Typesafe by Definition for Languages with Explicit Deallocation

Robert Johan van den Berg

July 8, 2018

Abstract

A definitional interpreter is an interpreter which uses the semantics of its own host language to define those of its object language. Traditionally, a seperate type safety proof is used for such an interpreter. Using a "typesafe-by-construction" approach, where the typesafety is proven by expressing the type system of the object language in the type system of the host language is a new approach recently used for imperative languages.

In this paper a proof-of-concept is made to show that the technique of "typesafe-by-construction" can be also applied to interpreters for languages with explicit deallocation. This is done by making such an interpreter for a language called ML-dealloc, which is a basic version of ML extended with explicit allocation and deallocation. The interpreter is written in agda, which type system can be used to express ML-dealloc.

1 Introduction

For a programming language, the property of type safety is essential. A well-typed program should be executed properly. As (Milner, 1978) puts it: “well typed programs do not get stuck”, meaning that they always stay consistent and progressing. This does not mean that well-typed programs are always correct, only that they always stay within a programmer defined state.

A programming language semantic can be defined using an interpreter, as explained by (Reynolds, 1972). The language is defined as the interpreter which interprets it. That means that every property, including type safety that holds for the interpreter holds for the language. If the interpreter is typesafe, the language is also typesafe.

Traditionally type safety is established using a seperate proof. Recently, researchers have used the dependently typed language Agda (Norell, 2008) to construct definitional interpreters which guarantee type safety intrinsically. This approach is called "typesafe-by-construction".

It has been shown by (Poulsen et al., 2018) that proving type safety through a definitional interpreter can be done for imperative languages. These researchers did this for the simply typed lambda calculus with references, and middleweight Java. Neither of these languages have explicit deallocation. This paper applies this method to proving a weaker version of type safety for a language with explicit deallocation. This paper applies writing a "typesafe-by-construction" definitional interpreter for a language which has explicit deallocation.

1.1 Scope

The goal of this research is to show the application of the typesafe-by-construction method for languages with explicit deallocation. Definitional interpreters written with this method can be used as a reference for the language. In this reference, some properties of programs, like termination or well-typedness, can be proven. It is not intended for such an interpreter to be used in a production system, although it could be used as a reference for an interpreter or compiler that can be used in such a context.

1.2 Research Basis

This paper builds upon (Poulsen et al., 2018). Many techniques that are used in that paper for their interpreters are used here too. Their interpretation of the simply typed lambda calculus extended with references forms the basis of the interpreter for the language with explicit deallocation in this paper.

1.3 Type Safety Property

A type safety property is a property that hold for all valid programs within a language, as explained by (Pierce, 2002). This is related to the language itself and the guarantees it can make about programs written in it. The two properties of type safety are *progress*, meaning that the program continues executing, and *type-preservation*, meaning that the type of a value in a program does not suddenly change.

Type safety can take multiple forms for different programming languages. The strongest form of type safety requires every terminating program or subroutine to always return a value of the correct type. This can be problemetic in some languages.

Some languages have weaker type safety properties than others. A language with weaker type safety could have additional conditions for type safety to apply, like the correct usage of all internal pointers. Another way for type safety to be weakened is the possibility of run-time errors, like exceptions, as a result of a function call, instead of always having that function return a correct value. This is the approach that was chosen for this research to facilitate explicit deallocation in the object language.

1.4 Explicit deallocation Problem

The method of proving strong type safety by construction within a dependently typed host language has not yet been applied to a language with explicit deallocation, like C. This can be difficult to prove for strong type safety as (Pierce, 2002) claims for strong type safety that "... it is extremely difficult to achieve type safety in the presence of an explicit deallocation operation." The presence of a manually called deallocation function "weakens" the type safety property, as the usage of well-typed pointers might still result to run-time errors or undefined behavior if the pointer points to a (formally) deallocated value. This is known as the dangling pointer problem.

An example in pseudocode is given below. When q is allocated it might point to the same memory previously pointed to by p. As such, p now points to a different type of value than earlier, which is a violation of the strong type safety property, which guarantees consistent typing.

```
int* p; char* q; alloc p; *p = 42; dealloc p; alloc q; *q = 'h';
```

To solve this problem, a weaker version of type safety is used then the strictest possible variation. It is weakened by saying that a well-typed function might raise an error, which causes execution to terminate. This is different than always returning a value of the correct type, whhic is the strongest possible form.

1.5 Contributions

The research hypothesis is: "The technique of "typesafe-by-construction" definitional interpreters is applicable to languages with explicit deallocation."

This paper demonstrates the possibility of a typesafe by construction language with explicit deallocation, while keeping things as typesafe as the programming language C, which has explicit deallocation. C allows the possiblility of well-typed programs to raise errors during execution, for example, a segmentation fault.

Our contribution is proving this method works for languages with explicit deallocation, by writing a corresponding interpreter in Agda (Norell, 2008), a dependently typed language which can be used for this purpose. The interpreter is written for a minimal language, hereafter refered to as ML-dealloc, which has an equivalent of explicit deallocation. (Poulsen et al., 2018) also had a similar construct for null pointers.

1.5.1 Paper Organization

This paper has the following organization: Section 2 describes the methodology of this research, and why it proves that typesafe-by-construction interpreters work for languages with explicit deallocation. Section 3 describes the interpreter and its development in incremental steps. It also includes the usage of Agda lemmas for this interpreter. Section 4 describes the testing that was performed on the interpreter. Section 5 shows related work, and section 6 discusses the results of this research and the direction of future work, as well as alternative approaches to the explicit deallocation problem.

2 Methodology

This section contains the methodology used to prove that typesafety by means of typesafe-by-construction definitional interpreters works for languages with explicit deallocation.

To show that it is possible to apply the typesafe-by-construction method to languages with explicit deallocation, an interpreter is written using that method. The interpreter, written in Agda uses Agda's dependent type system to ensure that only well-typed programs are able to be executed.

2.1 Writing a Definitional Interpreter

By writing a definitional typesafe-by-construction interpreter for a small language that will be called ML-dealloc with explicit deallocation, we show that it is possible to use this method for such a language. This interpreter is described in section 3. Properties of this interpreter then hold for the entire language, assuming that the interpreter is a proper representation of the language. To check whether the interpreter is correct, some test programs are written in ML-dealloc, to show that every expression works as intended. These are described in section 4.

2.2 Validity

Writing an interpreter in Agda means that Agda's dependent typesystem can be used for the object lannguage as well. By defining the semantics of the object language in Agda, we can ensure certain properties about the programs expressed. If these properties do not hold, Agda does not run the interpreter with the program as input.

For example, the expression "let int x = true" tries to put a Boolean in place of an integer value for addition. Addition is only defined for integers, and as such Agda does not run the evaluation for this expression, should it arise.

2.3 Agda's Typesystem

Agda has a dependently typed typesystem. That is to say, types can depend on other types and values. This is used in our interpreter to ensure that types are preserved in expressions. It also has proving constructs that are able to be used to prove properties of execution, for example, that the store used only grows with new values. This is used in the interpreter to ensure

2.4 Language

As an object language a small version of ML extended with explicit deallocation is used, called ML-dealloc. It is a basic language with arithmetic, let bindings, branches and loops, extended with explicit allocation expressions. Pointers to values in the store can be allocated and freed at runtime.

In ML-dealloc, pointers are strongly typed, meaning that pointers can never refer to a value of an incorrect type.

There are no function calls in ML-dealloc.

2.5 Execution

Each program in ML-dealloc is an expression. Evaluation is done recursively. Mutable state is limited to the store.

To model state during execution, stores and environments are used. The environment contains all static variables in scope. The staic variables are immutable. The store contains all values pointed to by pointers. The store only grows during program execution. To model explicit deallocation despite that, any value in the store can be marked as *freed*, after which no referencing or mutation is allowed.

Because explicit deallocation is added to this language, some decisions must be made about how this works in ML-dealloc. The goal here is to be as close to the programming language C in this regard. This is to ensure that this research applies to "normal" programming languages with explicit deallocation.

A pointer on which the free() function is called still refers to the memory that it pointed to before the call. Dereferencing this pointer now results in an error. Trying to mutate the value referred to is also invalid.

In C a double free() on the same pointer also results in an error, provided that the memory is still out of bounds. If the same address is allocated again however, it might result in undefined behaviour, as the new value might have a different type. In our interpreter this last option cannot occur, for no address is reused in this way. As such, double deallocation could be made legal in our interpreter. However, the absense of an error in our interpreter indicates the absense of both an error and undefined behaviour in the actual program.

Allocation is also done explicitly, to keep as close to C as possible. That means that some pointers point to undefined values. The same rules for derefencing freed pointers apply, namely that it results in an error. Setting a value is allowed, after which the value is no longer undefined. Dereferencing is also permitted as normal.

3 Implementation

This section describes the definitional interpreter for ML-dealloc. This section starts with describing a basic interpreter for various expressions. In each following section, something is added to the interpreter, which is described together with required changes to the interpreter. Agda code for the interpreter, and the language representations is also included.

The complete final interpreter can be found in the repository at: https://gitlab.ewi.tudelft.nl/sv/typesafe-c-flat.agda/tree/master in the src/interpreter.agda file.

3.1 Basics

In this subsection the basic parts of ML-dealloc are explained.

3.1.1 Types and Values

The only types that are implemented in this subsection are boolean, integer and void. The Agda code for this part is:

```
data MType : Set where
  mIntType : MType
  mBoolType : MType
  mVoidType : MType

data MValue : (t : MType) → Set where
  mInt : (num : Int) → MValue mIntType
  mBool : (bool : Bool) → MValue mBoolType
  mVoid : MValue mVoidType
```

Note that every MValue is indexed by a MType. This is how the interpreter tracks the types of values, and later expressions, to ensure that they match. An expression requiring a integer but given a bool will not be evaluated because of this.

3.1.2 Expressions

Every program in ML-dealloc is basically one expression, which might contain other expressions, which are evaluated recursively. For example, addition is an expression requiring two other expressions.

constants which contain values.

branches in the form of if statements.

arithmetic with addition and multiplication.

```
data MExpression : MType → Set where
  mConstant : ∀ {type : MType} → (value : (MValue type)) → MExpression type
  mAdd : (l : MExpression mIntType) → (r : MExpression mIntType) → MExpression mIntType
  mMult : (l : MExpression mIntType) → (r : MExpression mIntType) → MExpression mIntType
  mIf : ∀ {t : MType} → (c : MExpression mBoolType) →
    (l : MExpression t) → (r : MExpression t) → MExpression t
  mSeq : ∀ {t : MType} → MExpression mVoidType → MExpression t → MExpression t
```

The expressions are typed, and can contain other expressions, which can have requirements for types as well.

3.1.3 Interpretation

Interpretation is done by the eval function given in the following code.

```
eval : ∀ {ty : MType} → MExpression ty  → MValue ty
-- constants
eval (mConstant val) = val
--- arithmatic
eval (mAdd l r) =
  bind (eval l) λ {
    (mInt nl) → bind (eval r) λ {
      (mInt nr) → mInt (nl + nr)}}
eval (mMult l r) = bind (eval l) λ {
  (mInt nl) → bind (eval r) λ {
    (mInt nr) → mInt (nl * nr)}}
-- branches
eval (mIf c l r) =
  bind (eval c) λ {
  (mBool true) → eval l;
  (mBool false) → eval r}
eval (mSeq a b) = eval b --There is no mutable state, so evaluating a does nothing yet.
```

A bind function is used to avoid with statements. This function is given in figure 4.

```
bind : ∀ {A B : Set} → (a : A) → (f : (A → B)) → B
bind a f = f a
```

3.2 Environments

To add variables to this language an environment is used. This environment is passed recursively by the eval function, as such its signature changes, as shown in fugure 6.

An environment for ML-Dealloc is defined as a list of values indexed by type, as shown in figure 5:

An environment is indexed by a list of types, called an MTypeContext.

```
MTypeContext = List MType
MEnv : MTypeContext → Set
MEnv x = All (λ t → MValue t) x
```

De-Bruijn style indices are used to refer to those environments in the MVar expression. The environment is a mapping from a list of types to values of those types, and the "a in b" pointer refers to it

When a variable is added to the environment the expression MAssign is used. MAssign is an expression which takes two expressions, adds the result of one expression to its environment to create a new environment, and interprets the other expression with the new environment. This value will never change after assignment This is equivalent to a let-binding.

An expression also refers to an environment. As such the signature for expression becomes:

Values added to the environment have limited scope. To illustrate this clearly, the following expression is written in pseudocode below:
(let (int a = 4) b); c
Where a is added to the environment of expression b, but not to that of expression c.

M̃Seq just executes two statements in succesion. As there is no way to pass any information from the first expression to the last, it is redundant right now, except to show the limited scope of environments. It has more relevant applications later when mutable state is added, which are described in the following subsections.

But note that the environment of the expression marked with b does not contain the value added in the expression marked with a.

3.3 Mutable Stores

In this section mutable stores are introduced. A value on the store is referenced by a pointer using a de-Bruijn style index. Values in the store are mutable. The Agda code describing these types and values is below.

```
mPointerType : (t : MType) → MType
```

To use de-Bruijn style indices, the store has to grow in a monotone way. That means that stores only grow and never shrink.

Agda needs to know the store only grows. To that end we use the following lemma for supersets. These supersets are stricter then normal, because the order of elements matter. This means that a superset always ends with it subset. In other words, no new elements are present after the start of the subset.

```
data _⊇_ : ∀ {A : Set} (a : List A) (b : List A) → Set where
  extension : ∀ {A : Set} {tl1 l2 : List A} → {hd : A} → tl1 ⊇ l2 → (hd :: tl1) ⊇ l2
  equal : ∀ {A : Set} → (l : List A) → l ⊇ l

-- This function returns a predicate that a ends with c from two predicates.
-- One that says a ends with b and one that says that b ends with c.
-- That is correct because a superset contains all subsets of a set.
⊇trans : ∀ {A : Set} {a b c : List A} → (a ⊇ b) → (b ⊇ c) → (a ⊇ c)

-- This function returns a predicate that an element in list a is also in a superset of a.
⊇complete : ∀ {A : Set} {x : A} {l1 l2 : List A} → (x ∈ l2) → (l1 ⊇ l2) → x ∈ l1
```

We also prove transitivity meaning that if a is a superset of b and b is a superset of c then a is a superset of c. We also introduce a small proof guerenteeing that the superset contains all elements of the subset.

Note that values now refer to a typecontext. This means that values and expressions need to refer to new typecontexts during execution if the store grows. We introduce weakening functions to make values and expressions refer to a new typecontext.

```
weaken-val : ∀ {ty : MType} {ctx ctx2 : MTypeContext} → (MVal ty ctx) →
  ( (ctx2 ⊇ ctx)) → (MVal ty ctx2)
```

```
weaken-env : ∀ {Γ ctx ctx2 : MTypeContext} → (MEnv Γ ctx) →
  ((ctx2 ⊇ ctx)) → (MEnv Γ ctx2)
weaken-expr : ∀ {ty : MType} {ctx ctx2 Γ : MTypeContext} → (MExpr ctx Γ ty) →
  (ctx2 ⊇ ctx) → (MExpr ctx2 Γ ty)
weaken-store′ : ∀ {ctx ctx2 ctx3 : MTypeContext} → (MStore′ ctx ctx2) →
  (ctx2 ⊇ ctx) → (ctx3 ⊇ ctx2) → MStore′ ctx ctx3
weaken-store : ∀ {ctx ctx2 : MTypeContext} → (MStore ctx) →
  (ctx2 ⊇ ctx) → MStore′ ctx ctx2
```

These functions rebind a language element to refer to a new typecontext.
The approach of using lemmas and weakening stores was also done by (Poulsen et al., 2018)
Stores are defined using the following code:

```
MStore : MTypeContext → Set
MStore ctx = All (λ t → MVal t ctx) ctx
```

But in a head-tail list construction, sometimes it becomes needed to have only a part of the store. This however produces a problem, because the typecontext of the value needs to be the same as that of the store.

We solve this by introducing partial stores:

```
MStore′ : MTypeContext → MTypeContext → Set
MStore′ x ctx = All (λ t → MVal t ctx) x
```

Just as environments, stores are indexed by a typecontext. This also has relevance for expressions:

Loops are also introduced, to make non-terminating programs possible. This was not done earlier because of the lack of mutable state, which would make using them ineffective,as without mutable state, a loop cannot terminate. To use them in a way Agda can use, Agda needs a way to ensure it does terminate when evaluated by Agda. To ensure this, we introduce a fuel counter, which decreases for every expression evaluated until it reaches zero or all subexpressions are evaluated. This is added to the evaluation function, after the example of (Poulsen et al., 2018)

Which makes our evaluation function signature:

Because programs can now run out of fuel, in which case they do not return a valid value, some sort of error mechanism needs to be introduced. In the above code, it refers to a result, which can be a success or a timeout.

```
data MResult  (A : Set) : Set where
  success : (x : A) → MResult A
  timeout : MResult A
```

To use them in a propagating manner, the bind and return functions are redefined to work with results instead of values:

```
bind : ∀ {A : Set} {B : Set} → MResult A → (A → MResult B) → MResult B
return : ∀ {A : Set} →   A   → MResult A
```

Note that the store is passed along as a result, together with a lemma guarenteeing that the new store contains all variables of the old store. These variables may have changed values, but their types and location stay the same.

The difference between a store and an environment is that a store does not have a scope. When a value is changed in the store, all following expressions use the changed value, no matter the scope.

The following expressions are added to make use of the store:

```
mDeref : {t : MType} → MExpr ct Γ (mPointerType t) → MExpr ct Γ t
mAssign : {t : MType} → MExpr ct Γ t → MExpr ct Γ (mPointerType t)
```

The M̃Assign expression adds a value to the store. The M̃Deref expression retrieves the value from the store, and the M̃Mut expression changes a value in the store. Explicit deallocation is added later.

The following new rules are added to the evaluation function:

3.4 Explicit Allocation and Deallocation

Adding explicit allocation and deallocation requires the possibility of freed memory. To do that, we make it so that the store does not hold values anymore, but rather StoreValues, which can either contain correct values, or a FreedVal, or an UnassignedVal. All of those are still refered to by types in the same way.

```
data StoreVal : (t : MType) → (ctx : MTypeContext) → Set where
  storeVal : {t : MType} {ctx : MTypeContext} → MVal t ctx   → StoreVal t ctx
  freedVal : (t : MType) (ctx : MTypeContext) → StoreVal t ctx
  unassignedVal : (t : MType) → (ctx : MTypeContext)   → StoreVal t ctx
```

To account for the possibility of code dereferencing out of bounds memory, we will add the following possibility to the Result, which is returned on either reading or writing to a deallocated value, or reading from an unintialized value. All of these are invalid in C.

```
error : MResult A
```

An error works similar to a timeout. Instead of meaning the interpreter is out of fuel an error means that memory was used in an invalid way.

The following expressions are added:

```
mAlloc : (t : MType) → MExpr ct Γ (mPointerType t)
mFree : {t : MType} → (MExpr ct Γ (mPointerType t)) → MExpr ct Γ mVoidType
```

And the evaluation function now has the following rules added:

```
-- add unassigned variable to strore
eval {ctx = ctx} (suc x) env store (mAlloc t) =
  return (t :: ctx , (mPointer ((t :: ctx)) (here refl) ,
    (add-to-store t store , extension (equal ctx))))
-- Free variable in store
eval {Γ = Γ} {ctx = ctx} (suc x) env store (mFree a) =
  bind (eval x env store a) λ {
    (ctx1 , val , store1 , p1) → bind (free-store val store1) λ store2 →
      return (ctx1 , ((mVoid ctx1) , store2 , p1))}
```

So now the language has explicit deallocation.

4 Testing

To test the interpreter, some test cases were made. They can be found in the repository at: https://gitlab.ewi.tudelft.nl/sv/typesafe-c-flat.agda/tree/master in the src/tests.agda file.

Testing was done in Agda. Proper interpreter functionality was checked by writing unit tests for specific expressions. No testing for typechecking was done, as the interpreter is typesafe-by-construction.

All expressions have at least one test case. 22 test cases were made in total.

An example of a basic unit test for the expression (3 == 3) is shown below. 17 for correct interpretation of expressions that return a success result. 1 to check if timeouts work properly, which checks whether the interpreter indeed returns a timeout. 1 to check if a loop can be made non-terminating and timeout in that case. 3 test were made for successful handling of expressions that throw errors.

```
test-eq-true-int : interp 100 [] [] (mEq (mConstant (mInt (+ 3) [])) (mConstant (mInt (+ 3) []))) ≡
  success ([] , (mBool true []))
test-eq-true-int = refl
```

In this test (mEq (mConstant (mInt (+ 3) [])) (mConstant (mInt (+ 3) []))) is the equivalent of ˝(3 == 3). Agda checks if the expression equals the expected success result. In this case, it expects a boolean value true.

Interp is the same as eval, except only the typecontext and the return-value are returned. The store and extension lemmas are purposely removed. Although the store is not passed along for the result, the typecontext of the store is part of the return value. As such, the typecontext of the store is also returned by the interp function.

The success means interpretation was a success. In this example, it contains a boolean value true, to indicate 3 equals 3.

The t̃est-eq-true-int = refl line means that Agda automatically sees that the interpreted expression returns the expected result. Other expressions might return an error or timeout, in which the same construction can still be used.

Other tests are done in a similar way. A test case saying interpreting an expression should return a certain result is made, after which it is proven with Agda through a refl statement.

5 Discussion

The analysis of this research shows that it is possible to write a definitional interpreter which incorporates typesafe-by-construction for a language with explicit deallocation. By marking memory as freed so that using that memory causes an error an effective model of deallocation can be made. As this model can be used to distinguish between errors and results, memory safety of programs can be reasoned about as well. One could use this model to prove a program does not cause an error at any point. Any extended version of this model, as well as other models incorporating this technique, will have this property as well.

Some programming languages have other sources of unsafe behavior. For example, pointer arithmetic is unsafe. This work cannot be used prove these techniques do not cause error inside of a program, as the language we implemented did not have those constructs. The technique we used considering memory (de)allocation might still be implemented in such research. However, it is not applicable to all such languages though. For example, reasoning about pointer arithmetic requires numerical pointers, while we used de-Bruijn indices. Using pointer arithmatic would also require knowing the size of the memory pointed to by a pointer, which basically means a seperate store per pointer inside the store, which would require additional work as well.

The technique we used was to mark memory as inaccesible, either because it contains no valid value yet (so it cannot be correct to access) or because it is freed. This could apply to other causes of inaccessibility as well. For example, for concurrent programming it could be useful to show that no memory is in use that is locked by a different thread. A security system that restricts certain code from accessing certain parts of memory, could also benefit from this.

5.1 Alternative Approaches

In this paper memory was marked in the store as freed, to indicate the possibility of inaccessible memory. There are other approaches possible, two of which were briefly considered. They are listed here.

A seperate list of Booleans which indicate availability was another possible solution. Doing so could have allowed a store of values directly instead of a unassigned/freed/actual value construction. However, it would have been more complicated to implement, because it means an extra list as argument for evaluation. Lemmas for extending that list would also have to be made. The adaptation to apply marked memory as locked or otherwise inaccesible to model concurrent programs would be preserved, however.

Another possible way of marking memory as inaccessible would be to actually remove the values from the list. This is similar to an actual free function. However, using De-Bruijn indices would be very hard. Firstly, our lemmas for only monotonely grow stores would be impossible to use, as values can now be removed. By removing values, and thus invalidating the "x in y" construction of pointers, one would have to correct those pointers during execution, which means checking the environment, the store, and all not yet executed expressions for such a value. This is very complex to do, as you basically rewrite the program during execution.

If another kind of index was used, like a numerical one, lemmas can be avoided. This causes another issue, however. A position might be freed, and that position could later be assigned a different value, of a possibly different type. If this was allowed, it could cause type-safety errors, or possible undefined behavior. It would be an error in both cases, as a programmer should not expect a freed pointer to suddenly point to another correct value. That means that we would need to check for this situation. This could be solved by not allowing memory to be reused, however that would remove the advantage of freeing memory, as no memory gets freed for the interpreter for later use. Another approach is to check every pointer in the program during execution and mark those who should not be usable. Again, this completely rewrites programs during execution. This would be a lot of work, and we could not reuse most of (Poulsen et al., 2018), because that uses de-Bruijn indices. Marking memory as locked for other reasons then deallocation, for example, concurrency, would also require the same work that was done to mark values as freed, as you cannot "unfree" memory in this approach.

6 Related Work

Definitional interpretation: (Poulsen et al., 2018) forms the basis for this research. This work on typesafe-by-construction type systems for definitional interpreters shows this technique is applicable to languages with mutable state and more complex languages then earlier used. This research was based on their simply typed lambda calculus. Their form of type-safety is stronger then the one proven here, since they did not include explicit (de-)allocation. Their implentation does not have an error system, except for the timeout mechanism with fuel that ensures terminating programs.

Dependently typed programming: For this research the dependently typed language agda was used. Other dependently typed languages which could be used are Coq (Coquand et al., 1984), and IDRIS (Brady, 2013), to which the techniques used here could potentially be reused.

Typesafety: The idea of typesafety, as in “well-typed programs cannot go wrong” comes from (Milner, 1978). Type-systems for programming languages are discussed in (Pierce, 2002). This also includes proving methodology of typesafety. Proving typesafety of systems in a syntactic way is discussed in (Wright and Felleisen, 1994).

Proving other properties of correctness: The usage of dependently typed programming can be used for other applications then proving the property of typesafety of a language. (Brady and Hammond, 2009) proposes using dependently typed languages like IDRIS to ensure non-functional properties like correct usage of resources like memory in a DSL with explicit (de)allocation, or the file system. To prove their “correct-by-construction” properties dynamically they use “holes” which are assumptions the programmer can make, but have to be verified at run-time. This results in stronger proofs of “soundness” at the expense of having to either assume some axioms or fill holes with proofs at or before runtime. It would be interesting to know how these techniques can be applied to existing languages through this paper.

7 Conclusions and Future Work

This work shows that the concept of “typesafe-by-construction” for definitional interpreters is applicable to languages with explicit deallocation. Although the demonstrated typesafety property is weaker than other type-safety properties, like in the work of (Poulsen et al., 2018), it is still as typesafe as the original C concerning deallocation, because C can have segmentation faults or undefined behaviour at runtime.

Future work might expand on the usage of typesafe-by-construction for languages with explicit deallocation to prove properties of programs written in those languages. Because we introduced the error result, work can also be done on proving that programs do not return such a result on execution. Also, the constructs of our store with marked freed memory might be able to be used to validate program properties concerning the amount of memory used. Other future work might expand on other unsafe memory constructs, like typecasting and pointer arithmetic.

References

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

Edwin Brady and Kevin Hammond. Ensuring correct-by-construction resource usage by using full-spectrum dependent types, 2009.

Thierry Coquand, Gérard Huet, et al. The coq proof assistant, 1984.

Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

Ulf Norell. Dependently typed programming in agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018.

John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740. ACM, 1972.

Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.