is inspired by many pragmatic reasons. Firstly, the identifier approach corresponds closely to the concrete syntax of the language being processed, which makes it easy to relate both, for instance, during debugging. Secondly, the approach is compatible with nearly any tool implementation language. Thirdly, the (deceitful) simplicity of the approach makes it highly accessible.

This paper addresses this stand-off. We propose a down-to-earth approach for dealing with binders that is both easy to use and compatible with current practice. Our approach consists of a specification language, INBOUND.

Our specific contributions are:

1. We define the INBOUND specification language and illustrate its use with a number of examples (Section\cite{3}).

2. We provide a semantics for INBOUND in terms of an elaboration into attribute grammars (Section\cite{3}). In this elaboration every context attribute gives rise to a number of derived attributes that implement the boilerplate functions.

3. The generic derivation of these attributes is non-trivial: free variable computation invert the data flow, while renaming and substitution require freshening for capture avoidance.

4. We report on our implementation of INBOUND in terms of the Utrecht University Attribute Grammar Compiler, and discuss a number of interesting semantic extensions and practical features (Section\cite{5}).

5. We demonstrate the usefulness of INBOUND in two case studies (Section\cite{6}). One of these integrates INBOUND in an existing project, a small type checker for Haskell written in Haskell.

There is quite a bit of related work, which we discuss in Section\cite{7}.

2. Overview

This section gives an overview of the variable binding boilerplate that arises when implementing meta-programs like type-checkers and compilers.

2.1 Background: Conventional Approach

As a starting point for illustrating the conventional approach we take the following textbook definition for the simply typed $\lambda$-
is not relevant for this paper, and we go with a pragmatic string-
extensive and the discussion is still ongoing. The particular choice variables. The literature on different variable representations is
of algebraic data types:

In the Haskell implementation of a meta-program,

Abstract Syntax In the Haskell implementation of a meta-program, we usually capture such an abstract syntax definition in a number of algebraic data types:

```
data Type = BaseT
    | TArr Type Type

data Term = Lam TermVar Type Term
    | Var TermVar
    | App Term Term
```

Additionally, we need to choose a particular representation for term variables. The literature on different variable representations is extensive and the discussion is still ongoing. The particular choice is not relevant for this paper, and we go with a pragmatic string-based representation.

```
type TermVar = String
```

Note that the scoping rules of the language are absent in this data type declaration. Of course the scoping rules stated above still apply, but it is the programmer’s responsibility to make sure they are obeyed by the meta-program.

Binder Boilerplate Now comes the tedious boilerplate: There are two ubiquitous binder-related operations that are widely used in meta-programming tools: 1) computing the free variables that are not bound in a term, and 2) substituting all occurrences of a variable for a term.

For our example the free variables definitions is as follows:

```
freeVars :: Term -> Set TermVar
freeVars (Lam x t e) = delete x (freeVars e)
freeVars (Var x) = singleton x
freeVars (App e1 e2) = freeVars e1 `union` freeVars e2
```

The substitution function is substantially more verbose:

```
subst :: TermVar -> Term -> Term -> Term
subst x e (Var y)
  | x ≡ y = e
  | otherwise = Var y
subst x e1 (Lam y t e2)
  | x ≡ y = Lam y t e2
  | y `member` freeVars e1 = let y' = fresh (freeVars e1 `union` freeVars e2)
  | otherwise = Lam y (subst x e1)
subst x e (App e1 e2)
  = App (subst x e1) (subst x e2)
```

Moreover, to avoid variable capture, we need two additional definitions. The function `fresh` picks a variable name that is distinct from any of the names in the given set.

```
fresh :: Set TermVar -> TermVar
fresh s = head [v | v ← vs, v ∉ s]
where
  vs = map (λn → ’v’ : show n) [0 ..]
```

The function `rename` is a more specialized variant of `subst` that replaces one variable name by another.

```
rename :: TermVar -> TermVar -> Term -> Term
rename x x' (Var y)
  | x ≡ y = Var x'
  | otherwise = Var y
rename x x' (Lam y t e2)
  | x ≡ y = Lam y t e2
  | y ≡ x' = let y' = fresh (insert x' (freeVars e2))
  | otherwise = Lam y (rename x x' e2)
rename x x' (App e1 e2)
  = App (rename x x' e1) (rename x x' e2)
```

In meta-theoretic expositions these definitions are often elided because they are uninteresting. The understanding is that the informal exposition contains enough clues for any reader to easily derive them, if so desired. Unfortunately, the programming languages in which meta-programming tools are written are bad at reading between the lines and do require all operations to be written out explicitly. This brings us to the aim of this paper.

2.2 Challenge: Minimal Clues for Binder Boilerplate

The scoping rules are implicitly reflected in the above functions. Indeed, the functionality depends only on the structure of the language and its scoping rules. As a consequence, the process is repetitive; the same pattern is applied time and again. This makes the implementation of boilerplate functions time-consuming and error-prone.

Genericity In fact, this similarity challenges us to identify and formalize the generic recipe at the heart of binder boilerplate, and to isolate the minimal amount of information necessary to instantiate the generic recipe for a particular language. Then the programmer need only to supply the latter to get all the boilerplate automatically.

Expressivity Of course many such generic recipes are possible, depending on the class of languages that are covered by it. Our aim here is to be expressive and useful in practice. Hence our solution should go beyond small toy examples and cover a large set of realistic binding forms and situations like, for instance:

- multiple kinds of variables with interdependencies, e.g., type and term variables in System F:

  `λa.λ(x : a).e`

- structured binders, e.g., nested patterns in Haskell:

  `case e1 of
    (x, (y, z)) → e2`

- complex scoping, e.g., recursive scopes in OCaml:

  `let rec f x = g (x - 1)
  g x = if x > 0 then f x else 1
  in f 5 + g 7`

2.3 Solution: INBOUND

Our solution is INBOUND, a domain-specific language for specifying abstract syntax with binders. In exchange for a small amount of binding information, INBOUND automatically derives the tedious boilerplate functions.

\[ \tau ::= A | \tau_1 \rightarrow \tau_2 \]

\[ e ::= \lambda x : \tau.e | x | e_1 e_2 \]
Below is the INBOUND specification for our running example.

```plaintext
namespaces TermVar \triangleright Term

sort Type  ::=
  Unit
  | TArr (T₁, T₂ : Type)
sort Term  ::=
  inh ctxx : [TermVar]
  | Var (x@ctxx)
  | Lam (x : TermVar) (T : Type) (t : Term) t.ctxx = lhs.ctxx, x
  | App (t₁ t₂ : Term) t₁.ctxx = lhs.ctxx
  t₂ ctxx = lhs.ctxx

This specification is very similar to the algebraic data type definitions in Haskell. As before we have declarations of the syntactic sorts Type and Term, each with their data constructors. One important difference is that no concrete representation is chosen for term variables. Instead, TermVar is declared as a namespace. This tells INBOUND that TermVar is a kind of variable that can be bound and referenced. The annotation \triangleright Term means that a TermVar is a Term variable, i.e., that it can be substituted by a Term. (Note that the choice of representation is left up to INBOUND.)

Term variables are used inside terms. This is declared in the form of the context attribute ctxx associated with the Term sort. The declaration means that there is a context of term variables at every Term node in the abstract syntax tree. In this case the context is inherited, i.e., the contexts of the subterms are derived from the context of the parent term. Each constructor specifies what context each of its subterms inherits: The subterms of the App constructor inherit the context lhs.ctxx of their parent. The Lam constructor binds a new term variable x which is in scope at its subterm t. Hence, t inherits its parents context extended with x.

Finally, the Var constructor references a term variable X from the context ctxx. This information is declared in the form of field declaration X@ctxx.

In summary, we can already see on this example that INBOUND requires very little binding information on top of the basic abstract syntax definition. Nevertheless this is enough to generate all the tedious boilerplate. The rest of this paper explains how it manages syntax definition. Nevertheless this is enough to generate all the tedious boilerplate. The rest of this paper explains how it manages syntax definition.

3. Grammars and Binding Specifications

This section introduces INBOUND, our domain-specific language for specifying the abstract syntax of programming languages and their associated variable binder information.

3.1 Formal INBOUND Syntax

Figure 1 shows the grammar of INBOUND. The toplevel syntactic object is a INBOUND specification spec. Such a specification consists of a sequence of namespace declarations α₁\triangleright S₁, ..., αₘ\triangleright Sₘ that name the different kinds of variables and their associated sorts, followed by declarations sortdecl of the different syntax sorts S.

A sort declaration sortdecl consists of context attribute declarations and constructor declarations. There are two kinds of context attributes:

1. inherited contexts inhdecl pass context information down from a term to its immediate subterms, and
2. synthesized contexts syndecl assemble a context for the term from the contexts of its subterms.

<table>
<thead>
<tr>
<th>Labels</th>
<th>Sort label</th>
</tr>
</thead>
<tbody>
<tr>
<td>S, T, U</td>
<td>Constructor label</td>
</tr>
<tr>
<td>K</td>
<td>Field label</td>
</tr>
<tr>
<td>X, Y, Z</td>
<td>Meta-variable</td>
</tr>
<tr>
<td>A, B, C</td>
<td>Attribute label</td>
</tr>
<tr>
<td>α, β, γ</td>
<td>Namespace label</td>
</tr>
</tbody>
</table>

Declarations and definitions

```
<table>
<thead>
<tr>
<th>spec</th>
<th>::= namespace α \triangleright S sortdecl</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>sortdecl</td>
<td>::= sort S syndecl inhdecl cdelem</td>
<td>Sort</td>
</tr>
<tr>
<td>syndecl</td>
<td>::= syn A :: [α]</td>
<td>Inherited attr</td>
</tr>
<tr>
<td>inhdecl</td>
<td>::= inh A :: [α]</td>
<td>Synthesized attr</td>
</tr>
<tr>
<td>cdelem</td>
<td>::= K efield sfield attrdef</td>
<td>Term constructor</td>
</tr>
<tr>
<td>efield</td>
<td>::= X : α</td>
<td>Variable binder</td>
</tr>
<tr>
<td>sfield</td>
<td>::= F : S</td>
<td>Subterm</td>
</tr>
<tr>
<td>attrdef</td>
<td>::= N.A = e</td>
<td>Attribute def.</td>
</tr>
<tr>
<td>N</td>
<td>::= lhs</td>
<td>F</td>
</tr>
</tbody>
</table>

Context expressions

expr, e, a, b, c ::= N.A Context reference
| | e, X Context extension

Figure 1. Grammars with binding specifications

Values of a particular sort are built from constructors K. There are two kinds of constructors: term constructors and variable constructors. A variable constructor has exactly one field of the form X@A. This field X references a variable in context A. In contrast, a term constructor has two kinds of fields, declared in the constructor declaration:

- every variable binder X : α introduces a new variable X of namespace α, and
- every subterm F : S denotes a subterm of sort S with associated field label F.

In addition to the different field declarations, a constructor declaration also explains how to compute a number of context attributes: a definition lhs.A = e defines the term’s synthesized attribute A, and the definition F.A = e defines the inherited attribute A of subterm F.

There are three different kinds of context expression: the empty context is [], the context extension (e, X) extends the context e with newly bound variable X and the context reference N.A refers to context attribute A of node N (either the current node lhs or a subterm F).

3.2 Examples

The following examples of rich binder forms illustrate the expressive power of INBOUND.

Multiple contexts Figure 2 shows the INBOUND specification of System F. We start with the declaration of two namespaces TypeVar and TermVar for type and term variables respectively, which is followed by the declarations of System F’s two sorts: Types and Terms.

Both sorts have an inherited context attribute ctxx : [TypeVar] that collects the type variables in scope. These contexts are respectively extended by the constructors TLam for type functions and TAbs for type abstractions that both bind new type variables. The only constructor that references a type variable in ctxx is TVar.
namespaces

sort Type
inh tctx : [TypeVar]
  | TVar (X:@tctx)
  | TLam (X : TypeVar) (T : Type) T.tctx = lhs.tctx, X
  | TArr (T1 T2 : Type) t1.tctx = lhs.tctx t2.tctx = lhs.tctx

sort Term
inh ctxx : [TypeVar]
inh ctx : [TermVar]
  | Var (x@ctxx)
  | Lam (x : TermVar) (t : Term) t.ctx = lhs.ctx, x
  | App (t1 t2 : Term) t1.tctx = lhs.tctx t2.tctx = lhs.tctx
t1.ctx = rhs.ctx t2.ctx = rhs.ctx
  | TAbs (X : TypeVar) (t : Term) T.tctx = lhs.tctx, X t.ctx = rhs.ctx
  | TApp (t : Term) (T : Type) T.tctx = lhs.tctx t.ctx = rhs.ctx

Figure 2. Example specification of System F

Terms have a second inherited context attribute ctx : [TermVar] for term variables in scope. This context is extended by constructor Lam for term abstraction and referenced by the constructor Var.

In all other cases the contexts are simply inherited without modification from parent to children. As this case occurs frequently, INBOUND tries to insert this definition automatically when no explicit definition is provided. For this the parent needs an inherited attribute with the same label and type. However, we allow parent and child to be of different sorts. Hence, we could have elided 11 out of the 14 attribute definitions in this example. We do so in all further examples.

Binding forms Figure 3 shows the INBOUND specification of a simply-typed lambda calculus with products and destructuring let bindings. A destructuring let binding Let p t1 t2 matches term t1 against pattern p; the variables bound by p are in scope in term t2. The complication here is how to add variables bound inside p to the context of t2. This cannot happen in the let-binding itself, because the variables are not listed at this level – they are inside the pattern p. Yet inside p we have the problem that t2 and its context are not available.

Our solution use two context attributes: an inherited attribute setctx that is used to pass the outer context down into the pattern, and a synthesized attributed setctx that is passed up from the pattern in which all variables of the pattern are adjoined to the outer context. In the case of a product pattern, the context is chained from the pattern p1 for the first component to the pattern p2 of the second component.

3.3 Well-Formed INBOUND Specifications

Not all INBOUND specifications make sense. The well-formedness relation ⊨ spec in Figure 4 enforces that attribute definitions are type-correct with respect to sorts and namespaces.

In order to not overburden the definition of this relation and its subsidiaries, we assume that some information is available globally:

- IA: all inherited attributes, and
- SA: all synthesized attributes.

The single rule WFSPEC expresses that a specification is well-formed if each of its sort declarations is well-formed.

Rule WFSDECL in turn states that a sort declaration is well-formed if each of its constructor declaration is well-formed with respect to the declaration's sort.

Rule WFVARDECL states that a variable constructor K (X @ A) is well-formed if the context attribute A is an inherited attribute of the constructor's sort.

Rule WFTERMDECL expresses that term constructors are well-formed if the bodies of its attribute definitions are well-typed. The typing discipline of these bodies is linear with respect to the variables X bound in the constructor; this requirement is motivated below. Linearity means that every variable X must be used exactly once. Hence, we split the set of bound variables into disjoint subsets (, called local environments, one for each body expression.

Rule WFTCDECL requires that the attribute definitions for the constructor declaration are well-formed with respect to the enclosing declaration for sort S and with respect to the local environment Σ that includes the references and fields of the constructor.

The relation Σ ⊩ e : [α] denotes that context expression e is well-typed for namespace α with respect to the local environment Σ and field typing θ. It enforces linearity of the local environment Σ. The relation is defined by four rules: Rule WFNIL states that the empty context [] is well-typed for any namespace α in an empty local environment. Rule WFCONS states that a context extension is well-typed, if the namespace of the new variable X matches
the namespace of the context expression \( e \). Note that the local environment is split to enforce linearity. Finally rule WFNI and rule WFSYN express that the types of attribute references match the types in the attribute environments \( IA \) and \( SA \) respectively.

**Linearity** The linearity requirement is a consequence of our choice for functional purity. We want all boilerplate operations to yield equal results when repeated. This means that whenever we choose a fresh name for a variable \( x \), we do so in a pure way: like in Section 2 we compute a name that is distinct from all names that live in the same contexts as \( x \).

If \( x \) could appear twice in a context, we would have to avoid the circularity pitfall of attempting to choose a name that is distinct from itself. Instead of performing a complicated analysis to detect and avoid this pitfall, we rule it out entirely with a simple linearity requirement. So far this has not turned out to be a costly requirement: none of our examples or case studies suffers from it.

Linearity would not be required if we opted for impure generation of fresh variable names. However, we would then lose the good properties of a pure semantics.

**Circularity** For the final extraction of functions from the specification we also require that the dependency between attributes is non-circular \([9]\). However, for the sake of brevity we omit the details and refer to related work dealing with checking for non-circularity of attribute grammars \([7,9]\).

## 4. Elaboration Semantics

In this section we define the semantics of well-formed INBOUND-specifications by elaboration into a lower-level attribute grammar system. A central part of the elaboration is to turn the binding specifications into attributes that implement the necessary computations for the boilerplate operations.

### 4.1 Target language

Figure 3 shows the target attribute grammar language AG that we use for the elaboration. The language contains declarations for sorts, constructors and attributes.

Notably, AG has no special support for variables and binders. As a consequence there are two major differences with INBOUND:

1. There is no notion of variables in AG, and thus no notion of binding and referencing occurrences of variables in constructors either. Constructors can have two types of fields: subtrees of any sort, and terminals of the type \( A \) explained below.

2. In INBOUND each attribute represents a context. In contrast, attributes in AG can have arbitrary types and purposes.

Our elaboration makes use of this flexibility when elaborating each context into multiple attributes that implement the different syntactic operations.

Because of the wider scope of attributes in AG, its type and expression syntax are much richer. It supports list, set, (finite) map and tuple data types and their corresponding expression forms. There is also a primitive type \( A \) for identifiers, which we leave abstract in this paper.

### 4.2 Variable Elaboration

The INBOUND-specification language itself does not fix a specific variable representation. Yet the choice of representation influences the elaboration. For two reasons we work in this paper with the traditional first-order representation that uses identifiers of type \( A \) for variables: First, the definitions are intuitive and we assume that the reader is comfortable with the handling of variables using this

---

\( \text{WFDECL} \)  
\( \Gamma \vdash \theta \vdash S \text{ cdecl} [\alpha] : [\alpha] \)  
\( \text{WFVARDECL} \)  
\( \Gamma \vdash \theta \vdash S [\alpha] : [\alpha] \)  
\( \text{WFCONS} \)  
\( \theta \vdash \Gamma \vdash \psi \vdash e : [\alpha] \)  
\( \text{WFNYC} \)  
\( \theta \vdash \Gamma \vdash \psi \vdash F.A : [\alpha] \)  
\( \text{WFDECL} \)  
\( \theta \vdash \Gamma \vdash \psi \vdash \theta \vdash S.\text{cdecl} [\alpha] : [\alpha] \)  

---

Figure 4. Well-formed specifications
Declarations and definitions

\[
\begin{align*}
\text{ag}_\text{prog} &::= \text{ag}_\text{sortdecl} \quad \text{Program} \\
\text{ag}_\text{sortdecl} &::= \text{sort } S \text{ ag}_\text{syndecl} \text{ ag}_\text{inhdecl} \text{ ag}_\text{cdecl} \\
\text{ag}_\text{syndecl} &::= \text{syn } A : \text{ag}_\text{type} \quad \text{Synthesized attr.} \\
\text{ag}_\text{inhdecl} &::= \text{inh } A : \text{ag}_\text{type} \quad \text{Inherited attr.} \\
\text{ag}_\text{cdecl} &::= \text{cdecl } F : S \quad \text{Constructor} \\
\text{ag}_\text{field} &::= \text{field } X : \text{Atom} \quad \text{Atom field} \\
\text{ag}_\text{attrdef} &::= N.A = \text{ag}_\text{expr} \quad \text{Attr. def.}
\end{align*}
\]

Types

\[
\begin{align*}
\text{ag}_\text{type} &::= S \quad \text{Sort type} \\
\text{Atom} &\quad \text{Atom type} \\
[\text{ag}_\text{type}] &\quad \text{List type} \\
\text{Set } \text{ag}_\text{type} &\quad \text{Set type} \\
\text{ag}_\text{type}_1 \times \text{ag}_\text{type}_2 &\quad \text{Tuple type} \\
\text{ag}_\text{type}_1 \rightarrow \text{ag}_\text{type}_2 &\quad \text{Map type}
\end{align*}
\]

Expressions

\[
\begin{align*}
\text{ag}_\text{expr}, \text{ae} &::= x \quad \text{Variable} \\
&\mid X \quad \text{Atom field ref.} \\
&\mid N.A \quad \text{Attribute ref.} \\
&\mid \emptyset \quad \text{Empty set} \\
&\mid \{ \text{ae} \} \quad \text{Singleton set} \\
&\mid \text{ae}_1 \cup \text{ae}_2 \quad \text{Set union} \\
&\mid \text{ae}_1 - \text{ae}_2 \quad \text{Set difference} \\
&\mid [] \quad \text{Empty list} \\
&\mid [\text{ae}] \quad \text{Singleton list} \\
&\mid \text{ae}_1 : \text{ae}_2 \quad \text{List extension} \\
&\mid \epsilon \quad \text{Empty map} \\
&\mid \text{ae}_1 \circ (\text{ae}_2 \rightarrow \text{ae}_3) \quad \text{Map extension} \\
&\mid (\text{ae}_1, \text{ae}_2) \quad \text{Tuple} \\
&\mid \text{let} (x_1, x_2) = \text{ae}_1 \text{ in } \text{ae}_2 \quad \text{Tuple elimination} \\
&\mid K \overline{\text{ae}} \quad \text{Term} \\
&\mid \text{let } x = \text{ae}_1 \text{ in } \text{ae}_2 \quad \text{Let binding}
\end{align*}
\]

### Figure 5. Attribute grammar syntax

**A** : [α] in **INBOUND** is turned into a corresponding AG attribute **AP** : **Set** α. For example, elaborated sort **Term** in Figure 8 has an attribute **ctx**, where the original **Term** has an attribute **ctx**.

There is one important catch: we need to invert the dataflow. Contexts flow variables form their binders to their references, while free variables flow variable references to their binders. This flow inversion means that in rule Fv-SDECL, inherited attributes are elaborated into synthesized attributes and vice versa.

Similarly, context expressions are inverted in the elaboration. Indeed, Lam X t extends its context with X before passing it to t, but subtracts X from t’s free variables before passing them up. Hence, we get the following elaboration for Lam:

\[
t.\text{ctx} = \text{lhs.} \text{ctx}, X \leadsto \text{lhs.} \text{ctx} = t.\text{ctx} = \text{ctx} - \{ X \}
\]

In general, every binding occurrence of a variable gives rise to a subtraction, and every referencing occurrence to an addition of a variable. The latter is captured in rule Fv-TERMDECL and illustrated by the elaborated attribute **lhs**. ctx = { X } of Var X. The former is handled by rule Fv-TERMDECL which is responsible for inverting all the flows from and to subterms. All inherited contexts A of the current node, require definitions of free variable synthesis, and all synthesized contexts B of the subterms require definitions of inherited free variables. To calculate these definitions, we invert and combine all expressions that use A (respectively B). The App t₁ t₂ constructor is an example that involves combined inversion. The context **lhs. ctx** is inherited by both t₁ and t₂. Hence, **lhs. ctx. fvs** is defined as the union of t₁. ctx. fvs and t₂. ctx. fvs.

Our formalisation avoids a clever up-front analysis of the uses; instead it combines all expressions and suppresses the irrelevant ones during inversion. The joint inversion and suppression happens in the auxiliary function ![ECL](image-url). This function computes the contribution of context definition N₁, A₁ = e to the free variable definition N₂, A₂. It does so by inverting empty sets into empty sets and context extensions into free variable subtractions. Whenever a use of N₁, A₁ occurs, it is inverted into the free variables N₂, A₂ of the user. Whenever another attribute is used instead of N₁, A₁ the expression is suppressed by inverting it into an empty set.

### Free variable functions

We provide free variable functions as a front-end interface to the elaborated free variable attributes. There is one such function for each syntactic sort S. This function takes initial values for the inherited attributes to final values for the synthesized attributes:

\[
w_{\text{fvs}} : S \rightarrow \text{Set } A \rightarrow \text{Set } A
\]

For example, the elaborated attributes in Figure 8 give rise to two functions:

\[
w_{\text{fvs}} \text{Term} : \text{Term} \rightarrow () \rightarrow \text{Set } A \\
w_{\text{fvs}} \text{Pat} : \text{Pat} \rightarrow \text{Set } A \rightarrow \text{Set } A
\]

For Term we get a function that computes the free variables of a given Term. The function on patterns expects a set of variables and returns the remainder of variables which are not bound by the pattern.

The free variable function plays an important role in capture-avoiding substitution: it is used to identify which variables mentioned in a substitution run the risk of being captured when moving that substitution under a binder. For instance, the following derived function determines the free variables in a term substitution:

\[
w_{\text{fvs} \text{Subst}} : (\text{h} \mapsto \text{Term}) \rightarrow \text{Set } A \\
w_{\text{fvs} \text{Subst}} = \text{unions } \{ \{ x \} \cup w_{\text{fvs}} \text{Term}, t | (x \mapsto t) \leftarrow \text{sub} \}
\]

We derive such a function for every namespace.
4.5 Renaming elaboration

Variable renaming replaces all references to a variable by references to a different variable. It is used as an auxiliary function of full substitution: in order to avoid variable capture, bound variables are renamed to fresh ones. One renaming may lead to a cascade of further recursive renamings, as the operation itself is a simplified form of substitution that must avoid variable capture. We generically deal with the problem by conservatively freshening all bound variables during renaming; no attempt is made to accurately determine whether variable capture would actually happen.

Figure 7 shows the elaboration of variable renaming in two parts: analysis and synthesis. The analysis part pushes a mapping from variables to freshened variables down the context flow. This mapping is used in the synthesis part to construct the renamed term in a bottom-up fashion.

Analysis Rule $\text{RENAMING} \text{SORTDECL}$ elaborates every attribute declaration to one of the same kind; the flow is not inverted. The types of every elaborated attribute is a Cartesian product of declaration to one of the same kind; the flow is not inverted. The Rule $\text{RENAMING} \text{SORTDECL}$ elaborates every attribute declaration to one of the same kind; the flow is not inverted.

Synthesis Rule $\text{RS-SORTDECL}$ declares a new synthesized attribute $\text{rename} : S$ that holds the renamed term. In rule $\text{RS-VARDECL}$ this attribute is defined for a variable constructor $K (X @ A)$ as $K (\rho (X))$ where $\rho$ is the renaming map for attribute $A$ that is computed by the analysis part.

For a term constructor rule $\text{RS-TERMDECL}$ constructs a new term with the same constructor, and with renamed bound variables and the recursively renamed subterms. The former are proactively renamed to fresh variables to avoid potential variable capture. We choose the new variable name fresh with respect to the renamed context that receives it; this renamed context is computed by the analysis part. Auxiliary function $\text{rename} : S$ takes care of retrieving the appropriate renamed context and computing the fresh name. The renamed subterms are trivially obtained through the $\text{rename}$ attributes of the subterms.

Example Figure 8 contains the renaming elaboration for the simply-typed lambda calculus. The two interesting constructors are those that bind variables: lambda abstractions and pattern variables. In each case a let-binding matches against the previous freshened context $\text{ctx}$ and renaming mapping $\text{rho}$. Using $\text{fresh}$ we can choose a new fresh name and construct the pair again with an updated context and mapping.

Renaming function The elaborated attributes give rise to renaming functions for abstract syntax terms. Specifically, the elaborated attributes in figure 8 give rise to two functions.

\[
\begin{align*}
\text{rename}_{\text{Term}} : & \quad \text{Term} \to ([A], (A \mapsto \hat{A})) \to \text{Term} \\
\text{rename}_{\text{Pat}} : & \quad \text{Pat} \to ([A], (A \mapsto \hat{A})) \to (\text{Pat}, [A], (A \mapsto \hat{A}))
\end{align*}
\]

The function $\text{rename}_{\text{Term}}$ takes as argument the freshened context and the renaming mapping and will return the synthesized renamed term. $\text{rename}_{\text{Pat}}$ returns the freshened pattern and additionally the updated context and mapping.

4.6 Substitution elaboration

The elaboration of substitution is similar to that of renaming, except that it replaces variables by terms instead of other variables. It too proceeds in two steps, analysis and synthesis. Analysis pushes a map from variables to terms down the context flow and synthesis builds the new term with the variables replaced and the necessary context avoiding renamings performed. Both steps are defined in Figure 10.

Analysis Rule $\text{SA-SDECL}$ elaborates the attribute declarations into ones that hold substitution maps. As rule $\text{SA-VARDECL}$ shows, we do not need to define substitution maps for variable constructors. In rule $\text{SA-TERMDECL}$ we see that term constructors...
only propagate the substitution maps down the context flow, without modification, to make them available for the synthesis part.

**Synthesis** Rule SS-SDECL introduces a new synthesized attribute `rename` to hold the newly synthesized term. Rule SS-TERMDECL synthesizes a new constructor term using the same constructor, renamed bound variables (to avoid variable capture) and substituted subterms.

The actual substitution happens in rule SS-VARDECL. First the referenced variable \( X \) is renamed to \( y \) using the renaming map \( \rho \) to account for variable capture avoidance. The resulting term is then obtained by either looking up \( y \) in the propagated substitution map or, if \( y \) is not in the substitution map, by simply returning \( K y \).

**Example** Figure 8 contains the substitution elaboration for the simply-typed lambda calculus. The interesting case is the term variable case. The variable is first renamed and then either substituted by a different term or by the renamed reference.

**Substitution function** The elaborated attributes give rise to substitution functions on abstract syntax terms. For each syntactic sort \( S \) with a variable constructor \( \alpha : S \rightarrow S \) this is a function that expects values for each inherited attribute and in turn delivers values for each synthesized attribute. Specifically, the elaborated attributes for renaming and substitution in Figure 8 give rise to the function:

\[
\begin{align*}
\text{rename} & \colon \text{Term} \rightarrow \text{Set } \alpha \rightarrow (\alpha \rightarrow \alpha) \\
\text{term} & \colon \text{Term} \rightarrow \text{Set } A \rightarrow (A \rightarrow A) \\
\end{align*}
\]

The first argument is the original term. The second argument is the freshened context and the third the renaming map. Finally, the fourth argument specifies the substitution map. From this function we can derive a proper substitution function in the following way:

- The freshened context is initially the outer context which includes the free variables of the term, the variables to be substituted and the free variables of the substitutes.

Figure 8. Free variable elaboration of STLC Prod
The renaming map is initially empty.

For terms the result is

\[ \text{substTerm} :: \text{Term} \rightarrow (\lambda \rightarrow \text{Term}) \rightarrow \text{Term} \]
\[ \text{substTerm} \ t \ \text{sub} = \]
\[ \text{substTerm} \ t \ (\text{funSub} \ t \cup \text{funSub} \ \text{sub}) \ [\ ] \ \text{sub} \]

5. Implementation

This section briefly discusses our INBOUND compiler and explains how it differs from the elaboration of Section 4.

Our INBOUND compiler is written partly in Haskell and partly in terms of the Utrecht University Attribute Grammar Compiler system UUAGC [21]. Moreover, our compiler elaborates INBOUND specifications into UUAGC attribute grammars, which support all the required target language features. UUAGC in turn compiles attribute grammars to Haskell code. This makes it very easy to use INBOUND specifications for meta-programming tools written in Haskell. As another convenience, the UUAGC system also checks the required non-circularity for us.

The whole INBOUND compiler consists of about 350 lines of Haskell code and 1400 lines of attribute grammar code.

5.1 Implemented extentions

The line count above includes additional features, on top of those presented in Sections 3 and 4, that make the specification language more practical. We list the most important extensions below.

Namespaces without substitution The well-formedness relation presented in Section 5 makes sure that every namespace can support substitution. Our compiler does not uniformly enforce this property, but also allows namespaces that only support renaming. A practical application for this relaxation are type constructors. While type constructors come with a notion of scoping and can be renamed, substitution does not make sense for them.

Terminals It is not practical to define primitive datatypes, such as ints, floats and strings, in terms of INBOUND sorts. Hence, our compiler supports sort term constructors with terminals of any Haskell type. These terminals are simply copied as is in the elaboration of the syntactic operations.

Variable constructor fields In practice, variable constructors do not follow the prototypical example of a single atom field. Variables commonly annotated with additional information (e.g., source code location, type or kind) to ease the implementation of the meta-program.

To enable this practice, our implementation also allows variable constructors to have additional terminals and subtrees. In the elaboration of substitution these are preserved during renaming but are dropped when the variable is substituted. This is the semantics that is usually applied in this situation. However, it is up to the user to verify that it coincides with his intentions.

5.2 Future extensions

There are a few desirable extensions worth mentioning, that we intend to investigate in future work. They can further facilitate the use of INBOUND.
Polymorphic type constructors  Polymorphic type-constructors for reusable data types like lists is high on our list. Lists are ubiquitous in abstract syntax and writing many different specialized definitions is rather tedious.

Non-free syntax  Variable binding of abstract syntax exhibits a monad interpretation of substitution \(\text{Bind}\). In essence a substitution function corresponds to the bind of a Monad.

\[
\text{subst} :: \text{Term} \rightarrow (\text{Var} \rightarrow \text{Term}) \rightarrow \text{Term}
\]

In this light the structure that we enforce on abstract syntax specifications is the structure of a free monad for which we then can derive the bind automatically.

However, not all languages can be cast into this scheme. For example, the let-normal-form restricts the shape of compound expressions: an application takes two term variables as arguments.

type \(V = \text{String}\)
data \(\text{Expr} = \text{Var} V \mid \text{Lam} V \text{Expr} \mid \text{App} V V \mid \text{Let} V \text{Expr} \text{Expr}\)

It is still possible to define a substitution function for this language? Yes, but this would require renormalization after substituting the variables of an application. However, deriving this substitution function automatically seems as challenging as automatically deriving a Monad instance for an arbitrary data type declaration. Still, we would like to support such languages and ask the user to only specify the essential parts of the monadic bind.

Name resolution  INBOUND, like other specification languages for binding, focuses on resolved programs, i.e., programs in which names are already associated with the originating declaration. It does not address the name resolution process, but takes it for granted. This typically means that the meta-program first has to perform a syntactic analysis before using the INBOUND syntax and operations.

NERON et al. \cite{NERON} describe name resolution in a language-independent way by means of a scope graph that represents the naming structure of a program but which abstracts away language dependent parts of the abstract syntax. On top of that they develop a resolution calculus, which describes how to resolve references to declarations within a scope graph.

Attributes conveniently allow this contextual information to flow from the declaration (or the top-level) to the point where it is used. We would like to make the types of attributes richer and allow arbitrary information to flow around the abstract syntax and from this build the scope graphs that can be used for name resolution.

This would result in a formal specification language that covers parts of name binding that are commonly described in an ad-hoc way such as module imports.

6. Case Studies

To show the expressiveness of our specification language and the benefits of our approach we have performed two case studies.

6.1 LetPoly

We have implemented from scratch a compiler from a lambda calculus with let-polymorphism, LETPOLY, to System F.

The compiler features two syntactic sorts of terms, one for the source language and one for the target language, each with its own namespace of term variables, and one syntactic sort of types. Additionally, we specify typing environments, i.e. mappings from term variables to types, as part of the abstract syntax to get boilerplate operations for them for free.

The compiler consists of two major meta-programming tasks:
Elaboration of LetPoly into System F

This involves computations of free variables, fresh name generation for System F types and type variables and type substitution in System F types, terms and typing environments which were all generated by INBOUND.

Normalisation of System F

This uses free variable checks for η-reduction and term substitution for β-reduction and relies on capture avoidance.

6.2 Typing Haskell in Haskell

Our second case study integrates INBOUND in an existing project: We have replaced the existing syntax definitions and boilerplate operations for Haskell in Jones’ Typing Haskell in Haskell (THH) by a corresponding INBOUND specification.

The type language fragment of the abstract syntax has four distinct namespaces: type constructor names, type class names, unification variables and schematic type variables bound by type-schemes. In the original code unification variables use a nominal representation and schematic variables use de Bruijn indices. We translated the latter to use a nominal approach as well. The binding structure is rather flat, the only binding forms are type-schemes. We managed to express an important invariant of the type-checker in the binding specification: In a type scheme of the form

\[
\forall \alpha : k. C \tau \Rightarrow \tau'
\]

the qualified type \( C \tau \Rightarrow \tau' \) contains no unification variables and the only schematic variables are the \( x \) bound in the type scheme, i.e. type schemes are closed with respect to type variables.

Both kinds of variables support substitution (or instantiation) but their original implementations do not account for capture and do not need to do so because of the flat binding structure. As a consequence, the renaming boilerplate is also not needed.

The type-checker itself only makes use of boilerplate operations on types. We have however written a specification for the entire abstract syntax including terms. The term language features intricate scoping rules in binding groups and nested patterns. We can express another invariant of the type-checker: After dependency analysis a binding group

\[
\text{type BindGroup} = ([\text{Exp}],[[\text{Impl}]])
\]

consists of a list of explicitly typed bindings \([\text{Exp}]\) and a topologically sorted list of lists of implicitly typed bindings \([[[\text{Impl}]]]\). This is necessary to obtain the most general types possible. For a binding group \(\{e_0, e_1, \ldots, e_n\}\) our specification makes clear that no implicitly typed group depends on later ones by scoping them sequentially. Moreover, an implicit group is binds recursively over itself and the entire binding group is recursively scoped in the explicit ones \(e_i\).

Conclusion

Table 1 gives an overview of the size of the specifications and the generated code. The second and third column list the size of the original datatype definitions and boilerplate functions. The fourth and fifth column list the sizes of the sort declarations and binding specifications of the languages. The sixth column is the size of the attribute grammar code that is produced by the INBOUND compiler and the seventh column in turn shows the size of the Haskell code produced by the UUAG compiler.

In case of THH the declarations are slightly larger due to the additional declaration of namespaces and the unfortunate inlining of lists. However, the size of the binding specifications are a fraction of the boilerplate in the original source code.

The generated code, however, is significantly larger. INBOUND produces 1600 lines of AG code which in turn are translated to 6500 lines of Haskell code. Both, INBOUND and the UUAG compiler produce a lot of copy rules. However, this does not add to the complexity of the functions and most of the generated code can be drastically simplified by the Haskell compiler. The overall amount of code that is generated by the INBOUND compiler is linear in the size of the specification.

7. Related and Future Work

7.1 Pragmatic programming with binders

Unbound The Unbound library [22] is a Haskell library for programming with abstract type. It’s specification language consists of a set of reusable type combinators that specify variables, bindings and commonly found scoping rules like recursive and sequential scoping. The library also features a combinator Shift to express more exotic non-linear scoping rules. The end user is provided with syntax operations for free which are implemented by means of datatype-generic programming.

Our specification language INBOUND subsumes the binding specifications of Unbound but is also strictly more powerful: Each of the Unbound combinators Bind, Rebind and Rec assume that all the variables of a pattern are bound simultaneously while INBOUND supports selecting multiple different subsets for different bindings. Figure 11 contains an example of a INBOUND specification that is not expressible in Unbound: Patterns specify two distinct but interleaved sets of variable bindings and the term abstraction uses them to bind in two different sub-terms.

Ott

The Ott tool [19] allows the definition of concrete syntax of programming languages and inductive relations on terms for the formalization of meta-theory for programming languages. Ott in-

Table 1. Size statistics of the specifications and boilerplate.

<table>
<thead>
<tr>
<th>Original Declarations</th>
<th>Boilerplate</th>
<th>Generated Grammar</th>
<th>Binding</th>
<th>AG Code</th>
<th>HS Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LETPOLY</td>
<td>N/A</td>
<td>N/A</td>
<td>15</td>
<td>4</td>
<td>170</td>
</tr>
<tr>
<td>System F</td>
<td>N/A</td>
<td>N/A</td>
<td>15</td>
<td>5</td>
<td>310</td>
</tr>
<tr>
<td>THH (Types)</td>
<td>15</td>
<td>80</td>
<td>25</td>
<td>20</td>
<td>230</td>
</tr>
<tr>
<td>THH (Full)</td>
<td>50</td>
<td>80</td>
<td>100</td>
<td>120</td>
<td>1600</td>
</tr>
</tbody>
</table>

Figure 11. Example specification not expressible by Unbound and Romeo
cludes a language of binding specifications that allow the definition of functions for calculating the sets of variables bound by patterns. The expressivity of their binding specifications corresponds to using at most one inherited attribute per sort and namespace (and as many synthesized attributes as desired) in INBOUND. Furthermore, Ott binding specification are restricted to specifying which new variables are bound in a binding form, specifying that certain expressions are closed is not supported.

7.2 Binding-safe programming

**Fresh look and nameless painless** Pouillard and Pottier [18] and Pouillard [17] provide a novel approach to binding-safe programming by using a fine grained and abstract interface for names that is disconnected from and actual representation. To control the use of names, they introduce an abstract notion of world and associate a world with each name. Consequently, types for abstract syntax are then indexed by one or more worlds for the names they contain. There is a direct relation between the concepts used in this paper and the concepts used by Pouillard and Pottier.

1. Our contexts correspond to worlds. In particular the empty context corresponds to the empty world.
2. Context attributes correspond to world indices of abstract syntax types.
3. In constructor declarations our evaluation rules of context attributes correspond to the computations that define the world indices.
4. The occurrence of a name is what we call a variable reference and weak links are variable bindings.

In comparison we lack the support of strong links. The corresponding concept would be a variable binding that is structurally guaranteed to be fresh for the enclosing context, i.e. a variable binding that does not shadow any previous variables. While we also chose to implement the generation of fresh variables (strong links) in a pure way, we do not keep track of this information.

Pouillard and Pottier [18] show how to implement renaming and substitution for the untyped lambda calculus but do not show how to derive this functionality generically from the specification and in particular how to do so when multiple sorts have variables which is the goal of this paper. In turn binding-safe programming is beyond the scope of this paper but we intend to address it in future work.

**Romeo** ROMEO’s binding specification language is also based on attribute grammars. Yet, it restricts every sort to one inherited and one synthesized attribute. These attributes are usually heterogeneous: they simultaneously specify the context of all namespaces. This allows more flexibility in the specification of binders than UNBOUND’s type combinators, but is still not powerful enough to express the specification of Figure 11. ROMEO’s binding specification is strictly less expressive than Ott’s binding specifications and also do not support closing terms.

7.3 Variable binding in mechanization

There is a wealth of work [2] [10] [11] [13-15] on dealing with variable binding in the mechanization of programming language meta-theory. In particular, these system focus on addressing the proof boilerplate of variable binding and try to make proving as simple as possible. Often this comes at the cost of expressivity by for example restricting to single-variable binders or restricting the number of namespaces. Moreover, these systems are all restricted to a single inherited context with linear scoping.

8. Conclusion

We presented the specification language INBOUND for abstract syntax with binders. This language makes scoping rules of languages explicit and automatically derives boilerplate functions from them. Our implementation is available from [https://github.com/skeuchel/inbound](https://github.com/skeuchel/inbound).

The language provides ample opportunity for follow-on work, like automatic name resolution and support for non-free monads.

References