Idris implementation of a type-correct, stack-safe compiler including exception handling

Advanced Software Analysis

KSADSOA1KU

Morten Skøtt Knudsen
mskk@itu.dk

Jeppe Vinberg
jepv@itu.dk

Francisco Martínez Lasaca
frml@itu.dk

December 15, 2020

Abstract

In this report we describe the implementation of a dependently-typed abstract syntax of an expression language, an evaluator for this expression language, a compiler that translates the expressions into sequences of instructions for an abstract stack machine, and an interpreter for such instructions. Furthermore, the report also describes the implementation of a statement language that allows assigning such evaluated expressions to variables, which can then be used in other assigned expressions.

The implementation relies on the dependent type system of the programming language Idris to ensure properties of the semantics of the languages, such as well-formedness of expressions, type-correctness and stack-safety. Using this approach for implementing compilation and interpretation ensures well-formedness, and that only operations that do not cause unexpected behaviour can be executed on a stack of a certain type.

Upon having implemented the languages, we have acknowledged that Idris allows writing highly expressive abstract syntaxes, which in turn makes it easier to perform compilation into type-correct and stack-safe machine code. We first implemented this for an expression language that allows expressions to throw exceptions, and then went on implementing a statement language, that allows assigning expressions to variables and using those variables in other expressions.

1 Introduction

Compilers are complex pieces of software consisting of many parts that are difficult to implement, even more so proving that they are correct. Conventional approaches to ensure correctness have been focused on formalised proofs. However, with the use of dependently-typed languages, it has been shown that it is possible to use those systems for proof-checking, thus integrating the proof directly into the implementation of the programming language [3].

In this report, we describe the implementation of an evaluator and a compiler for a simple imperative language with variable assignment. Plus, an implementation of an interpreter for the compiled code is described.

The language used for the implementations is the purely functional language Idris. In Idris, types are first-class constructs, which means that types can be passed to and returned from functions. It is also possible to put constraints on data that is enforced at compile time through types. This encourages a type-driven approach to development where it is possible to prove the correctness of the implementation by being very precise about types throughout the design of the implementation [2].

The motivation for this implementation is that, given a dependently typed abstract syntax of the language, we can define a type-preserving evaluator that, given an expression, evaluates to a value of that same type. The compiler translates the typed expressions into machine code emulating operations on a stack machine. These operations will be dependently typed on the stack that they operate on, ensuring then type-safety on the stack. The dependent type system will also allow us to ensure stack-safety. Lastly, the interpreter will take as input the compiled machine-code and execute it [4].

As an extension to the expression language, semantics for throwing and catching exceptions are also added. This introduces new challenges in changing control flow at runtime. Here, we will also use the dependent type system to ensure that any well-formed expressions do not throw unhandled exceptions. That is, that all thrown exceptions will be caught [3].

Lastly, we describe how to build a statement language based on the expression language. This language will allow the programmer to declare new variables and assign values to them, which are based on the evaluation of expressions.

2 Background

Implementing an abstract syntax in a dependently-typed language gives the ability to express the semantics of the language using the typing system. In this way, the type-correctness of any constructed expression is guaranteed.

Constructing a compiler that translates expressions of this language into operations for an abstract stack machine in a dependently-typed language also allows us to express some constraints on these operations directly in the types of the host language. For example, we can ensure type-safety on the stack, as we are only able to perform an `ADD` operation if there are two natural numbers on top of the stack. We can also ascertain that no underflow can occur, as we impose restrictions on the type that limit their uses, as with the `pop` operation: there must be an element on top of the stack before this operation can be executed.

This chapter describes the relevant theory used in the implementation. Dependent types in Idris are presented in 2.1, and, in 2.2, we introduce proofs in Idris.

2.1 Dependent Types

Our implementation relies heavily on the notion of dependent types. Here we present the dependent types system in Idris.

Dependent types are types that depend on values of another type, adding a new layer of precision to the traditional approach to types. Generic types are types that depend on other types; dependent types add on top of this. A simple example is the vector type, `Vect n ty`, which is parametrized by not only the type `ty` of the elements in the vector, but also the number of elements in the vector (`n`). As the refined types contain more information, operations over them are more restricted. For instance, trying to inspect the head of an empty list of type `Vect 0 Int` will be detected and prevented at compile time.

Whenever a function that takes a value of a dependent type as input is defined, the values on which the type depends on are passed to the function as implicit arguments, allowing using these values in the execution of the function. In this example, we extract the implicit argument `n`, which is the length of the argument, which is used in the function.

```
length: Vect n ty → Int
length _ {n} = n
```

2.2 Proofs

Given that Idris is able to express types as first-class citizens, it is possible to increase the precision of the of the expressed types and, consequently, increase the precision of the implemented functions. It is possible through the type to express both properties about the equality of values and the relationships between data [1]. In the latter case, it is possible to express assumptions about the data which will be checked by the type checker when the functions are called at compile time. The predicate creates a contract between the arguments used by the function and it is enforced that all arguments satisfy these these assumption before it is even possible to invoke the function.

The way to do this in Idris is by explicitly making this requirement part of the type. For example, it may be desired to retrieve an element from a vector only in the case that such element actually belongs to the vector. Instead of returning a result of type `Maybe` to handle the case in which the element is not there, it is a more precise approach to describe in the type whether the element is contained in the vector or not. This way, we create a precondition for the function at compile time that the element is in fact contained in the vector. The following code is the `Elem` dependent type defined in `Data.Vect`. An instance of `Elem` proves that a value of type `a` is contained in a given vector of type `Vect k a` (i.e. vectors of any size with elements of the same type as the value provided).

```
data Elem : a → Vect k a → Type where
    Here : Elem x (x :: xs)
    There : (later : Elem x xs) → Elem x (y :: xs)
```

Instances of this data structure can be understood as proofs. `Here` proves that the value `x` is the first element of a vector, while `There` proves that, if `x` is in `xs`, then `x` must also be in `y :: xs`. Types like the `Elem` can be passed directly to functions and used to describe the relationships between data used in the function.

3 Expression language

3.1 Expression Abstract Syntax

The abstract syntax of the expression language ensures type-correctness by taking advantage of the dependent type system in the composition of expressions. The abstract syntax is expressed using the `Exp` type as shown below.

```
data TyExp = NatTy | BoolTy

data V : TyExp → Type where
  VNat : Nat → V NatTy
  VBool : Bool → V BoolTy

data Exp : Bool → TyExp → Type where
  SingleExp : (v : V t) → Exp False t
  PlusExp : (x : Exp a NatTy) → (y : Exp b NatTy) → Exp (a || b) NatTy
  IfExp : (cond : Exp a BoolTy) → (x : Exp b t) → (y : Exp c t) →
    Exp (a || b || c) t
  ThrowExp : Exp True t
  CatchExp : (x : Exp a t) → (h : Exp b t) → Exp (a && b) t
```

The `V` ('value') type is a way to wrap `Nat` and `Bool` values in the same type. In this way, the enumeration type `TyExp` works as a tagging mechanism to indicate whether a value is of type `Bool` or `Nat`.

The type of `Exp` depends on `TyExp`. This allows us, in the same way, to tag expressions with whatever type it will evaluate to. This is convenient when we want to impose type constraints on composite expressions such as `IfExp` and `PlusExp`. For example, the conditional expression of `IfExp` must be tagged with `BoolTy`; that is, it must evaluate to a value of type `Bool`. Furthermore, the type constraints state that the type of each of the branches must be the same. `PlusExp` also imposes constraints on the input expressions, as they should both be tagged with `NatTy`, indicating that both expressions must evaluate to values of type `Nat`.

The `Exp` type also depends on a boolean value used in relation to exception handling, as the expression language includes expressions for throwing and catching exceptions. The boolean value is used to indicate whether an expression can throw an unhandled exception or not. Note how `ThrowExp` can only be constructed with this boolean flag being true. The `IfExp` and the `PlusExp` constructors use logical ORs in order to convey whether any of the child expressions throw an exception, in which case the whole expression should throw an exception. The semantics of `CatchExp` is to return an expression `x` if it does not throw an handled exception, and execute a handler `h` otherwise, catching thus the exception. The dependent boolean flag for the entire `CatchExp` expression depends on the logical AND of both of the flags of `x` and `h`: it is true if and only if both expressions throw an unhandled expression.

An example of an expression returning a natural number that does not throw an unhandled exception is presented below. The expression is a `CatchExp`, with the main expression being an addition between an integer of value 60 and an expression that throws an exception. This expression has type `Exp True NatTy` (i.e. an expression returning a natural number that throws an unhandled exception). However, it can be catched with the handler, which simply returns the number 30. Therefore, the whole expression `expr` does not throw an unhandled exception, as the exception is catched by the handler.

```
expr : Exp False NatTy
expr = CatchExp (PlusExp (SingleExp (VNat 60)) ThrowExp)
                (SingleExp (VNat 30))
```

3.2 Expression Evaluation

The evaluation of expressions translates an expression depending on a `TyExp` into a value `V` depending on that same `TyExp`. This is indicated by the type of the `eval` function as seen below.

```
eval : (b = False) → Exp b t → V t
eval p (SingleExp v) = v
eval p (PlusExp x y) = evalPlusExp p x y
eval p (IfExp cond x y) = evalIfExp p cond x y
eval Refl ThrowExp impossible
eval p (CatchExp x h) = evalCatchExp p x h
```

Along with the expression to be evaluated, the function also requires a proof that `b` is `False` as an input, which ensures that the expression being evaluated cannot throw an exception. That is why the pattern match on `ThrowExp` tells us that this execution path is impossible. The reason why it is necessary to define the type as `(b = False) → Exp b t → V t` and not just `Exp False t → V t` is that `Exp False t` does not cover the `Exp (True && False)` case, for example, as seen in `evalCatchExp` below.

In order to see how this works in details, we inspect the function `evalCatchExp`:

```
1 evalCatchExp : (p : (a && b) = False) →
2                (x : Exp a t) →
3                (h : Exp b t) → V t
4 evalCatchExp {a = False} {b} p x h = eval Refl x
5 evalCatchExp {a = True} {b = False} p x h = eval Refl h
6 evalCatchExp {a = True} {b = True} Refl _ _ impossible
```

Here, the proof `p` is deconstructed, allowing us to inspect the implicit dependent type arguments so that we can tell which part of the expression will throw an exception. In line 4 we see that the value of `a` is false, which means that the first expression of the catch expression will not throw an exception. As a result, we evaluate only the first expression. In line 5, we see that the first expression will in fact throw an exception, so we evaluate only the second expression of the catch expression. This expression is also referred to as the 'handler'. The last line, line 6, states that if both expressions passed to the catch expression throw an exception, then it is not a well-formed expression, and, as seen the proof already expresses this in the type.

3.3 Code Abstract Syntax

The expression language is our source language, and the target language is the `Code` language, which describes operations on a stack.

This stack is described by the following types:

```
mutual
  StackType : Type
  StackType = List Ty

  data Ty = Han StackType StackType | Val TyExp
```

The `StackType` is a list of types, i.e., it describes the types of the elements on the stack. The elements on the stack can be one of two things: a `Val` that has a certain `TyExp` type indicating that a value of type `TyExp` is on the stack, or a `Han`, which is short for 'handler'. The handler relates to the catching of exceptions. The `Han` is a type that contains two `StackType`'s used to represent a specific portion of the Code which will be executed in case of an exception. The `El` function converts a type `Han` to the intermediate type `Code` used to represent instructions working on the stack.

The abstract syntax of the compiled code looks as follows:

```
data Code : (s : StackType) → (s' : StackType) → Type where
  PUSH : V tyExp → Code (Val tyExp :: s) s' → Code s s'
  ADD : Code (Val NatTy :: s) s' → Code (Val NatTy :: Val NatTy :: s) s'
  IF : (c1 : Code s s') → (c2 : Code s s') → Code (Val BoolTy :: s) s'
  THROW : Code (s'' ++ (Han s s') :: s) s'
  MARK : (h : Code s s') → (c : Code ((Han s s') :: s) s') → Code s s'
  UNMARK : Code (t :: s) s' → Code (t :: (Han s s') :: s) s'
  HALT : Code s s
```

The `Code` type depends on two values of the `StackType` type. The first is the type of the stack before the operation, and the second is the type of the stack after the operation. In this manner, we set constraints on the type of operations that can follow each other.

An important thing to realize about the `Code` type is the fact that each of the constructors, except for `HALT` and `THROW`, takes another `Code` as an argument. This is the operation that is to follow the operation we are currently constructing. That means that any sequence of operations must end with a `HALT` or a `THROW` operation. The sequencing of operations gives us the ability to convey many constraints on the relationships between the types of the stack between operations.

A good example of how this works is the `ADD` constructor. It takes a `Code` as an argument, which starts with a single natural number on top of the stack. This indicates that the stack after the addition operation has a natural number on the stack, which is the result of the `ADD` operation. The constructor itself returns a `Code`, which tells us that there must be two natural numbers on top of the stack before the operation is executed.

It is also worth studying the `THROW`, `MARK` and `UNMARK` constructors. The idea behind the `MARK` operation is to allow us to put a piece of code on the stack such that, if an exception is thrown, we can unwind the stack until the point where the handler was placed, and start executing the code of the handler, disregarding any values that were placed on the stack by the erroneous branch. The `UNMARK` operation is placed at the end of the potentially erroneous branch in order to remove the handler from the stack, if not exception is thrown. The `THROW` operation simply indicates that an exception is thrown. It does not accept another `Code` value because no operation can follow immediately after a thrown exception. `THROW` changes the control flow of the program, such that the `Code` placed on the stack by `MARK`, is instead executed.

4 Expression compilation

The compilation of expressions translates an expression into a sequence of `Code` elements. The main compilation function looks as follows.

```
comp : (b = False) → Exp b ty → Code (Val ty :: s) s' → Code s s'
comp p (SingleExp v) c = PUSH v c
```

```
comp p (PlusExp x y) c = compPlusExp p x y c
comp p (IfExp cond x y) co = compIfExp p cond x y co
comp p (CatchExp x h) c = compCatchExp p x h c
comp Refl ThrowExp _ impossible

compile : (b = False) → Exp b ty → Code s (Val ty :: s)
compile p e = comp p e HALT
```

Like the evaluation function, the `compile` function takes a proof `b` indicating that no exception is thrown as well as the expression to be compiled. Initially, the `HALT` instruction is added to the stack of instructions, so that when the stack is executed it will terminate. Like previously, the proof needs to be handled specifically for each case, and as such the logic has been moved into separate functions.

The `comp` function is called with the `HALT` instruction on the instruction stack. It takes a proof `b` that indicates that no exceptions are thrown, the expression to be compiled, and the intermediary code, `Code (Val ty :: s) s'`, which is the instruction stack here assumed to contain an element on top. The function returns a `Code s s'` constructed from one of the `Code` constructors, e.g. `Push`, which takes the expression and the `Code`.

Consider the function `compCatchExp` and the type of the function `compCatch` describing how the compilation of the `CatchExpr` happens:

```
compCatch : Exp b ty → Code (Val ty :: (s'' ++ (Han s s') :: s)) s'
    → Code (s'' ++ (Han s s') :: s) s'
-- ...

compCatchExp : (p : (a && b) = False) → (x : Exp a ty) → (h : Exp b ty) →
  (c : Code ((Val ty) :: s) s') → Code s s'
compCatchExp {a = False} Refl x h c = comp Refl x c
compCatchExp {a = True} {b = False} p x h c =
  MARK (comp Refl h c) (compCatch {s'' = []} x (UNMARK c))
compCatchExp {a = True} {b = True} Refl _ _ _ impossible
```

`compCatchExp` takes as argument a proof ensuring that no exceptions are thrown, the expression to evaluate, the handler and finally the accumulated stack of instructions so far. Next, one of two different events can happen. If a proof that `a = False` is provided, then control is immediately returned to the calling function using `Refl` as proof. Otherwise, if `a = True` and `b = False`, then it is known that an exception will be thrown that must be caught and handled. The way it is handled conceptually is by first placing the `MARK` instruction, then the code that will potentially throw an exception, and then finally the `UNMARK` instruction is added.

4.1 Code Execution

Once an expression has been turned into a sequence of `Code` instructions, we want to be able to execute these. As the type of the `Code` indicates, these operations are performed on a stack. The type of this stack is as follows:

```
El : Ty → Type
El (Han t t') = Code t t'
El (Val NatTy) = Nat
El (Val BoolTy) = Bool
```

```
data Stack : (s : StackType) → Type where
  Nil : Stack []
  (::) : El t → Stack s → Stack (t :: s)
```

The stack depends on an element of `StackType`, which allows us to indicate the type of element at each position of the stack. The most interesting thing about the `Stack` is the first agument to the `(::)` constructor. The `El` type function allows us to deduce the type of the element added to the stack. For instance, if an element of type `Nat` is added to the stack, according the the `El` function, the type of the top element of the stack will become a `Val NatTy`. This function allows an easy translation between the actual elements of the stack and their corresponding types in the `StackType`.

The execution function itself looks as follows:

```
exec : Code s s' → Stack s → Stack s'
exec (PUSH (VNat x) c) s = exec c (x :: s)
exec (PUSH (VBool x) c) s = exec c (x :: s)
exec (ADD c) (m :: n :: s) = exec c ((n + m) :: s)
exec (IF c1 c2) (True :: s) = exec c1 s
exec (IF c1 c2) (False :: s) = exec c2 s
exec THROW s = fail s
exec (MARK h c) s = exec c (h :: s)
exec (UNMARK c) (x :: h :: s) = exec c (x :: s)
exec HALT s = s

fail : Stack (s'' ++ Han s s' :: s) → Stack s'
fail {s'' = []} (h' :: s) = exec h' s
fail {s'' = (_ :: _)} (_ :: s) = fail s
```

The function accepts a `Code` which has a starting type of `StackType s` and must finish with a stack of type `StackType s'`. The second argument is a `Stack` which has exactly the `Stacktype s`, which is what our `Code` expects in order to be able to execute. The function outputs a `Stack` with `StackType s'`, which is also the final `StackType` of the `Code`.

All operations in this function manipulate the input stack in some way, that is expected by the input `Code`. The most interesting part, again, is the `THROW`, `MARK` and `UNMARK`. `MARK` places the handler argument `h` on the stack, which allows us to return to this place, should the execution of the `Code` throw an exception. `UNMARK` removes this handler. `THROW` calls the function `fail`, which expects an input stack whose `StackType` has an element of type `Han s s'` in it. The fail function is called recursively, gradually removing the top elements of the stack, until the type of the top element is of type `Han s s'`. At this point the execution continues with the `Code` that corresponds to this handler.

5 Statement language

Once we are able to express, compile and execute expressions, we can extend the language in order to be able to declare variables and assign them to expressions in form of *statements*. The pseudo-code example below is of a well-formed program in the language including four different statements.

```
x ← 1;
y ← 3 + x;
z ← True;
w ← if z then x else (x + y);
```

By following a type-driven approach, we can extend the type-correct and stack-safe nature of expressions to whole programs by only allowing well-formed programs to be executed. This means that expressions behave appropriately, but also variables are ensured to be defined before they can be referenced. The following example shows an invalid program that does not type-check because of variable `v` not being previously defined.

```
x ← 1;
y ← v + 2;
```

5.1 Expression Abstract Syntax

To use variables within expressions, we need to refine the `Exp` type and add a new constructor `VarExp` which allows referencing variables in expressions.

```
VariableId : Type
VariableId = String

data Exp : Bool → TyExp → List (VariableId, TyExp) → Type where
  VarExp : (vId : VariableId) → {auto p : Elem (vId, ty) env} → Exp False ty env
  SingleExp : (v : V t) → Exp False t env
  PlusExp : (x : Exp a NatTy env) → (y : Exp b NatTy env) → Exp (a || b) NatTy env
  IfExp : (cond : Exp a BoolTy env) → (x : Exp b t env) → (y : Exp c t env)
          → Exp (a || b || c) t env
  ThrowExp : Exp True t env
  CatchExp : (x : Exp a t env) → (h : Exp b t env) → Exp (a && b) t env
```

The type of `Exp` now encodes an *environment*, which contains all the variables and the types that the expression references. The environment is represented by a list of pairs assigning `VariableIds`, represented by strings, to `TyExps` (i.e. either `NatTys` or `BoolTys`). This information is necessary for the `VarExp` constructor as we need to proof that the variable with id `vId` already exists in the environment before we can use it. To prove that `vId` exists in the environment with type `ty`, we mark the proof as an auto-implicit parameter. The compiler can then derive the proof automatically hereby freeing the user from writing the proof manually. In addition, all the constructors have been modified to return the environment, which is simply passed around unaltered.

5.2 Programs

Expressions are now expressive enough to operate on variables existing in a given environment, which allows creating statements and chaining them together in the form of `Programs`.

```
data Program : List (VariableId, TyExp) → List (VariableId, TyExp) → Type where
  EmptyProgram : Program env env
  Declaration : (vId : VariableId) →
                (exp : Exp b t env) →
                {auto expExecutable : b = False} →
                (continuing : Program ((vId, t) :: env) env') →
                (Program env env')
```

```
  Assignment : (vId : VariableId) →
               (exp : Exp b t env) →
               {auto expExecutable : b = False} →
               (continuing : Program ((vId, t) :: env) env') →
               {auto prf' : Elem (vId, t) env} →
               (Program env env')
```

Programs allow variable declarations, which introduce fresh variables into the environment, and assignments, which assign new values to already declared variables. The type of `Program` depends on two environments. The first one lists the variables that the expression of the current statement requires to be evaluated, while the second one lists the defined and assigned variables of the entire program in an orderly fashion.

Both declarations and assignments take as arguments the variable ID `vId` of the current statement, its expression `exp` of type `t`, and a proof that `exp` does not throw unhandled exceptions. Both return a program that uses the variables defined in `env` and that defines or assigns the variables of `env'`. They require the continuing program, which has the current ID and its type on top of the required variables, and the `env'` untouched.

5.3 Program evaluation

In order to evaluate programs, we introduce a new data type, `ValuesEnv`, which allows keeping track of the different computed values of the executed statements of a program given an environment.

```
data ValuesEnv : List (VariableId, TyExp) → Type where
  EmptyValuesEnv : ValuesEnv []
  MoreValuesEnv : (vId: VariableId) → V ty → ValuesEnv envTy
                  → ValuesEnv ((vId, ty) :: envTy)
```

`ValuesEnv` can either be empty (`EmptyValuesEnv`) or have some elements by taking a variable ID, its computed value, and an existing environment. It returns an extended environment with the given variable ID and its type on top of the previous `ValuesEnv`.

To evaluate programs, we first need to evaluate the expressions it contains. For that, the type signature of the `eval` function has to be refined to deal with environments and variables. We do so by adding a `ValuesEnv` of the same environment of the expression `e` being evaluated, and the `VarExp` constructor to the case split analysis of `e`.

```
eval : (e : Exp b t tEnv) → (valuesEnv : ValuesEnv tEnv)
       → {auto prf : b = False} → V t
eval (VarExp vId {p}) valuesEnv = evalVarExp valuesEnv p
eval (SingleExp v) valuesEnv = v
eval (PlusExp x y) valuesEnv {prf} = evalPlusExp x y valuesEnv prf
eval (IfExp cond x y) valuesEnv {prf} = evalIfExp cond x y valuesEnv prf
eval ThrowExp _ {prf = Refl} impossible
eval (CatchExp x h) valuesEnv {prf}= evalCatchExp x h valuesEnv prf
```

Thanks to the `VarExp` constructor, we are certain that `vId` belongs to `tEnv` with type `ty`. By doing pattern matching with this proof and the `ValuesEnv` over the same environment, we can search the value of the variable in the environment and return it.

```
evalVarExp : (valuesEnv : ValuesEnv tEnv) → (p : Elem (vId, ty) tEnv) → V ty
```

```
evalVarExp (MoreValuesEnv _ val _) Here = val
evalVarExp (MoreValuesEnv _ _ valueEnv) (There later) = evalVarExp valueEnv later
```

In order to evaluate a program, we need the `ValuesEnv` of the variables it requires, and we return a `ValuesEnv` of all its statements. The evaluation of the empty program is the provided `valuesEnv`, as whatever was computed in the previous statements has to be returned. The evaluation of a declaration or an assignment is the result of adding the mapping of the current variable ID with the value of the current evaluated expression to the `valueEnv` recursively evaluated on the continuing program.

```
evPro : (p : Program env env') → (valueEnv : ValuesEnv env) → ValuesEnv env'
evPro EmptyProgram valueEnv = valueEnv
evPro (Declaration vId exp continuing) valueEnv =
  let evaluated = eval exp valueEnv in
  evPro continuing (MoreValuesEnv vId evaluated valueEnv)
evPro (Assignment vId exp continuing) valueEnv =
  let evaluated = eval exp valueEnv in
  evPro continuing (MoreValuesEnv vId evaluated valueEnv)
```

The syntax for evaluating whole programs can be further simplified, as we are certain that they do not require any variable to be defined before being evaluated, which results in the `Program [] env'` type. Users are expected to evaluate their programs using the following `evalProgram` function.

```
evalProgram : (p : Program [] env') → ValuesEnv env'
evalProgram p = evPro p EmptyValuesEnv
```

5.4 Compilation

The type presented in the previous section for compiled code (namely, `Code`) is not expressive enough to represent compiled programs. We need to introduce a `HeapType`, on which a later `Heap` will depend on in the execution phase. The idea is that the heap keeps track of the computed variables, so that `Code` operations can read or write into it. A `HeapType` has the same type as an environment: mapping variable IDs with their types.

```
HeapType : Type
HeapType = List (VariableId, TyExp)
```

The `Code` data type changes slightly in order to accommodate the reading and writing of variables. As explained in the previous section, the `StackTypes` encode how the types on the stack change from before to after the `Code` element has been executed. In the same way, it now depends on two values of the `HeapType`. These types encode which variables are available on the heap before the execution of the `Code` element, and after it.

The new operations added to the `Code` type are `STORE` and `LOAD`. `STORE` accepts a variable name for a value to be associated with. Its semantics is that it pops the top element off of the stack, and stores its value on the heap instead. `LOAD` also accepts the name of the variable we want to load, but it requires a proof that such variable belongs to the heap. If this is the case, the variable is then put on top of the stack. The rest of the constructors simply pass the heaps intact.

The function `El` has also to be modified to adhere to the new `Code` type.

```
El : Ty → Type
El (Han t t' h h') = Code t t' h h'
El (Val NatTy) = Nat
```

```
El (Val BoolTy) = Bool

data Code : (s : StackType) → (s' : StackType) → (h: HeapType) →
            (h': HeapType) → Type where
  STORE : (vId: VariableId) → (c: Code s s' ((vId, ty)::h) h') →
          Code ((Val ty)::s) s' h h'
  LOAD : (vId: VariableId) → {auto prf: Elem (vId, ty) h} →
         (c: Code ((Val ty)::s) s' h h') → Code s s' h h'
  -- ...
```

The `comp` function to compile expressions now changes: it adds the case of an expression being a `VarExp`, and convers it into a `LOAD` instruction in a straightforward way. It is worth noting that the variable type environment in which the expression is being evaluated is the same as the input type of the heap for the resulting `Code` element. In essence, this means that the expression being compiled can access the same variables that the `Code` element has access to at the beginning of its execution.

```
comp : (b = False) → Exp b ty tenv → Code (Val ty :: s) s' tenv h' →
       Code s s' tenv h'
comp p (VarExp vId) c = LOAD vId c
-- ...
```

Once the `comp` function is implemented, compiling programs is straightforward. Compiling the empty program returns the empty `Code` (i.e. `HALT`). For declarations and assignments, the expression of the current statement is compiled and connected with a `STORE` of the current statement's variable ID `vId` and the compilation of the continuing program.

The input and output `StackTypes` are both empty in the return type of the `compile` function. This comes from the fact that, by nature of expressions, the compiled code of an expression leaves a single value on the top of the stack. In the same way, a statement is compiled to a `STORE` instruction, which will remove a single element from the stack. Therefore, any compiled statement will start with an empty stack and end with an empty stack.

```
compile : (prog: Program tenv tenv') → Code [] [] tenv tenv'
compile (Declaration vId exp {expExecutable} continuing) =
  comp expExecutable exp (STORE vId (compile continuing))
compile (Assignment vId exp {expExecutable} continuing) =
  comp expExecutable exp (STORE vId (compile continuing))
compile EmptyProgram = HALT
```

5.5 Execution

In order to execute the new `LOAD` and `STORE` instructions, we need a `Heap` whose type depends on `HeapType`. It can be either empty or contain some value if a variable ID, its value and another heap is provided.

```
data Heap : (h : HeapType) → Type where
  HeapNil : Heap []
  HeapCons : (vId: VariableId) → V t → Heap h → Heap ((vId, t) :: h)
```

Now, to execute code, we require a heap in the `exec` function. Also, not only the modified stack is returned, but also a heap, which is modified while executing `STORE` instructions.

```
mutual
  lookup: Heap h → (p: Elem (vId, ty) h) → V ty
  lookup (HeapCons vId val tl) Here = val
  lookup (HeapCons _ _ tl) (There later) = lookup tl later

  partial
  exec : Code s s' h h' → Stack s → Heap h → (Stack s', Heap h')
  exec (LOAD {prf} vId c) s h = case lookup h prf of
    (VNat v) ⇒ exec c (v :: s) h
    (VBool v) ⇒ exec c (v :: s) h
  exec (STORE vId c) {s=(Val NatTy)::_} (x :: tl) h =
    exec c tl (HeapCons vId (VNat x) h)
  exec (STORE vId c) {s=(Val BoolTy)::_} (x :: tl) h =
    exec c tl (HeapCons vId (VBool x) h)
  -- ...
```

The relevant changes of the `exec` function are on the `LOAD` and `STORE` instructions. For the `LOAD`, the value of `vId` is looked up in the heap `h` given the proof that `vId` belongs to `h` in the function `lookup`, and the result is put on top of the stack. The execution continues recursively. For the `STORE`, the next command is executed with an extended heap by setting `vId` to its value `x`, which we are certain that exists on top of the stack thanks to the type signature of `STORE`.

6 Discussion

The behaviour of the `Assignment` and `Declaration` program constructors are remarkably similar. The only difference between them is that it is required that a variable had been declared beforehand in order to assign a value to it using an `Assignment`. Currently, however, there is no way to ascertain that the user actually uses the `Assignment` statement when re-assigning values. Consequently, either of them can be used to re-assign values to variables. Therefore, in our implementation, when a new variable is added to the heap, we always add the variable to the head of the `HeapType` and the heap itself. That means that, at the end of the program execution, the heap contains a history of every value a variable was assigned to during the execution. The most recent ones will the closest to the top of the heap. This is definitely not an optimal implementation, given the execution of large programs. It was initially implemented this way in order to simplify the types, and allow easier implementation. Ideally, it would be changed to an implementation that updates values on the heap if they are assigned to a new one. Another thing to note is that the declaration does not require any proofs that the variable name being added is not already assigned. It would be a natural extension forbidding adding a variable with a name which has already been declared.

With regards to the `Code` data type, we choose the solution of allowing almost all the `Code` constructors to take as input another `Code` element, which would be the representation of the instruction to be executed after it. This gives the `Code` type a recursive structure, which allows us to define naturally functions that recurse over the `Code` elements. This means that any `Code` sequence must be built from the bottom up as each element depends upon next one. An alternative way of structuring the code would be to define a concatenation operation, that would join together two `Code` elements into one. The reason for deciding against doing this is that the `IF` constructor will always need to contain two `Code` elements, corresponding to the two branches it can take. In this case, it is intuitive that either of the two `Code` elements will follow the `IF` code element. That means that if we apply this logic to the other constructors, then the nature of the `IF` constructor will apply naturally to the others as well.

Generally, the process of developing Idris code has been about working intensively with the types until a solution was accepted by the Idris compiler. This can yield code consisting of total functions which in terms of parameters are adhering to very strict typing. It can be, however, a frustrating process for the aspiring Idris developer and sometimes the first solution that compiled might be accepted even if it is sub-optimal.

7 Conclusion

During this project an implementation consisting of several coherent components has been implemented in the independently-typed language Idris. Initially a basic expression language was implemented that would read a defined abstract syntax and evaluate it to a natural number [3]. The language uses a compilation stage that compiles expressions into something resembling machine-code instructions that will work on a stack-machine. This was then extended with exception handling making it possible to both throw and catch exceptions within the abstract syntax [4].

Finally on top of this, a very small imperative language was implemented with the ability to declare variables and assign values to these while using the expression language to work on the variables.

Due to the independently-typed nature of the host language Idris, it is possible to guarantee typing through all steps of the implementation and therefore both type-correct and stack-safe machine code can be guaranteed.

References

[1] Edwin Brady. *Type-driven development with Idris*. Manning Publications Company, 2017.

[2] Edwin Brady. *Idris: A Language for Type-Driven Development*. https://www.idris-lang.org/. Accessed: 07 December 2020.

[3] James McKinna and Joel Wright. "A type-correct, stack-safe, provably correct, expression compiler in Epigram". In: *Journal of Functional Programming*. Citeseer. 2006.

[4] Mitchell Pickard and Graham Hutton. "Dependently-Typed Compilers Don't Go Wrong". In: *Proc. ACM Program. Lang* 1.1 (2020).