Implementation

Sune Skaanning Engtorp

21/05/2024

1 Implementation

1.1 Representing Syntax

The generalized editor calculus[1] assumes that it is given an abstract syntax that is represented by a set of sorts $\mathcal{S}$, an arity-indexed family of operators $\mathcal{O}$, and a sort-indexed family of variables $\mathcal{X}$, as per Robert Harper's notation [5].

A criterion for the good solution of this project is also the implementation of being able to pretty-print a program into a concrete syntax. For this, it is also necessary for the user to provide the concrete for a language they wish to edit.

It can be a challenge from the user's perspective to provide a specification based on what the calculus assumes. Therefore, it is ideal for the implementation to provide other means of describing the syntax of a language.

In terms of early examples, Metal[9] has been used in the Mentor [3] and CENTAUR [2] systems. Metal compiles a specification containing concrete syntax, abstract syntax and tree building functions for a formalism F into a Virtual Tree Processor (VTP) formalism, a concrete syntax parser produced by YACC[8] and a tree generator which uses VTP primitives to construct abstract syntax trees.

Another example with the same purpose is Zephyr ASDL (Abstract Syntax Description Language)[13], where the authors have built a tool that converts an ASDL specification into C, C++, Java and ML data-structure definitions. The authors consider ASDL a simpler alternative to other abstract syntax description languages, such as ASN.1 [10].

However, both examples have lack of binding mechanisms in abstract syntax in common. This motivates another possibility of defining a specification language for the to-be-implemented generalized editor itself, which can assist the user in describing the syntax. This would also require a parser that can parse the necessary information assumed by the calculus (including binders). Picking this route allows the project to avoid spending time analyzing different tools and developing a workaround for binders.

1.1.1 Specification language

The specification language is chosen to expect some syntactic categories followed by concrete and abstract syntax in BNF notation. Every syntactic category is represented by one or more non-terminals with a term, arity and operator name. The term might refer to other syntactic categories or its own. Each term is the concrete syntax of an operator, while the arity, in combination with the operator name, is a concise representation of the abstract syntax of an operator. The abstract syntax only makes use of the defined syntactic categories and, inspired by Harper[5], binders can be specified in the arity description with a dot ('.'), e.g. $x.s$ specifies that variable x is bound within the scope of s.

From this specification language, it is possible to extract what is assumed by the generalized editor calculus[1]. The set of sorts $\mathcal{S}$ is the set of syntactic categories. For example, a syntactic category $e \in Exp$ can get parsed into a sort s_e. The family of arity-indexed operators $\mathcal{O}$ can be extracted from the BNF notation since each derivation rule represents an operator with a specified arity.

Example 1: Syntax of a small C language

Following is a specification of a subset of the C language[7], per the described specification language.

$$
\begin{array}{ll}
p \in \mathit{Prog} & s \in \mathit{Stmt} \\
vd \in \mathit{VariableDecl} & fd \in \mathit{FunDecl} \\
t \in \mathit{Type} & id \in \mathit{Id} \\
e \in \mathit{Exp} & b \in \mathit{Block} \\
fa \in \mathit{Funarg} & cond \in \mathit{Conditional} \\
int \in \mathit{Int} & char \in \mathit{Char} \\
bool \in \mathit{Bool} & string \in \mathit{String}
\end{array}
$$

Sort	Term	Arity	Operator
$p ::=$	fd	$(fd)p$	*program*
$b ::=$	bi	$(bi)b$	*block*
$bi ::=$	vd	$(vd)bi$	*blockdecls*
$\mid$	s	$(s)bi$	*blockstmts*
$\mid$	ϵ	$()bi$	*blockdone*
$vd ::=$	$t\ id$ ”=” e ”;” bi	$(t, e, id.bi)s$	*vardecl*
$fd ::=$	$t_1\ id_1$ ”(” $t_2\ id_2$ ”)” ”{” b ”}” fd	$(t_1, id_1.fd, t_2, id_2.b)fd$	*fundecl1*
$\mid$	$t_1\ id_1$ ”(” $t_2\ id_2$ ”,” $t_3\ id_3$ ”) {” b ”}” fd	$(t_1, id_1.fd, t_2, t_3, id_2.id_3.b)fd$	*fundecl2*
$\mid$	ϵ	$()fd$	*fundecldone*
$s ::=$	id ”=” e ”;”	$(id, e)s$	*assignment*
$\mid$	id ”(” fa ”);”	$(id, fa)s$	*stmtfuncall*
$\mid$	”return ” e ”;”	$(e)s$	*return*
$\mid$	$cond$	$(cond)s$	*conditional*
$\mid$	$s\ s$	$(s, s)s$	*compstmt*
$fa ::=$	$t\ id$	$(t, id)fa$	*funarg*
$\mid$	$t\ id$ ”,” fa	$(t, id, fa)fa$	*funargs*
$cond ::=$	”if (” e ”) {” b_1 ”} else {” b_2 ”}”	$(e, b_1, b_2)cond$	*ifelse*
$t ::=$	”*int*”	$()t$	*tint*
$\mid$	”*char*”	$()t$	*tchar*
$\mid$	”*bool*”	$()t$	*tbool*
$e ::=$	int	$(int)e$	*int*
$\mid$	$char$	$(char)e$	*char*
$\mid$	$bool$	$(bool)e$	*bool*
$\mid$	e_1 ”+” e_2	$(e_1, e_2)e$	*plus*
$\mid$	e_1 ”==” e_2	$(e_1, e_2)e$	*equals*
$\mid$	id ”(” fa ”)”	$(id, fa)e$	*expfuncall*
$\mid$	id	$(id)e$	*expident*
$id ::=$	%string	$()id$	*ident*

’%int’, ’%char’, ’%string’ and ’%bool’ are meta-variables representing any parseable integer, character, sequence of characters and boolean constant by the C language. This subset is chosen as it can represent a C program consisting of only function declarations at the top level, where one of them might represent a `main` function, the entry point of a C program. The identifier of a function declaration is bound within the following function declarations (e.g. id_1 is bound within *fd* in the *fundecl1* operator).

A limitation of this specification is recursive function calls. Ideally, the identifier in a function declaration is bound both within the function block and the sequence of

following function declarations. However, this would result in the same identifier appearing twice in the arity definition. E.g. the arity for the *fundecl1* operator is $(t_1, id_1.fd, t_2, id_2.b)fd$ where id_1 is the identifier of the function, which ideally would be bound in both *fd* (the following sequence of function declarations) and *b* (the function block).
Another limitation, or something that might seem unnecessary, is having the *blockdone* and *fundecldone* operators. They are necessary to allow for a block to end with a *vardecl* operator and to end a sequence of function declarations with the $fundecl1$ or $fundecl2$ operator. This a pattern that allows operators to bind identifiers within the following terms.

Example 2: Syntax of a small SQL language

Below is the syntax of a subset of the PostgreSQL[4] dialect of SQL:

$$\begin{array}{ll}
q \in \mathit{Query} & \\
\mathit{cmd} \in \mathit{Command} & \mathit{id} \in \mathit{Id} \\
\mathit{const} \in \mathit{Const} & \mathit{clause} \in \mathit{Clause} \\
\mathit{cond} \in \mathit{Condition} & \mathit{exp} \in \mathit{Expression}
\end{array}$$

Sort	Term	Arity	Operator
query ::=	”SELECT ” id_1 ” FROM ” id_2 *clause*	$(id_1, id_2, clause)query$	*select*
cmd ::=	”INSERT INTO ” id_1 ” AS ” id_2 *query*	$(id_1, id_2.query)cmd$	*insert*
id ::=	%string	$()id$	*id*
const ::=	%number	$()const$	*num*
|	”’”%string”’”	$()const$	*str*
clause ::=	”WHERE ” *cond*	$(cond)clause$	*where*
|	”HAVING ” *cond*	$(cond)clause$	*having*
cond ::=	exp_1 ”¿” exp_2	$(exp_1, exp_2)cond$	*greater*
|	exp_1 ”=” exp_2	$(exp_1, exp_2)cond$	*equals*
exp ::=	*const*	$()exp$	*econst*
|	*id*	$()exp$	*eid*

where ’%string’ and ’%number’ and ’%char’ are placeholders for any parsable sequence of characters and numbers by the PostgreSQL language.
This subset is chosen as it can represent simple select queries and insert commands. Notably, a binder is used in the *insert* operator, where the alias of id_1, specified as id_2 is bound within the sub-query.

To make the specification language parseable, a more computer-friendly format is presented in figure Fig. 1. Every syntactic category is expected on its own line, followed by a blank line and all derivations. Each derivation is expected to be a syntactic category, followed by '::=' and every term (which acts as the concrete syntax of an operator), arity and operator name, separated with a vertical bar '—'. Every term, arity and operator are separated with a number-sign '#'. See figure Fig. 2 for an example.

Figure 1: Specification language in BNF notation

Figure 2: Subset of syntax of a small SQL language in a parseable format

```
query in Query
cmd in Command
id in Id
clause in Clause

query ::= " SELECT " id " FROM " id clause # (id,id,clause)query # select
cmd ::= " INSERT INTO " id " AS " id query # (id,id.query)cmd # insert
```

It is also assumed that the first non-terminal from the derivations is the starting symbol.

1.2 Code generation versus generic model

Another important thing to consider for the implementation is whether part of the editor's source code should be generated automatically, or if a generic model might suffice.

Automatic generation of source code offers the benefit of directly representing provided operators, along with their arity and sort, within an algebraic data type (referred to as type in Elm and data in Haskell). This ensures that only well-formed terms can be represented using the algebraic data types.

However, opting for this method might require automatic updates to both the definitions and signatures of some functions. This presents a challenge in ensuring that these functions maintain their intended behaviour after the updates.

Example 3: Algebraic data types for a small C syntax

Given example Example 1, one can generate the following custom types in Elm:

```
type alias Bind a b =
    ( a, b )
type Prog
    = Program FunDecls
type Block
    = Block BlockItems
type BlockItems
    = BlockDecl VarDecls
    | BlockStmts Statements
    | BlockDone
type VarDecls
    = VarDecl Type Id Exp BlockItems
type FunDecls
    = FunDecl1 Type (Bind Id FunDecls) Type (Bind Id Block)
    | FunDecl2 Type (Bind Id FunDecls) Type (Bind Id (Bind Id Block))
    | FunDeclDone
type Statement
    = Assignment Id Exp
    | StmtFunCall Id Funargs
    | Return Exp
    | Conditional Conditional
type Funargs
    = ArgSingle Funarg
    | ArgCompound Funargs Funarg
type Funarg
    = Funarg Type Id
type Conditional
    = IfElse Exp Block Block
type Statements
    = SSingle Statement
    | SCompound Statements Statement
type Type
    = TInt
    | TChar
    | TBool
type Exp
    = Num
    | Char
    | Bool
    | Plus Exp Exp
    | Equals Exp Exp
    | ExpFunCall Id Funargs
    | ExpId Id
```

```
type Id
    = Ident String
```

The *Bind* type alias is simply a tuple of 2 given type variables. This would be a part of the generated source code.

In contrast, a generic solution without the need for generating new source reduces the risk of syntax errors in the target language for the editor itself. A generic solution involves designing a model capable of holding the necessary information from any syntax. An approach for this is provided in figure **??**. However, this approach requires additional well-formedness checks on any given term concerning the syntax.

As an alternative to having specific types for each given syntactic category (as in example Example 3), one can design a generic model with records in Elm, as in listing Listing 1.

```
type alias Syntax =
{ synCats : [String]
, operators : [Operator]
}

type alias Operator =
{ name : String
, arity : (Maybe [String], String)
, concSyn : String
}
```

Listing 1: Elm Records for storing syntax information

The arity in the Operator type is a tuple, where the first entry is a potential list of identifiers bound within the term in the second element of the tuple.

In Haskell, this corresponds to named fields[6], which have a very similar syntax.

The implementation will proceed with automatic generation of source code, including algebraic data types, due to their advantage in handling ill-formed terms effectively. If given an ill-formed term, it is considered ill-typed by the editor, which poses an advantage over the generic solution requiring thorough checking to ensure that given terms are well-formed.

1.3 Generating source code

Elm CodeGen[11] is an Elm package and CLI tool (command-line interface tool) to generate Elm source code. The tool is an alternative to the otherwise obvious (and arguably tedious) strategy of having a source code template, where certain placeholders are replaced with relevant data or code snippets associated with the parsed syntax.

Besides offering the ability to generate source code, it offers offers automatic imports and built-in type inference. Example usage of Elm CodeGen from the documentation[11] is given in figure Fig. 3.

Figure 3: Elm CodeGen usage

Following declares an Elm record and passes it to a `ToString` function:

```
Elm.declaration "anExample"
    (Elm.record
        [ ("name", Elm.string "a fancy string!")
        , ("fancy", Elm.bool True)
        ]
    )
    |> Elm.ToString.declaration
```

The above will generate following string:

```
anExample : { name : String, fancy : Bool }
anExample =
    { name = "a fancy string!"
    , fancy = True
    }
```

Using Elm CodeGen offers the advantage of integrating a parser, which, if implemented in Elm as well, can directly produce the data to enable Elm Codegen to produce source files for the editor.

More specifically, the implementation will use the built-in `Parser` Elm library to parse a `RawSyntax` object (Listing 2), which is a direct representation of the syntax as specified in the specification language (Fig. 1).

```
type alias RawSyntax =
    { synCats : List RawSynCat
    , synCatRules : List RawSynCatRules
    }

type alias RawSynCatRules =
    { synCat : String
    , operators : List RawOp
    }

type alias RawOp =
    { term : String
```

```
    , arity : String
    , name : String
    }

type alias RawSynCat =
    { exp : String
    , set : String
    }
```

Listing 2: Raw syntax model in Elm

If parsing of the raw syntax is successful, the raw model will be transformed into a separate model built around the `Syntax` type alias (listing Listing 3). Transformations include converting the string-representation of the arity into its own `Arity` type, which is a simple list of tuples, where the first element is a list of variables to be bound within the second element.

```
type alias Syntax =
    { synCats : List SynCat
    , synCatOps : List SynCatOps
    }

type alias SynCat =
    { exp : String
    , set : String
    }

type alias SynCatOps =
    { ops : List Operator
    , synCat : String
    }

type alias SynCatOps =
    { ops : List Operator
    , synCat : String
    }

type alias Operator =
    { term : Term
    , arity : Arity
    , name : String
    , synCat : String
    }

```

```
type alias Term =
    String

type alias Arity =
    List ( List String, String )
```

Listing 3: Syntax model

Having all expected sets of sorts and family of operators, i.e. the abstract syntax extended with *hole* and *cursor* operator $\mathcal{S}, \mathcal{O}$, cursorless trees $\hat{\mathcal{O}}, \hat{\mathcal{S}}$, cursor contexts $\mathcal{S}^C, \mathcal{O}^C$ and well-formed trees $\dot{\mathcal{O}}, \dot{\mathcal{S}}$, the CodeGen package can generate algebraic data types for every sort and its operators and separate them into their own separate modules (files).

Example 4: From specification parser to Elm CodeGen for small C-language

Given the specification in example Example 1, the parser can produce following `Declaration` for the `Statement` algebraic data type in example Example 3:

```
Elm.customType "Statement"
            [ Elm.variantWith "Assignment"
                [ Elm.Annotation.named [] "Id",
                  Elm.Annotation.named [] "Exp" ]
            , Elm.variantWith "StmtFunCall"
                [ Elm.Annotation.named [] "Id",
                  Elm.Annotation.named [] "Funargs" ]
            , Elm.variantWith "Return"
                [ Elm.Annotation.named [] "Exp" ]
            , Elm.variantWith "Conditional"
                [ Elm.Annotation.named [] "Conditional" ]
    ]
```

This declaration, if passed to Elm CodeGen's `File` function, would generate a source file with following contents:

```
type Statement
    = Assignment Id Exp
    | StmtFunCall Id Funargs
    | Return Exp
    | Conditional Conditional
```

Having generated all sorts and operators as algebraic data types in Elm, the next step is to implement functionality for editor expressions. For example, consider the cursor substitution operator (definition **??**), where the editor calculus enforces that operators can only be substituted with operators of same sort. Initially, a straightforward approach involves having a substitution function for each sort $s \in \mathcal{S}$. This approach of course leads an implementer to consider generalization, i.e. how a single function can take any cursor-encapsulated abt and replace it with an operator of the same sort. A solution to this could be using type classes, where we might have a type class called **substitutable** and having an instance for each sort. Having a typeclass is possible in Haskell (see listing Listing 4), but typeclasses are not directly supported by Elm. They can however be simulated with Elm records, as shown in the "typeclasses" Elm package[12]. An example of such a simulation is shown in listing Listing 5. This typeclass simulation in Elm has the disadvantage of forcing an explicit reference to the typeclass "instance" in a generic function, in contrast to Haskell. This leads to more verbose code and more source code generation.

```haskell
-- typeclass
class Substitutable a where
    substitute :: a -> a -> a

-- instance of typeclass
instance Substitutable a where
    substitute _ replacement = replacement

-- example usage
doIntSub :: Int
doIntSub = substitute 1 2
```

Listing 4: Haskell typeclass example

```elm
{-| Simulate a type class
-}
type alias Substitutable a =
    { substitute : a -> a -> a }

{-| Generic instance of the typeclass, we don't need any
    specific implementation for each type/sort, we just
   want to assure that the expression and replacement
   are of the same type. This is constrained by the `
   substitute` function signature in the (simulated)
   typeclass.
-}
substituteAny : Substitutable a
substituteAny =
    { substitute = \_ replacement -> replacement }

{-| Polymorphic function that can be used with any type
   that has an instance of the `Substitutable` typeclass
```

```
    .
-}
substitute : Substitutable a -> a -> a -> a
substitute substitutable expression replacement =
    substitutable.substitute expression replacement

{-| Example usage
-}
doIntSub : Int
doIntSub =
    substitute substituteAny 1 2
```

Listing 5: Elm typeclass simulation example

1.4 Decomposing trees

The implementation needs to be able to check if a given abt is well-formed, or in other words, if it can be decomposed into $C[\dot{a}]$, where C is a cursor-context (definition **??**) and $\dot{a}$ is a well-formed abt (definition **??**).

First of all, it is necessary to know which symbol in the given syntax representation is intended as the starting symbol. Only the syntactic category of the starting symbol would need functions supporting decomposition. In other words, it does not make sense to decompose an abt of some sort which cannot be derived to a well-formed program. This aspect has not been considered yet, and as a temporary solution, a type wrapper called `Base` has been introduced, which has an entry for every syntactic category in the given syntax. For example, see listing Listing 6 for an excerpt of the `Base` type generated for the C language in example Example 1.

```
type Base
    = Prog Prog
    | Block Block
    | BlockItems BlockItems
    | VarDecls VarDecls
    | FunDecls FunDecls
```

Listing 6: Example of the Base type

This allows for any operator of any sort to be decomposed, although the ideal solution would be having a way to specify the starting symbol in the syntax representation.

The generalized editor calculus[1] defines well-formed trees as abt's with exactly one cursor either as the root or as an argument of the root. However this definition, in conjunction with the cursor context definition, leads to multiple valid decompositions for an abt in certain scenarios, which is also mentioned in [1]. This can occur when the cursor is located at one of the immediate children of the root. In that case, the cursor context can be interpreted as either the tree where the cursor has been replaced with a context hole, or it can be interpreted as an empty context. For example, consider a small tree created from the syntax given in example Example 2 (including cursor and hole operators) and their two possible decompositions in figure Fig. 4.

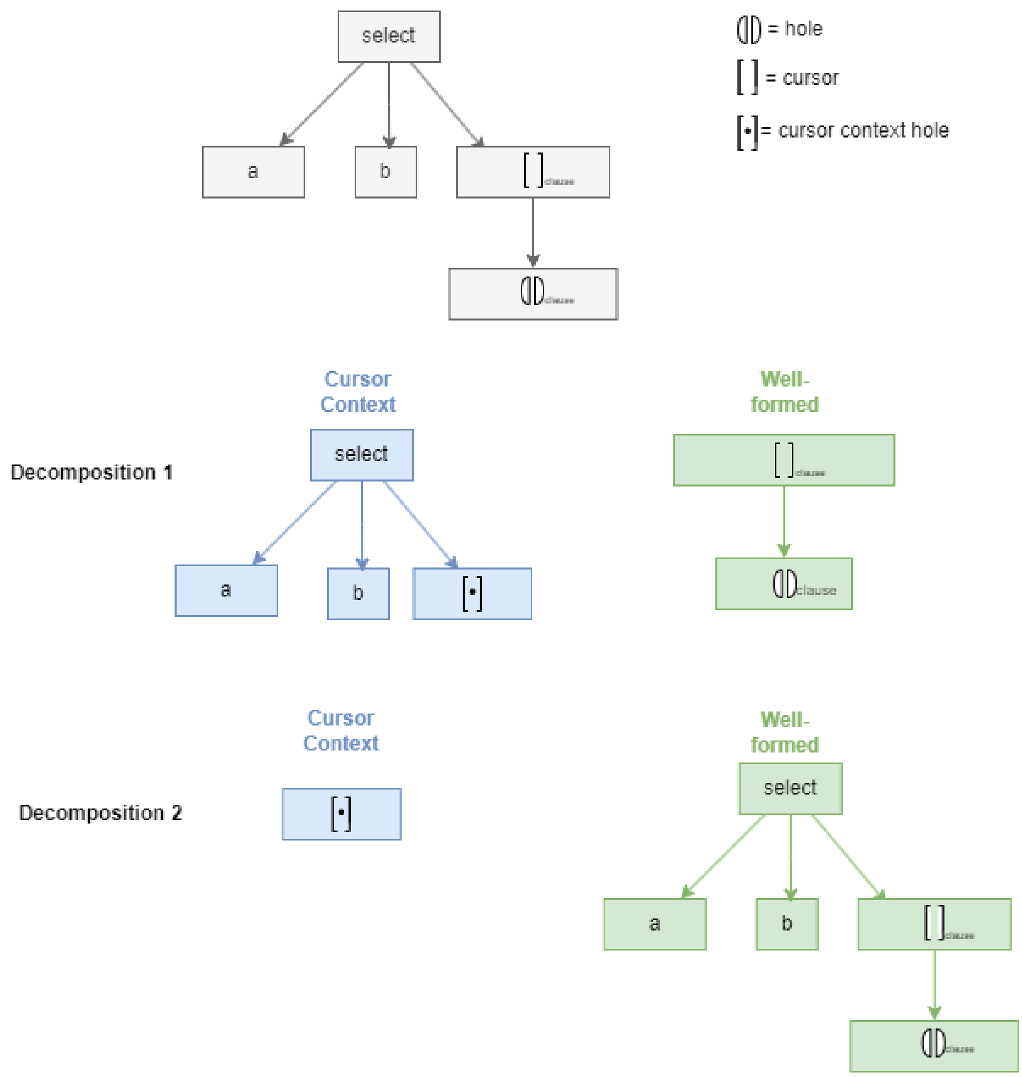

Figure 4: Two different decompositions of the same term

Decomposition should be generic and be possible on a valid abt of any given syntax. For this we have a typeclass called `decomposable` that contains a method called `decompose` which takes a statement made up by the operators of sort $\mathcal{S}$, and returns a cursor-context and well-formed-tree pair, if decomposition is possible.

In order to instantiate the `decomposable` type class for any sort, it is then necessary to specify how we for any term in any sort, can decompose uniquely into a cursor context and well-formed-tree pair.

Unique decomposition of an abt can be defined algorithmically and divided into following sub-tasks:

- Locate the cursor in the tree to be decomposed and generate a path to the cursor
- Generate an abt of sort $s^C \in \mathcal{S}^C$ based on the cursor path
- Generate an abt of sort $\dot{s} \in \dot{\mathcal{S}}$ based on the rest of the tree that was not traversed when generating the cursor context

The following will explain in more details how the steps above can be done, and how we always get a unique decomposition, as long as the abt to be decomposed is well-formed.

1.4.1 Cursor path

Generating a path to the cursor in the abt of sort $s \in \mathcal{S}$ extended with hole and cursor operators (definition **??**) simplifies the process of generating the cursor context. The path tells us which operator $o^C \in \mathcal{O}^C$ replaces each $o \in \mathcal{O}$. The list can be generated by performing pre-order traversal of the tree to be decomposed, extending the list with every i, representing which argument in an operator of arity $(\vec{s}_1.s_1, ..., \vec{s}_i.s_i, ..., \vec{s}_n.s_n)\mathcal{S}$ was followed to locate the cursor.

The implementation of such a function depends on the set of sorts $\mathcal{S}$ and arity-indexed family of operators $\mathcal{O}$ given by the abstract syntax of a language. The Elm CodeGen package[11] has been used to generate a `getCursorPath` function for a `Base` type (Listing 7). The function makes use of the helper function `getBranchList` which is left out for brevity, but its purpose is to generate a case expression for every syntactic category in the given syntax. See Example 5 for an excerpt of the generated `getCursorPath` function for the small C language (Example 1).

```
createGetCursorPath : Syntax -> Elm.Declaration
```

```
createGetCursorPath syntax =
    Elm.declaration "getCursorPath" <|
        Elm.withType
            (Type.function
                [ Type.list Type.int
                , Type.named [] "Base"
                ]
                (Type.list Type.int)
            )
            (Elm.fn2
                ( "path", Nothing )
                ( "base", Nothing )
                (\_ base ->
                    Elm.Case.custom base
                        (Type.named [] "Base")
                        (getBranchList syntax)
                )
            )
```

Listing 7: getCursorPath function generator

Example 5: Generated cursor path finder function

1.4.2 Cursor context

Having the cursor path, the cursor context can be generated by replacing every operator $o \in \mathcal{O}$ with its corresponding cursor context operator $o^C \in \mathcal{O}^C$, with respect to which argument in the operator was followed to locate the cursor. This is also done by performing pre-order traversal of the tree, but it will stop when the cursor is reached (i.e. when the cursor path is empty) and replace the operator reached with the context hole operator $[\ \cdot\ ] \in C$. The rest of the tree which has not been traversed will be passed to the next step, generating the well-formed tree.

Functions supporting this are generated by Elm CodeGen, and can be seen in listing Listing 8, where a `toCCtx` function is generated for the `Base` type in conjunction with a `toCCtx_s` for every s syntactic category in the given syntax.

```
createToCCtxFuns : Syntax -> List Elm.Declaration
createToCCtxFuns syntax =
  List.map createToCCtxFun syntax.synCatOps ++
  [ Elm.declaration "toCCtx" <|
```

```
  Elm.withType
    (Type.function
      [ Type.named [] "Base"
      , Type.list Type.int ]
      (Type.tuple
        (Type.named [] "Cctx")
        (Type.named [] "Base"))
    ) <|
    Elm.fn2
      ( "base", Nothing )
      ( "path", Nothing )
      (\base path ->
        Elm.Case.custom base
          (Type.named [] "Base")
          (List.map
            (\synCatOp ->
              Elm.Case.branchWith
                synCatOp.synCat
                1
                (\exps ->
                  Elm.apply
                    (Elm.val <|
                    "toCCtx_" ++ synCatOp.synCat)
                    (exps ++ [ path ])
                )
            )
            syntax.synCatOps
            )
      )
]
```

Listing 8: toCCtx function generator

1.4.3 Well-formed tree

The well-formed tree is generated by performing pre-order traversal of the rest of the tree that was not traversed when generating the cursor context. This is done by first replacing the cursor with the well-formed operator $\dot{o} \in \dot{\mathcal{O}}$ of arity $(\hat{s})\dot{s}$ indicating that the cursor encapsulates the root of a cursorless abt of sort $\hat{s}$. After this, the rest of the tree is traversed, and every operator $o \in \mathcal{O}$ is replaced with its corresponding cursorless operator $\hat{o} \in \hat{\mathcal{O}}$. Like when generating functions supporting cursor context, a very similar approach is taken here, and can be seen in listing Listing 9.

```
createToWellFormedFun : Syntax -> Elm.Declaration
createToWellFormedFun syntax =
  Elm.declaration "toWellformed" <|
    Elm.withType
      (Type.function
        [ Type.named [] "Base" ]
        (Type.named [] "Wellformed")) <|
      Elm.fn
          ( "base", Nothing )
          (\base ->
            Elm.Case.custom
                (Elm.apply
                  (Elm.val "consumeCursor")
                  [ base ])
                (Type.named [] "Base")
                (List.map
                  (\synCatOp ->
                    branchWith synCatOp.synCat
                        1
                        (\exps ->
                          Elm.apply
                              (Elm.val <|
                                "Root_" ++
                                synCatOp.synCat ++
                                "_CLess")
                              [ Elm.apply
                                (Elm.val <|
                                  "toCLess_" ++
                                  synCatOp.synCat)
                                exps
                              ]
                        )
                  )
                  syntax.synCatOps
                )
          )
```

Listing 9: toWellFormed function generator

The cursor context and well-formed tree pair as defined above will decompose any well-formed abt into a unique pair of a cursor context and a well-formed tree.