Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Formally Verified Insertion of Reference Counting Instructions

Bachelorarbeit von

Marc Huisinga

an der Fakultät für Informatik

$$\frac{\Gamma \Vdash e : \tau}{\Gamma, x : \mathbb{B} \Vdash e : \tau}$$

$$\frac{\Gamma, x : \mathbb{B}, x : \mathbb{B} \Vdash e : \tau}{\Gamma, x : \mathbb{B} \Vdash e : \tau}$$

$$\frac{\Gamma, \mathtt{x} : \tau, \mathtt{x} : \mathbb{O} \Vdash \mathtt{F} : \mathbb{O}}{\Gamma, \mathtt{x} : \tau \Vdash \mathtt{inc\ x;\ F} : \mathbb{O}}$$

$$\frac{\Gamma \Vdash \mathtt{F} : \mathbb{O}}{\Gamma, \mathtt{x} : \tau \Vdash \mathtt{dec\ x;\ F} : \mathbb{O}}$$

$$\frac{\Gamma \Vdash \mathtt{x} : \mathbb{O}}{\Gamma \Vdash \mathtt{ret\ x} : \mathbb{O}}$$

```
inductive linear (β : const → var → lin_type)
  : type_context → typed_rc → Prop
notation Γ ` ⊩ `:1 t := linear Γ t
| weaken {Γ : type_context} {t : typed_rc} (x : var)
  (t_typed : Γ ⊩ t)
  : (x : 𝔹) :: Γ ⊩ t
| contract {Γ : type_context} {x : var} {t : typed_rc}
  (x_𝔹 : (x : 𝔹) ∈ Γ) (t_typed : (x : 𝔹) :: Γ ⊩ t)
  : Γ ⊩ t
| inc_𝕆 {Γ : type_context} {x : var} {F : fn_body}
  (x_𝕆 : (x : 𝕆) ∈ Γ) (F_𝕆 : (x : 𝕆) :: Γ ⊩ F :: 𝕆)
  : Γ ⊩ (inc x; F) :: 𝕆
| inc_𝔹 {Γ : type_context} {x : var} {F : fn_body}
  (x_𝔹 : (x : 𝔹) ∈ Γ) (F_𝕆 : (x : 𝕆) :: Γ ⊩ F :: 𝕆)
  : Γ ⊩ (inc x; F) :: 𝕆
| «dec» {Γ : type_context} (x : var) {F : fn_body}
  (F_𝕆 : Γ ⊩ F :: 𝕆)
  : (x : 𝕆) :: Γ ⊩ (dec x; F) :: 𝕆
| ret {x : var}
  : (x : 𝕆) :: 0 ⊩ (ret x) :: 𝕆
```

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Sebastian Ullrich
Abgabedatum:	30. Oktober 2019

Zusammenfassung

Ullrich und de Moura [IFL 2019, noch zu erscheinen] definieren eine formale funktionale Programmiersprache für den Zwischencode des Lean 4 Theorembeweisers, sowie einen Compiler, welcher Referenzzählungsinstruktionen in den Zwischencode einfügt. In einem Appendix beschreiben Ullrich und de Moura außerdem eine Korrektheitsbedingung für diesen Compiler in der Form eines linearen Typsystems, und beweisen daraufhin, dass der Compiler immer wohl-getypten Code ausgibt. Wir formalisieren die Zwischencode-Sprache, den Compiler, das lineare Typsystem und einen Teil des Korrektheitsbeweises in Lean 3. Außerdem implementieren wir die Listen-Funktion `group by` und beweisen einige Lemmata, um dessen Korrektheit zu garantieren.

Ullrich and de Moura [IFL 2019, to appear] define a formal functional programming language for the intermediate representation (IR) of the Lean 4 theorem prover, as well as a compiler that inserts reference counting instructions into the IR. In an appendix Ullrich and de Moura also provide a correctness condition for this compiler in the form of a linear type system and then prove that the compiler always produces well-typed output. We formalize the IR language, the compiler, the linear type system and part of the proof of correctness in Lean 3. We also implement the list function `group by` and prove some lemmas to ensure its correctness.

Contents

1 Introduction

Most high-level programming languages use some sort of garbage collection to prevent errors induced by incorrect memory management. Functional programming languages in particular often make use of this abstraction. One such functional programming language is used in the meta-programming framework of the Lean theorem prover [de Moura et al., 2015; Ebner et al., 2017].

The Lean theorem prover enables users to verify the correctness of proofs and programs. To reduce the effort of formalizing programs, theorems and their corresponding proofs, Lean provides a meta-programming framework for writing proof and code automation. To implement these meta-programs, users write Lean code, which is then compiled to an intermediate representation (IR) and executed in a virtual machine.

In Lean 3, the performance of this meta-programming framework is not competitive with proof automation implemented in programming languages with more efficient compilers. One reason for this inefficiency is the use of a primitive reference counting garbage collection algorithm in the virtual machine [Ullrich and de Moura, 2019a].

During the development of Lean 4, a new reference counting garbage collection algorithm was designed to improve the performance of the IR [Ullrich and de Moura, 2019a]. This new algorithm reduces the amount of reference counting (RC) instructions by saving RC instructions associated with references that are guaranteed to be kept alive by the caller of a function. To alleviate the burden on the user, a heuristic is provided to infer whether a reference should be kept alive. Additionally, the reference count is used to implement a heuristic for performing destructive updates. This is especially beneficial to improving the performance and reducing the memory footprint of pure functional programming languages like Lean [Férey and Shankar, 2016].

For theorem provers, it is of special interest that both the algorithms used in the theorem prover and their associated proofs of correctness are without error. Ullrich and de Moura [2019b] present a proof of correctness for the new garbage collection algorithm of Lean 4. To ensure that the proof is faultless, this thesis implements a formalization of most of the proof in Lean 3. Specifically, instructions related to the destructive update optimization are not modeled, while the function application and constructor cases are still missing from the proof. During the development of the formalization, we implemented `group by`, which subdivides a list into its equivalence classes. We did not end up using it, but believe that it provides a good example for algorithm verification in Lean.

In chapter 2, we discuss different garbage collection techniques, provide some formal background on linear type systems, explain the prior work of Ullrich and

de Moura [2019a] and present the basics of the Lean theorem prover. In chapter 3, we formalize the IR, a linear type system to classify the correctness of reference counting, restrict the IR to a well-formed subset and define a compiler that inserts RC instructions. In chapter 4, we present the formalized proof of correctness. Finally, in chapter 5, we describe an implementation of `group by` and several lemmas related to it.

2 Background

2.1 Garbage Collection in Pure Programming Languages

Garbage collection algorithms used in programming languages often fall into either the category of tracing garbage collection or the category of reference counting.

Tracing garbage collection determines objects that are unreachable by reference from a special set of root objects, which usually consists of objects on the stack and global objects. When an unreachable object is detected, the object is freed. Reference counting attaches a counter to every object that tracks the amount of references to that object. When the counter reaches zero, the object is freed.

Naive reference counting exhibits a number of deficiencies:

- Circular references are never collected. Every object in the circle is being referenced at least once by another object in the circle, even if the circle itself is unreachable and could otherwise be freed [McBeth, 1963]. This is not an issue in Lean due to the inability to create objects with circular references [Ullrich and de Moura, 2019a].
- Every object also needs to store a reference count, which induces a constant space overhead for all objects in memory.
- In multi-threaded environments, the reference count has to be updated atomically, which induces a significant runtime overhead for every counter update [Choi et al., 2018].
- Pause times may be unbounded [Boehm, 2004].

Some of the listed deficiencies can be mitigated: updates need not be atomic if an object is only accessed from a single thread and the amount of RC instructions can be significantly reduced with heuristics that identify redundant RC instructions.

One such heuristic is provided by Ullrich and de Moura [2019a] for a pure and functional language. Parameters to functions are marked as either borrowed or owned. Borrowed parameters are guaranteed to be kept alive by the caller of the function and as such the reference count of these parameters does not need to be updated. Owned parameters behave like regular references and require updates to the reference count.

The reference count can then be used for an optimization that is especially useful in pure and functional languages. It is common that objects are freed right before

the creation of a new object of the same type, for instance when a pure function performs the equivalent of a mutation on an object by reading from the object to then create a similar object with updated parameters [Ullrich and de Moura, 2019a]. When the reference is marked as owned and the reference count is 1 right before its last use, it can be guaranteed that the object in question will not be used anymore after returning from the respective function. The memory of that object can then be reused by a different object with the same type - an optimization known as a destructive update.

2.2 IR Language

The target language of our formalization will be the purely functional intermediate representation (IR) language introduced by Ullrich and de Moura [2019a]. Let us first introduce an informal syntax to motivate the abstract syntax formalized in section 3.1. The informal syntax is mostly the same as the one provided by Ullrich and de Moura [2019a]. As an example, we will take a look at an implementation of a map function and ignore reference counting for now.

```
map f xs = case xs of
  (ret xs)
  (let x = proj₁ xs; let xs' = proj₂ xs;
   let fx = f x; let fxs' = map f xs';
   let fxs = ctor₂ fx fxs'; ret fxs)
```

The IR is untyped. For this example, we assume that there are two constructors corresponding to the commonly used list constructors nil and cons, both encoded by an index, where nil takes no parameters and cons takes the head and the tail of the list as parameters.

`case xs of` (*case*1) (*case*2) executes *case*1 if the value of `x` has been created with the first constructor, nil, and *case*2 if it has been created with the second constructor, cons. `ret xs` returns the value of the variable `xs`, in this case the empty list. `let x =` *expr*`;` binds the result of *expr* to the variable `x`. Expressions cannot be nested, i.e. every `let` only contains a single operation. $\texttt{proj}_i$ `xs` accesses the i-th field of `xs`, for example the head of `xs` for $i = 1$ or the tail of `xs` for $i = 2$ if `xs` is a cons cell. `f x` applies the variable `x` to the function stored in the variable `f`. In `let fxs' = map f s;`, `map` is a constant, not a variable: constants are names for functions that can be accessed globally. The mathematical function that maps constants to their corresponding function is called a program and henceforth denoted by δ. The third way to do function application is not shown in this example: it is also possible to partially apply functions with `let y = pap f` $\texttt{x}_1$ $\texttt{x}_2$ `...;`. If we want to store a function `f` in a variable `y`, we can simply write `let y = pap f;` Finally, $\texttt{ctor}_i$ $\texttt{x}_1$ $\texttt{x}_2$ `...` invokes the i-th constructor.

There are also two RC instructions not shown so far: `inc x;` and `dec x;`, where `inc` increments the reference count of `x` and `dec` decrements it. The reference count

is modeled as a multi-set of tokens. Additionally, we do not use a global token multi-set, but instead a token multi-set that is local to the function and only tracks a local portion of the reference count. It turns out that this view of reference counting is a lot easier to handle formally, since its correctness can be captured by a linear type system, while considering the correctness of reference counting with a global reference counter would require formalizing the heap. At the end of section 2.3 we will see that programs which are correct in regards to reference counting with a local token multi-set are also correct in regards to reference counting with a global reference counter.

Tokens can be created and consumed: `inc` and `let` create new local tokens, while `ret`, `dec`, as well as constant-, partial- and constructor application consume tokens of their parameters. Every function receives one token for each of its parameters. Intuitively, tokens are consumed when they are discarded via `dec`, encapsulated in a constructor or passed to a different function that maintains the token. Meanwhile, tokens are either created directly via `inc`, received from a different function or built from a constructor.

To further explain the semantics of reference counting, we consider a few more examples that have been introduced by Ullrich and de Moura [2019a]. Let us first consider a function that does not require any RC instructions to be correct:

```
id x = ret x
```

`id` receives one token for `x`, which is then consumed by returning it immediately. Next, let us consider a function that requires an `inc` instruction.

```
mkPairOf x = inc x; let p = ctor_1 x x; ret p
```

`mkPairOf` receives one token for `x`. To create `p`, x is consumed twice, so an additional token needs to be created with an `inc` instruction. Now we consider a function that requires a `dec` instruction.

```
fst x y = dec y; ret r
```

`y` is never used, so the one token received as a parameter needs to be explicitly discarded. Finally, we reconsider the map function shown earlier and insert RC instructions to remark a small technicality.

```
map f xs = case xs of
(ret xs)
(let x = proj_1 xs; inc x; let xs' = proj_2 xs; inc xs';
 let fx = f x; let fxs' = map f xs';
 let fxs = ctor_2 fx fxs'; ret fxs)
```

We need an additional `inc` instruction just to use `x` once: Instructions of the form `let x = proj_i y;` are treated differently from `let` instructions with other expressions and need to be incremented manually for reasons explained later, in section 2.3.

Ullrich and de Moura [2019a] also provide `reset x;` and `let x = reuse y in ctor_i z_1 z_2 ...;` instructions to implement an optimization that uses the reference count for destructive updates. We will not consider these instructions here for reasons

explained in section 4.3.

We will not need the semantics of the IR, either. Ullrich and de Moura [2019a] provide a full and formal description of the semantics.

2.3 Linear Type Systems

Ullrich and de Moura [2019b] use a linear type system to define the correctness of token-based reference counting. Regular structural type systems usually implicitly assume the Exchange, Weaken and Contract rules: Exchange changes the order of elements in the type context, Weaken discards elements of the type context and Contract duplicates elements of the type context. Henceforth Exchange will be assumed implicitly by considering the type context as a multi-set instead of a list, since preserving the order of elements in the type context will not prove to be useful. Linear type systems disregard Weaken and Contract: by not assuming Weaken and Contract, variables that are declared once can only be used exactly once as well. This is useful when reasoning about resources: without Weaken and Contract, we can use the type context to represent the multi-set of tokens. Depending on the piece of IR code, we can then consume tokens from the type context or create new tokens to add to the type context.

As mentioned in section 2.1, we might however want to opt-out of updating the token multi-set when dealing with borrowed variables, as long as we can still guarantee the correctness of reference counting. For this purpose, the type system given by Ullrich and de Moura [2019b] provides two types (τ): $\mathbb{O}$ for owned variables that behave linearly, and $\mathbb{B}$ for borrowed variables where we assume the Weaken and Contract rules to be able to duplicate and discard tokens freely. Parameters are marked as either owned or borrowed. We assume a function β that maps constants to a list of types that correspond to the types of the parameters of that function.

We will now explain the typing rules introduced by Ullrich and de Moura [2019b] so that we will later be able to discuss differences between this type system and our Lean-formalized type system. The type system provides typing rules for several objects: variables (x, y, z), expressions (e), function bodies (F), constants (c) and programs (δ). We begin with Var, Weaken and Contract.

$$\textsc{Var}\ \frac{}{x : \tau \Vdash x : \tau} \qquad \textsc{Weaken}\ \frac{\Gamma \Vdash e : \tau}{\Gamma, x : \mathbb{B} \Vdash e : \tau} \qquad \textsc{Contract}\ \frac{\Gamma, x : \mathbb{B}, x : \mathbb{B} \Vdash e : \tau}{\Gamma, x : \mathbb{B} \Vdash e : \tau} \qquad \textsc{Contract-F}\ \frac{\Gamma, x : \mathbb{B}, x : \mathbb{B} \Vdash F : \mathbb{O}}{\Gamma, x : \mathbb{B} \Vdash F : \mathbb{O}}$$

The Var rule ensures that variables can only be typed if there is exactly one token for that variable present in the type context. There is no Weaken rule for function bodies since it is not needed in any of the proofs provided by Ullrich and de Moura [2019b].

Next, we consider rules for RC instructions.

$$\textsc{Inc}\ \frac{\Gamma, \mathtt{x} : \tau, \mathtt{x} : \mathbb{O} \Vdash \mathtt{F} : \mathbb{O}}{\Gamma, \mathtt{x} : \tau \Vdash \mathtt{inc}\ \mathtt{x};\ \mathtt{F} : \mathbb{O}} \qquad \textsc{dec}\ \frac{\Gamma \Vdash \mathtt{F} : \mathbb{O}}{\Gamma, \mathtt{x} : \tau \Vdash \mathtt{dec}\ \mathtt{x};\ \mathtt{F} : \mathbb{O}}$$

Incrementing a variable always creates a new owned token, even if the variable is borrowed. This means that if the variable is borrowed and we increment it, we still need to ensure that the variable is consumed at least once. In the compiler defined in section 2.4, this case will never occur, as we will never insert RC instructions when a variable is borrowed. On the other hand, this ensures that as long as a program is correct in regards to the type system, it does not matter which variables end up being marked as borrowed: if a variable is marked as owned while it could instead be borrowed, we merely insert redundant RC instructions, since our type system guarantees that owned variables can always be used in the same contexts that borrowed variables can be used in. If a variable is marked as borrowed while it should instead be owned, the compiler will have to insert an `inc` instruction to create an owned token so that borrowed variables can be used in an owned context. Meanwhile, borrowed variables cannot be decremented.

We now introduce rules for `ret` and `case`.

$$\textsc{Ret}\ \frac{\Gamma \Vdash \mathtt{x} : \mathbb{O}}{\Gamma \Vdash \mathtt{ret}\ \mathtt{x} : \mathbb{O}} \qquad \textsc{Case}\ \frac{\overline{\Gamma, \mathtt{x} : \tau \Vdash \mathtt{F} : \mathbb{O}}}{\Gamma, \mathtt{x} : \tau \Vdash \mathtt{case}\ \mathtt{x}\ \mathtt{of}\ \overline{\mathtt{F}} : \mathbb{O}}$$

The Ret rule enforces that `ret` can only be typed if the context contains exactly a single token for the variable to return. In the expression rules for function application, we will see that the return value of function application needs to be owned. Meanwhile, `case` is typed by typing every single control path. Reading `x` to decide the control path is a non-consuming operation.

Next, we consider the Let rule.

$$\textsc{Let}\ \frac{\Gamma, \overline{x : \mathbb{B}} \Vdash \mathtt{e} : \mathbb{O} \qquad \Delta, \overline{x : \mathbb{O}}, \mathtt{z} : \mathbb{O} \Vdash \mathtt{F} : \mathbb{O}}{\Gamma, \Delta, \overline{x : \mathbb{O}} \Vdash \mathtt{let}\ \mathtt{z} = \mathtt{e};\ \mathtt{F} : \mathbb{O}}$$

When `e` is a saturated (full) constant application, some of the parameters can be marked as borrowed. It is sound to locally mark some owned variables as borrowed, as long as we can ensure that these variables are only borrowed in the scope of the function application. This is similar to Wadler [1990]'s `let!` construct. We will now see that the result of all expressions except for `proj` expressions need to be owned, and hence `e` should be owned.

Let us now take a look at the rules for different expressions.

$$\textsc{Const-App-Full}\ \frac{\overline{\Gamma \Vdash \mathtt{y} : \beta(\mathtt{c})}}{\overline{\Gamma} \Vdash \mathtt{c}\ \overline{\mathtt{y}} : \mathbb{O}} \qquad \textsc{Const-App-Part}\ \frac{\beta(\mathtt{c}) = \overline{\mathbb{O}}}{\overline{\mathtt{y} : \mathbb{O}} \Vdash \mathtt{pap}\ \mathtt{c}\ \overline{\mathtt{y}} : \mathbb{O}}$$

$$\textsc{Var-App}\ \frac{}{\mathtt{x} : \mathbb{O}, \mathtt{y} : \mathbb{O} \Vdash \mathtt{x}\ \mathtt{y} : \mathbb{O}} \qquad \textsc{Ctor-App}\ \frac{}{\overline{\mathtt{y} : \mathbb{O}} \Vdash \mathtt{ctor}_i\ \overline{\mathtt{y}} : \mathbb{O}}$$

Parameters in Const-App-Full are typed according to β. The return value needs to be owned: if functions were allowed to return borrowed variables, those variables might escape the context in which they are guaranteed to be kept alive. To understand that variables should never escape the context in which they are borrowed, consider a function `id` that immediately returns its borrowed parameter and its use in `let y = id x; dec x; ret y`, where `x` is owned and temporarily passed as a borrowed variable to `id`. The single token for `x` is used up by decrementing it. Since `x` and `y` are identical, `y` is decremented as well. If the `dec` call frees `x`, `ret y` will be an instance of use-after-free, despite this piece of code being typeable if we allow borrowed parameters to escape the context in which they are borrowed.

For Const-App-Part, parameters to `pap c` need to be owned: if a parameter is borrowed and we return the resulting `pap` value, the parameters would again escape the context in which they are borrowed. When fully applying this value later, we cannot consider the value as borrowed, because it is not necessarily borrowed in this new context. Neither can we just consider it to be owned, since we never paid a token for it being an owned parameter.

Consequently, for Var-App, the parameter `y` in `x y` needs to be owned, since functions can only be stored in variables by using `pap`, and `pap` parameters need to be owned, so `y` should be owned as well. Meanwhile, since `x` may already contain parameters that have been partially applied, `x` should be owned as well: the token for `x` is created from the tokens of owned parameters that have been partially applied to create `x`. Since those parameters are owned, we need to consume tokens for them, which we can only do by consuming a token of `x`, and this is only possible if `x` is owned.

Ctor-App is similar to Const-App-Part - the parameters to $\texttt{ctor}_i$ need to be owned. If we passed a borrowed variable to $\texttt{ctor}_i$ and returned the resulting value, the borrowed variable would escape the context in which it is marked as borrowed and we would not be able to guarantee that the borrowed variable is still alive by the time that it is accessed.

Lastly, projections are handled specially:

$$\textsc{Proj-Bor}\ \frac{\Gamma, \mathtt{x} : \mathbb{B}, \mathtt{y} : \mathbb{B} \Vdash \mathtt{F} : \mathbb{O}}{\Gamma, \mathtt{x} : \mathbb{B} \Vdash \mathtt{let\ y = proj}_i\ \mathtt{x;\ F} : \mathbb{O}} \qquad \textsc{Proj-Own}\ \frac{\Gamma, \mathtt{x} : \mathbb{O}, \mathtt{y} : \mathbb{O} \Vdash \mathtt{F} : \mathbb{O}}{\Gamma, \mathtt{x} : \mathbb{O} \Vdash \mathtt{let\ y = proj}_i\ \mathtt{x;\ inc\ y;\ F} : \mathbb{O}}$$

Projected variables can be borrowed if the constructor that is being projected from is borrowed as well. For this reason, $\texttt{let y = proj}_i\ \texttt{x; F}$ does not provide a token for `y` as in the other `let` rules. Meanwhile, owned constructors always project to owned variables, and as such need an explicit `inc` instruction to create the owned token explicitly. `y` cannot be borrowed, because it might outlive `x` and would thus not be kept alive.

The type system of Ullrich and de Moura [2019b] provides a third type $\mathbb{R}$, which is used for variables that can be used for destructive updates, as well as more rules to work with this type. In our shortened version of the type system, expressions and function bodies are always owned, while the expressions of Ullrich and de Moura

[2019b] might be of type $\mathbb{R}$ as well. We will not consider that part of the type system here.

Finally, constants are typed by typing the function body that belongs to the constant in a specific program, while programs are typed by typing all constants of the program.

$$\frac{\textsc{Const}\quad \delta(c) = \lambda\, \overline{y}.\ F \qquad \overline{y : \beta(c)} \Vdash F}{\delta \Vdash c} \qquad \frac{\textsc{Prog}\quad \forall c \in dom(\delta).\ \delta \Vdash c}{\Vdash \delta}$$

Ullrich and de Moura [2019a] define δ to be partial and require F in the Prog rule to be well-formed. Due to the partiality, only constants in the domain of δ need to be considered. We will later demand the well-formedness separately instead of requiring it in Prog.

Now that we have defined rules that ensure the correctness of reference counting with a local token multi-set, we might worry about the correctness of reference counting with a global reference counter, which is how reference counting would likely be implemented in practice. For this, Ullrich and de Moura [2019b] prove a semantics preservation theorem. Informally, it states that it does not matter whether we evaluate well-typed programs according to a formal semantics that disregards reference counting, or whether we evaluate according to a formal semantics that implements global reference counting. In order for the formal semantics that implements global reference to evaluate to a value, the program needs to halt and use-after-free may not occur. In this sense, the local type system guarantees that use-after-free will never occur with a global reference counter for halting programs. This does not yet prove that the type system implies freedom from memory leaks, though Ullrich and de Moura [2019b] refer to Chirimar et al. [1996] where this property is proved for a very similar type system and formal language. We will not formalize either proof here.

2.4 Insertion of RC Instructions

In the compiler given by Ullrich and de Moura [2019a], RC instruction insertion is provided as a mathematical function over the IR language introduced earlier. During the compilation, we will need a map β_l, initialized to $[\overline{\mathtt{y}} \mapsto \beta(c), ... \mapsto \mathbb{O}]$, that keeps track of whether variables are borrowed or owned, where parameters of a function $\lambda\, \overline{\mathtt{y}}.\ \mathtt{F}$ belonging to an arbitrary constant c are initially tracked according to β, while other entries default to being owned.

We begin by introducing the functions that prepend RC instructions to a specific

function body.

$$\mathbb{O}^+_{\mathtt{x}}(V, \mathtt{F}, \beta_l) = \mathtt{F} \qquad \text{if } \beta_l(\mathtt{x}) = \mathbb{O} \wedge \mathtt{x} \notin V$$
$$\mathbb{O}^+_{\mathtt{x}}(V, \mathtt{F}, \beta_l) = \mathtt{inc\ x;\ F} \qquad \text{otherwise}$$

$$\mathbb{O}^-_{\mathtt{x}}(\mathtt{F}, \beta_l) = \mathtt{dec\ x;\ F} \qquad \text{if } \beta_l(\mathtt{x}) = \mathbb{O} \wedge \mathtt{x} \notin \mathtt{FV(F)}$$
$$\mathbb{O}^-_{\mathtt{x}}(\mathtt{F}, \beta_l) = \mathtt{F} \qquad \text{otherwise}$$

$$\mathbb{O}^-(\mathtt{[]}, \mathtt{F}, \beta_l) = \mathtt{F}$$
$$\mathbb{O}^-(\mathtt{x}\,\overline{\mathtt{x'}}, \mathtt{F}, \beta_l) = \mathbb{O}^-(\overline{\mathtt{x'}}, \mathbb{O}^-_{\mathtt{x}}(\mathtt{F}, \beta_l), \beta_l)$$

$\mathbb{O}^+_x$ is always applied right before a variable is used in an owned context that consumes tokens. V represents the set of variables that are alive after incrementing `x`. Variables are not incremented if they are owned and not used anymore. This way, the last token of a variable is consumed on the last use of the variable. If the variable is instead borrowed, it needs to be incremented in order for there to be an owned token to use in the owned context. Meanwhile, variables are only decremented if they are owned and dead so that there are always tokens for those variables before their last use.

We will now consider the compilation of `ret` and `case`.

$$C(\mathtt{ret\ x}, \beta_l) = \mathbb{O}^+_{\mathtt{x}}(\varnothing, \mathtt{ret\ x}, \beta_l)$$
$$C(\mathtt{case\ x\ of}\ \overline{\mathtt{F}}, \beta_l) = \mathtt{case\ x\ of}\ \overline{\mathbb{O}^-(\overline{y}, C(\mathtt{F}, \beta_l), \beta_l)}$$
$$\text{where } \{\overline{y}\} = \mathtt{FV(case\ x\ of}\ \overline{\mathtt{F}}\mathtt{)}$$

Borrowed variables are incremented before being returned. At the start of every case, those owned variables that are not used in that specific case are decremented. To do this constructively, the set `FV(case x of` $\overline{\mathtt{F}}$`)` needs to be sorted into a list by some order. The compiler keeps the token count minimal; variables are only incremented right before their use and will only be duplicated before function application, where the duplicate tokens will immediately be used up. Because of this, we will only always have a single token for every variable at our disposal in between instructions, and that token is either consumed by passing it to some other function, returning it, using a constructor or calling `dec` once.

`proj` is treated specially once more:

$$C(\mathtt{let\ y = proj}_i\ \mathtt{x;\ F}, \beta_l) = \mathtt{let\ y = proj}_i\ \mathtt{x;\ inc\ y;}\ \mathbb{O}^-_{\mathtt{x}}(C(\mathtt{F}, \beta_l), \beta_l)$$
$$\text{if } \beta_l(\mathtt{x}) = \mathbb{O}$$
$$C(\mathtt{let\ y = proj}_i\ \mathtt{x;\ F}, \beta_l) = \mathtt{let\ y = proj}_i\ \mathtt{x;}\ C(\mathtt{F}, \beta_l[\mathtt{y} \mapsto \mathbb{B}])$$
$$\text{if } \beta_l(\mathtt{x}) = \mathbb{B}$$

Since $\beta_l(\mathtt{y}) = \mathbb{O}$ by default, β_l does not need to be updated in the case where $\beta_l(\mathtt{x}) = \mathbb{O}$.

To implement the compilation of the different expressions for function application, an auxiliary function C_{app} is defined:

$$C_{app}(\texttt{[]}, \texttt{[]}, \texttt{let z = e; F}, \beta_l) = \texttt{let z = e; F}$$
$$C_{app}(y\,\overline{y'}, \mathbb{O}\,\overline{b}, \texttt{let z = e; F}, \beta_l) = \mathbb{O}^+_y(\overline{y'} \cup \texttt{FV(F)}, C_{app}(\overline{y'}, \overline{b}, \texttt{let z = e; F}, \beta_l))$$
$$C_{app}(y\,\overline{y'}, \mathbb{B}\,\overline{b}, \texttt{let z = e; F}, \beta_l) = C_{app}(\overline{y'}, \overline{b}, \texttt{let z = e; } \mathbb{O}^-_y(\texttt{F}, \beta_l), \beta_l)$$

The second parameter designates the expected type of variables in the first parameter. When the parameter is expected to be owned, the variable passed to that parameter is incremented if it is borrowed or not used in the remaining parameter list or function body. When the parameter is borrowed, the variable passed to that parameter is decremented after the application, as long as it is owned and dead. In the case that the parameter is owned, the variable does not need to be decremented, since its last token would be consumed by the application.

C_{app} can now be used to implement the other cases:

$$C(\texttt{let z = c } \overline{\texttt{y}}\texttt{; F}, \beta_l) = C_{app}(\overline{\texttt{y}}, \beta(\texttt{c}), \texttt{let z = c } \overline{\texttt{y}}\texttt{; } C(\texttt{F}, \beta_l), \beta_l)$$
$$C(\texttt{let z = pap c } \overline{\texttt{y}}\texttt{; F}, \beta_l) = C_{app}(\overline{\texttt{y}}, \beta(\texttt{c}), \texttt{let z = pap c } \overline{\texttt{y}}\texttt{; } C(\texttt{F}, \beta_l), \beta_l)$$
$$C(\texttt{let z = x y; F}, \beta_l) = C_{app}(\texttt{x y}, \mathbb{O}\,\mathbb{O}, \texttt{let z = x y; } C(\texttt{F}, \beta_l), \beta_l)$$
$$C(\texttt{let z = ctor}_i\ \overline{\texttt{y}}\texttt{; F}, \beta_l) = C_{app}(\overline{\texttt{y}}, \overline{\mathbb{O}}, \texttt{let z = ctor}_i\ \overline{\texttt{y}}\texttt{; } C(\texttt{F}, \beta_l), \beta_l)$$

Ullrich and de Moura [2019a] describe an important subtlety when passing in the same parameter as both borrowed and owned, in that order:

$$\beta(\texttt{c}) = \mathbb{B}\,\mathbb{O},\ \beta_l = [\texttt{y} \mapsto \mathbb{O}],\ F' = \texttt{let z = c y y; F}$$

If `y` is alive in `F`, then `y` is guaranteed to be kept alive over the course of the function call. If `y` is however dead in `F`, we need to ensure that the second use of `y` does not consume `y` entirely. `c` is not aware that its second parameter is the same as the first one, and might decide to free the second parameter when it is no longer needed. This will however also free the first parameter, which `c` assumes is kept alive by the caller. Luckily, the compiler already prevents this issue. If `y` is dead in `F`, the variable is decremented after the application. Then, the use of `y` in the second parameter is no longer the last use of `y` in that function body, leading to `y` being incremented, which ensures that `y` will be kept alive over the entirety of the call. Specifically, for `F = ret z`, the full compilation process looks like this:

$$\begin{aligned}
&C(\texttt{let z = c y y; ret z}, \beta_l)\\
&= C_{app}(\texttt{y y}, \mathbb{B}\,\mathbb{O}, \texttt{let z = c y y; } C(\texttt{ret z}, \beta_l), \beta_l)\\
&= C_{app}(\texttt{y y}, \mathbb{B}\,\mathbb{O}, \texttt{let z = c y y; ret z}, \beta_l)\\
&= C_{app}(\texttt{y}, \mathbb{O}, \texttt{let z = c y y; } \mathbb{O}^-_{\texttt{y}}(\texttt{ret z}, \beta_l), \beta_l)\\
&= C_{app}(\texttt{y}, \mathbb{O}, \texttt{let z = c y y; dec y; ret z}, \beta_l)\\
&= \mathbb{O}^+_{\texttt{y}}(\{\texttt{y}, \texttt{z}\}, C_{app}(\texttt{[]}, \texttt{[]}, \texttt{let z = c y y; dec y; ret z}, \beta_l), \beta_l)\\
&= \mathbb{O}^+_{\texttt{y}}(\{\texttt{y}, \texttt{z}\}, \texttt{let z = c y y; dec y; ret z}, \beta_l)\\
&= \texttt{inc y; let z = c y y; dec y; ret z}
\end{aligned}$$

Finally, a full program δ is compiled by compiling every constant and immediately decrementing any parameter that is dead in the function body. The new program is called δ_{RC}:

$$\delta_{RC}(c) = \lambda\, \overline{\mathtt{y}}.\ \mathbb{O}^{-}(\overline{\mathtt{y}}, C(\mathtt{F}, \beta_l)) \text{ where } \delta(c) = \lambda\, \overline{\mathtt{y}}.\ \mathtt{F} \text{ and } \beta_l = [\overline{\mathtt{y}} \mapsto \beta(c), ... \mapsto \mathbb{O}]$$

2.5 Lean

Lean is an interactive theorem prover with dependent types, quotient types, support for tactic-style proofs and a universe hierarchy with an impredicative and proof-irrelevant `Prop` universe at the bottom of the hierarchy. Propositions are encoded as inductive types in the `Prop` universe and then proved by constructing a term of this type. Since `Prop` is proof-irrelevant, we can use classical mathematics for proofs [de Moura et al., 2015].

We provide some examples to illustrate the syntax of Lean 3. To prevent confusion with types already built-in into Lean 3, the names of the following types are indexed. We can define simple inductive types using the `inductive` keyword:

```
inductive nat₂ : Type
| zero : nat₂
| succ : nat₂ → nat₂
#check nat₂.succ (nat₂.zero)
```

Inductive types can be parameterized. In the following example, α is fixed in `list`.

```
inductive list₂ (α : Type) : Type
| nil : list₂
| cons : α → list₂ → list₂
#check list₂.cons 1 (list₂.nil ℕ)
```

Alternatively we could also explicitly name the parameters to `cons`. `Type*` is a placeholder for `Type u` with an arbitrary universe `u`. We can have Lean infer the type of `list₃ α` by not stating it.

```
inductive list₃ (α : Type*)
| nil : list₃
| cons (val : α) (tail : list₃) : list₃
```

If a parameter can be inferred from context, we can tell Lean to infer it by replacing the parameter with `_`.

```
#check list₃.cons 1 (list₃.nil _)
```

Inductive propositions can be defined similarly.

```
inductive false₂ : Prop

inductive true₂ : Prop
| intro : true₂

inductive and₂ (a b : Prop) : Prop
| intro : a → b → and₂

inductive or₂ (a b : Prop) : Prop
| left : a → or₂
| right : b → or₂
```

Inductive types with a single constructor can be defined more succinctly with `structure`. We can call the constructor of an inductive type with a single instructor using ⟨/-...-/⟩. Should Lean not be able to infer a type from context, we can always help it with type annotations.

```
structure and₃ (a b : Prop) : Prop := (h₁ : a) (h₂ : b)
#check and₃.mk (true₂.intro) (true₂.intro)
#check (⟨true₂.intro, true₂.intro⟩ : and₃ true₂ true₂)
#check (⟨⟨⟩, ⟨⟩⟩ : and₃ true₂ true₂)
```

Functions can be defined similarly to inductive types by pattern matching. Parameters left to : are fixed for the recursion, while parameters right to : are recursed on. Parameters can also always be marked to be left implicit with {/-...-/} if they should be inferred from context by default. Here, α is left implicit and fixed for the recursion, while we recurse on both `list₂` α parameters.

```
def append₂ {α : Type} : list₂ α → list₂ α → list₂ α
| (list₂.nil _) ys := ys
| (list₂.cons x xs') ys := list₂.cons x (append₂ xs' ys)
#reduce append₂ (list₂.cons 1 (list₂.cons 2 (list₂.nil _)))
  (list₂.cons 3 (list₂.cons 4 (list₂.nil _)))
```

We can still explicitly pass in parameters by prefixing the function with `@`.

```
#reduce @append₂ ℕ (list₂.cons 1 (list₂.cons 2 (list₂.nil _)))
  (list₂.cons 3 (list₂.cons 4 (list₂.nil _)))
```

Types can be restricted via type classes by demanding a specific type class instance using [/-...-/]. Lean will then infer a fitting instance in the context where the respective function is called. We will only use type classes, not define them.

```
def add {α : Type} [has_add α] (a : α) (b : α) : α := a + b
```

For definitions over `Prop`, we use `lemma` and `theorem`. The implementation of a lemma is the proof of the proposition designated by the type of the lemma.

```
lemma and_assoc₂ {a b c : Prop}
  : and₂ (and₂ a b) c → and₂ a (and₂ b c)
| ⟨⟨h_a, h_b⟩, h_c⟩ := ⟨h_a, ⟨h_b, h_c⟩⟩
```

We will make extensive use of Lean 3's effective standard library, mathlib [Avigad et al., 2019]. It provides basic data structures, definitions for many mathematical objects, a plethora of lemmas for these objects and a number of tactics.

2.6 Related Work

Chirimar et al. [1996] provide the basis for using linear type systems to write formal proofs related to reference counting. They prove that a local correctness predicate, implemented by a linear type system, has all the desired properties for reference counting with a global reference counter.

Moreau and Duprat [2001] implement a formal proof of correctness in the Coq theorem prover [The Coq Development Team, 2019] for a distributed reference counting algorithm. The algorithm is implemented by an abstract distributed machine model and a distributed protocol, whereas we use a formal programming language and a compiler. The correctness condition is global, while we use a local condition described by a linear type system.

Different kinds of tracing garbage collection have been formalized numerous times in multiple different interactive and automatic theorem provers [Burdy, 2001; Lin et al., 2007; Hawblitzel and Petrank, 2009; Sandberg Ericsson et al., 2017; Zakowski, 2017; Havelund, 1999; Jackson, 1998].

3 Formalized Definitions

In this chapter, we first develop a formalization of the abstract syntax that represents the IR. Then, we define what it means for a program with RC instructions to be correct in regard to reference counting. Next, we restrict the IR to a specific well-formed subset. Lastly, we implement the compiler that inserts RC instructions.

3.1 Abstract Syntax of the IR

For the formalization, we first define variables, constants and constructors as natural numbers:

```
def var := ℕ
def const := ℕ
def cnstr := ℕ
```

We choose natural numbers because we will later need decidable equality and a total order on variables, constants and constructors. For the purpose of executing and testing the compiler we might also define a pretty-printing procedure for the abstract syntax, in which case it is also useful that Lean already provides a pretty-printing procedure for natural numbers.

Next, we define the abstract syntax for expressions.

```
inductive expr : Type
| const_app_full (c : const) (ys : list var) : expr
| const_app_part (c : const) (ys : list var) : expr
| var_app (x : var) (y : var) : expr
| ctor (i : cnstr) (ys : list var) : expr
| proj (i : cnstr) (x : var) : expr
```

As in the informal syntax, expressions cannot be nested and will only be used in `let` statements. Function and constructor parameters are modeled as lists, not vectors with a size `n` dependent on `c` or `i`: it turns out that we will only need the arity of functions rarely, and as such it is easier to demand arity-correctness in a separate predicate instead of maintaining an invariant for the arity. Indeed, we will not need the arity of constructors at all.

We now implement function bodies:

```
inductive fn_body : Type
| ret (x : var) : fn_body
| «let» (x : var) (e : expr) (F : fn_body) : fn_body
| case (x : var) (Fs : list fn_body) : fn_body
| inc (x : var) (F : fn_body) : fn_body
| dec (x : var) (F : fn_body) : fn_body
```

As with function and constructor application, cases are modeled as a list. Every function body ends with a `ret`.

Function bodies are used in functions, which also contain the parameter list:

```
structure fn := (ys : list var) (F : fn_body)
```

Finally, we use a program type for mapping constants to functions so that `const_app_full` and `const_app_part` applications reference the correct functions.

```
def program := const → fn
```

For convenience, we also define a sum type that represents either `expr` or `fn_body`.

```
inductive rc : Type
| expr (e : expr) : rc
| fn_body (F : fn_body) : rc
```

We define coercions from both `expr` and `fn_body` to `rc` so that they need not be converted explicitly in contexts where `rc` is needed. `rc` will prove to be convenient when defining the type system, which we will define over `rc` instead of `expr` and `fn_body` separately.

Before we proceed, we implement some Lean-friendly notation for our abstract syntax. In Lean, notation can be used both for writing prettier Lean code and for prettifying the proof state displayed in an editor like Visual Studio Code. To implement a notation, every parameter to the notation needs to be delimited by some symbol to distinguish it from function application. This means that we cannot use exactly the same notation as in the informal syntax mentioned earlier, since syntax like `let fxs' = map f s;` does not delimit the parameters `map`, `f` and `s` with symbols. Additionally, `let` is already a reserved keyword in Lean. One alternative to notation for syntax like `f x` is a coercion to a function type, where `f` is coerced to a function `var → expr` via `expr.var_app f`. This approach has two key disadvantages: one cannot pattern match on the coercion and the coercion does not show up in the proof state. Instead, we choose the following notation:

```
notation c `⟦` ys `...` `⟧` := expr.const_app_full c ys
notation c `⟦` ys `...` `, ` `_` `⟧` := expr.const_app_part c ys
notation x `⟦` y `⟧` := expr.var_app x y
notation `⟪` ys `⟫` i := expr.ctor i ys
notation x `[` i `]` := expr.proj i x
```

```
notation x ` := ` e `; ` F := fn_body.let x e F
notation `case ` x ` of ` Fs := fn_body.case x Fs
notation `inc ` x `; ` F := fn_body.inc x F
notation `dec ` x `; ` F := fn_body.dec x F
```

`ret` is the only constructor for which we do not define any notation, using it directly. To provide an example of the new notation, we adapt the map example from before:

```
case xs of [
  (ret xs),
  (x := xs[1]; inc x; xs' := xs[2]; inc xs';
   fx := f⟦x⟧; fxs' := map⟦[f, xs']...⟧;
   fxs := ⟪[fx, fxs']⟫2; ret fxs)
]
```

In Lean, the full example with natural number encoding would look like this:

```
def map_body : fn_body := case 1 of [
   (ret 1),
   (3 := 1[1]; inc 3; (4 := 1[2]; inc 4;
   (5 := 2⟦3⟧; (6 := 0⟦[2, 4]...⟧;
   (7 := ⟪[5, 6]⟫2; ret 7)))))
]
def map : fn := ⟨[0, 1], map_body⟩
def id : fn := ⟨[0], ret 0⟩
def map_prog : program := λ c, if c = 0 then map else id
```

Luckily, we will never have to write full programs in this syntax.

Next we need to implement some technical machinery to ensure that we can both recurse over and use induction on values of type `fn_body`. Lean can automatically prove the well-foundedness of structural recursion. When this fails, Lean uses the `assumption` tactic, i. e. the fact that the object to recurse over is structurally smaller than the initial object already needs to be present in the proof context. When interacting with `fn_body.case x Fs`, we might however want to use higher-order functions like `map f Fs` to recurse over `F ∈ Fs` in `f`. Lean cannot automatically prove that this recursion is well-founded. Lean 3 also does not generate useful recursors for nested inductive types like `fn_body`, where the inductive type is nested inside of another type like `list`. Since we will only need `map f Fs` to write functions for `fn_body.case`, as well as a recursor on `fn_body`, we only implement the necessary machinery to prove the well-foundedness of recursive calls in `f` and a special recursor for `fn_body.case`.

A new `map` function is defined that passes a proof that the size of each element is smaller than the size of the list itself as a second parameter to the function being mapped to every element. With this parameter, Lean can prove that recursion over the element being mapped is well-founded.

```
def map_wf {α β : Type*} [has_sizeof α]
  (xs : list α)
  (f : Π (a : α), (sizeof a < 1 + sizeof xs) → β)
  : list β
```

It is also proved that the new `map_wf` behaves exactly like `map`.

```
lemma map_wf_eq_map {α β : Type*} [has_sizeof α]
  {xs : list α} {f : α → β}
  : map_wf xs (λ a _, f a) = map f xs
```

Lastly, the new recursor is defined, which provides an induction hypothesis for all `F ∈ Fs`.

```
def {l} fn_body.rec_wf (C : fn_body → Sort l)
  («ret» : Π (x : var), C (ret x))
  («let» : Π (x : var) (e : expr) (F : fn_body)
    (F_ih : C F), C (x ≔ e; F))
  («case» : Π (x : var) (Fs : list fn_body)
    (Fs_ih : ∀ F ∈ Fs, C F), C (case x of Fs))
  («inc» : Π (x : var) (F : fn_body)
    (F_ih : C F), C (inc x; F))
  («dec» : Π (x : var) (F : fn_body)
    (F_ih : C F), C (dec x; F))
  : Π (x : fn_body), C x
```

`map_wf`, `map_wf_eq_map` and `fn_body.rec_wf` were provided by Sebastian Ullrich.

Finally, we can now use this new machinery to define the free variables of a function body. To avoid duplicate variables, we use the `finset` type by Avigad et al. [2019] to represent the set of free variables.

```
def FV_expr : expr → finset var
| (c⟦xs...⟧) := xs.to_finset
| (c⟦xs..., _⟧) := xs.to_finset
| (x⟦y⟧) := {x, y}
| (⟪xs⟫i) := xs.to_finset
| (x[i]) := {x}

def FV : fn_body → finset var
| (ret x) := {x}
| (x ≔ e; F) := FV_expr e ∪ ((FV F).erase x)
| (case x of Fs) := insert x (finset.join
    (Fs.map_wf (λ F h, FV F)))
| (inc x; F) := insert x (FV F)
| (dec x; F) := insert x (FV F)
```

Since Avigad et al. [2019] do not yet provide a `finset.join`, we implement our own.

```
def join {α : Type*} [decidable_eq α] (xs : list (finset α))
  : finset α
```

We also prove an important lemma to work with `join` and add it to the set of lemmas of the `simp` tactic.

```
@[simp] theorem mem_join {α : Type*} [decidable_eq α]
  {x : α} {xs : list (finset α)}
  : x ∈ join xs ↔ ∃ S ∈ xs, x ∈ S
```

3.2 Type System for RC-Correct Programs

We begin by implementing the types of our type system.

```
@[derive decidable_eq]
inductive lin_type : Type
  | 𝕆 | 𝔹

structure typed_rc := (c : rc) (ty : lin_type)

@[derive decidable_eq]
structure typed_var := (x : var) (ty : lin_type)
```

Next, we define some notation for typed variables and typed `rc`.

```
notation x ` : `:2 τ := typed_var.mk x τ
notation xs ` [:] `:2 τ := (list.map (λ x, (x : τ)) xs
  : multiset typed_var)
notation xs ` {:} `:2 τ := multiset.map (λ x, (x : τ)) xs
notation c ` :: `:2 τ := typed_rc.mk c τ

abbreviation type_context := multiset typed_var
```

The type context of our type system is a multi-set by Avigad et al. [2019], which implicitly assumes the Exchange rule. Multi-sets still keep track of the count of every element in the multi-set, so we can still disregard Weaken and Contract for owned variables. As a quotient type over lists, they are also finite.

We will now introduce the formalized rules of the type system in the same order as in section 2.3 and explain the differences. We define the type system as an inductive type family that yields a proposition:

```
inductive linear (β : const → var → lin_type)
  : type_context → typed_rc → Prop
  notation Γ ` ⊩ `:1 t := linear Γ t
```

The notation is local, while β is fixed in both `linear` and the local notation. The type system is defined over `rc`; we will later consider predicates for constants and programs. In this type system, `rc` will always be owned. Despite that, we still

use `typed_rc` to allow for the possibility of different types: if we decide to add the `reset` and `reuse` instructions by Ullrich and de Moura [2019b] in the future, the type system will not have to be changed significantly to account for the fact that expressions can be of type $\mathbb{R}$ as well. Another important difference is that β is not a function `const → list lin_type`, but instead `const → var → lin_type`. In that sense, we assume that every variable has a type assigned by β, not just those that are passed in as parameters. This will not prove to be an issue, since the default type of those parameters in the list will never play a role - it will however simplify several arguments, since we would otherwise need an extra hypothesis that the parameter list has the same length as the one provided by β. Henceforth we assume `(Γ Δ : type_context)`, `(t : typed_rc)`, `(x y z : var)`, `(xs ys : list var)`, `(e : expr)`, `(F : fn_body)`, `(Fs : list fn_body)`, `(c : const)` and `(i : cnstr)`, using implicit parameters where possible.

Next, we implement `weaken` and `contract` rules:

```
| weaken (t_typed : Γ ⊩ t)
  : (x : 𝔹) :: Γ ⊩ t
| contract (x_𝔹 : (x : 𝔹) ∈ Γ) (t_typed : (x : 𝔹) :: Γ ⊩ t)
  : Γ ⊩ t
```

There is no `var` rule: we inline all the occurrences of `var` into the respective rules. For completeness, `weaken` and `contract` work for both expressions and function bodies. The `contract` rule uses `(x : 𝔹) ∈ Γ`. It is usually easier to prove that something is a member of Γ than to prove that Γ can be written in a certain way, since proving the latter often involves proving the former. Not every rule can be changed in this way, but we will still use this where possible.

```
| inc_ℕ (x_ℕ : (x : ℕ) ∈ Γ) (F_ℕ : (x : ℕ) :: Γ ⊩ F :: ℕ)
  : Γ ⊩ (inc x; F) :: ℕ
| inc_𝔹 (x_𝔹 : (x : 𝔹) ∈ Γ) (F_ℕ : (x : ℕ) :: Γ ⊩ F :: ℕ)
  : Γ ⊩ (inc x; F) :: ℕ
| «dec» (F_ℕ : Γ ⊩ F :: ℕ)
  : (x : ℕ) :: Γ ⊩ (dec x; F) :: ℕ
```

Instead of a single `inc` rule for a $\tau \in \{\mathbb{N}, \mathbb{B}\}$, there are now two `inc` rules for separate types, making it slightly easier to apply the rule.

```
| ret : (x : ℕ) :: 0 ⊩ (ret x) :: ℕ
| case_ℕ (x_ℕ : (x : ℕ) ∈ Γ) (Fs_ℕ : ∀ F ∈ Fs, Γ ⊩ ↑F :: ℕ)
  : Γ ⊩ (case x of Fs) :: ℕ
| case_𝔹 (x_𝔹 : (x : 𝔹) ∈ Γ) (Fs_ℕ : ∀ F ∈ Fs, Γ ⊩ ↑F :: ℕ)
  : Γ ⊩ (case x of Fs) :: ℕ
```

In `ret`, we inline the use of the `var` rule.

```
| const_app_full : list.map (λ y, y : β c y) ys ⊩ c⟦ys...⟧ :: 𝕆
| const_app_part : ys [:] 𝕆 ⊩ c⟦ys..., _⟧ :: 𝕆
| var_app : (x : 𝕆) :: (y : 𝕆) :: 0 ⊩ x⟦y⟧ :: 𝕆
| ctor_app : ys [:] 𝕆 ⊩ (⟪ys⟫i) :: 𝕆
| «let» (xs_𝕆 : (xs [:] 𝕆) ⊆ Δ) (e_𝕆 : Γ + (xs [:] 𝔹) ⊩ e :: 𝕆)
  (F_𝕆 : (z : 𝕆) :: Δ ⊩ F :: 𝕆)
  : Γ + Δ ⊩ (z ≔ e; F) :: 𝕆
```

After inlining, it is obvious that all the application rules look the same: The type context has to contain the parameter list and their types in order for the application to be typed. Lastly, we define the rules for projection.

```
| proj_𝔹 (x_𝔹 : (x : 𝔹) ∈ Γ) (F_𝕆 : (y : 𝔹) :: Γ ⊩ F :: 𝕆)
  : Γ ⊩ (y ≔ x[i]; F) :: 𝕆
| proj_𝕆 (x_𝕆 : (x : 𝕆) ∈ Γ) (F_𝕆 : (y : 𝕆) :: Γ ⊩ F :: 𝕆)
  : Γ ⊩ (y ≔ x[i]; inc y; F) :: 𝕆
```

For readability, we also define a non-local notation.

```
notation β `; ` Γ ` ⊩ `:1 t := linear β Γ t
```

Finally, we define the predicates for typing constants and programs.

```
inductive linear_const (β : const → var → lin_type)
  (δ : program)
  : const → Prop
  notation ` ⊩ `:1 c := linear_const c
| const (F_𝕆 : β; (δ c).ys.map (λ y, y : β c y) ⊩ (δ c).F :: 𝕆)
  : ⊩ c

notation β `; ` δ ` ⊩ `:1 c := linear_const β δ c

inductive linear_program
  (β : const → var → lin_type)
  : program → Prop
  notation ` ⊩ `:1 δ := linear_program δ
| program {δ : program} (const_typed : ∀ c : const, (β; δ ⊩ c))
  : ⊩ δ

notation β ` ⊩ `:1 δ := linear_program β δ
```

δ is now a total function as well.

3.3 Well-Formedness of the Pure IR

For some arguments, we will require the pure IR to follow a set of rules.

- Whenever a variable is used, it must have been defined beforehand.

- Variables that have been defined must be used later.
- Defined variables must be fresh.
- The length of the parameter list to a function must equal the arity of the function.
- Parameters to functions that are used in a `pap` must be marked as owned in β to guarantee that the owned variables partially applied to a constant can eventually be used to fully apply the function.

We provide well-formedness introduction rules for all instructions except for `inc` and `dec`: we will use the well-formedness predicate to restrict the input to our compiler when proving it correct, and we assume that there are no RC instructions in the input to the compiler. Otherwise, the compiler input might already be incorrect, and we would not be able to prove the correctness of the output. Since there are no well-formedness introduction rules for `inc` and `dec`, IR code that contains these instructions is not well-formed by default. Ullrich and de Moura [2019b] handle this differently: there are two separate function body structures, one with RC instructions and one without. Henceforth we will call the portion without RC instructions pure and enforce purity using our well-formedness predicate instead.

```
inductive fn_body_wf (β : const → var → lin_type) (δ : program)
  : finset var → fn_body → Prop
```

We also define a notation for this predicate:

```
notation β `; ` δ `; ` Γ ` ⊢ `:1 F := fn_body_wf β δ Γ F
```

In our predicate, Γ represents the set of variables that have already been defined. The rules of `fn_body_wf` are formalized exactly as defined by Ullrich and de Moura [2019b], the only slight difference is that the well-formedness rule for `let` has been inlined into the rules for the expression, i. e. instead of there being a `let` and a `const_app_full` rule, there is a `let_const_app_full` rule. While this does introduce some duplication when proving things about `fn_body_wf`, applying rules of `fn_body_wf` directly yields all the assumptions for the respective expression.

Once again we define predicates for constants and programs. Constants are well-formed if the function body of their corresponding function is well-formed and the parameter list contains no duplicates. Programs are well-formed if all their constants are well-formed.

```
inductive const_wf (β : const → var → lin_type) (δ : program)
  : const → Prop

notation β `; ` δ ` ⊢ `:1 c := const_wf β δ c
```

```
inductive program_wf (β : const → var → lin_type)
  : program → Prop

notation β ` ⊢ `:1 δ := program_wf β δ
```

The full definition of the well-formedness predicate can be found at `https://github.com/mhuisi/rc-correctness`.

3.4 Insertion of RC Instructions

We will now implement the compiler. $\mathbb{O}_x^+$, $\mathbb{O}_x^-$ and $\mathbb{O}^-$ are implemented in the same way as Ullrich and de Moura [2019a] implement them, with the exception that the recursion in $\mathbb{O}^-$ is replaced by a `foldr`.

```
def inc_𝕆_var (x : var) (V : finset var) (F : fn_body)
  (β_l : var → lin_type) : fn_body :=
if β_l x = 𝕆 ∧ x ∉ V then F else inc x; F

def dec_𝕆_var (x : var) (F : fn_body) (β_l : var → lin_type)
  : fn_body :=
if β_l x = 𝕆 ∧ x ∉ FV F then dec x; F else F

def dec_𝕆 (xs : list var) (F : fn_body) (β_l : var → lin_type)
  : fn_body :=
xs.foldr (λ x acc, dec_𝕆_var x acc β_l) F
```

`dec_𝕆` is difficult to handle formally. While decrementing, `FV F` is changing all the time because of the additional `dec` instructions, making it difficult to apply induction. If `xs` does not contain duplicates, every variable for which we call `dec_𝕆_var` will not appear in any of the `dec` instructions already added prior in this `dec_𝕆` call, meaning that we can substitute the `F` in `FV F` with the `F` passed to `dec_𝕆`. We do this by defining a second variant of `dec_𝕆` that inlines `dec_𝕆_var` and substitutes `F`:

```
def dec_𝕆' (xs : list var) (F : fn_body) (β_l : var → lin_type)
  : fn_body :=
xs.foldr (λ x acc,
  if β_l x = 𝕆 ∧ x ∉ FV F then dec x; acc else acc) F
```

We can then prove that `dec_𝕆` and `dec_𝕆'` are equal if `xs` does not contain duplicates:

```
lemma dec_𝕆_eq_dec_𝕆'_of_nodup {ys : list var} (F : fn_body) (β_l
   : var → lin_type)
  (d : list.nodup ys) : dec_𝕆 ys F β_l = dec_𝕆' ys F β_l
```

We will now use `dec_𝕆` to define the compiler and only fall back to `dec_𝕆'` in proofs.

We begin by defining the auxiliary function C_{app}.

```
def C_app : list (var × lin_type) → fn_body → (var → lin_type)
  → fn_body
| [] (z ≔ e; F) βl := z ≔ e; F
| ((y, t)::xs) (z ≔ e; F) βl :=
  if t = 𝕆 then
    inc_𝕆_var y ((xs.map prod.fst).to_finset ∪ FV F)
      (C_app xs (z ≔ e; F) βl) βl
  else
    C_app xs (z ≔ e; dec_𝕆_var y F βl) βl
| xs F βl := F
```

This definition uses another important design pattern: instead of taking two lists, one for the parameter list and one for their expected borrow status, we take a list over the product of both types. By construction, this ensures that both lists have the same length. With two separate lists, the alternative would be to either pass in a proof that the length of both lists is equal, or to use a dependent type that stores the length of the list and parameterize both types with the same length. In our definition of C, we will see that neither is needed, and we can construct the list over the product of both types without needing to prove anything, essentially guaranteeing that both lists have the same length by construction. `C_app` is also defined as a total function, not a partial one, doing nothing in cases where it would not be defined otherwise. Defining partial functions is possible in Lean, but requires more formal overhead, especially when using the function in a context where we would otherwise not have to worry about partiality. Since we will always ensure that `C_app` is defined by using the well-formedness predicate, we do not bother with defining `C_app` as a partial function.

We will now implement the function body compiler and consider `ret` and `case` first.

```
def C (β : const → var → lin_type) : fn_body → (var → lin_type)
    → fn_body
| (ret x) βl := inc_𝕆_var x finset.empty (ret x) βl
| (case x of Fs) βl :=
  case x of Fs.map_wf (λ F h,
    dec_𝕆 ((FV (case x of Fs)).sort var_le) (C F βl) βl)
```

In order to create a list from `FV (case x of Fs)`, we sort the `finset` according to the order induced by the natural numbers that back `var`. Next, we define the `proj` rule.

```
| (y ≔ x[i]; F) βl :=
  if βl x = 𝕆 then
    y ≔ x[i]; inc y; dec_𝕆_var x (C F (βl[y ↦ 𝕆])) βl
  else
    y ≔ x[i]; C F (βl[y ↦ 𝔹])
```

Here, variables in β_l do not default to being owned, but are instead marked as owned

whenever they are defined. This is not an issue, since we will never need to access the borrow status of variables that have not been defined yet. However, due to the fact that we do not need to keep track of which variables are owned by default and which variables have been assigned their borrow status by β, this simplifies arguments related to β_l.

```
| (z ≔ c⟦ys...⟧; F) βl :=
  C_app (ys.map (λ y, ⟨y, β c y⟩))
    (z ≔ c⟦ys...⟧; C F (βl[z ↦ 𝕆])) βl
| (z ≔ c⟦ys..., _⟧; F) βl :=
  C_app (ys.map (λ y, ⟨y, β c y⟩))
    (z ≔ c⟦ys..., _⟧; C F (βl[z ↦ 𝕆])) βl
| (z ≔ x⟦y⟧; F) βl :=
  C_app ([⟨x, 𝕆⟩, ⟨y, 𝕆⟩]) (z ≔ x⟦y⟧; C F (βl[z ↦ 𝕆])) βl
| (z ≔ ⟪ys⟫i; F) βl :=
  C_app (ys.map (λ y, ⟨y, 𝕆⟩)) (z ≔ ⟪ys⟫i; C F (βl[z ↦ 𝕆])) βl
| F βl := F
```

Here, our definition of β as `const → var → lin_type` instead of `const → list lin_type` enables us to ensure that the two lists passed to C_{app} are of the same size by construction. Once again, we do nothing for cases where `C` is not defined, i.e. cases where we would compile RC instructions.

Finally, we define the compilation of programs.

```
def C_prog (β : const → var → lin_type) (δ : program) (c : const)
    : fn :=
  let (βl, f) := (β c, δ c) in ⟨f.ys, dec_𝕆 f.ys (C β f.F βl) βl⟩
```

4 Formalized Proof

In this section, we provide a short overview of the proof that the compiler is correct in regards to the type system formalized in section 3.2 and provide some insights on technical decisions we made. Specifically, we formalize the proof of the following theorem:

```
theorem rc_insertion_correctness
  (β : const → var → lin_type) (δ : program) (wf : β ⊢ δ)
  : β ⊪ C_prog β δ
```

The original paper proof is provided by Ullrich and de Moura [2019b]. For the full formalization, we refer to `https://github.com/mhuisi/rc-correctness`. We will henceforth assume `(δ : program)`, `(β : const → var → lin_type)`, `(β_l : var → lin_type)`, `(x z : var)`, `(ys : list var)`, `(Γ Γ' Γ'' V : finset var)`, `(e : expr)`, `(F F1 F2 : fn_body)`, `(r : rc)` and `(τ : lin_type)`, using implicit parameters where possible. We begin by proving some lemmas about the types we defined in chapter 3.

4.1 Free Variables

Our first lemma captures the intuition that function bodies can only be well-formed if all its free variables have been defined beforehand:

```
theorem FV_sub_wf_context (h : β; δ; Γ ⊢ F)
  : FV F ⊆ Γ
```

The proof is a straight forward induction on `F` with a generalized Γ. For this induction, we use the new recursor `rec_wf` as well as the lemma `map_wf_eq_map` defined in section 3.1, which we will need whenever working with `C` or `FV`. Henceforth, when we say that we use induction on a function body `F`, we will always use `rec_wf` as the recursor. We also use a pattern that is applied in all following proofs: instead of using the `simp` tactic directly, we use the `squeeze_simp` tactic by Avigad et al. [2019] and then call `simp` with the list of lemmas printed by `squeeze_simp`. This allows more control over which lemmas are applied, since `simp` is occasionally simplifying terms too much, leading to terms that are either hard to read or hard to use in proofs. Additionally, it increases the stability of proofs: when `simp` is called somewhere in the middle of a proof, the output of `simp` depends on the lemmas it applied. If the global set of lemmas that simp can use changes, it may happen that proofs no longer type-check due to a change in the term produced by `simp`. We have also found that it significantly reduces the time it takes to check proofs in some cases. These

advantages come at a cost: simplification is now much more verbose.

Next, we want to prove that compiling a program does not change its free variables.

```
theorem FV_C_eq_FV : FV (C β F β_l) = FV F
```

This theorem is more general than necessary, since β_l and β are not required to be related in any way. To show this lemma, we need lower and upper bounds on the set of free variables over `inc_𝕆_var` and `dec_𝕆`:

```
lemma FV_inc_𝕆_var_eq_FV (h : x ∈ FV F)
  : FV (inc_𝕆_var x V F β_l) = FV F

lemma FV_sub_FV_dec_𝕆 : FV F ⊆ FV (dec_𝕆 ys F β_l)

lemma FV_dec_𝕆_filter : FV (dec_𝕆 ys F β_l)
  = ys.to_finset.filter (λ y, β_l y = 𝕆 ∧ y ∉ FV F) ∪ FV F

lemma FV_dec_𝕆_sub_vars_FV
  : FV (dec_𝕆 ys F β_l) ⊆ ys.to_finset ∪ FV F
```

`FV_dec_𝕆_sub_vars_FV` is a direct corollary of `FV_dec_𝕆_filter`, which is proved using `FV_sub_FV_dec_𝕆`. We then show `FV_C_eq_FV` using these lemmas by induction on `F` and generalizing β_l. The function application cases require an additional lemma:

```
lemma FV_Capp_eq_FV {xs : list (var × lin_type)}
  (heq : FV F1 = FV F2)
  (h : ∀ xτ ∈ xs, xτ.1 ∈ FV (z ≔ e; F1))
  : FV (C_app xs (z ≔ e; F1) β_l) = FV (z ≔ e; F2)
```

In the proof of `FV_C_eq_FV`, we sometimes use explicit recursors. This is due to the fact that the `cases` tactic in Lean 3 sometimes takes very long for relatively small problems. When this happens, we use the explicit recursors instead to prevent the code from taking too long to check.

4.2 Well-Formedness

In the proof of `rc_insertion_correctness`, we will often need to show the well-formedness of a program that is structurally smaller than the one we induct on in order to apply the induction hypothesis. To obtain a well-formedness predicate for this structurally smaller program, we prove a lemma that we will henceforth refer to as the "sandwich lemma".

```
lemma wf_FV_sandwich (Γ'_low : FV F ⊆ Γ') (Γ'_high : Γ' ⊆ Γ)
  (h : β; δ; Γ ⊢ F)
  : β; δ; Γ' ⊢ F
```

As seen in `FV_sub_wf_context`, Γ at least needs to contain the free variables of `F` in order to be the context of a well-formedness predicate. But Γ cannot get too

big either, since it must not contain variables that will be defined via `let` in `F`. The sandwich lemma ensures that given some upper bound Γ which guarantees the well-formedness of a function body `F`, all `Γ'` with `FV F ⊆ Γ' ⊆ Γ` also ensure the well-formedness of `F`. We prove `wf_FV_sandwich` as a corollary of a more general sandwich lemma and the fact that a context which only contains `FV F` already ensures well-formedness for `F`.

```
lemma wf_sandwich (Γ_sub_Γ' : Γ ⊆ Γ') (Γ'_sub_Γ'' : Γ' ⊆ Γ'')
  (hΓ : β; δ; Γ ⊢ F) (hΓ'' : β; δ; Γ'' ⊢ F)
  : β; δ; Γ' ⊢ F

lemma FV_wf (h : β; δ; Γ ⊢ F)
  : β; δ; FV F ⊢ F
```

We prove `wf_sandwich` by induction on `F` with generalized Γ, `Γ'` and `Γ''`, while we show `FV_wf` by induction on `h`. To use the induction hypothesis in `FV_wf`, we again need to use `wf_sandwich`, as well as `FV_sub_wf_context`. Due to the duplication in the definition of ⊢, many of the arguments needed in order to prove `wf_sandwich` and `FV_wf` are duplicated as well. We tame this duplication by using Lean's tactic combinators. Instead of proving every goal in order, we use the `any_goals` combinator to close any goal that can be closed by the provided argument. If multiple goals can be closed by the same argument, `any_goals` will close them all at once. This has the advantage of massively reducing duplication, but it also reduces the readability of the proof, since it is not immediately obvious which goals are solved by an argument.

4.3 RC Insertion Correctness

The actual induction for the proof of `rc_insertion_correctness` requires some initial setup work. Many arguments will work differently depending on whether a variable in the type context is owned or borrowed, so we split up the type context into two parts: the owned part, and the borrowed part. Specifically, we define the following:

```
let y𝕆 := filter (λ y, β c y = 𝕆) ys,
let y𝔹 := filter (λ y, β c y = 𝔹) ys,
```

We will later forget these definitions for a more general induction. Instead, we prove a few facts about `y𝕆` and `y𝔹` which will provide all we need in the induction.

```
obtain ⟨y𝕆_𝕆, y𝔹_𝔹⟩
  : (∀ y ∈ y𝕆, β c y = 𝕆) ∧ (∀ y ∈ y𝔹, β c y = 𝔹),
obtain ⟨y𝕆_sub_ys, y𝔹_sub_ys⟩ : (y𝕆 ⊆ ys ∧ y𝔹 ⊆ ys),
obtain ⟨nd_y𝕆, nd_y𝔹⟩ : multiset.nodup y𝕆 ∧ multiset.nodup y𝔹,
```

`nd_y𝕆` and `nd_y𝔹` can be proved from the lack of duplicates in `ys`, which is ensured by the well-formedness predicate. We then subdivide `ys` and Γ.

```
have ys_subdiv : ↑ys = y𝕆 + y𝔹,
have Γ_subdiv : ↑(list.map (λ (y : var), y : β c y) ys)
  = (y𝕆 {:} 𝕆) + (y𝔹 {:} 𝔹),
```

The proofs for all these lemmas use `filter_congr` and `map_congr` to rewrite the functions passed to `filter` and `map`. If we would instead rewrite normally, Lean 3 would also rewrite the inferred type class instances to match the rewritten term. It then occasionally infers different type class instances than the ones created by the rewrite, leading to errors that are difficult to debug, which can be mitigated by using `filter_congr` and `map_congr` instead. We do not use the `finset` implemented by Avigad et al. [2019] to carry around `nd_y𝕆` and `nd_y𝔹`, because we found that proving and maintaining the `nodup` statements manually to be easier than using functions for `finset` that maintain the invariant. For the most part, we do not want to treat `y𝕆` and `y𝔹` as `finsets`, which means that most effort in maintaining the invariant is wasted. We will also commonly need an upper bound on the set of owned variables.

```
have y𝕆_sub_FV : y𝕆.to_finset
  ⊆ FV (dec_𝕆 ((δ c).ys) (C β ((δ c).F) (β c)) (β c)),
```

Our goal now looks like this:

```
β; y𝕆 {:} 𝕆 + y𝔹 {:} 𝔹 ⊩ ↑(dec_𝕆 ((δ c).ys) (C β ((δ c).F) (β c)
  ) (β c)) :: 𝕆
```

To inductively decrement `(δ c).ys`, we prove the following lemma using `dec_𝕆_eq_dec_𝕆'_of_nodup` from section 3.4 by induction on `ys` while generalizing `y𝕆` and `y𝔹`:

```
lemma inductive_dec {y𝕆 y𝔹 : multiset var}
  (y𝕆_sub_ys : y𝕆 ⊆ ↑ys) (ys_sub_vars : ↑ys ⊆ y𝕆 + y𝔹)
  (d : list.nodup ys)
  (y𝕆_𝕆 : ∀ y ∈ y𝕆, β_l y = 𝕆) (y𝔹_𝔹 : ∀ y ∈ y𝔹, β_l y = 𝔹)
  (nd_y𝕆 : nodup y𝕆) (nd_y𝔹 : nodup y𝔹)
  (h : β; (filter (λy, y ∈ FV F) y𝕆 {:} 𝕆) + (y𝔹 {:} 𝔹) ⊩ F :: 𝕆)
  : β; (y𝕆 {:} 𝕆) + (y𝔹 {:} 𝔹) ⊩ dec_𝕆 ys F β_l :: 𝕆
```

We then prove the facts related to `y𝕆` from before for `y𝕆' := filter (λ (y : var), y ∈ FV (C β ((δ c).F) (β c))) y𝕆`, which is the set of owned variables after inductively decrementing `(δ c).ys`. Using `FV_C_eq_FV`, `FV_sub_wf_context` and `wf_FV_sandwich`, we also obtain `y𝕆'.to_finset ⊆ FV (δ c).F` and `(β; δ; to_finset y𝕆' ∪ to_finset y𝔹 ⊢ (δ c).F)`. We now forget the definition of `y𝕆'`, rename it to `y𝕆` and proceed to only use the lemmas we have proved about it, while also renaming those lemmas accordingly.

Our goal is `(y𝕆 {:} 𝕆) + (y𝔹 {:} 𝔹) ⊩ ↑(C β ((δ c).F) (β c)) :: 𝕆`. Before we begin with our induction, we generalize further: `(δ c).F` is replaced by an arbitrary `F` and `β c` is replaced by an arbitrary β_l. Due to the fact that Lean 3's `generalizing` keyword for induction only works on variables, not terms, we

generalize using the `generalize` tactic, rewrite and then forget the definition so it does not show up in our induction hypothesis when using induction.

```
generalize h : β c = β_l,
rw h at *,
clear h,
generalize h : (δ c).F = F,
rw h at *,
clear h,
```

Then, we begin the actual proof by induction on `F` while generalizing `y𝕆`, `y𝔹` and β_l. Right before the induction, our goal looks like this:

```
nd_y𝕆 : nodup y𝕆,
nd_y𝔹 : nodup y𝔹,
y𝕆_𝕆 : ∀ (y : var), y ∈ y𝕆 → β_l y = 𝕆,
y𝔹_𝔹 : ∀ (y : var), y ∈ y𝔹 → β_l y = 𝔹,
wf : β; δ; to_finset y𝕆 ∪ to_finset y𝔹 ⊢ F,
y𝕆_sub_FV : ∀ {|x : var|}, x ∈ y𝕆 → x ∈ FV F
⊢ β; (y𝕆 {:} 𝕆) + (y𝔹 {:} 𝔹) ⊪ ↑(C β F β_l) :: 𝕆
```

The `ret` case is proved using `y𝕆_sub_FV` and `nd_y𝕆`. Before closing the goal using `linear.ret`, we need to remove `y𝔹 {:} 𝔹` from the type context, which is done using an extra lemma that is shown by multiset induction on `y𝔹 {:} 𝔹`.

```
lemma inductive_weakening {ys : multiset typed_var}
  {y𝔹 : multiset var}
  (h : β; ys ⊪ r :: τ)
  : β; ys + (y𝔹 {:} 𝔹) ⊪ r :: τ
```

For `let`, we only prove the `proj` case; the other cases have yet to be shown. The induction hypothesis for all instructions i that look like i; `F` is always the same:

```
∀ {y𝕆 y𝔹 : multiset var} (β_l : var → lin_type),
  nodup y𝕆 → nodup y𝔹 →
  (∀ (y : var), y ∈ y𝕆 → β_l y = 𝕆) →
  (∀ (y : var), y ∈ y𝔹 → β_l y = 𝔹) →
  (β; δ; to_finset y𝕆 ∪ to_finset y𝔹 ⊢ F) →
  (y𝕆.to_finset ⊆ FV F) →
  (β; (y𝕆 {:} 𝕆) + (y𝔹 {:} 𝔹) y𝔹 ⊪ ↑(C β F β_l) :: 𝕆)
```

In order to apply this induction hypothesis, we typically need to make heavy use of our well-formedness predicate, `wf_FV_sandwich`, `FV_sub_wf_context` and `FV_C_eq_FV`. In fact, most of the work required to prove the `proj` case involves applying the induction hypothesis in different contexts and satisfying the conditions of the hypothesis.

The same is true for the `case` case: most of the proof works on satisfying the induction hypothesis. Before the application of the induction hypothesis, we also need to do some setup work with `inductive_dec`. `inc` and `dec` cannot be compiled, which is ensured by the well-formedness predicate.

During the proof of this theorem, we found that the original well-formedness predicate by Ullrich and de Moura [2019b] was insufficient to prove the correctness of the `reuse` instruction. Specifically, it did not guarantee the linearity that their type system demands, since we were allowed to `reset` the same variable `x` multiple times. While a `reset` token for `x` was then guaranteed to exist, it was not guaranteed to be unique, which would be required for linearity. Ullrich and de Moura [2019b] fix this issue by demanding that both defined and reset variables are fresh in a separate linear context Δ. We would now have to reconsider our theorems about well-formedness, since theorems like `FV_wf` do not hold anymore. A possible solution would be to consider two different sets of free variables, one for variables that appear in Γ, and one for variables that appear in Δ, but we chose to instead disregard `reset` and `reuse` for now.

5 Formalized Group By

In this chapter, we provide a short overview for an implementation of the `group by` function and lemmas we have proved about this implementation. The full proofs can be found at `https://github.com/mhuisi/rc-correctness`. Intuitively, `group by` subdivides a list into equivalence classes according to some equivalence relation. Originally, we intended to use `group by` to subdivide the type context in chapter 4, but it did not end up being the right abstraction, with the concrete subdivision being easier to handle than an application of `group by`. Despite that, we believe that it provides a nice and simple example for algorithm verification in Lean.

5.1 Group By on Lists

We begin by defining the `group` function, which represents `group by`. We do not explicitly pass the equivalence relation that we use to identify the groups to `group`, but instead require α to be a `setoid`, which is a type with an equivalence relation. This way, Lean's type class resolution can implicitly look up an appropriate equivalence relation for α.

```
def group {α : Type*} [p : setoid α] [decidable_rel p.r]
  : list α → list (list α)
| []        := []
| (x :: xs) :=
  have list.sizeof (filter (not ∘ (≈ x)) xs) < 1 + list.sizeof xs,
    from /- [...] -/,
  (x :: filter (≈ x) xs) :: group (filter (not ∘ (≈ x)) xs)
```

To help Lean with proving that `group` terminates, we prove that the filtered list is smaller than the initial list. When proving things about `group` by induction, regular induction on lists will not be sufficient, since `group (filter (not ∘ (≈ x)) xs)` may be significantly smaller than `xs`. To solve this issue, we define strong induction for lists.

```
@[elab_as_eliminator] def strong_induction_on
  {α : Type*} {p : list α → Sort*}
  : ∀ xs : list α,
    (∀ xs, (∀ ys, length ys < length xs → p ys) → p xs) → p xs
```

Our first lemma is that the equivalence classes of `group xs` contain exactly the same elements as `xs`:

```
lemma join_group_perm
  {α : Type*} [p : setoid α] [decidable_rel p.r] (xs : list α)
  : join (group xs) ~ xs
```

To prove this, we need another lemma:

```
lemma filter_append_not_filter_perm {α : Type*}
  (p : α → Prop) [decidable_pred p] (xs : list α)
  : filter p xs ++ filter (not ∘ p) xs ~ xs
```

Both lemmas are shown with strong induction on `xs`. We also employ a trick that we will use for all proofs involving permutations: Avigad et al. [2019] provide a lemma `list.perm_iff_count` that roughly states `l₁ ~ l₂ ↔ ∀ (a : α), count a l₁ = count a l₂` as long as the equality on α is decidable. Using classical mathematics in Lean, all propositions are decidable [Avigad et al., 2017, chapter 11.6], and we can grab a decidablity instance for equality on α out of thin air using `letI := classical.dec_eq α`, resulting in lemmas for `group` requiring the same hypotheses as `group` itself. `count a xs` is considerably easier to handle than `~` due to the fact that we do not need to explicitly construct permutations in order to prove `l₁ ~ l₂`. Henceforth we will always assume `(α : Type*)`, `[p : setoid α]`, `[decidable_rel p.r]` and `(xs : list α)` for every lemma, using implicit parameters where possible.

Next, we show that the elements of `group xs` are actually equivalence classes.

```
lemma group_equiv : ∀ g ∈ group xs, ∀ x y ∈ g, x ≈ y
```

We also prove that the elements of `group xs` are pairwise disjoint, even when using equivalence instead of equality.

```
lemma pairwise_equiv_disjoint_group
: pairwise (λ g1 g2 : list α, ∀ x1 ∈ g1, ∀ x2 ∈ g2, ¬(x1 ≈ x2))
  (group xs)
```

Since `pairwise R xs` ensures that `R` holds between elements at different indices in `xs`, the proof of this lemma involves proving that `group xs` does not contain any duplicates and that all `g1 g2 ∈ group xs` with `g1 ≠ g2` are disjoint.

We also classify the groups of `group xs` in two different ways: every `x ∈ xs` has a group assigned to it and every group `g_hd :: g_tl` can be written as `filter (≈ g_hd) xs`.

```
lemma filter_equiv_mem_group {x : α} (h : x ∈ xs)
  : filter (≈ x) xs ∈ group xs

lemma cons_eq_filter_of_group {g_hd : α} {g_tl : list α}
  (h : (g_hd :: g_tl : list α) ∈ group xs)
  : (g_hd :: g_tl : list α) = filter (≈ g_hd) xs
```

`cons_eq_filter_of_group` can then be used to prove that finding two equivalent elements in two separate groups is sufficient to ensure that the groups are already equal.

```
lemma group_eq_of_mem_equiv {g1 g2 : list α}
  (h1 : g1 ∈ group xs) (h2 : g2 ∈ group xs)
  (h : ∃ x ∈ g1, ∃ y ∈ g2, x ≈ y)
  : g1 = g2
```

5.2 Lifting Group By

Since the order of groups or elements in a group is rarely relevant, we want to lift `group` to `multiset` by first forgetting the order of groups and the order of elements in the group. After this, we forget the order of the input `xs`. To forget the order of the output of `group`, we define `group'`:

```
def group' {α : Type*} [p : setoid α] [decidable_rel p.r]
  (xs : list α) : multiset (multiset α)
```

In order to forget the order of `xs`, we need to ensure that the order of the input does not affect the output, i. e. that our definition is independent of representatives.

```
lemma group'_eq_of_perm (h : xs ~ ys) : group' xs = group' ys
```

To prove that the groups of `group'` are equal, we need to show that `group' xs` and `group' ys` are permutations of each other. We do this using the `list.perm_ext` lemma by Avigad et al. [2019], which states that `l₁ ~ l₂ ↔ ∀ (a : α), a ∈ l₁ ↔ a ∈ l₂` . This is only true if `l₁` and `l₂` do not contain duplicates, which we show using a more general lemma:

```
lemma nodup_map_coe_of_perm_nodup (xs : list (list α))
  (h : list.pairwise (λ a b, ¬(a ~ b)) xs
  : list.nodup (list.map coe xs : list (multiset α))
```

We finish the proof of `group'_eq_of_perm` by using a lemma which states that permuted inputs result in permuted groups.

```
lemma group_perm_of_perm (h : xs ~ ys)
  : ∀ gx ∈ group xs, ∃ gy ∈ group ys, gx ~ gy
```

We prove `group_perm_of_perm` using `join_group_perm`, `filter_equiv_mem_group` and `cons_eq_filter_of_group`.

Finally, we port the most relevant lemmas of `group` to `group'` and easily lift both `group'` and its corresponding lemmas to `multiset` using Lean's built-in quotient types.

```
def group {α : Type*} [p : setoid α] [decidable_rel p.r]
  (s : multiset α)
  : multiset (multiset α)
```

6 Conclusion

We have implemented definitions and a part of the proof of correctness by Ullrich and de Moura [2019a] in the Lean theorem prover and found an issue with the proof of the `reuse` case. This work and our implementation and verification of `group by` provide evidence that it is feasible to use Lean 3 for program verification, even when working with complex correctness conditions.

The full IR language by Ullrich and de Moura [2019a] with `reset` and `reuse` instructions has yet to be formalized, while the cases in the proof of correctness for function application are still missing. One could also imagine an implementation of the semantics preservation theorem by Ullrich and de Moura [2019b] in a theorem prover or a formalization of a proof which ensures that the linear type system guarantees freedom from memory leaks.

Bibliography

Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*, 2017. URL `https://leanprover.github.io/theorem_proving_in_lean/axioms_and_computation.html#the-law-of-the-excluded-middle`. [Online, accessed 12-October-2019].

Jeremy Avigad, Reid Barton, Mario Carneiro, Johan Commelin, Floris van Doorn, Sébastien Gouëzel, Simon Hudon, Chris Hughes, Robert Y. Lewis, Patrick Massot, and Scott Morrison. *Lean mathlib*, 2019. URL `https://github.com/leanprover-community/mathlib`. [Online, accessed 12-October-2019].

Hans-J. Boehm. The space cost of lazy reference counting. *SIGPLAN Not.*, 39 (1):210–219, January 2004. ISSN 0362-1340. doi: 10.1145/982962.964019. URL `http://doi.acm.org/10.1145/982962.964019`.

Lilian Burdy. B vs. Coq to prove a garbage collector. In *the 14th International Conference on Theorem Proving in Higher Order Logics: Supplemental Proceedings*. Citeseer, 2001.

Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, 1996. doi: 10.1017/S0956796800001660.

Jiho Choi, Thomas Shull, and Josep Torrellas. Biased reference counting: Minimizing atomic operations in garbage collection. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 35:1–35:12, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5986-3. doi: 10.1145/3243176.3243195. URL `http://doi.acm.org/10.1145/3243176.3243195`.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21401-6.

Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP):34:1–34:29, August 2017. ISSN 2475-1421. doi: 10.1145/3110278. URL `http://doi.acm.org/10.1145/3110278`.

Gaspard Férey and Natarajan Shankar. Code generation using a formal model of reference counting. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *NASA Formal Methods*, pages 150–165, Cham, 2016. Springer International Publishing. ISBN 978-3-319-40648-0.

Klaus Havelund. Mechanical verification of a garbage collector. In *International Parallel Processing Symposium*, pages 1258–1283. Springer, 1999.

Chris Hawblitzel and Erez Petrank. Automated verification of practical garbage collectors. *SIGPLAN Not.*, 44(1):441–453, January 2009. ISSN 0362-1340. doi: 10.1145/1594834.1480935. URL `http://doi.acm.org/10.1145/1594834.1480935`.

Paul B. Jackson. Verifying a garbage collection algorithm. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics*, pages 225–244, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-49801-8.

Chun-Xiao Lin, Yi-Yun Chen, Long Li, and Bei Hua. Garbage collector verification for proof-carrying code. *Journal of Computer Science and Technology*, 22(3): 426–437, May 2007. ISSN 1860-4749. doi: 10.1007/s11390-007-9049-z. URL `https://doi.org/10.1007/s11390-007-9049-z`.

J. Harold McBeth. Letters to the editor: On the reference counter method. *Commun. ACM*, 6(9):575–, September 1963. ISSN 0001-0782. doi: 10.1145/367593.367649. URL `http://doi.acm.org/10.1145/367593.367649`.

Luc Moreau and Jean Duprat. A construction of distributed reference counting. *Acta Inf.*, 37(8):563–595, May 2001. ISSN 0001-5903. doi: 10.1007/PL00013315. URL `http://dx.doi.org/10.1007/PL00013315`.

Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. A verified generational garbage collector for CakeML. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, pages 444–461, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66107-0.

The Coq Development Team. The Coq proof assistant, version 8.10.0, October 2019. URL `https://doi.org/10.5281/zenodo.3476303`.

Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: Reference counting optimized for purely functional programming, 2019a.

Sebastian Ullrich and Leonardo de Moura. *Counting Immutable Beans - Appendix*, 2019b. URL `https://leanprover.github.io/papers/beans_appendix.pdf`. [Online, accessed 12-October-2019].

Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

Yannick Zakowski. *Verification of a Concurrent Garbage Collector*. PhD thesis, 2017.

Erklärung

Hiermit erkläre ich, Marc Huisinga, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift