MPRI M2 Internship report : Micro-F* in F*

Simon Forest, under the supervision of Cătălin Hriţcu, INRIA Paris

August 21, 2015

The general context

The F* project. F* [1, 13] is a joint project between INRIA and Microsoft Research. It is aimed at achieving three different goals at once: First, it is an OCaml-like functional programming language. Second, it is a verification system, like Why3 or Dafny, that is able to prove semi-automatically that a program satisfies a certain specification. Finally, it is a proof assistant, like Coq and Agda, that allows users to manually construct proofs and checks that these proofs are correct. As a tool aimed at achieving these three goals simultaneously, F* is the state of the art; related tools include Zombie/Trellys [5] and HTT [9].

As a proof assistant, F* provides a combination of SMT-based automation and constructive proofs, which should lead to more maintainable proofs compared to tactic proofs in Coq. The proofs in F* are typed functional programs, whereas in Coq they are produced by untyped imperative meta-programs written in Ltac. F* also provides a semantic termination check based on a well-founded order on expressions; this check is much more convenient and flexible than syntactic termination checks e.g. in Coq. In particular, in F*, one can define custom termination metrics, using lexicographic orders for instance, which leads to natural definitions and concise proofs. However, F* became a proof assistant only very recently [13], so, at the moment, there are few examples of proofs written with it. The biggest example before this internship was the metatheory of System F-omega, which is only around 2000 lines of code.

Mechanized metatheory. The proofs of programming language metatheories are long and tedious, often not very deep and very easy to get wrong on paper. One can use proof assistants to help construct such proofs interactively and mechanically check these proofs, following the POPLMark spirit [3].

The research problem

The question I have studied is whether a rich fragment of F* can be formalized within F* itself. This is an interesting and challenging problem for two main reasons.

First, it gives a mechanized metatheory *of* F*, which is of first importance in order to gain confidence in the use of F* as a verification tool. Also, the current metatheory is very complex and easy to get wrong, and is still work in progress. Until now, there were only paper proof sketches with a lot of holes, with some parts that were not up-to-date, and with some serious errors. So there was a need to get all of this straight.

Second, it gives a mechanized metatheory *in* F*, which is also important in order to see what are the limits and the capabilities of the F* tool. Until now, there were not many examples of F* used as a proof assistant and all these examples are quite small. Moreover, since F* has features that are not present in standard proof assistants, it was an opportunity to discover interesting proof patterns that are specific to F*. Finally, the extensive use of F* for such a big proof was also an opportunity to find bugs in the implementation.

Other attempts like this one have been done with other proof assistants, for example, Coq in Coq [4]. But this was new to F*; in the self-certification effort for the previous (a lot less expressive) F* variant [12] the metatheory was done in Coq and the verification effort was limited to producing Coq certificates [11].

Contributions

We have mechanically checked in F* most of the progress and preservation proofs for a rich subset called micro-F*, featuring dependent types and kinds, type operators, subtyping, subkinding, semantic termination checking, and a weakest-precondition calculus based on predicate transformers. Micro-F* is defined using 7 mutually defined judgments (typing, subtyping, kinding, subkinding etc.) which are defined by a total of almost 100 rules. This mechanization became the largest case study of F* as a proof assistant. At the moment, the code is about 6500 lines of code.

During this process, we discovered effective proof patterns allowed by the features of F* that can be re-used for formalizing the metatheory of other programming languages. First, we defined parallel substitution in a

concise way using the lexicographic metrics of F* to show termination. Then, we proved a strong version of the substitution lemma [10], which is inspired by category theory [7, 8] and which entails many other usual lemmas, like weakening, bound strengthening, and the usual substitution lemma, that are usually proved one by one in usual proofs about lambda-calculus. We also simplified a lot the paper preservation proof by using a different induction scheme. Finally, we defined a generic inversion lemma, which entails all other inversion lemmas, and we used it extensively in the proof of progress. We have also used these patterns to simplify the proofs of the other mechanized metatheory examples in F*, such as STLC and $\lambda\omega$.

The paper proof sketch of micro-F* was also fixed and improved by the experience acquired by the mechanization: we fixed the proof of the substitution lemma, while finding a way to allow both CBV and CBN reductions in the operational semantics of pure expressions. Our solution involved significantly generalizing one of the logical validity rules of F*. We fixed the proof of the progress lemma by extending the definition of values and adding some new reduction rules. We changed the way recursive functions are defined, using a fixpoint operator instead of 'let rec', which is easier to handle in the mechanization. Finally, we fixed a lot of small technical bugs in the rules of the language that were hard to detect in the paper proof sketch.

This experiment also brought about improvements to the F* tool. Indeed, during the proof, we found two practical inconsistencies (ways to derive false from a consistent environment) that we filed as bugs and are now fixed. Also, features were added in order to make this mechanization easier to work on: an interactive mode to work easily on a part of the code without having to verify all the code every time, and a way to focus on cases of a proof in order for the check to be quicker. Moreover, our efforts informed several new features we plan to add to the F* tool that could have been useful in the proofs, like a tactic language and a way to print the proof context.

Arguments supporting its validity

The final result of this work is a mechanized proof of micro-F* in F*; this was considered valuable enough to be part of a paper submitted to POPL 2016 [13]. Even though the correctness of the proof still depends on the correctness of F* itself, machine-checked proofs are generally considered more reliable than paper proofs [3]. While we have already written the most interesting parts of the proof, we needed to admit some lemmas for technical reasons (e.g. current limitations in the F* tool), but we strongly believe that these lemmas are correct.

Conclusion and future work

To summarize, after proving such a big mechanized metatheory proof in F*, there are three main takeaways: First, F* can naturally express carefully designed lemmas and proofs, such that the category theory inspired substitution lemma, which can greatly simplify the proofs. In particular, we often used functions and proofs that have a non-trivial termination argument, often based on lexicographic metrics, and this allowed us to write the most general version of functions and proofs and saved us a lot of time. Second, the constructive proofs in F* are readable, easy to adapt to changes in definitions, and generally more maintainable than proofs made with a tactic language. Third, while SMT-automation relieved us from a lot of tedious code one would have to write manually in a prover like Coq, we found this kind of automation to be a mixed blessing. When not working correctly, there are few ways to get around the limitations. This explains why some proofs of this project could not be fully completed. In order to address this problem, a tactic language is planned to be added to F*, in order to be able to write the proofs manually when the SMT-automation fails.

Several aspects of future work have been identified: Concerning F* as a proof assistant, adding an Mtac-inspired tactic language would increase completeness of the tool. Also, having a way to debug the proofs would be useful. Finally, having an Agda-style editor mode would make writing proofs more convenient.

Concerning the metatheory of F*, a lot of work still remains to be done. In the short term, it would be nice to finish the last holes in our soundness proof for micro-F* and to extend this proof to bigger F* fragments (e.g. the lattice of effects, higher-order state, polymorphism, etc). In the medium term, the main goals are formally proving normalization and consistency for F*, and ideally machine checking these proofs (they are currently done only on paper and only for a small subset of F*), and also formalizing and proving the soundness of the SMT encoding. In the long term, we are aiming for the self-certification of F* in F*, while still producing certificates that can be checked by Coq in order to build strong confidence in the proofs [11].

1 Technical background

1.1 Overview of F* as a proof assistant

As a programming language, F* is an ML-like functional programming language with dependent types and kinds, type operators, subtyping, subkinding, refinement types, inductive types, a lattice of effects and, most importantly, a weakest precondition calculus based on predicate transformers. The syntax F* expressions is basically the same as Caml-light or F#. F* differs from these languages at the type and kind level. In particular F* allows programs to be given total or partial correctness specifications and includes a weakest precondition calculus that produces proof obligations that are discharged using an SMT solver.

The type syntax of F* allows type-level lambda abstractions on expressions and types, and also type-level applications on expression and types.

In order to be able to write proofs with it, the type system includes a logical part with a validity judgment on formulas. In this setting, formulas are types and types are formulas, which let the user mix the two when making proofs. In particular, the type syntax includes usual logical operators like $\Longrightarrow, \wedge, \vee, \forall$.

In the next paragraph, we present the weakest precondition calculus used in F*.

Weakest preconditions. Dijkstra [6] defines the semantics of programs as predicate transformers: functions that produces preconditions on the input of the program from postconditions on the output of the program. This approach is inspired from Hoare logic, where the postcondition holds after the execution of the program on a state where the precondition is true. With predicate transformers, there are no specific precondition or postcondition, only a function that can build a precondition out of *any* postcondition, which happens to be more flexible in practice, but a bit less intuitive for humans. In the case of Dijkstra [6], the preconditions produced by the predicate transformer are weakest preconditions, that is, sufficient and necessary conditions such that the postconditions hold. Also, this was originally stated for an imperative language.

In the following, we denote $wp(\mathsf{p}, post)$ the weakest precondition for the program p such that the postcondition *post* holds after the execution.

One of the nice thing about this kind of predicate transformers is that the computation of the weakest-precondition behaves well on most of usual program syntax. For example, for a simple imperative language that acts on a heap, where preconditions and postconditions are just predicates written as formulas on this heap:

- assignment: $wp(\mathsf{x{:=}e}, post) = post[\mathsf{e}/\mathsf{x}]$.
- sequencing: $wp(\mathsf{p_1}; \mathsf{p_2}, post) = wp(\mathsf{p_1}, wp(\mathsf{p_2}, post))$.
- conditional (where the if branch is executed when the guard is equal to 0, the else branch otherwise): $wp(\mathsf{if\ e\ then\ p_1\ else\ p_2}, post) = (\mathsf{e}{=}0 \Longrightarrow wp(\mathsf{p_1}, post)) \wedge (\mathsf{e} \neq 0 \Longrightarrow wp(\mathsf{p_2}, post))$.

For example: $wp(\mathsf{if\ x\text{-}3\ then\ x{:=}x{+}1\ else\ x{:=}0}, \mathsf{x{=}4}) = (\mathsf{x\text{-}3} = 0 \Longrightarrow \mathsf{x{=}3}) \wedge (\mathsf{x\text{-}3} \neq 0 \Longrightarrow 0 = 4)$, which is equivalent to $x = 3$.

This setting can be changed a bit in order to work with a pure functional programming languages: basically, the postconditions are now predicates on the values returned by expressions after computation, and preconditions are predicates on the free-variables of the expression. For example, the weakest precondition for the expression $\mathsf{if\ y} = 0\ \mathsf{then}\ 42\ \mathsf{else}\ \text{-}42$ and the postcondition $\lambda(\mathsf{x{:}int}).\mathsf{x} \geq 0$ (the predicate which tests whether the returned value is positive) is $\mathsf{y} = 0$.

As we have seen, the weakest precondition calculus let us compute the condition (that is, the precondition) with which a conclusion (that is, the postcondition) would be true. This can be used to build a verification system. Indeed, if we want to prove a property of a program, that is a conclusion under certain premises, then one can build the weakest precondition associated to the conclusion, and check, using for example SMT-solver, that the precondition is implied by the premises of the property.

Instead of using weakest precondition calculus as a tool to analyze a program from an outside point of view, one can try to directly integrate it in the programming language. In [14], a type system which includes this weakest-precondition calculus in the typing judgment of expressions. This extension gives a structure of monad at the type level, which is called the Dijkstra monad. In more recent work on F* [13], this weakest precondition calculus is generalized to a lattice of effects including divergent, stateful and exception-throwing computations. At the end, the new typing judgment looks like this: $\Gamma \vdash \mathsf{e} : \mathsf{M\ t\ wp}$ where M is the kind of computation (pure, divergent etc.), t the usual type, and wp the predicate transformer built from the weakest precondition calculus. These triples of three elements (monadic effect, type and WP) are called **computation types**. The WPs live at the type level and should still be seen as functions from postconditions to preconditions, that also live at the type level. And the built WPs let F* check the properties we want to show.

In order to verify something with a proof assistant, one needs to write definitions describing the concepts involved, write the properties about these definitions, and write the proof of these properties. In F* there are two ways for this: a logical style and a constructive style.

Logical style. In the logical style, the verification process is devolved to the SMT-solver, which basically have to check that some formulas are correct, imply other formulas etc. In this style, writing a definition is writing a formula or an expression that computes something , writing a property is writing the formulas that need to be true or implied, and writing a proof is helping the SMT-solver to prove itself by adding in its context already-proved properties.

In order to introduce the properties we want to prove and their associated proofs in this style, we can use two mechanisms: **intrinsic proofs** and **extrinsic proofs**. With intrinsic proofs, we write the property we want about an expression directly in the declaration of this expression. It is convenient for properties used very often, because one does not need use explicitly a lemma in order to get this property.

For example, if we want to specify that the usual append function on list that concatenates two lists l1 and l2 produces a list of length that is the sum of the two lists.

```
let rec append l1 l2 =
  match l1 with
  | [] → l2
  | a:l1' → a:(append l1' l2)
```

We first need to define what is the length of a list with a simple function:

```
let rec length l =
  match l with
  | [] → 0
  | x:l' → 1+(length l)
```

Then we write the property we want to show. With intrinsic proofs, we specify the property in the declaration of the function, before the definition:

```
val append: #a:Type → l1:list a → l2:list a → Tot (list a) (requires True) (ensures (length (append l1 l2) = length l1 + length l2))
```

In this example, we declare a polymorphic function (on the variable a of kind Type) append that takes two lists as arguments and which is total (the Tot is a syntax-sugar notation for computation types that are total computations), which does not require anything on the data (the requires part), and which ensures the property on lengths that we wanted (the ensures part). Now, every time append is used, the property about the lengths will be added in the context and used transparently. Also, in this case, since the property we want to prove is simple, F* will be able to prove just with the code of append.

But intrinsic proofs are limited because they can only be chosen at declaration time, and because they can harm performance: the property is always added to the context even if it is not used.

We can also write extrinsic proofs that are separate from the definition. Using the same example, we can write the property on lengths in a extrinsic style like this:

```
val append_property: #a:Type → l1:list a → l2:list a →
  Lemma (requires True) (ensures (length (append l1 l2) = length l1 + length l2))
let rec append_property a l1 l2 =
  match l1 with
  | [] → ()
  | x:l1' → append_property l1' l2
```

Here we declare the property we want to prove the same way we declare a function, except we return a Lemma instead of a real type. And then we provide the body of the proof with a let, which can be recursive just like usual let. As one can see, the proof is quite small, because F* is able to automate most of the reasoning: it can unfold itself the expressions appearing in the proof (in the example, the body of length and append are unfolded), it can reduce the expressions that are present, and it can use equalities in context to replace expressions by others, and it can automate the small reasoning about arithmetics using the SMT-solver.

In general, the way to write proofs in F* is just to call the minimum number of lemmas at a particular point in order to introduce the needed properties into the context, then we let the SMT solver figure the details out. Note that we can only prove properties that involve pure expressions (that is, that F* already considers as pure). This is because F* does not allow impure expressions to appear at the type level, and that all the specifications and properties we write live at the type level.

Constructive style. In the constructive style, the verification process is devolved to the type system. In this style, writing a definition is defining an inductive type, writing a property is declaring a function which takes terms of some types as arguments (the premises) and returns a term of some type (the conclusion), and writing a proof is about writing an expression of the right type.

Note that this style is standard in type-theory-based theorem provers, and in particular, it is used in Coq. But contrary to Coq, the constructive proofs are made by writing expressions of the right type instead of using tactics.

For example, take simply typed lambda-calculus with DeBruijn indices:

```
type var = nat
type typ =
  | TUnit: typ
  | TArr: typ → typ → typ
type exp =
  | EVar: var → exp
  | EApp: exp → exp → exp
  | ELam: typ → exp → exp
```

with the definition of environments:

```
type env: var → Tot(option exp)
val extend: env → typ → Tot env
let extend g t x = if x = 0 then Some t else g (x1)
```

First, we would like to define what is a value. In the logical style, it would be by defining a predicate:

```
val is_value : exp → Tot bool
let is_value e =
  match e with
  | EVar x → true
  | EApp e1 e2 → false
  | ELam t e → true
```

In the constructive style, it would be by defining a type that represents values:

```
type value : exp → Type =
  | VVar : x:var → value (EVar x)
  | VLam : t:typ → e:exp → value (ELam t e)
```

Then, we want to define the typing rules. We can do it easily in constructive style.

```
type typing: g:env → e:exp → t:typ → Type =
  | TyVar: g:env → x:var{is_Some (g x)} → typing g (EVar x) (Some.v (g x))
  | TyApp: g:env → e1:exp → e2: exp → t:typ → t':typ
      → typing g e1 (TArr t t')
      → typing g e2 t
      → typing g (EApp e1 e2) t'
  | TyLam: g:env → t:typ → e:exp → t':typ
      → typing (extend g t) e t'
      → typing g (ELam t e) (TArr t t')
```

We can also define the reduction rules using inductive types:

```
type reduction: exp → Type =
  | RedBeta: t: typ → e: exp → e': exp → reduction (EApp (ELam t e) e')
  | RedApp1: e1: exp → e2:exp → reduction e1 → reduction (EApp e1 e2)
  | RedApp2: e1: exp → e2:exp → reduction e2 → reduction (EApp e1 e2)
```

Now, in order to define properties in a constructive style, we declare functions which take the premises as arguments and return the conclusion. Their associated proof is made by building a term of the right type. For example, we can define the progress property in a constructive style:

```
val constructive_progress: e:exp{not (is_value e)} → t:typ → typing empty e t → Tot (reduction e)
let rec constructive_progress e t h =
  let TyApp _ e1 e2 t t' hfun harg = ht in
  if not (is_value e1) then
    RedApp1 e1 e2 (constructive_progress e1 (TArr t t') hfun
  else if not (is_value e2) then
    RedApp2 e1 e2 (constructive_progress e2 t harg)
  else
    let TyAbs _ t e t' hbody = hfun in
    RedBeta t e e2
```

Note that we still need to rely on the logical definition of a value, because we can not directly refer to what is not a value using the constructive definition. With the logical definition, we just use the logical negation.

We saw that there are two main ways (that can be mixed) to write a definition or a property. Both of them have pros and cons. The main advantage of the logical way is that it is easy to manipulate the definition

using logical operators, like negation, as we have seen for the value definition. On the other hand, reasoning about it heavily relies on SMT-solver, which can fail to prove anything if the definitions or the properties are too complicated. It is hard to get the negation of a property defined by a constructive type, but it is easier to reason with: it is just about constructing and deconstructing terms, which needs less SMT-solving, so more things can be proved in this way. Also inductive types are a very comfortable way to represent inference rules of type systems, that will be useful for the work of this internship.

Termination. When defining lemmas or expressions that are total, F* needs to check that the computation terminates. In order to do it, a metric is defined on some of the types: for natural numbers, F* uses the classical decreasing metric, and for inductive types, a term of a certain type is considered lesser than another one if the former is a subterm of the latter. When defining a recursive function which is supposed to be total, F* will check that all the calls to the defined function are made with smaller arguments: by default, it checks that the tuple of non-functional arguments decreases with a lexicographic ordering based on the metrics of the different types. But one can define other decreasing metrics on the tuples, by writing (decreases $\%[e_1;...;e_n]$) where $\%[e_1;...;e_n]$ defines a lexicographic metric using n expressions.

For example, take the Ackermann function:

```
val ackermann: n:nat → m:nat → Tot nat
let rec ackermann n m =
  if m=0 then n + 1
  else if n = 0 then ackermann 1 (m  1)
  else ackermann (ackermann (n  1) m) (m  1)
```

F* will refuse this code, because it is not able to show that the arguments of ackermann decrease, based on the default termination check mode. One can see that the recursive calls to ackermann are made with either a smaller m or with the same m but with a smaller n. So we can make F* understand that this code terminates by specifying another lexicographic metric in the declaration:

```
val ackermann: n:nat → m:nat → Tot nat (decreases %[m;n])
```

Note that the expressions appearing in the user-defined lexicographic clauses can be any total expression. So we can write anything we want for them: one of the argument of the function, a computation returning a type with a metric (natural numbers for example), or even a non computable expression which still expresses a property of the arguments (for example, all0 f, which returns 0 if f x = 0 for all x, 1 otherwise). This termination check also work with mutually defined functions: F* checks that the arguments of the calls to other mutual recursive functions decrease. It can also be used with proofs and let the user define non-trivial induction.

1.2 Overview of micro-F*

Micro-F* is a sub-language of F* and shares a lot of features and concepts: monadic effects, WPs, type operators, subtyping, subkinding, logical fragment with types as formulas. One way to see it is as an extension of $\lambda\omega$. Even though it is only a fragment of F*, it is still very complex and hard to understand entirely. Since most of the ideas here are not specific to Micro-F*, the description of Micro-F* will stay high-level in this report. We only present some overview. The complete details can be found in [13].

Syntax. Micro-F* is based on three principal syntactic classes: expressions, types and kinds (as usual in a language with type operators). There is also syntax for computation types, which are tuples of 3 elements: a monadic effect (in this report, we will almost only talk about the PURE effect), a concrete type and a WP. The grammar for the syntax is presented in figure 1.

effects	M	::=	PURE | ALL
expression constants	ec	::=	() | n | h | l | bang | assign | select | update | fixpure | fixomega
expressions	e	::=	x | ec | e e | λx:t.e
values	v	::=	() | n | h | l | λx:t.e | partially applied constants
type constants	tc	::=	unit | int | heap | location | false | and | forallexpr | foralltype k | prec
types	t	::=	a | tc | t e | t t | t → c | λ x:t . t | λ a:k. t
kinds	k	::=	Type | x:t → k | a:k → k
computation types	c	::=	M t t

Figure 1: Syntax of micro-F*

Some details about this syntactic definition:

$$\frac{}{(\lambda x : t.e)\ e' \to e[e'/x]}\ (\text{Ps-Beta}) \qquad \frac{t \to t'}{\lambda x : t.e \to \lambda x : t'.e}\ (\text{Ps-LamT}) \qquad \frac{}{\mathsf{fixpure}\ e\ e'\ v \to e'\ v\ (\mathsf{fixpure}\ e\ e')}\ (\text{Ps-FixPure})$$

$$\frac{}{\mathsf{update}\ h\ l\ i \to h[l \to i]}\ (\text{Ps-Update}) \qquad \frac{}{\mathsf{select}\ h\ l \to h[l]}\ (\text{Ps-Select}) \qquad \frac{}{(\lambda x : t.t')e \to t'[e/x]}\ (\text{Ts-EBeta}) \qquad \frac{}{(\lambda a : k.t')t \to t'[t/a]}\ (\text{Ts-TBeta})$$

$$\frac{t \to t'}{(\lambda a : k.t) \to (\lambda a : k.t')}\ (\text{Ts-TLamT}) \qquad \frac{t_1 \to t_1'}{(\lambda x : t_1.t) \to (\lambda x : t_1.t)}\ (\text{Ts-TLamE}) \qquad \frac{e \to e'}{t\ e \to t\ e'}\ (\text{Ts-TAppE})$$

Figure 2: Reduction rules of micro-F*

- for expression constants: `n` are integers, `h` are heaps (functions from integers to integers), `l` are locations in the heap (or references). `bang` and `assign` are the effectful functions to read and write references (so they do not take heaps as argument). `select` and `update` are the pure functions to read and write in heaps (they take heaps as argument), and they are used to write properties of effectful computations at the type level. `fixpure` and `fixomega` are the two fixpoint operators, one for pure computations and the other for effectful ones. They are parametrized by types that will define its type (since micro-F* does not have polymorphism, it is a way to have polymorphic fixpoints).

- for type constants: `false`, `and` are the usual logical operators. `forallexpr` and `foralltype` are used to introduced a forall quantification for expressions and types at the logical level. `prec` is used to build inductive inequalities between expressions.

- computation types: as we said earlier, are triples with an effect, a type for the result of the computation, and a WP, which is also encoded as a type. We will use the syntactic sugar `Tot t` to denote the computation type `PURE t wp`, where `wp` requires the given postcondition to hold for all elements of the type.

- also note that we distinguish between expression variables and type variables, we have lambda-abstraction both at the type and expression levels.

Reduction Micro-F* has reduction rules on both types and expressions, that are mutually recursive. We list here some of the most important reduction rules in figure 2. The definitions need to be mutually recursive because of (Ps-LamT) and (Ts-TAppE). The reduction strategy of micro-F* involves both call-by-value (because of the fixpoint) and call-by-name (because of the β-reduction).

Type system. The type system is composed of 7 judgments that are mutually recursive:

- the typing judgment (figure 3), which assigns a **computation type** `c` for an expression `e` in an environment Γ: $\Gamma \vdash$ `e` : `c`. It may have been better to call it the computation judgment, but still it can be seen as an extension of the usual typing judgment. Sometimes in this report, to keep explanations simple, we will do as if this judgment was only assigning a type to an expression, as in standard typing judgment for lambda-calculus. There are two specificities here. First, there are two application rules: since the type system asks for expressions that appear in types to be total, we have a rule checking that the argument of the function is total when the arrow is dependent, which is T-App1. But when the arrow is not dependent, we allow applications of arguments that are not necessarily total with T-App2. Second, there are two other rules that are not syntax directed and that only change the computation type part: there is T-Sub which replaces the computation by a weaker one (according to a subcomputation judgment), and the T-Ret rule which acts only on total computations and returns a typing judgment where the WP is strengthened. Intuitively, the T-Ret rule allows the F* type system to reason about the concrete code of pure expressions, not just about their type. We call these two rules the **non-syntactic rules**.

- the kinding judgment (figure 4), which assigns a kind `k` for a type `t` in an environment Γ: $\Gamma \vdash$ `t` : `k`. The rules of this judgment are what one could expect from a lambda-calculus with subkinding and type operators.

- the kind well-formedness judgment, which establishes that a kind `k` is well-formed in an environment Γ: $\Gamma \vdash$ `k` *kwf*. The definition of this judgment is not very interesting regarding what is following. It just checks that the types appearing in the kind are kindable. We will not talk much about it.

$$\text{(T-Var)}\ \frac{x : t \in \Gamma}{\Gamma \vdash x : \mathsf{Tot}\ t} \qquad \text{(T-Abs)}\ \frac{\Gamma \vdash t : \mathsf{Type} \quad \Gamma, x{:}t \vdash e : M\ t\ wp}{\Gamma \vdash \lambda x{:}t.\, e : \mathsf{Tot}\ (x{:}t \to M\ t\ wp)} \qquad \text{(T-Const)}\ \frac{}{\Gamma \vdash ec : \mathsf{Tot}\ (t_{ec})}$$

$$\text{(T-App1)}\ \frac{x \in FV(t') \quad \Gamma \vdash e_1 : M\ (x : t \to M\ t'\ wp)\ wp_1 \quad \Gamma \vdash e_2 : \mathsf{Tot}\ t}{\Gamma \vdash e_1\ e_2 : M\ (t'[e_2/x])\ (\mathsf{bind}_M\ wp_1\ \lambda\_.wp[e_2/x])}$$

$$\text{(T-App2)}\ \frac{x \notin FV(t') \quad \Gamma \vdash e_1 : M\ (x : t \to M\ t'\ wp)\ wp_1 \quad \Gamma \vdash e_2 : M\ t\ wp_2}{\Gamma \vdash e_1\ e_2 : M\ t'\ (\mathsf{bind}_M\ wp_1\ (\lambda\_.\mathsf{bind}_M\ wp_2\ \lambda x.wp))}$$

$$\text{(T-If0)}\ \frac{\Gamma \vdash e_0 : M\ \mathsf{int}\ wp_0 \quad \Gamma \vdash e_1 : M\ t\ wp_1 \quad \Gamma \vdash e_2 : M\ t\ wp_2}{\Gamma \vdash \mathsf{if}_0\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : M\ t\ (\mathsf{ite}_M\ wp_0\ wp_1\ wp_2)}$$

$$\text{(T-Sub)}\ \frac{\Gamma \vdash e : M'\ t'\ wp' \quad \Gamma \vdash M'\ t'\ wp' <: M\ t\ wp}{\Gamma \vdash e : M\ t\ wp} \qquad \text{(T-Ret)}\ \frac{\Gamma \vdash e : \mathsf{Tot}\ t}{\Gamma \vdash e : PURE\ t\ (\mathsf{return}\ t\ e)}$$

Figure 3: Typing rules of micro-F*

$$\text{(K-Var)}\ \frac{a : k \in \Gamma}{\Gamma \vdash a : k} \qquad \text{(K-Const)}\ \frac{}{\Gamma \vdash tc : k_{tc}} \qquad \text{(K-Arr)}\ \frac{\Gamma \vdash t_1 : Type \quad \Gamma, x : t_1 \vdash t_2 : Type \quad \text{(some condition on wp)}}{\Gamma \vdash t_1 \to M\ t_2\ wp : Type}$$

Figure 4: Some of the kinding rules of micro-F*

- the subtyping judgment (figure 5), which establishes that a type t' is a subtype of a type t in an environment Γ: $\Gamma \vdash$ t' <: t. Note that it can not directly be used to apply subtyping in the typing judgment. Also note that we can get a subtyping relation from an equality at the logical level (validity judgment below). So expression and type conversions are done in F* via subtyping and involve the SMT solver.
- the subkinding judgment, which establishes that a kind k' is a subkind of a kind k in an environment Γ: $\Gamma \vdash$ k' <: k.
- the validity judgment (figure 6), which establishes that a formula (which is a type with kind Type) is logically true in an environment Γ: $\Gamma \models \phi$. The validity judgment is essentially used to get type equalities, that are used to build subtyping relations, and implications between WPs that are used to build subcomputation derivations. In the F* tool validity is discharged using the SMT solver.
- the subcomputation judgment (figure 7), which establishes that a computation type c' is a subtype from another computation type c in an environment Γ: $\Gamma \vdash$ c' <: c. This is a judgment with only one rule which asks to prove that the type t' in c' is a subtype of the type t in c, and that the WP in c is "stronger" than the WP in c'.

The environment Γ contains expression variables (which are introduced by the typing judgment with expression abstractions at the expression level, by the kinding judgment with expression abstractions at the type level etc.) and type variables (which are introduced by the kinding judgment with type abstractions at the type level, by the validity judgment with forall introduction etc. . .). In order to say that the types and kinds in the environment are well defined, there is also a **well-formedness judgment** for environments.

Take away. In order to keep the explanations high level, but still to be able to imagine what were the challenges faced when writing the proofs, we make the following summary.

First, about the main features of micro-F*: it is a lambda-calculus, of which $\lambda\omega$ can be seen as a fragment. It

$$\text{(Sub-Conv)}\ \frac{\Gamma \vdash t_1 : Type \quad \Gamma \vdash t_2 : Type \quad \Gamma \models t_1 =_{Type} t_2}{\Gamma \vdash t_1 <: t_2}$$

$$\text{(Sub-Trans)}\ \frac{\Gamma \vdash t_1 <: t_2 \quad \Gamma \vdash t_2 <: t_3}{\Gamma \vdash t_1 <: t_3} \qquad \text{(Sub-Fun)}\ \frac{\Gamma \vdash t <: t' \quad \Gamma, x : t \vdash c' <: c}{\Gamma \vdash t' \to c' <: t \to c}$$

Figure 5: Subtyping rules of micro-F*

$$\text{(V-Assume)}\ \frac{\Gamma \vdash e\ :\ t}{\Gamma \models t} \qquad \text{(V-RedE)}\ \frac{e \to e'}{\Gamma \models e = e'} \qquad \text{(V-SubstE)}\ \frac{\Gamma \models e = e' \quad \Gamma \models \phi[e/x]}{\Gamma \models \phi[e'/x]} \qquad \text{(V-AndIntro)}\ \frac{\Gamma \models \phi_1 \quad \Gamma \models \phi_2}{\Gamma \models \mathsf{and}\ \phi_1\ \phi_2}$$

$$\text{(V-FalseElim)}\ \frac{\Gamma \models \mathsf{false} \quad \Gamma \vdash \phi : Type}{\Gamma \models \phi} \qquad \text{(V-ForallExpIntro)}\ \frac{\Gamma \vdash t : Type \quad \Gamma, x : t \models \phi}{\Gamma \models \mathsf{forallexp}(x : t).\phi}$$

$$\text{(V-DistinctTH)}\ \frac{t_1, t_2 \text{ in head normal form} \quad t_1 \neq t_2}{\Gamma \models \mathsf{not}\ (t_1 = t_2)}$$

Figure 6: Some validity rules of micro-F*

$$\text{(SCmp)}\ \frac{M' \leq M \quad \Gamma \vdash t' <: t \quad \Gamma \models wp \Longrightarrow wp'}{\Gamma \models M'\ t'\ wp' <: M\ t\ wp}$$

Figure 7: Subcomputation rule of micro-F*

has more features, like subkinding, subtyping, a logical fragment ... This calculus has expression variables and type variables. It has expression and type variables because there are lambda abstraction at both expression level and type level. Second, about the complexity of its specification. The specification of micro-F* is really big: 97 rules in total, more than a thousand lines in the code. It is also a challenge to write a proof about this calculus because both the syntactic forms and the judgments of the type system are mutually defined. So it is not possible to handle what is going on at the kind level, then go to the type level then finish at the expression level. Instead, a property on the type system needs to be proved mutually with all the judgments.

Additional formalization details. In order to write the proofs for this language in F*, different choices of implementation were made.

First, the formalization uses a nameless representation of variables based on DeBruijn indices. The numbering of expression variables and type variables are independent. As a consequence, we will stop showing named variables from here. For example, lambda abstraction will be denoted by λt.e instead of λx:t.e.

Also, environments are represented as a pair of two functions: one from variables to the types of micro-F*, and one from variables to the kinds of micro-F*. The result is returned using an option type, in case the variables are not defined in the environment. When the environment are extended with other variables, it is always at the 0 position. Remember that the environment binds variables to types and kinds that can themselves refer to variables of the environment. This implies that, when the environment is extended, the types and kinds in the environment must be shifted, that is, the free variables appearing in them must be increased, in order to match the extended environment.

Finally, the judgments of the type system are written in F* using inductive types in order to write constructive proofs. The reduction of terms is also represented with inductive types.

We refer the reader to the appendix (A.1) for the implementation details of the definitions.

Goal. The goal of this internship was essentially to prove that micro-F* satisfies the progress and preservation properties. We only attempted the proof for PURE computation types.

2 Substitution

In this section, we describe how we dealt with the definitions and the properties involving the general notion of substitution. We first explain how we defined the substitution of terms for micro-F* using parallel substitution. Then, we show how we extended this parallel substitution to the level of the type system to get a strong substitution lemma, which is needed for the preservation property among others.

2.1 Defining parallel substitution

Substitution is part of the formalization of micro-F*, since it appears in the type judgments and in the reduction relation. In order to simplify proofs, a concise and general definition of substitution was needed. In this section, we show how we defined substitution for micro-F* in detail, since it is a key component of the

formalization, and also because it shows some interesting termination problem that can be solved in F*. We define parallel substitution functions that handle both expression and type variables.

We first define parallel substitution functions for each syntactic form, that is, a function that substitutes the free variables inside expressions, types, kinds by expressions for expression variables and by types for type variables. We substitute both expression variables and type variables at the same time. Since all the syntactic forms are mutually defined, the parallel substitution functions are also mutually defined. There is a substitution function for each syntactic form: esubst for expression substitution, tsubst for type substitution etc. Each of these functions takes two arguments:

- an argument s that we call a **sub** (for parallel substitution), which is a pair of functions, one from variables to expressions (which we refer to with s.es), and one from variables to types (which we refer to with s.ts). This argument defines what will be substituted for the free variables met in the terms
- the syntactic form in which the substitution occurs

As one could imagine, these functions just go recursively on the structure of the terms, and when a free variable is encountered, it is replaced by the corresponding element defined in the sub argument.

Lambda abstraction and termination. But some cases bring troubles. Indeed, if we forget about lambda abstractions, we have substitution functions that are mutually defined and with a simple decreasing clause, which is the term being substituted. For example, for application at the expression level, the substitution is called on the left-hand side of the application and on the right-hand side, which are considered as strictly lesser than the application itself by the metric of inductive types. But it is not as simple for lambda abstractions. In this case, we also need to transform the sub argument. Indeed, when we cross a lambda, the way expression and type variables are substituted by the the sub needs to change, because of the new variable introduced by the lambda. Consider an expression abstraction at the expression level: λt.e. The substitution on this expression is defined like this: esubst s (λt.e) = λ(tsubst s t).(esubst s' e) where s' is defined the following way:

s'.es(0) = 0
s'.es(x+1) = esubst (eshift) (s.es(x))
s'.ts(a) = tsubst (eshift) (s.ts(a))

where eshift is a sub which just shifts the type variables:

eshift.es(x) = x+1
eshift.ts(a) = a

What s' does is to move expression substitution one-level up, because of the variable introduced by the lambda, and to shift the free variables in terms of the sub accordingly.

One can see that the simple termination argument does not hold anymore: when going under a lambda, we not only substitute a smaller term (the body of the lambda), but we also do a substitution on the elements of the sub argument, which are not subterms of the lambda abstraction. So we need a more complex decreasing clause to prove termination.

First, we can see that the shift we need to apply to the sub is a simple sub: it only replaces variables by other variables (and not more complicated terms). We call these simple subs **renamings**. One can also check easily that the transformation s $\rightarrow$ s' is very simple when done with renamings and does need to use a complicated substitution functions and can be done directly. This transformation also preserves renamings.

One solution could be to define substitution in two steps: first define the substitution when the subs are renamings, and then define the general substitution based on the one for renamings. But this would duplicate the code for substitution functions and make proofs about it more complicated.

Instead, we define the general substitution functions directly, using the lexicographic metrics to show termination. This metric should reflect the following steps to define completely the substitution:

- First, we define the substitution functions when variables are substituted. That is easy: we just replace the variables with what is inside the sub argument, without other recursive calls. So termination is trivial.
- Then we define the substitution functions when the sub argument is a renaming, by doing an induction over the terms. When going under lambdas, and transforming the sub, the substitutions done over the elements of sub is done on variables (because s is a renaming), so it terminates. So there are no problems when going under lambdas.
- Finally, we define the substitution for non-renamings, still with an induction over the terms. When going under lambdas, and transforming the sub, the substitution is done with a renaming (because shifting is a renaming), so it terminates.

This induction was proposed by Altenkirch and Reus in [2]. At the end, we get the following decreasing clauses for the different functions:

- for esubst s e: [is_evar e; is_renaming s; e]
- for tsubst s t: [is_tvar t; is_renaming s; t]
- for ksubst s k: [1;is_renaming s; k] (in the first position, there is a 1 because there are no variables at the kind level)

with is_evar and is_tvar functions that return 0 when their argument is a variable and 1 otherwise, and is_renaming a function that returns 0 if its argument is a renaming, 1 otherwise.

```
let is_evar e = if is_EVar e then 0 else 1
let is_tvar t = if is_TVar t then 0 else 1

type renaming (s:sub) = (forall (x:var). is_EVar (Sub.es s x)) ∧ (forall (a:var). is_TVar (Sub.ts s a))

let is_renaming (s:sub) = if excluded_middle (renaming s) then 0 else 1
```

Note that is_renaming is not computable. It uses the function from F*'s classical logic library excluded_middle which is not defined, but only assumed as a function of this type:

```
val excluded_middle: p:Type → Tot (b:bool{b = true ⟺ p})
```

So excluded_middle returns true if and only if the property passed as argument is true. When checking termination, F* will be able to do a case analysis on the result of excluded_middle, that is, on whether the substitution is a renaming. So we are able to use it to finish our induction using this non-trivial property. Note also that, once the decreasing clauses are given, the proof of termination is almost completely automated. We still need to write some refinement types that concern renamings but that's all.

Putting out the sub transformation. Now we have terminating mutually defined substitution functions. But there is still a problem: the sub transformation when going under lambdas is defined locally in the substitution functions and we can not refer to it from outside. Later, when we will want to prove some things about this substitution, we will not be able to refer to this transformation:

```
let rec esubst s e =
match e with
...
| ELam t e →
(
  let s' = (*transformation of s to s'*) in
  ELam (tsubst s t) (esubst s' e)
)
(*We can not refer to s' from here, because it is a local variable*)
```

One thing that can be done is to define again this transformation globally after the substitution functions and prove that this global transformation does the same thing than the local transformation. But, due to F* limitations, this method slows down the verification process. So we need a better way.

A better way is to define this transformation mutually recursively with the 3 other substitution functions, instead of defining it locally every time a lambda is crossed. So we define sub_elam (the transformation when going under expression lambdas) and sub_tlam (the transformation when going under type lambdas) which takes one argument, which is the sub s on which the transformation is to be applied. The decreasing clauses need to be changed in order for the substitution functions to be able to call these transformations. That is, call to the substitution functions need to be one level higher than the calls to the substitution transformations. To do this, we simply add another parameter to the decreasing clause that will be 1 for the substitution functions and 0 for the sub transformations.

At the end we get the following decreasing clauses:

- for esubst s e: [is_evar e; is_renaming s; 1; e]
- for tsubst s t: [is_tvar t; is_renaming s; 1; t]
- for ksubst s k: [1; is_renaming s; 1; k]
- for sub_elam s: [1; is_renaming s; 0; (void)]
- for sub_tlam s: [1; is_renaming s; 0; (void)]

The last parameter of the decreasing clause is never relevant for termination of sub_elam and sub_tlam, what is why it is replaced by void. This was a good example of an aspect of writing proofs with F*, which is finding the right decreasing clause that makes the computation terminate.

Basic subs. Here is a list of the subs and sub transformations that are common in the code:

- sub_id, the identity sub
- sub_elam, sub_tlam: they are the transformations applied to the subs when the substitution functions go under an expression-level or a type-level lambda-abstraction: esubst s (λt.e) =λ(tsubst s t).(esubst (sub_elam s) e), and tsubst s (λk.t) =λ(ksubst s k).(tsubst (sub_tlam s) t) for example.
- eshift, tshift: these subs, that are renamings, increase the free expression or type variables by one.
- sub_ebeta, sub_tbeta: these subs represent the β-reductions at the expression and at the type level: e[e'/0] is done with esubst (sub_ebeta e') e, and t[t'/0] with tsubst (sub_tbeta t') t for example.

Category of sub. One can quickly see that the set of subs with composition of sub forms a category. More precisely, it forms a monoid (with identity being the identity substitution). We can also easily prove that, in this category, sub_elam and sub_tlam are functors.

2.2 Substitution of judgments

The judgments of the type system we are considering all have the same shape: on the left-hand side, an environment, in which the judgment is proven, and on the right-hand side, a tuple of terms that can be substituted.

In general, when proving the metatheory of some lambda-calculus, there are always a bunch of lemmas about judgments that one has to prove. These lemmas are similar in the following way: they expect a judgment in some environment Γ_1 as a premise, and give as a conclusion a judgment of the same kind but in an environment Γ_2 and with the right-hand side substituted. For example, weakening, substitution, strengthening of the environment ...

Even though they are similar, these lemmas are usually proved one by one separately, using an induction over the judgments of the type system. For micro-F*, it would have been a complete pain to write a proof for each of these lemmas, since there are many judgments. Instead, we tried to adapt a technique evoked by a colleague of Cătălin Hriţcu, Steven Schäfer [10] which consists in proving a more general lemma, of which all the lemmas mentioned above would be direct instances.

Idea. The idea is to generalize the parallel substitution for terms to the judgments of the type system. When one looks at a derivation of a specific judgment of micro-F*, one can see that this looks like a tree with different kinds of judgments appearing in the nodes. But the leaves of these derivations are always the same: T-Var rules, K-Var rules or constant rules, which are neutral with substitution. So, as we defined the substitution functions by induction, substituting the variables at the leaves of terms by other terms, we can try to extend this substitution to judgments, by substituting the T-Var and K-Var rules at the leaves of judgments by other judgments. This is exactly how the usual substitution lemma in simply typed lambda-calculus works, but it is only substituting one variable and misses the more general lemma. Note also that these T-Var and K-Var rules are the only rules that make the environment interfere with the judgment. In all other rules, the content of the environment does not matter. Actually, from a particular judgment in an environment Γ_1, one can get another judgment in an environment Γ_2 by substituting the "holes", which are the T-Var and K-Var rules, by terms of the corresponding type in Γ_2. For example, if x and y are expression variables of type int in Γ_1, s_x and s_y two terms such that $\Gamma_2 \vdash s_x : \text{int}$, $\Gamma_2 \vdash s_y : \text{int}$, we can substitute a judgment involving x and y, for example $\Gamma_1 \vdash x + y : \text{int}$, to get a judgment in Γ_2 matching the substituted terms, $\Gamma_2 \vdash s_x + s_y : \text{int}$.

Substitution arrow. In order to be able to write this lemma, we first need to generalize the notion of subs to typing judgments. We call this generalization **substitution arrows**. They are defined in the code under the subst_typing type. This type is parametrized by three things: the environment we start with, Γ_1, the sub applied on the terms in the judgment, s, and the environment we finish in, Γ_2. Like the subs, substitution arrows (denoted usually hs) are defined using a pair of functions, one from expression variables to typing judgments (referred to as hs.ef), and one from type variables to kinding judgments (referred to as hs.tf). For every expression variable x that is present in the environment Γ_1 with type t, hs.ef gives a typing derivation $\Gamma_2 \vdash$ s.es(x) : tsubst s t. The same for hs.tf: for every type variable a that is present in the environment Γ_2 with kind k, hs.tf gives a kinding derivation $\Gamma_2 \vdash$ s.tf(a) : ksubst s k. When a substitution arrow for the sub s between environments Γ_1 and Γ_2 can be defined, we denote this by $(\Gamma_1) \xrightarrow{s} (\Gamma_2)$. Note that in the following, the exact definition a substitution arrow $(\Gamma_1) \xrightarrow{s} (\Gamma_2)$ is not important. The only fact that it exists matters.

In order to be able to do the same kind of induction and termination argument we have used for the substitution functions, we define an equivalent notion of renaming for substitution arrows: a substitution arrow hs is a renaming arrow if the judgments substituted by hs.ef are only T-Var rules, and the judgments

substituted by hs.tf are only K-Var rules. So the effect on a typing derivation of this kind of arrow is to replace T-Var leaves by other T-Var leaves, and K-Var leaves by other K-Var leaves.

Details specific to micro-F*. Once the idea is understood for a simple calculus, like the simply typed lambda-calculus, the following differences with micro-F* need to be emphasized: first, contrary to STLC, or even $\lambda\omega$, there is no stratification between the terms. There are all mutually recursive. The same for the type judgments. So, contrary to STLC, where from a typing judgment we get another typing judgment where the only thing which is replaced is the expression, here we substitute all the elements of the right hand side, since free variables can appear in any of them. Plus, remember that we have both expression variables and type variables to handle. Also, we have to handle what the substitution does on the complicated encodings that appear in the WPs in the computation types.

Statement and proof. Once we have defined these substitution arrows, we state how we can use them to substitute whole judgments. So, suppose we have built a substitution arrow $(\Gamma_1) \xrightarrow{s} (\Gamma_2)$. Then we have the following statements on each kind of judgments:

Theorem 1. *(Substitution lemma) If a substitution arrow $(\Gamma_1) \xrightarrow{s} (\Gamma_2)$ can be built then the following statements hold:*

- *Typing judgment: for e expression and c computation, if we have a derivation $\Gamma_1 \vdash e : c$, then we can build the derivation $\Gamma_2 \vdash (esubst\ s\ e) : (csubst\ s\ e)$*
- *Kinding judgment: for t type and k kind, if we have a derivation $\Gamma_1 \vdash t : k$, then we can build the derivation $\Gamma_2 \vdash (tsubst\ s\ t) : (ksubst\ s\ k)$*
- *Kind well formedness judgment: for k kind, if we have a derivation $\Gamma_1 \vdash k\ kwf$, then we can build the derivation $\Gamma_2 \vdash (ksubst\ s\ k)\ kwf$*
- *Subtyping judgment: for t', t types, if we have a derivation $\Gamma_1 \vdash t' <: t$, then we can build the derivation $\Gamma_2 \vdash (tsubst\ s\ t') <: (tsubst\ s\ t)$*
- *Subkinding judgment: for k', k kinds, if we have a derivation $\Gamma_1 \vdash k' <: k$, then we can build the derivation $\Gamma_2 \vdash (ksubst\ s\ k') <: (ksubst\ s\ k)$*
- *Validity judgment: for t type, if we have a derivation $\Gamma_1 \models t$, then we can build the derivation $\Gamma_2 \models (tsubst\ s\ t)$*
- *Subcomputation judgment: for c', c computations, if we have a derivation $\Gamma_1 \vdash c' <: c$, then we can build the derivation $\Gamma_2 \vdash (csubst\ s\ c') <: (csubst\ s\ c)$*

The proof of this big lemma needed different kinds of ingredients.

First, since there are some substitutions appearing in the rules of the type judgment, and in the encodings of WPs, we need to prove some commutation laws about the subs. For example, if $\Gamma \vdash e_1 : t \to t'$ and $\Gamma \vdash e_2 : t$, then, using dependent application, the final type is obtained with a substitution: $\Gamma \vdash e_1\ e_2 : t'[e_2/0]$. In this case, we need to show that substitution commutes well with the substitution on t'.

These proofs were completely carried out.

Secondly, we need to prove that substitution of terms behaves nicely on encodings, that is to say that we have some kind of morphism property on the encodings. For example, we need to prove that the tot function, which produces a computation for a type t which encodes the totalness property, verify the following property: csubst s (tot t) = tot (tsubst s t). These proofs were not carried out completely. The reasons for this are that these proofs are generally tedious and non-interesting, that there are a lot of encodings that need proofs, and that it is difficult to write this proof to the end on complicated encodings, because at some point we reach a limitation with the SMT automation. Still, these proofs help to find bugs in the encodings of the WPs that are done by end, in a DeBruijn fashion, that are very difficult to read and write.

Finally, we can write the big induction on all the judgments of the type system. The general way to write each case is to apply substitution on the premises, reapply by the constructor, and then use the morphism property of the encodings to conclude. But, as with substitution functions, we face some problems when going under a lambda. Indeed, when going under an expression or type lambda, the sub is transformed with sub_elam or sub_tlam, so we need to change the substitution arrow accordingly: we need to define a substitution arrow transformation, as we have done for the subs. And, since there are substitutions happening during this transformation, we need to make these transformations mutually recursive, as before. We call these two transformations elam_hs and tlam_hs.

Let's see what transformation we need to write. So suppose we have a substitution arrow $(\Gamma_1) \xrightarrow{\mathsf{s}} (\Gamma_2)$ and we are trying to substitute a T-Abs rule in the environment Γ_1 into Γ_2 with s : from $\Gamma_1 \vdash \lambda$t.e : t $\to$ t' , we want to build $\Gamma_2 \vdash$ esubst s (λt.e) : tsubst s (t $\to$ t'). (we just write the types instead of the whole computation types for clarity). By deconstructing the T-Abs rule in Γ_1, one can get the judgment $\Gamma_1, t \vdash$ e : t', and in order build the conclusion in Γ_2 with the T-Abs rule, one needs the judgment Γ_2, tsubst s t $\vdash$ esubst (sub_elam s) e : tsubst (sub_elam s) t'. We can see that we can get the latter judgment using the substitution lemma on the former judgment with the substitution arrow $(\Gamma_1, t) \xrightarrow{\mathsf{sub\_elam\ s}} (\Gamma_2, \mathsf{tsubst\ s\ t})$. In order to conclude, we need to build a substitution arrow transformation that builds a substitution arrow $(\Gamma_1, t) \xrightarrow{\mathsf{sub\_elam\ s}} (\Gamma_2, \mathsf{tsubst\ s\ t})$ out of the substitution arrow $(\Gamma_1) \xrightarrow{\mathsf{s}} (\Gamma_2)$. We explain in detail how this transformation is done.

Lambda transformation We need to build a term hs' that represents the arrow $(\Gamma_1, t) \xrightarrow{\mathsf{sub\_elam\ s}} (\Gamma_2, \mathsf{tsubst\ s\ t})$ from the hs term. First, let's build hs'.ef. In what follows, we denote $\uparrow$ the substitution with eshift, that is, the shifting of expression variables inside terms. There are two cases:

- for x = 0, we have that x is a variable of type t in (Γ_1, t), so we need to give a derivation that (sub_elam s).es(x), which is 0, is of type (tsubst s t) in $(\Gamma_2,$ tsubst s t) which is trivial.
- for $x = n + 1$, we have that x is a variable of type $\uparrow$ (t') in Γ_1, t, where $x' = n$ is a variable of type t' in Γ_1 (remember that an extended environment has its variables shifted). So we need build a derivation Γ_2, (tsubst s t) $\vdash$ (sub_elam s).es(x) : tsubst (sub_elam s)($\uparrow$ t'). Unfolding the definition of sub_elam, and using a commuting property with eshift this amounts to the derivation Γ_2, (tsubst s t) $\vdash\uparrow$ (s.es(x')) :$\uparrow$ (tsubst s t') But from hs.es(x'), we know a derivation $\Gamma_2 \vdash$ (s.es(x')) : tsubst s t'. In order to conclude, we just need to apply the substitution lemma on this last judgment with the substitution arrow $(\Gamma_2) \xrightarrow{eshift} (\Gamma_2, \mathsf{tsubst\ s\ t})$, which we can easily build.

It is the same for hs'.tf. And we define the same way tlam_hs. So we are able to correctly define these transformations, up to the termination check that gets trickier, as with substitution functions, that we solve the same way.

We write the same kind of transformation for type lambdas. And this finish the global scheme of the proof.

Deriving usual lemmas. We can now enjoy the power of our strong lemma and derive, painlessly, all the interesting lemmas we want, just by providing a substitution arrow !

- Weakening: for t a type, we build the substitution arrow $(\Gamma) \xrightarrow{\mathsf{eshift}} (\Gamma, t)$ (this arrow is used in the proof to build the lambda transformation)
- Substitution: for t a type, and e such that $\Gamma \vdash$ e : Tot t, we build the substitution arrow $(\Gamma, t) \xrightarrow{\mathsf{sub\_ebeta\ e}} (\Gamma)$
- Supertyping in the environment: for t, t' types such that $\Gamma \vdash$ t' <: t, we build the substitution arrow $(\Gamma, t) \xrightarrow{\mathsf{sub\_id}} (\Gamma, t')$
- Superkinding in the environment: basically the same than before but for kinds instead of types

So we get all the usual lemmas that were supposed to be proved one by one, with an induction each time. With this method, only one was necessary. And the amount of work that needs to be done to build a substitution arrow is substantially lower than with an induction. And even the substitution arrows are flexible: if you want some other lemma which is a mix of the lemmas above, you can compose the existing substitution arrows instead of writing a specific one.

A challenge. Writing this lemma was a challenge for several reasons. First, the general substitution lemma was not part of the paper proof sketch. We turned to it because there were too many lemmas about substitution and they were not flexible enough.

The proof took a very long time to write (1 month) since it is an induction on **all** the judgments. At the end, this proof is 700 LoC.

Also, during this proof, the first F* limitations started to appear. First, as the proof went on, the verification process took longer. Also SMT automation started to fail as the code got bigger. So a very awkward strategy was used to get around this: all the code for working cases was admitted. Only the case being written was not commented. Partly because of this awkwardness, an interactive mode was added to F* in order to avoid compiling every time from the beginning. Also, an option was added in order to better handle big inductions on a lot of cases. Even with this feature, we could not make the proof completely work.

Some simplifications were also needed in order to make proofs easier to write. First, we changed the way environment well-formedness was checked. In the paper proof version, environment well-formedness is another

mutually recursive judgment which is required at the leaves of typing and kinding judgments, in particular for variables introduction:

$$\text{(old T-Var)}\ \frac{\Gamma, x : t, \Gamma' \text{ well-formed}}{\Gamma, x : t, \Gamma' \vdash x : \mathsf{Tot}\ t} \qquad \text{(old K-Var)}\ \frac{\Gamma, a : k, \Gamma' \text{ well-formed}}{\Gamma, a : k, \Gamma' \vdash a : k}$$

It enforced an implicit check of the well-formedness of environment: every judgment of the type system involving Γ implies the well-formedness of Γ. But it causes troubles because the substitution lemma can not substitute these well-formedness judgments. After different attempts, we chose to remove the implicit check for environment well-formedness (as one can see on the T-Var and K-Var rules in figure 3. Now, the environment well-formedness is an extrinsic property, that appears as a premise of the lemmas that need it (like the preservation lemma). Another change is the way recursive functions are defined. In the first version, it was defined as a let rec constructor, that is like a double abstraction on argument and function. Due to the complexity of the proof with this style, we switch to a fixpoint style, which is just a constant (see figure 2 to see the reduction for the fixpoint constant).

Moreover, since it was the first proof made, we discovered some errors in the formalization during the process that needed to be fixed. One of them was the way we introduced validity judgment from typing judgment with the V-Assume rule. The old version allowed only to introduce variables that were present in the environment:

$$\text{(old V-Assume)}\ \frac{x : t \in \Gamma}{\Gamma \models t}$$

Since the substitution lemma substitutes these variables, we could not longer apply this rule . So now any typed expression can be introduced in the validity judgment (see figure 3 for the new V-Assume).

Another problem was caused by a conflict between the old reduction strategy. Indeed, at the time, the reduction strategy was call-by-value.

It was so because one can prove that values are total computations, and the substitution lemma can only substitute with expressions that are total. For example, if we want to prove the preservation property on the β-redex $(\lambda \mathsf{t.e})\ \mathsf{e'} \rightarrow \mathsf{e}[\mathsf{e'}/0]$, we need to apply the substitution lemma, and show that e' is total.

Also, in the old definition of values, variables were considered as values. But, during the process of substitution, these variables could be substituted by non-values in such way that the reduction rules, that were CBV, did not apply anymore.

We fixed this problem by removing variables from values.

This fix was working but it removed some completeness of micro-F*. Indeed, with this fix, it was not possible for the logical level of micro-F* to deduce that $(\lambda \mathsf{x{:}t.42})\ \mathsf{y}$ can reduce to 42.

In order to keep the completeness, we wanted to switch to call-by-name. But as said earlier, we would have not been able to prove the preservation property for β-redex, when the argument is not a value.

We explain later in the section devoted to the preservation lemma how we found a way to switch to CBN.

We refer the reader to the appendix (A.4) for implementation details about the substitution lemma.

Category theory. The remark that we did before for subs can be done for substitution arrows. It is even why we named them arrows. Indeed, one prove that there is a category of substitution arrows where environments are the objects and subs are the arrows. In this setting, the transformations elam_hs and tlam_hs are functors. This categorical interpretation guided the way the substitution lemma was written: for example, it is only when elam_hs and tlam_hs were found to be functors, that the code for them where properly isolated and generalized, which made the termination argument simpler. Also, having in mind which diagrams elam_hs and tlam_hs make commute helped to write the code for them. This categorical interpretation was not only a design guide, but also a way to talk about this proof: the most fruitful discussions about this lemma were only possible with commutative diagrams. See the appendix (B) for more details.

3 Simple inversion scheme and preservation

In this section, we talk about the proof of the usual preservation lemma. One of the specificities of this lemma for micro-F* is that it handles both type-level reductions and expression-level reductions, since they are mutually-recursively defined. We will only give details about the proof for expressions. One of the contributions of this internship was to simplify the proof of preservation lemma, and in particular, to remove the need to write the complicated proofs of some lemmas called inversion lemmas.

Inversion lemmas. In STLC, when one has a judgment $\Gamma \vdash \texttt{e}_1\ \texttt{e}_2 : \texttt{t}$, one knows that the last rule applied is an application rule and can get directly that $\Gamma \vdash \texttt{e}_1 : \texttt{t'} \to \texttt{t}$ and $\Gamma \vdash \texttt{e}_2 : \texttt{t'}$. And it is the same story for lambda abstraction, variables introduction etc. So basically, one can reason about typing judgment syntactically: the last rule applied only depends on the syntax of what is in the judgment.

In Micro-F*, things are not as simple. For the typing judgment, there can be up to three rules that produces the same term, essentially because there are non-syntactic rules, which do not change the typed expression. So, when one has $\Gamma \vdash \texttt{e1 e2} : \texttt{t}$ in Micro-F*, one can not directly deconstruct and get the same judgments than in STLC. But still, it would be nice to be able to reason syntactically. Which is the goal of inversion lemmas: to let one invert syntactically typing judgments.

In the paper proof, these inversion lemmas were used in the preservation lemma, because of the way the induction was done. The preservation lemma (for typing judgments) takes a typing judgment $\Gamma \vdash \texttt{e} : \texttt{c}$ and a reduction step $e \to e'$ as arguments and returns a typing judgment $\Gamma \vdash \texttt{e'} : \texttt{t}$ as conclusion. In the paper proof of preservation, the induction was done on the reduction step, which is completely syntactic. So, this induction needs inversion lemmas, for all syntactic constructions at the expression level. And each one of them would have been a difficult to write, because of what was needed for the WPs, which are a pain to handle in general. So inversion lemmas are basically a bad idea in Micro-F*

Induction on typing derivation. So, is it possible to avoid these inversion lemmas during the preservation proof ?

The problem was present because of the gap between the induction on the reduction proof, which is completely syntactic, and the typing derivation, which has non-syntactic rules.

So one idea was to do an induction on the typing derivation instead of reduction in order to handle directly the non-syntactic rules as one step of the induction.

But then how do we relate the induction on the typing derivation to the induction on the reduction ? For, this we do a three-level matching: first, a match on typing induction in order to handle the non-syntactic rules, then a match on the reduction term, and then a final match on the typing derivation. We do this in three steps in order for F* to do some automation: after the first match, F* knows that the non-syntactic rules are already handled. After the second match, F* knows what is the syntax of the expression. With these two information, F* directly knows what is the last rule applied in the typing derivation. We show a piece of the proof that shows this three level match:

```
let rec pure_typing_preservation g e e' t wp post hwf ht hstep hv = (* ht: the typing derivation, hstep: the reduction proof*)
match ht with
| TySub ht' hsc → ... (*handling of TSub*)
| TyRet t htot → ... (*handling of TRet*)
| _ → (
      (* At this point, F* knows that ht is not built with TSub or TRet*)
      match hstep with
      | PsBeta t e e' →
        (
          (*At this point, F* knows that the typed expression is (\lambda t. e) e' *)
          (*And it can deduce that the last rule of the typing derivation is an application rule *)
          let TyApp htfuns htargs htotargs hktrets = ht in
          ... (*end of the proof for the beta case*)
        )
      | ... (*other cases*)
      )
```

This induction with the typing derivation and this three-level matching simplified a lot the initial proof of the preservation lemma, and removed the need for almost all inversion lemmas (the PsBeta case still needs an inversion for lambda-abstraction) where the paper proof needs them.

But during the proof, we found that other cases (which were forgotten by the paper proof) needed inversion lemmas: the fully-applied expression constants, and each of them needed a specific inversion lemma.

We also found out that the inversion lemmas were needed in other places than the preservation lemma. For example, the progress proof needs them too.

So, even though we simplified the proof of the preservation lemma, the need for a lot of inversion lemmas was still there.

Generic inversion lemma. But we can still try to avoid writing several inversion lemmas, by writing a generic inversion lemma that does not give the specific properties needed in each situation but allows to compute them outside of this lemma out of more general properties it would return. In order to do this, we first make the following remark: the typing derivation of $\Gamma \vdash e : c$ is a chain of non-syntactic rules (maybe empty) with a final typing rule which acts syntactically. For example, when we inductively go through the typing derivation of $\Gamma \vdash \lambda t.e : c$, we will first meet a number of T-Ret and T-Sub rules, and then we will hit a T-Abs rule.

Also, one can see that the information given by inversion lemmas can be derived from the information contained in the final syntactic rule and from the chain of non-syntactic rules.

So we just need to return these two pieces of information.

In order to handle the first part, we need to represent typing derivations where the last applied rule is a syntactic rule, that is, not a non-syntactic rule. We do this using F* refinements:

```
ht:typing g e c{not(is_TySub ht) && not (is_TyRet ht)}.
```

Note that we are using is_TySub and is_TyRet predicates that are automatically built by F* when defining inductive types.

In order to handle the second part, we define a type that represents the chain of non-syntactic rules that we call scmpex for SubCoMPutationEXtended. This type is parametrized by four elements: the environment g, the typed expression e, and the two computations c and c' that are at the borders of the chain.

The generic inversion lemma just builds and returns these two elements. And that's all. Now, we can use this lemma in all situations where inversion is needed and derive what we need in some specific case out of what is returned. The interested reader can see more details in the appendix (see A.5).

Beta-reduction case. As we said earlier, the fix done to the values definition during the proof of the substitution lemma made F* less complete. In order to keep the same completeness, we wanted to switch to CBN.

But switching to CBN raises a problem in the preservation lemma proof because of the following facts: First, in order to prove the β-case ($(\lambda t.e)\ e' \to e[e'/0]$), one needs to use the substitution lemma. Second, the substitution lemma needs the expressions it substitutes with (e' in the example) to be total.

But in general, expressions are not total. However, values are total. So we can prove the β-reduction case for CBV but not for CBN.

In order to switch to CBN, we tried to find other conditions under which an expression was total. We introduced the notion of satisfiability of WPs at this point:

Definition 1. *(Satisfiability) A WP wp is said to be satisfiable in the environment* Γ *if there exists a postcondition p such that the precondition wp post is satisfied in* Γ *:* $\Gamma \models wp\ p$.

One can show as a first property that an expression typed with a satisfiable WP is total. Also, as a second property, one can show that if the WP of an application $e_1\ e_2$ is satisfiable, then the WP of e_2 is satisfiable.

So, in order to use CBN anyway, we weakened the preservation lemma for expressions, and asked as premise that the WP associated to the typing judgment is satisfiable.

With the two properties in mind, the new preservation property can be proved. Indeed, in the β-reduction case $(\lambda t.e)\ e'$, one can get the satisfiability of e' with the premise, and deduce the fact that e' is total. So we can apply the substitution lemma and conclude that $e[e'/0]$ has the same computation type than $(\lambda t.e)\ e'$.

But, it was not possible to prove the second property in this mechanized proof simply. So we added a rule in the validity judgment axioms which entails this second property, and should be seen as an admissible rule:

$$\frac{\Gamma \vdash e : \mathsf{PURE}\ t\ wp}{\Gamma \models \mathsf{forall}\ p{:}\mathsf{postcondition}.\ wp\ p \Longrightarrow p\ e}\ (\text{V-Construct})$$

Results. At the end, we partially proved the following lemma:

Theorem 2. *(Preservation) If* $\Gamma \vdash e : \mathsf{PURE}\ t\ wp$, *e reduces to e', and wp is satisfiable in* Γ *then* $\Gamma \vdash e' : \mathsf{PURE}\ t\ wp$.
If $\Gamma \vdash t : k$ *and t reduces to t' then* $\Gamma \vdash t' : k$.

By changing the induction done for the preservation lemma, and by writing this generic inversion lemma, we substantially simplified the proof for preservation as it was first stated in the paper proof: in the proof of

the preservation lemma, we do not need anymore to apply an inversion lemma at each step. So we do not need anymore to prove difficult lemmas about WPs transformations that were needed before. These lemmas had quit difficult proofs, were returning big and complicated types, were not flexible and were used only once. So this is a huge gain.

Also this design is generic enough to handled all situations where inversion is needed in Micro-F*. Plus, the complexity of the proofs are reduced since we only return a simple structure with a typing judgment and a scmpex in every case, instead of different complicated structures, which are all case specific So this design is really useful and could be reused easily in other contexts.

Even with these new techniques, it was too difficult to write a complete proof. Even though most of the cases are handled, the most problematic one, which is the fix-point case, is not: the encoding of the type of the fix-point is too heavy to handle for F* at this point. Also, there are two holes in two other cases, due to the difficulty to get a subkinding relation out of an expression/typing equality. For example, if we have a kind k with a free variable x and an equality judgment $\Gamma \models \mathsf{e} = \mathsf{e'}$, we need to build a subkinding relation $\Gamma \vdash \mathsf{k[e/x]} <: \mathsf{k[e'/x]}$. Because of DeBruijn indices, this is actually very difficult, and we are still trying to figure out how to write a simple proof for it. Otherwise, the results are encouraging.

Also, during the proof, we were able to find a bug in the specification, which is that preservation does not necessarily hold in an inconsistent environment.

The reader can refer to the appendix (A.6) for details of implementation.

4 Progress

We want to prove here the usual progress lemma: an expression which is typable in the empty environment and which is not a value can be reduced.

The definition of values is quite heavy because of constants: every partial application of constants is considered as a value. Without automation, this would make the proof quite difficult since we would need to show that all the non-values cases are handled by hand. But hopefully, F* is able to check automatically that all the possible cases are handled.

The proof is done by induction on the typing derivation. Most of the cases for this match on the derivation are trivial: either we apply progress recursively (T-Sub, or T-Ret ...), or we return a simple reduction term (T-If0). The only difficult case is the application case $\mathsf{e}_1\ \mathsf{e}_2$. There are different cases:

- if e_1 is not a value, we apply progress recursively on e_1
- if e_2 is not a value, we apply progress recursively on e_2
- if e_1 is a lambda abstraction, we do a beta reduction
- otherwise, we are facing a fully applied constant

Fully applied constants are responsible for most of the work and the code on progress proof. Each one of them need to be treated individually. Here are the main cases:

- Pure fix-point: this is the easiest case since we do not need to check anything about the arguments and we can directly apply the reduction step for fixpoint
- Pure operations: these are pure manipulations of heaps at the logical level. There are two operations: the select, which takes a heap and a location as arguments and returns the value in the heap at this location, and the update, which takes a heap, a location and an integer as arguments and returns the modified heap.
- Impure operations: these are constants that are not possible in a pure computation: the bang, which reads a reference, the assign, which assigns an integer to a reference, and the impure fixpoint. But it is not possible to remove these cases automatically. We need to show that such cases trigger a contradiction at the type-level.

The case of pure operations. What prevents us to apply the reduction step on these fully applied constants is that the reduction steps associated to these operations behave syntactically: the argument, that is expected to be of type heap/integer/locations, needs to be a syntactic heap/integer/location. So, we proceed the following way:

- we use inversion lemma to get the necessary information about the arguments of the fully applied constants: we build a typing judgment for each argument, such that the argument which is supposed to be a heap/integer/location at the end is typed as heap/integer/location.

- we show that, in the empty environment, a value of type heap/integer/location is a syntactic heap/integer/location

These two points are not trivial and will be explained in detail in the next two paragraphs.

Multiple Inversions for Application scheme. So, how do we show that the argument which is supposed to be a heap/integer/location at the end can be typed as heap/integer/location ? In order to keep things simple, we will focus on only one example, which is the update constant. The only constraint about the arguments of this constant is in the type of this constant, which is heap → location → int → heap (only the types of computation types are written). We call this type the **base type**.

But relating the type of the constant to the type of the arguments is not simple, because of the subtyping rules that can be applied anywhere, and in particular on the constant. Since the subtyping relation includes beta reduction at the type level, we can not even assume that we are handling an arrow type with three arguments at each step. The only thing we know about the shape of the handled types is that whenever an argument is applied to the (partially applied) constant, the type of the (partially applied) constant is t → t' and the type of its argument is t. We call these types the **upper types**. What we want to do is to relate the base type to the upper types. We do this in three steps (see figure 4):

- Unroll the type derivation of the fully applied constants, using the generic inversion lemma. This produces scmpex relations between the different upper types (the blue arrows on the figure)
- Hit the constant introduction and introduce the type for it (bottom of the figure)
- Apply back the arguments and use the subtyping relations in the scmpex to build a subcomputation relation between the arguments types in the base type and the argument types in the upper types (red arrows on the figure), and also to get a subcomputation relation for the returned type (the green arrow on the figure)

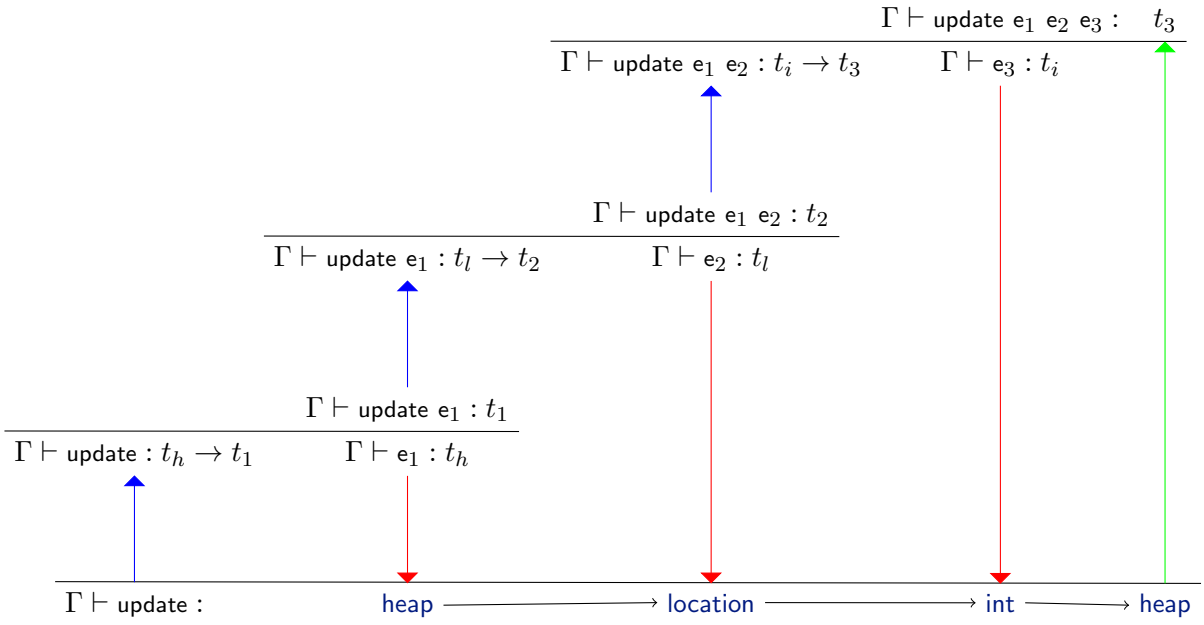

Figure 8: Multiple inversions for application

Once we have the subtyping relation, we can apply subtyping to show that the arguments can be typed with the right type.

This scheme, where one applies several times sequentially inversion lemmas to handle an application with several arguments, is used in other places, with something different each time. We call this design the **multiple inversions for application scheme**.

Getting syntactic heap/integer/location. So now we have values of type heap/integer/location. How do we prove that they are syntactically heaps/integers/locations ? The way we proceed is to handle all possibles values and show that only one is possible, by reaching a contradiction in all other cases. The way we show a contradiction is to build a validity term of false in micro-F*, and use an axiom which raises the proof of false in micro-F* to a contradiction at F* level: this axiom is basically the assumption that the empty environment

is consistent (that is, that you can not build a syntactic proof of false). Once false is in the context of the proof, we can finish the proof directly.

In order to show a contradiction, we go recursively in the typing derivation and we maintain an equality proof term between the type we began with (heap/integer/location) and the type of the current subtree of the derivation, until we hit a rule which is not a non-syntactic one. Once we have this equality, we can use an axiom in the validity judgment which states that two types in normal form that are syntactically different are different for the validity judgment. For example, if we start with e of type int, we maintain a judgment $. \models t = \text{int}$ as we go recursively in the typing derivation of e. Then, we hit a rule which is not a subtyping one. There are different cases that are basically all the kinds of values we can encounter:

- if it is an integer introduction, we get the result of the lemma
- if it is a heap or a location introduction, we have an equality $. \models t = \text{int}$ with $t = \text{heap}$ or $t = \text{location}$ from which we deduce a contradiction
- if it is a lambda introduction, we have an equality $. \models t_1 \to t_2 = \text{int}$ which can be converted to a contradiction
- if it is a partially applied constant, one proves that the final type is equal to an arrow, which produces the same contradiction than with lambda abstraction. In order to do this, a multiple inversions for application scheme is used, this time to relate the returned type at the base level to the returned type at the upper level. For example, if we have a judgment $. \vdash \text{update } e_1\ e_2 : t$, we show that $. \vdash \text{int} \to \text{heap} <: t$. From this, we can prove, using a helper lemma, that t itself is equal to an arrow type $t_1 \to t_2$, and, by transitivity, $t_1 \to t_2 = \text{int}$ and we get a contradiction.

Once we proved this lemma, we can do a reduction step on update and select operations.

The case of impure computations. There are still one case left in the proof of progress, which are the fully applied constants that lead to impure computations. For this paragraph, we introduce back the notion of effect of computations. The idea for this case is to do again a multiple inversions for application scheme in order to get the fact that an effectful computation is a subcomputation of a pure one. For example, for the assign operation: the assign operation has the following type: location → int → ALL unit wp. When we have a typing derivation $. \vdash \text{assign } e_1\ e_2 : c$, where $c = \text{PURE } t\ wp'$, we use the multiple inversions for application scheme to get a subcomputation term: $. \vdash \text{ALL unit wp} <: \text{PURE } t\ wp'$, which is impossible since the subcomputation constructor prevents syntactically an effectful computation to be a subcomputation of a pure computation. This introduces the contradiction in F* and let us finish the proof.

Results. At the end, we were able to almost completely prove the progress lemma:

Theorem 3. *(Progress) If* $\Gamma \vdash e : c$ *and* e *is not a value then* e *reduces to some* e'.

Contrary to the preservation proof, all the cases are handled. But the proof still relies on lemmas about WPs that are partially proved. This proof was also an occasion to develop several variants of the multiple inversions for application scheme, and a way to test the flexibility of the generic inversion lemma. The results are quite satisfying and it should make the case for reusing these schemes in other proofs about programming languages.

We also fixed the metatheory of micro-F*: at some point, the select and update constants were not reducible when fully applied (so Ps-Select and Ps-Update did not exist). They were just used at the logical level and the the properties about them were axioms at the logical level.

But still, when fully applied, select would produce a value of type int. This made the following typable expression not reducible: if select h l then e else e', which is not a value and so breaks the progress lemma. So we introduced reduction rules for select and update in order to fix the problem.

We refer the reader to the appendix (A.7) for implementation details.

Conclusion

Due to space constraint, we refer the reader to page 2 for a summary of contributions and future work.

A Pieces of code of the project

A.1 Definitions of terms

```
type eff =
  | EfPure
  | EfAll

type econst =
  | EcUnit
  | EcInt : i:int → econst
  | EcLoc : l:loc → econst
  | EcBang
  | EcAssign
  | EcSel
  | EcUpd
  | EcHeap : h:heap → econst
  | EcFixPure : tx:typ → t':typ → t'':typ → wp:typ → econst
  | EcFixOmega : tx:typ → t':typ → wp:typ → econst


and tconst =
  | TcUnit
  | TcInt
  | TcRefInt
  | TcHeap

  | TcFalse
  | TcAnd

  | TcForallE
  | TcForallT : k:knd → tconst

  | TcEqE
  | TcEqT : k:knd → tconst

  | TcPrecedes

and knd =
  | KType : knd
  | KKArr : karg:knd → kres:knd → knd
  | KTArr : targ:typ → kres:knd → knd

and typ =
  | TVar : a:var → typ
  | TConst : c:tconst → typ

  | TArr : t:typ → c:cmp → typ
  | TTLam : k:knd → tbody:typ → typ
  | TELam : t:typ → tbody:typ → typ
  | TTApp : t1:typ → t2:typ → typ
  | TEApp : t:typ → e:exp → typ

and exp =
  | EVar : x:var → exp
  | EConst : c:econst → exp
  | ELam : t:typ → ebody:exp → exp
  | EIf0 : eguard:exp → ethen:exp → eelse:exp → exp
  | EApp : e1:exp → e2:exp → exp
and cmp =
  | Cmp : m:eff → t:typ → wp:typ → cmp
```

A.2 Definitions of reduction

```
type tstep : typ → typ → Type =
  | TsEBeta : tx:typ →
              t:typ →
              e:exp →
              tstep (TEApp (TELam tx t) e) (tsubst_ebeta e t)
  | TsTBeta : k:knd →
```

```
                t:typ →
                t':typ →
                tstep (TTApp (TTLam k t) t') (tsubst_tbeta t' t)
  | TsArrT1 : #t1:typ→
                #t1':typ→
                c:cmp →
                =ht:tstep t1 t1' →
                tstep (TArr t1 c) (TArr t1' c)
  | TsTAppT1 : #t1:typ →
                 #t1':typ →
                 t2 : typ →
                 =ht:tstep t1 t1' →
                 tstep (TTApp t1 t2) (TTApp t1' t2)
  | TsTAppT2 : t1:typ →
                 #t2:typ →
                 #t2':typ →
                 =ht:tstep t2 t2' →
                 tstep (TTApp t1 t2) (TTApp t1 t2')
  | TsEAppT : #t:typ →
                #t':typ →
                e:exp →
                =ht:tstep t t' →
                tstep (TEApp t e) (TEApp t' e)
  | TsEAppE : t:typ →
                #e:exp →
                #e':exp →
                =he:epstep e e' →
                tstep (TEApp t e) (TEApp t e')
  | TsTLamT : k:knd →
                #t:typ →
                #t':typ →
                =ht:tstep t t' →
                tstep (TTLam k t) (TTLam k t')
  | TsELamT1 : #t1:typ →
                 #t1':typ →
                 t2:typ →
                 =ht:tstep t1 t1' →
                 tstep (TELam t1 t2) (TELam t1' t2)
and epstep : exp →exp →Type =
  | PsBeta : t:typ →
                ebody:exp →
                e:exp →
                epstep (EApp (ELam t ebody) e) (esubst_ebeta e ebody)
  | PsIf0 : e1:exp →
               e2:exp →
               epstep (EIf0 (eint 0) e1 e2) e1
  | PsIfS : i:int{i<>0} →
               e1:exp →
               e2:exp →
               epstep (EIf0 (eint i) e1 e2) e2
  | PsAppE1 : #e1:exp →
                #e1':exp →
                e2:exp →
                =ht:epstep e1 e1' →
                epstep (EApp e1 e2) (EApp e1' e2)
  | PsAppE2 : e1:exp →
                #e2:exp →
                #e2':exp →
                =ht:epstep e2 e2' →
                epstep (EApp e1 e2) (EApp e1 e2')
  | PsLamT : #t:typ →
               #t':typ →
               ebody:exp →
               =ht:tstep t t' →
               epstep (ELam t ebody) (ELam t' ebody)
  | PsIf0E0 : #e0:exp →
                #e0':exp →
                ethen:exp →
                eelse:exp →
                =ht:epstep e0 e0' →
```

```
                epstep (EIf0 e0 ethen eelse) (EIf0 e0' ethen eelse)
  | PsFixPure : tx:typ → t':typ → t'':typ → wp:typ →
                 d:exp → f:exp → v:value →
                 epstep (eapp3 (EConst (EcFixPure tx t' t'' wp)) d f v)
                (eapp2 f v (eapp2 (EConst (EcFixPure tx t' t'' wp)) d f))
  | PsUpd : h:heap → l:loc → i:int →
            epstep (eapp3 (EConst EcUpd) (eheap h) (eloc l) (eint i)) (eheap (upd_heap l i h))
  | PsSel : h:heap → l:loc →
            epstep (eapp2 (EConst EcSel) (eheap h) (eloc l)) (eint (h l))
```

A.3 Definition of type system

```
type typing : env → exp → cmp → Type = (*typing judgment*)

| TyVar : #g:env → x:var{is_Some (lookup_evar g x)} →
            typing g (EVar x) (tot (Some.v (lookup_evar g x)))

| TyConst : g:env → c:econst → ecwf g c →
            typing g (EConst c) (tot (econsts c))

| TyAbs : #g:env →
          #t1:typ →
          #ebody:exp →
          m:eff →
          t2:typ →
          wp:typ →
          =hk:kinding g t1 KType →
           typing (eextend t1 g) ebody (Cmp m t2 wp) →
           typing g (ELam t1 ebody) (tot (TArr t1 (Cmp m t2 wp)))

| TyIf0 : g:env → e0 : exp → e1:exp → e2:exp → m:eff →
          t:typ → wp0 : typ → wp1 : typ → wp2:typ →
          typing g e0 (Cmp m tint wp0) →
          typing g e1 (Cmp m t wp1) →
          typing g e2 (Cmp m t wp2) →
          typing g (EIf0 e0 e1 e2) (Cmp m t (ite m t wp0 wp1 wp2))

| TyApp : #g:env → #e1:exp → #e2:exp → #m:eff → #t:typ →
          #t':typ → #wp:typ → #wp1:typ → #wp2:typ →
          =ht1:typing g e1 (Cmp m (TArr t (Cmp m t' wp)) wp1) →
      =ht2:typing g e2 (Cmp m t wp2) →
          =htot:option (typing g e2 (tot t)){teappears 0 t' ⟹ is_Some htot} →
          =hk:option (kinding g (teshd t') KType){not (teappears 0 t') ⟹ is_Some hk} →
      typing g (EApp e1 e2) (Cmp m (tsubst (sub_ebeta e2) t') (tyapp_wp m e2 t t' wp wp1 wp2))

| TyRet : #g:env → #e:exp → t:typ →
          typing g e (tot t) →
          typing g e (Cmp EfPure t (return_pure t e))

| TySub : #g:env → #e:exp → #c':cmp → #c:cmp →
          =ht:typing g e c' →
          =hsc:scmp g c' c →
              typing g e c

and scmp : g:env → c1:cmp → c2:cmp → Type = (*subcomputation judgment*)

| SCmp : #g:env → m':eff → #t':typ → wp':typ →
          m:eff{eff_sub m' m} → #t:typ → wp:typ →
         =hs:styping g t' t →
         =hk:kinding g wp (k_m m t) →
         =hvmono:validity g (monotonic m t wp) →
         =hk':kinding g wp' (k_m m' t') →
         =hvmono':validity g (monotonic m' t' wp') →
     =hvsub :validity g (sub_computation m t wp m' t' wp') →
             scmp g (Cmp m' t' wp') (Cmp m t wp)


and styping : g:env → t':typ → t:typ → Type = (*subtyping judgment*)
| SubConv : #g:env → #t:typ → t':typ →
            =hv:validity g (teqtype t' t) →
```

```
            =hk:kinding g t KType →
        =hk':kinding g t' KType →
                styping g t' t

| SubFun : #g:env → #t:typ → #t':typ →
           #c':cmp → #c:cmp →
           =hst:styping g t t' →
           =hsc:scmp (eextend t g) c' c →
       =hk':kinding g (TArr t' c') KType →
                styping g (TArr t' c') (TArr t c)

| SubTrans : #g:env → #t1:typ → #t2:typ → #t3:typ →
              =hs12:styping g t1 t2 →
              =hs23:styping g t2 t3 →
                    styping g t1 t3
and tcwf : g:env → tc:tconst → Type = ... (*type constant wellformedness*)
and ecwf : g:env → ec:econst → Type = ... (*expression constant wellformedness*)
and kinding : g:env → t : typ → k:knd → Type =

| KVar : #g:env → x:var{is_Some (lookup_tvar g x)} →
           kinding g (TVar x) (Some.v (lookup_tvar g x))

| KConst : #g:env → #c:tconst →
             tcwf g c →
             kinding g (TConst c) (tconsts c)

| KArr : #g:env → #t1:typ → #t2:typ → #phi:typ → #m:eff →
         =hk1:kinding g t1 KType →
         =hk2:kinding (eextend t1 g) t2 KType →
         =hkp:kinding (eextend t1 g) phi (k_m m t2) →
         =hv :validity (eextend t1 g) (monotonic m t2 phi) →
             kinding g (TArr t1 (Cmp m t2 phi)) KType

...

and skinding : g:env → k1:knd → k2:knd → Type= ... (*subkinding judgment *)
and kwf : env → knd → Type = ... (*kind wellformedness *)
and validity : g:env → t:typ → Type = (*validity judgment*)

| VRedE : #g:env → #e:exp → #t:typ → #e':exp →
           =ht :typing g e (tot t) →
           =ht':typing g e' (tot t) →
           =hst:epstep e e' →
                validity g (teqe t e e')


| VSubstE : #g:env → #e1:exp → #e2:exp → #t':typ → t:typ →
             =hv12 :validity g (teqe t' e1 e2) →
             =ht1 :typing g e1 (tot t') →
             =ht2 :typing g e2 (tot t') →
             =hk :kinding (eextend t' g) t KType →
             =hvsub:validity g (tsubst_ebeta e1 t) →
                    validity g (tsubst_ebeta e2 t)


| VForallIntro : #g:env → #t:typ → #phi:typ →
                   =hk:kinding g t KType →
                   =hv:validity (eextend t g) phi →
                       validity g (tforalle t phi)

| VAndIntro : #g:env → #p1:typ → #p2:typ →
               =hv1:validity g p1 →
               =hv2:validity g p2 →
                    validity g (tand p1 p2)

| VFalseElim : #g:env → #t:typ →
                =hv:validity g tfalse →
                =hk:kinding g t KType →
                    validity g t
```

```
| VDistinctTH : #g:env → #t1:typ{is_hnf t1} →
                          #t2:typ{is_hnf t2 && not (head_eq t1 t2)} →
              =hk1:kinding g t1 KType →
              =hk2:kinding g t2 KType →
                   validity g (tnot (teqtype t1 t2))
```

We also define a judgment for the well-formedness of environments:

```
type ewf : env → Type =

| GEmpty : ewf empty

| GType : #g:env → #t:typ →
            =hw:ewf g →
            =hk:kinding g t KType →
                ewf (eextend t g)

| GKind : #g:env → #k:knd →
            =hw:ewf g →
            =h:kwf g k →
               ewf (textend k g)
```

A.4 Substitution of judgments

We first define in the code what are substitution arrows:

```
type subst_typing : s:sub → g1:env → g2:env → Type =
| SubstTyping : s:sub→  g1:env → g2:env →
                ef:(x:var{is_Some (lookup_evar g1 x)} →
                    Tot(typing g2 (Sub.es s x) (tot (tsubst s (Some.v (lookup_evar g1 x)))))) →

                tf:(a:var{is_Some (lookup_tvar g1 a)} →
                    Tot(kinding g2 (Sub.ts s a) (ksubst s (Some.v (lookup_tvar g1 a))))) →
                subst_typing s g1 g2
| RenamingTyping : s:sub → g1:env → g2:env →
                ef:(x:var{is_Some (lookup_evar g1 x)} →
                    Tot(hr:typing g2 (Sub.es s x) (tot (tsubst s (Some.v (lookup_evar g1 x))))){is_TyVar hr})) →

                tf:(a:var{is_Some (lookup_tvar g1 a)} →
                    Tot(hr:kinding g2 (Sub.ts s a) (ksubst s (Some.v (lookup_tvar g1 a)))){is_KVar hr})) →
                subst_typing s g1 g2
```

Then, we prove the lemma by declaring mutually recursive functions of the following types:

```
val typing_substitution : #g1:env → #e:exp → #c:cmp → s:sub → #g2:env →
    h1:typing g1 e c →
    hs:subst_typing s g1 g2 →
    Tot (hr:typing g2 (esubst s e) (csubst s c) {is_RenamingTyping hs ∧ is_TyVar h1 ⟹ is_TyVar hr} )
(decreases %[is_tyvar h1; is_renaming_typing hs; 1;h1])
val scmp_substitution : #g1:env → #c1:cmp → #c2:cmp → s:sub → #g2:env →
    h1:scmp g1 c1 c2 →
    hs:subst_typing s g1 g2 →
    Tot (scmp g2 (csubst s c1) (csubst s c2) )
(decreases %[1; is_renaming_typing hs; 1; h1])
val styping_substitution : #g1:env → #t':typ → #t:typ → s:sub → #g2:env →
    h1:styping g1 t' t →
    hs:subst_typing s g1 g2 →
    Tot (styping g2 (tsubst s t') (tsubst s t) )
(decreases %[1;is_renaming_typing hs; 1; h1])
val tcwf_substitution : #g1:env → #c:tconst → s:sub → #g2:env →
                        h1:tcwf g1 c →
             hs:subst_typing s g1 g2 →
             Tot (tcwf g2 (tcsubst s c))
(decreases %[1;is_renaming_typing hs; 1; h1])
val ecwf_substitution : #g1:env → #ec:econst → s:sub → #g2:env →
                        h1:ecwf g1 ec →
             hs:subst_typing s g1 g2 →
             Tot (ecwf g2 (ecsubst s ec))
(decreases %[1;is_renaming_typing hs; 1; h1])
val kinding_substitution : #g1:env → #t:typ → #k:knd → s:sub → #g2:env →
    h1:kinding g1 t k →
    hs:subst_typing s g1 g2 →
```

```
    Tot (hr:kinding g2 (tsubst s t) (ksubst s k){is_RenamingTyping hs ∧ is_KVar h1 ⟹ is_KVar hr})
(decreases %[is_kvar h1; is_renaming_typing hs; 1; h1])
val skinding_substitution : #g1:env → #k1:knd → #k2:knd → s:sub → #g2:env →
    h1:skinding g1 k1 k2 →
    hs:subst_typing s g1 g2 →
    Tot (skinding g2 (ksubst s k1) (ksubst s k2) )
(decreases %[1; is_renaming_typing hs; 1; h1])
val kwf_substitution : #g1:env → #k:knd → s:sub → #g2:env →
    h1:kwf g1 k →
    hs:subst_typing s g1 g2 →
    Tot (kwf g2 (ksubst s k))
(decreases %[1;is_renaming_typing hs; 1; h1])
val validity_substitution : #g1:env → #t:typ → s:sub → #g2:env →
    h1:validity g1 t →
    hs:subst_typing s g1 g2 →
    Tot (validity g2 (tsubst s t))
(decreases %[1;is_renaming_typing hs; 1; h1])
val elam_hs : #g1:env → s:sub → #g2:env → t:typ →
                            hs:subst_typing s g1 g2 →
                            Tot (hr:subst_typing (sub_elam s) (eextend t g1) (eextend (tsubst s t) g2){is_RenamingTyping hs ⟹ is_Renaming
(decreases %[1;is_renaming_typing hs; 0; EVar 0])
val tlam_hs : #g1:env → s:sub → #g2:env → k:knd →
                            hs:subst_typing s g1 g2 →
                            Tot (hr:subst_typing (sub_tlam s) (textend k g1) (textend (ksubst s k) g2){is_RenamingTyping hs ⟹ is_Renaming
(decreases %[1;is_renaming_typing hs; 0; TVar 0])
```

Note that the lexicographic metrics given looks like the ones for the substitution functions.

A.5 Simple inversion scheme

We first define the extended subcomputations judgment used to get a simpler generic inversion lemma:

```
type scmpex : env → e:exp → cmp → cmp → Type =
| SExTrans : #g:env → #e:exp → #c1:cmp → #c2:cmp → #c3:cmp →
               scmpex g e c1 c2 →
               scmpex g e c2 c3 →
          scmpex g e c1 c3
| SExSCmp : #g:env → #e:exp → #c1:cmp → #c2:cmp →
              scmp g c1 c2 →
         scmpex g e c1 c2
| SExRet : #g:env → #e:exp → #t:typ →
             typing g e (tot t) →
        scmpex g e (tot t) (return_pure_cmp t e)
```

Then we define what is returned by the inversion lemma:

```
type remove_subtyping_res : env → exp → cmp → Type =
| RemoveSub : #g:env → #e:exp → #c:cmp →
  c' : cmp →
  hsc: scmpex g e c' c →
  ht': typing g e c'{not(is_TySub ht') ∧ not(is_TyRet ht')} →
  remove_subtyping_res g e c
```

So the structure `remove_subtyping_res g e c` contains three things:

- another computation type `c'` defined existentially
- an extended subcomputation judgment that can relate the judgments $g \vdash e : c$ and $g \vdash e' : c'$
- and a typing judgment $g \vdash e : c'$ refined such that the last rule applied is neither a T-Sub rule nor a T-Ret rule, that is a non-syntactic rule.

It is a very lightweight returned type compared to the result of the initial inversion lemmas.
We finally declare the actual inversion lemma:

```
val remove_subtyping : #g:env → #e:exp → #c:cmp →
     hwf:ewf g →
     ht:typing g e c →
     Tot(remove_subtyping_res g e c)
```

It asks for a proof that the environment `g` to be well-formed (with the argument of type `ewf g`), and for a typing judgment, and returns a structure of the type defined before.

A.6 Preservation property

We prove the property by defining the two following functions:

```
val pure_typing_preservation : #g:env → #e:exp → #e':exp → #t:typ → #wp:typ → #post:typ →
      hwf:ewf g →
      ht:typing g e (Cmp EfPure t wp) →
      hstep:epstep e e' →
      hv :validity g (TTApp wp post) →
      Tot (either (typing g e' (Cmp EfPure t wp)) (validity g tfalse))
val pure_kinding_preservation : #g:env → #t:typ → #t':typ → #k:knd →
      hwf:ewf g →
      hk:kinding g t k →
      hstep:tstep t t' →
      Tot (either (kinding g t' k) (validity g tfalse))
```

The returned type of these lemmas is a sum type, which is either the fact that the reduced expression/type has the same type, or the fact that the environment is inconsistent.

A.7 Progress property

We first define a type to represent the conclusion of this lemma. We use an existential construction in order to introduce the expression `e'` to which `e` reduces:

```
type pure_progress_res : exp → Type =
| PureProgress : #e:exp →
    e' :exp →
    epstep e e' →
    pure_progress_res e
```

We then prove the progress property by defining a function of the following type :

```
val pure_progress : #e : exp{not (is_value e)} → #t:typ → #wp:typ →
  ht:typing empty e (Cmp EfPure t wp) →
  Tot (pure_progress_res e)
```

B Categorical interpretation

We give here more details about the categorical interpretation of the substitution of judgments.

We define the category of substitution arrows *SubArr* where the objects are the environments Γ of expression and type variables, and the arrows between two environments Γ_1 and Γ_2 are the subs `s` such that for all `x:t` $\in \Gamma_1$, one can find a typing derivation $\Gamma_2 \vdash$ `s.es(x)` : (`tsubst s t`) and for all `a:k` $\in \Gamma_1$, one can find a kinding derivation $\Gamma_2 \vdash$ `s.ts(a)` : (`ksubst s k`).

One can show that the arrows are composable using the substitution lemma.

In this interpretation, the sub transformers `sub_elam` and `sub_tlam` have a special role. In the categorical interpretation, they are defining two parametrized endofunctors called `elam_hs` and `tlam_hs`.

For some type `t`, the functor `elam_hs`$_t$ sends the object Γ to Γ, t and the substitution `s` to `sub_elam s`.

The main property of this functor is that it makes the following diagram commute.

$$\begin{array}{ccc} \Gamma_1, t & \xrightarrow{\text{sub\_elam s}} & \Gamma_2, (\text{tsubst s t}) \\ {\scriptstyle \text{eshift}}\uparrow & & \uparrow{\scriptstyle \text{eshift}} \\ \Gamma_1 & \xrightarrow{\text{s}} & \Gamma_2 \end{array}$$

The same for `tlam_hs`: for some kind `k`, the functor `tlam_hs`$_k$ sends the object Γ to Γ, k and the substitution `s` to `sub_tlam s`.

It makes the following diagram commute:

$$\begin{array}{ccc} \Gamma_1, k & \xrightarrow{\text{sub\_tlam s}} & \Gamma_2, (\text{ksubst s k}) \\ {\scriptstyle \text{tshift}}\uparrow & & \uparrow{\scriptstyle \text{tshift}} \\ \Gamma_1 & \xrightarrow{\text{s}} & \Gamma_2 \end{array}$$

C Additional work

C.1 Free variable handling

Because of the T-App1 and T-App2, we needed to handle free variables.

In order to reason with them, we defined boolean predicates:

```
val eeappears : x:var → e:exp → Tot bool
val teappears : x:var → t:typ → Tot bool
val keappears : x:var → k:knd → Tot bool
val eceappears : x:var → ec:econst → Tot bool
val tceappears : x:var → tc:tconst → Tot bool
val ceappears : x:var → c:cmp → Tot bool

let rec eeappears x e =
match e with
| EVar y → x = y
| EConst c → eceappears x c
| ELam t ebody → teappears x t || eeappears (x+1) ebody
| EIf0 eg et ee → eeappears x eg || eeappears x et || eeappears x ee
| EApp e1 e2 → eeappears x e1 || eeappears x e2
and teappears x t = ...
```

Then, we wrote several properties about free variables, in order to be able to handle the T-App rules.

First, a property of positive preservation: For s sub and e expression, if s.es(x) = x and x appears in e, then x appears in esubst s e. This property is represented in the code like this:

```
val esubst_on_eappears : x:var → s:sub{Sub.es s x = EVar x} → e:exp →
   Lemma (requires (eeappears x e)) (ensures (eeappears x (esubst s e)))
```

Then, a property of negative preservation: if x does not appear in e, and for all expression variable y which is not x, x does not appear in s.es(y), and for all type variable a, lsi{x} does not appear in s.ts(a), then x does not appear in esubst s e. In the code, we first represent the properties needed on the sub with an external type:

```
type sub_neappears : sub → var → var → Type =
| SubEAppears : s:sub → x:var → y:var →
                ef: (x':var{x<>x'} → Lemma( not (eeappears y (Sub.es s x')))) →
                tf: (a:var → Lemma (not (teappears y (Sub.ts s a)))) →
                sub_neappears s x y
```

Then we prove the lemma of the following type:

```
val esubst_on_neappears : #s:sub → #x:var → #y:var →
                          hs:sub_neappears s x y →
                          e:exp →
                          Lemma (requires (not (eeappears x e))) (ensures (not (eeappears y (esubst s e))))
```

We prove a last negative result: For e expression and s sub, if x does not appear in any expression or type given by s, then x does not appear in esubst s x.

In the code, as before, we define in an external type the needed properties:

```
type sub_neappears_all : s:sub → x:var → Type =
| SubNEAppearsAll : s:sub → x:var →
            ef: (y:var → Lemma (not (eeappears x (Sub.es s y)))) →
          tf: (a:var → Lemma (not (teappears x (Sub.ts s a)))) →
          sub_neappears_all s x
```

Then we prove the lemma of the following type:

```
val esubst_on_neappears_all : #s:sub → #x:var →
              hs:sub_neappears_all s x →
         e:exp →
         Lemma (not (eeappears x (esubst s e)))
```

C.2 Derived lemma

Another lemma took a lot of effort to be proved: the derived lemma. This lemma gives simple properties about the typing judgments made in a well-formed environment.

Theorem 4. *(Derived lemma) If the environment* Γ *is proved well-formed, then the following statements hold:*

- *Typing judgment: for* e *expression and* M t wp *computation, if* $\Gamma \vdash e : M\ t\ wp$, *then* $\Gamma \vdash t : Type$
- *Kinding judgment: for* t *type and* k *kind, if* $\Gamma \vdash t : k$, *then* $\Gamma \vdash k\ kwf$
- *Subtyping judgment: for* t', t *types, if* $\Gamma \vdash t' <: t$, *then* $\Gamma \vdash t' : Type$ *and* $\Gamma \vdash t : Type$
- *Subkinding judgment: for* k', k *kinds, if* $\Gamma \vdash k' <: k$, *then* $\Gamma \vdash k'\ kwf$ *and* $\Gamma \vdash k\ kwf$
- *Validity judgment: for* phi *formula, if* $\Gamma \models phi$, *then* $\Gamma \vdash phi : Type$

Just like the substitution lemma, this lemma was proved using an induction on all the judgments. During the proof, we also discovered some bugs in the metatheory.

C.3 Mechanized proof for the weakest-preconditions calculus

At the beginning of my internship, when I was in Saarbrücken to learn how to use F*, I made a project with another student, Enrico Steffinlongo.

The goal of this project was to write a mechanized proof of the correctness and completeness of the weakest-precondition calculus on a small language in F*. The correctness part is about proving that the computed precondition is sufficient to get the postcondition after the execution of the program. The completeness part is about proving that the computed precondition is the weakest, compared to the other valid Hoare triples with the same postcondition.

Definitions In order to do this, we first defined the formulas and the deduction system on it:

```
type form =
| FFalse : form
| FImpl : form → form → form
| FAnd : form → form → form
| FForall : (heap → Tot form) → form
| FEq : #a:Type → a → a → form

type deduce : form → Type =
  | DFalseElim :
              f:form →
              deduce FFalse →
              deduce f
  | DImplIntro :
              f1:form →
              f2:form →
              (deduce f1 → Tot (deduce f2)) → (* <- meta level implication *)
              deduce (FImpl f1 f2)
  | DImplElim :
          f1:form →
              f2:form →
              deduce (FImpl f1 f2) →
              deduce f1 →
              deduce f2
  | DAndIntro :
              f1:form →
              f2:form →
              deduce f1 →
              deduce f2 →
              deduce (FAnd f1 f2)
...
```

Then we defined a small language in F*, with its operational semantics:

```
type com =
| Skip : com
| Assign : var:id → term:aexp → com
| Seq : first:com → second:com → com
| If : cond:bexp → then_branch:com → else_branch:com → com
| While : cond:bexp → body:com → inv:pred → com

type reval : com → heap → heap → Type =
| ESkip : h0:heap →
                reval Skip h0 h0

| EAssign : h0:heap → x:id → e:aexp →
```

```
                reval (Assign x e) h0 (update h0 x (eval_aexp h0 e))

| ESeq : #h0:heap → #c1:com → #c2:com → #h1:heap → #h2:heap →
                reval c1 h0 h1 →
                reval c2 h1 h2 →
                reval (Seq c1 c2) h0 h2

| EIfTrue : #h0:heap → be:bexp{eval_bexp h0 be = true} →
                ct:com → ce:com → #h1:heap →
                reval ct h0 h1 →
                reval (If be ct ce) h0 h1

| EIfFalse : #h0:heap → be:bexp{eval_bexp h0 be = false} →
                ct:com → ce:com → #h1:heap →
                reval ce h0 h1 →
                reval (If be ct ce) h0 h1

| EWhileEnd : #h0:heap → be:bexp{eval_bexp h0 be = false} →
                cb:com → i:pred →
                reval (While be cb i) h0 h0

| EWhileLoop : #h0:heap → be:bexp{eval_bexp h0 be = true} →
                cb:com → i:pred → #h1:heap → #h2:heap →
                reval cb h0 h1 →
                reval (While be cb i) h1 h2 →
                reval (While be cb i) h0 h2
```

We then defined what we call Hoare triple in two different ways: one semantically and one syntactically. The semantic definition:

```
type hoare (p:pred) (c:com) (q:pred) : Type =
  (#h:heap → #h':heap → reval c h h' → deduce (p h) → Tot (deduce (q h')))
```

Here, a Hoare triple {p}c{q} is seen as a function which, for any heaps h,h', takes as argument a reduction proof of the program c which changes the state from h to h', and the validity of the predicate p on h and produces the validity of q on h'.

The syntactic definition:

```
type hoare_syn : pred → com → pred → Type =
  | HoareSynAssign : q:pred → x:id → e:aexp →
                     hoare_syn (pred_sub x e q) (Assign x e) q
  | HoareSynConsequence : p:pred → p':pred → q:pred → q':pred → c:com →
                          hoare_syn p' c q' →
                          deduce (pred_impl p p') →
                          deduce (pred_impl q' q) →
                          hoare_syn p c q
  | HoareSynSkip : p:pred →
                   hoare_syn p Skip p
  | HoareSynSeq : p:pred → c1:com → q:pred → c2:com → r:pred →
                  hoare_syn p c1 q →
                  hoare_syn q c2 r →
                  hoare_syn p (Seq c1 c2) r
  | HoareSynIf : p:pred → q:pred → be:bexp → t:com → e:com →
                 hoare_syn (pand p (bpred be)) t q →
                 hoare_syn (pand p (pnot (bpred be))) e q →
                 hoare_syn p (If be t e) q
  | HoareSynWhile : p:pred → be:bexp → c:com →
                    hoare_syn (pand p (bpred be)) c p →
                    hoare_syn p (While be c p) (pand p (pnot (bpred be)))
```

We finally defined the weakest-precondition calculus:

```
val wlp : com → pred → Tot pred
let rec wlp c q =
  match c with
  | Skip → q
  | Assign x e → pred_sub x e q
  | Seq c1 c2 → wlp c1 (wlp c2 q)
  | If be ct ce → pif (bpred be) (wlp ct q) (wlp ce q)
  | While be c' i →
    pand i (fun _ → FForall (pimpl i (pif (bpred be) (wlp c' i) q)))
```

Results

For the semantic version, we were able to prove only correctness. The completeness part is false with the semantic definition anyway, because of the while loops. Indeed, the weakest-precondition calculus uses the invariant given by the loop to build the precondition. But this invariant might not be the most efficient one, and can introduce non-necessary conditions.

The proof of correctness is done by defining a function of the following type:

```
val wlp_sound : c:com → q:pred → Tot (hoare (wlp c q) c q)
```

For the syntactic version, we were able to prove completeness, by defining a function of the following type:

```
val wlp_weakest' : c:com → p:pred → q:pred →
                   hpcq:hoare_syn p c q →
                   Tot (deduce (pred_impl p (wlp c q)))
```

In the conclusion, (pred_impl p (wlp c q)) is the formula which states that the precondition wlp c q is weaker than p. So the conclusion states precisely what we want. []

References

[1] F*'s home page. `https://fstar-lang.org/`.

[2] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, pages 453–468. Springer, 1999.

[3] Brian E Aydemir, Aaron Bohannon, Matthew Fairbairn, J Nathan Foster, Benjamin C Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics*, pages 50–65. Springer, 2005.

[4] Bruno Barras. Coq en Coq. 1996.

[5] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. *ACM SIGPLAN Notices*, 49(1):33–45, 2014.

[6] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[7] C. Keller. Substitutions for simply-typed λ-calculus. Internship Report, Ecole Normale Superieure de Lyon, 2008.

[8] C. Keller. The category of simply typed λ-terms in Agda. Unpublished note, 2008.

[9] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5-6):865–911, 2008.

[10] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In Xingyuan Zhang and Christian Urban, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015*, LNAI. Springer-Verlag, Aug 2015. To appear.

[11] Pierre-Yves Strub, Nikhil Swamy, Cédric Fournet, and Juan Chen. Self-certification: bootstrapping certified typecheckers in F* with Coq. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 571–584, 2012.

[12] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013.

[13] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cedric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, and Jean-Karim Zinzindohoue. Dependent types and multi-monadic effects in F*. 2016. `https://www.fstar-lang.org/papers/mumon/`.

[14] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Notices*, volume 48, pages 387–398. ACM, 2013.