Formal verification of System F

Formal Verification 2021 Final Report

Andrea Gilot

Noé De Santo

1 Introduction

Lambda calculi constitute the theoretical underpinnings of functional programming, a paradigm which became more and more popular over the last decades. This makes studying lambda calculi of interest, as the properties which can be proven about them more or less translate into properties of languages built on top of them. Moreover, in order to model more advanced features of programming languages, one can complexify the simply typed lambda calculus (STLC). System F is an example of such an extension which introduces parametric polymorphism [1], a feature that many programming languages support nowadays.

In this project we first proved key properties of STLC, and afterwards extended the work already done to prove them for System F as well.

2 Theoretical Background

We will now formally define System F and its main theorems. We will recall key theorems and definitions, but this section does not aim at being exhaustive, and expects prior knowledge about System F. Most of what we will discuss in this section can be found in [2, Chapter 23]. We will however adapt everything to De Bruijn notation [2, Chapter 6]

System F is defined in fig. 1, and is a quite straightforward adaption from [2, Figure 23-1]. There are however a few things to note:

- Integers are De Bruijn indices, and are used to represent both type and term variables; the context will always allow to distinguish which is meant. See appendix A for some examples;
- Ground types are represented by strings (`char*`);
- Environments are represented using lists;
- $\uparrow$ represents a term variable shift, whereas $\downarrow$ represents a type variable shift;
- In the T-TAbs typing rule, instead of adding a new dummy element[1] to the environment, we shift the environment. The intuition behind this is discussed in appendix B.

For this definition to be complete, we must still define the two operations it relies on: shifting and substitution. Note that many variants of these operations are used: substitution of a term in a term, term shift, substitution of a type in a term,… since these definitions are a bit heavy, they are made available in the appendix (fig. 9).

We can now introduce the main theorems we want to prove:

[1]The element is usually an identifier, so adding it allows to know which identifiers are defined; using De Bruijn notation, those identifiers will always be an integer range $[0, n]$, so there is no real information to store.

Theorem 2.1 (Type judgment uniqueness). *For all environment $\Gamma \in \mathbf{\Gamma}$, term $t \in \boldsymbol{\lambda}$ and types $T_1, T_2 \in \mathbf{\Lambda}$, we have*

$$\Gamma \vdash t : T_1 \ \wedge \ \Gamma \vdash t : T_2 \implies T_1 = T_2$$

Theorem 2.2 (Progress). *For all term $t \in \boldsymbol{\lambda}$ such that $\emptyset \vdash t : T$ for some type $T \in \mathbf{\Lambda}$, then either $\exists t' \in \boldsymbol{\lambda} : \ t \longrightarrow t'$ or t is a value.*

Theorem 2.3 (Preservation). *For all environment $\Gamma \in \mathbf{\Gamma}$, terms $t, t' \in \boldsymbol{\lambda}$ and type $T \in \mathbf{\Lambda}$, we have*

$$\Gamma \vdash t : T \ \wedge \ t \longrightarrow t' \implies \Gamma \vdash t' : T$$

Theorem 2.2 raises two questions:

1. What is a value?
2. Does it allow a term which is both a value and reducible?

These two questions are in fact intertwined. The typical definition of a value is a term which is (at the top-level) an (type-)abstraction[2]; in this case, the answers to the second question must be yes. We assumed this definition in our project.

It is however possible to restrict the reduction relation, in particular to make it deterministic; one such restriction is the call-by-value reduction strategy [2, Chapter 5]. It restricts the reduction relation in the following way:

- R-Abs and R-Tabs are not allowed;
- R-App2 requires t_1 to be a value;
- R-AbsApp requires t_2 to be value (t_1 is also a value!).

With these restrictions, the second answer can be no.

In this project, we will prove theorem 2.1 and theorem 2.3 in the most general way, i.e. for full β-reduction, but will only prove theorem 2.2 for the call-by-value reduction strategy.

3 Implementation

3.1 Preliminaries

Our project has been implemented and verified using Stainless [3] and is available on Github. The code is organized as follows:

- `Utils.scala` Additional theorems on `Option` and `List`;
- `SystemF.scala` Definition of types, terms, free variables and theorems on free variables;
- `Transformations.scala` Definitions and theorems related to substitution and shifting;
- `Reduction.scala` Definition and theorems related to the reduction relation;

[2]This answer has to change when constants are added to the calculus; see section 3.5

Syntax:

$$\begin{aligned}\boldsymbol{\Lambda} &::= \mathbb{N} \mid \texttt{char}^* \mid \boldsymbol{\Lambda} \to \boldsymbol{\Lambda} \mid \forall.\, \boldsymbol{\Lambda}\\ \boldsymbol{\Gamma} &::= \boldsymbol{\Lambda} :: \boldsymbol{\Gamma} \mid \emptyset\\ \boldsymbol{\lambda} &::= \mathbb{N} \mid \lambda \boldsymbol{\Lambda}.\, \boldsymbol{\lambda} \mid \boldsymbol{\lambda}\ \boldsymbol{\lambda} \mid \textsc{Fix}\, \boldsymbol{\lambda} \mid \Lambda.\, \boldsymbol{\lambda} \mid \boldsymbol{\lambda}\, [\boldsymbol{\Lambda}]\end{aligned}$$

Reduction:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{R-App1} \qquad \frac{t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}\ \text{R-App2} \qquad \frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]}\ \text{R-TApp} \qquad \frac{t \longrightarrow t'}{\textsc{Fix}\, t \longrightarrow \textsc{Fix}\, t'}\ \text{R-Fix}$$

$$\frac{t \longrightarrow t'}{\lambda T.\, t \longrightarrow \lambda T.\, t'}\ \text{R-Abs} \qquad \frac{t \longrightarrow t'}{\Lambda.\, t \longrightarrow \Lambda.\, t'}\ \text{R-TAbs} \qquad \frac{}{(\lambda T.\, t_1)\ t_2 \longrightarrow \uparrow_0^{-1} \left(\left[0 \mapsto \uparrow_0^1 (t_2)\right]\ t_1\right)}\ \text{R-AbsApp}$$

$$\frac{}{\textsc{Fix}\, \lambda T.\, t \longrightarrow \uparrow_0^{-1} \left(\left[0 \mapsto \uparrow_0^1 (\textsc{Fix}\, \lambda T.\, t)\right]\ t\right)}\ \text{R-AbsFix} \qquad \frac{}{(\Lambda.\, t_1)\ [T_2] \longrightarrow \downarrow_0^{-1} \left(\left[0 \mapsto \downarrow_0^1 (T_2)\right]\ t_1\right)}\ \text{R-TAbsTApp}$$

Typing:

$$\frac{(\Gamma)\ [k] = T}{\Gamma \vdash k : T}\ \text{T-Var} \qquad \frac{T_1 :: \Gamma \vdash t : T_2}{\Gamma \vdash \lambda T_1.\, t : T_1 \to T_2}\ \text{T-Abs} \qquad \frac{\Gamma \vdash t_1 : T_2 \to T \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T}\ \text{T-App}$$

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \textsc{Fix}\, t : T}\ \text{T-Fix} \qquad \frac{\downarrow_0^1 (\Gamma) \vdash t : T}{\Gamma \vdash \Lambda.\, t : \forall.\, T}\ \text{T-TAbs} \qquad \frac{\Gamma \vdash t : \forall.\, T}{\Gamma \vdash t\ [T_2] : \downarrow_0^{-1} \left(\left[0 \mapsto \downarrow_0^1 (T_2)\right]\ T\right)}\ \text{T-TApp}$$

Figure 1. Definition of System F

- `Typing.scala` Definition of typing derivation and main theorems of System F i.e. type uniqueness and type safety.

As mentioned, the proofs were originally done for the simply typed lambda calculus (with fixpoint). These original proofs are also available on our repository. Note however that many improvements were made to the proofs once we started working on System F, and these improvements were not back-ported to the simply typed lambda calculus proofs.

Although our proofs mainly mirror the ones from [2], many tricks were required to mechanize the proofs. We will first introduce the more general points of the original implementation for the simply typed lambda calculus, and then dwell on the additions we did once this original part was done.

3.2 Typing derivations

To prove anything related to the type of a term, we first need to define a function which tells us whether $\Gamma \vdash t : T$. Since terms have a unique type in System F, having a function `typeof` which, given an environment and a term, outputs its type would be convenient. However, doing so raises several issues:

- Some terms are ill-typed; what should the function output in this case?
- Type uniqueness of System F is a property we would like to prove, but we already use it implicitly in our definition of `typeof`: this is a chicken-and-egg problem;
- The type of a term depends directly of the type of the subterms and which typing rule is being applied, which are things that could be useful for some proofs. Instead, here the typing is a black box;
- How can we make sure that the implementation of such a procedure is correct?

To overcome all of these issues, we decided to first work on typing derivations. A typing derivation is a proof that some term has a type, given an environment. Formally it is defined as a sequence of typing rules where each premise of a rule is either the conclusion of a previous typing rule, or an axiom. It has therefor a tree-like structure (as seen in fig. 2), where each node is a typing rule.

$$\dfrac{\dfrac{\dfrac{(T :: T)\ [0] = T}{T :: T \vdash 0 : T}\ \text{T-Var}}{T \vdash \lambda T.\, 0 : T \to T}\ \text{T-Abs} \qquad \dfrac{(T)\ [0] = T}{T \vdash 0 : T}\ \text{T-Var}}{T \vdash (\lambda T.\, 0)\ 0 : T}\ \text{T-App}$$

Figure 2. Example of a typing derivation

We call the conclusion $\Gamma \vdash t : T$ of the final typing rule the conclusion of the typing derivation, and call Γ its environment, t its term and T its type.

This definition is easily captured by Scala's case classes. The trait `TypingDerivation` represents a generic typing derivation, and each typing rule has itw own case class (e.g. `AppDerivation` for T-App).

However, a `TypingDerivation` instance might not represent a correct typing derivation. Indeed, we could easily build a `TypingDerivation` object whose subderivations do not

conclude the required premises of this derivation. We thus need a function, `isValid`, which checks whether a given typing derivation is correct. More formally, if σ is a typing derivation concluding $\Gamma \vdash t : T$ whose subderivations are $\{\sigma_i\}$, then `isValid` is true if:

- `isValid` is true for each σ_i;
- The environment and type of each σ_i must correctly relate with T and Γ; the exact relation depends on the typing rule being used (e.g. for most rules, σ_i's environment must be equal to Γ, but in the case of T-TAbs it must be equal to $\downarrow_0^1(\Gamma)$);
- The term of each σ_i is the correct subterm of t.

We were then able to implement a function `deriveType`, that given a term and an environment produces a typing derivation.

Now that all this machinery is settled, we can find solutions to all of our issues:

- If a term t is ill-typed then for all typing derivations with term t, `isValid` will be false, and `deriveType` can simply return `None`;
- We can prove that two valid typing derivations with the same term and environment are equal; in particular, they conclude the same type. This proves theorem 2.1, without using it as an assumption!
- We can track all the typing rules that have been used, and we know the type of all the subterms thanks to the generated typing derivation.
- Finally, we can prove the correctness of `deriveType` by proving the following two lemmas:

Lemma 3.1 (`deriveType` soundness)**.** *For all environment $\Gamma \in \mathbf{\Gamma}$ and term $t \in \boldsymbol{\lambda}$ such that `deriveType(`Γ`,t)` is defined, we have*

$$\texttt{deriveType}(\Gamma,t).\texttt{isValid}$$

Lemma 3.2 (`deriveType` completeness)**.** *For all valid typing derivation σ with environment Γ and term t, we have*

$$\texttt{deriveType}(\Gamma,t) = \sigma$$

It is even possible to define the `typeof` function we originally wanted:

```
def typeof(t: Term): Option[Type] = {
    deriveType(∅, t).map(σ => σ.typ)
}
```

Figure 3. Implementation of `typeof`

3.3 Reduction relation

The way we implemented and verified the reduction relation is a bit reminiscent of how typing relations were done, but not quite similar:

- The "specification" was first implemented as a `reducesTo` function, which is used to determine whether a term reduces to another. It is implemented such that it is possible, without too much trouble, to ensure that it matches the definition in fig. 1;
- We then defined a function which has to determine if a term does reduce, and if it is the case, to which other term. This function also defines a reduction strategy. We decided to use call-by-value as a strategy, so the function is `reduceCallByValue`;
- We can then prove progress for the chosen reduction strategy.

```
def reducesTo(t: Term, tp: Term): Option[ReductionRule]
def reduceCallByValue(t: Term): Option[Term]
```

Figure 4. The reduction functions

One not so obvious thing to verify before moving to progress is that the reduction strategy is sound:

Theorem 3.3 (`reduceCallByValue` soundness)**.** *For all term $t \in \boldsymbol{\lambda}$ such that `reduceCallByValue(t)` is defined, we have*

$$t \longrightarrow \texttt{reduceCallByValue}(t)$$

Note that there is no real "completeness" criterion for reduction strategies, except for the progress property, which can be seen as a kind of completeness.

It is then (surprisingly) trivial to prove progress:

```
def callByValueProgress(t: Term): Unit = {
  require(deriveType(Nil(), t).isDefined)
  t match{
    case Var(_) => ()
    case Abs(_, _) => ()
    case App(t1, t2) => {
      callByValueProgress(t1)
      callByValueProgress(t2)
    }
    case Fix(f) => callByValueProgress(f)
    case TAbs(t) => ()
    case TApp(t, typ) => callByValueProgress(t)
  }
}.ensuring(
  reduceCallByValue(t).isDefined || t.isValue
)
```

Figure 5. Stainless proof of theorem 2.2

The proof is however tied to one particular evaluation strategy. One could try to generalize as much as possible, i.e. by proving it for full β-reduction. It might even be possible to prove it for some kind of non-deterministic reduction

strategy[3], but the proof cannot be given for all evaluation strategies at once. In particular, any reduction strategy which disallows a computation rule, such as R-AbsApp, (trivially) cannot ensure progress.

3.4 Free variables and negative shifts

Dealing with De Brujin's notation was unarguably the most time-consuming part of the project. In particular, the evaluation relation requires the use of negative shifts. When performing negative shifts, it must be verified that no variable will end up being below 0. Therefore, theorems to know which variables are free (the bond between free variables and validity of negative shifts is discussed in appendix C) and others to know how shifting and substituting affect free variables were required. In the rest of this section, we will focus on free variables inside types; however the same explanation holds for free term variables inside terms and free type variables inside terms as well.

Formally, the set of free variables of a type is defined as follows:

$$\begin{aligned}
\mathrm{FV}(k) &:= \{k\} \\
\mathrm{FV}(\texttt{char}^*) &:= \emptyset \\
\mathrm{FV}(T_1 \to T_2) &:= \mathrm{FV}(T_1) \cup \mathrm{FV}(T_2) \\
\mathrm{FV}(\forall.\, t) &:= \{x - 1 \mid x \in (\mathrm{FV}(t) \setminus \{0\})\}
\end{aligned}$$

However, what is really interesting is not the set of free variables of a given type, but a weaker property: whether this type contains free variables in a particular range (this is again related to the "free variables – negative shift" relation).

This property translates in checking whether the intersection of the set of free variables and the range of interest is empty; however, manipulating sets and set intersections would quickly become cumbersome.

An easier solution arose using the relation in fig. 6, whose right hand side is either trivial (`false` and $k \in [c,\ c+d[$) or can be computed using the relation recursively.

We were thus able to define `hasFreeVariablesIn`, a function that recursively computes the right hand side of the relation (see fig. 7), and then prove its correctness:

Lemma 3.4 (`hasFreeVariablesIn` completeness)**.** *For all type $T \in \mathbf{\Lambda}$ and $c, d, k \in \mathbb{N}$ such that*

$$k \in \texttt{T.freeVariables} \ \wedge\ k \in [c,\ c+d[$$

we have `T.hasFreeVariablesIn(c, d)`*.*

Lemma 3.5 (`hasFreeVariablesIn` soundness)**.** *For all type $T \in \mathbf{\Lambda}$ and $c, d \in \mathbb{N}$ such that*

$$\forall k \in \texttt{T.freeVariables}:\ k \notin [c,\ c+d[$$

we have $\neg$`T.hasFreeVariablesIn(c, d)`*.*

[3]Either by having the relation being truly non-deterministic, as Stainless seems to have a randomness source, or by having the function return all reduction candidates.

Corollary. *`hasFreeVariablesIn` is correct, i.e.*

$$\texttt{T.hasFreeVariablesIn(c, d)} \iff \mathrm{FV}(T) \cap [c,\ c+d[\ \neq \emptyset$$

```
def hasFreeVariablesIn(c: BigInt, d: BigInt) = {
  require(c >= 0)
  require(d >= 0)
  this match {
    case VariableType(v) =>
        (c <= v) && (v < c+d)
    case BasicType(_) => false
    case ArrowType(t1, t2) =>
        t1.hasFreeVariablesIn(c, d) ||
        t2.hasFreeVariablesIn(c, d)
    case UniversalType(body) =>
        body.hasFreeVariablesIn(c+1, d)
  }
}
```

Figure 7. Implementation of `hasFreeVariablesIn`

Therefore, we could prove theorems for `hasFreeVariablesIn`, which was way easier since we did not have to manipulate set intersections.

First, we proved how to manipulate ranges that do not contain free variables. In fact, if there are no free variables in $[a,\ b[$ then there are no free variables in $[a',\ b'[$ for $a' \geq a$ and $b' \leq b$. In addition, if there are no free variables in the ranges $[a,\ b[$ and $[b,\ c[$, then there are none in $[a, c[$ as well.

After that, we needed to track free variables after shifts or substitutions, leading to two main lemmas:

Lemma 3.6. *For all type $T \in \mathbf{\Lambda}$ and $a, a', c, d \in \mathbb{N}$ such that T has no free variables in $[a,\ a'[$, we have that*

$$\downarrow_c^d (T) \text{ has no free variables in } \begin{cases} [a+d,\ a'+d[& \text{if } c \leq a \\ [a,\ a'[& \text{if } c \geq a \end{cases}$$

Lemma 3.7. *For all types $T, S \in \mathbf{\Lambda}$ and $c, c', d, d', j \in \mathbb{N}$ such that S has no free variables in $[c,\ c'[$ and T has no free variables in $[d,\ d'[$, we have that*

$$[j \mapsto S]\ T \text{ has no free variables in } [\max(c,d),\ \min(c',d')[$$

These theorems were used to prove that some types will have a range without free variables, which means that a negative shift can be applied to the resulting type; in particular:

Corollary. *The negative shift in the* T-TApp *typing rule (fig. 1) is valid, i.e. $\downarrow_0^{-1} \left(\left[0 \mapsto \downarrow_0^1 (T_2)\right] T\right)$ is a type without negative variables.*

As already mentioned, all of this can also be applied to term shifting & substitution as well as for type shifting & substitution inside of terms or environments. In particular, the same lemmas can be proved as well.

For $c, d \in \mathbb{N}$

$$\begin{aligned}
\mathrm{FV}\,(k) \cap [c,\ c+d[\neq \emptyset &\iff k \overset{?}{\in} [c,\ c+d[\\
\mathrm{FV}\,(\texttt{char}^*) \cap [c,\ c+d[\neq \emptyset &\iff \texttt{false} \\
\mathrm{FV}\,(t_1 \to t_2) \cap [c,\ c+d[\neq \emptyset &\iff \left(\mathrm{FV}\,(t_1) \cap [c,\ c+d[\neq \emptyset\right) \vee \left(\mathrm{FV}\,(t_2) \cap [c,\ c+d[\neq \emptyset\right) \\
\mathrm{FV}\,(\forall.\ t) \cap [c,\ c+d[\neq \emptyset &\iff \mathrm{FV}\,(t) \cap [c+1, c+d+1[\neq \emptyset
\end{aligned}$$

Figure 6. Free variable range relation

3.5 Constants

Before expanding our work to System F, we considered extending STLC instead. In particular, we started adding constants to the calculus. One simple solution to do so is to "hard-code" the constants in an environment, which will be used instead of the empty environment to type top-level terms.

The example in fig. 2 can be seen as such: 0 is the only constant of the calculus, and in particular the only value of type T, and as such can be interpreted as being `unit`.

This approach however does not work if an infinite number of constants is needed, e.g. to represent natural numbers.

We thus implemented constants as a new construct of the calculus (also available on our repository), under the constraint that constants must have a ground type (i.e. they cannot have a function type). This addition was not too complicated, and did not add anything difficult (nor interesting) to the proofs.

The constraint on their type however was annoying: one typical set of constants which is nice to have is the set of integers `Nat`; however, having integers without any operation on them (such as `Succ`, `Pred`, `Add`,...) does not make them worthwhile.

Lifting the constraint yields some problems when mixed with the fixpoint construct; assume $\emptyset \vdash \texttt{Succ} : \texttt{Nat} \to \texttt{Nat}$:

$$\textsc{Fix}\ \texttt{Succ} \longrightarrow ?$$

this term is well-typed and closed ($\emptyset \vdash \textsc{Fix}\ \texttt{Succ} : \texttt{Nat}$) but is not a value; thus progress applies, even though no reduction rule applies. Two possible solutions we thought of are:

1. Make so that constants have a variant of the function type (e.g. $\emptyset \vdash \texttt{Succ} : \texttt{Nat} \rightsquigarrow \texttt{Nat}$) to which the fixpoint combinator cannot be applied;
2. Add a new reduction rule:

$$\frac{c \in \textsc{Constants}}{\textsc{Fix}\ c \longrightarrow c\ \textsc{Fix}\ c}\ \text{R-CstFix}$$

One other thing to consider is the semantics of the constants with function type. Assume we have

$$\textsc{Constants} := \{\texttt{Add},\ 0,\ 1,\ 2,\ \dots\}$$

and consider $\emptyset \vdash \texttt{Add}\,0\,1 : \texttt{Nat}$. We found two possible interpretations of this situation:

1. Consider $\texttt{Add}\,0\,1$ to be a value, but we then have two values, 1 and $\texttt{Add}\,0\,1$, which are not syntactically equal, yet semantically equal;
2. Add some reduction rule such that $\texttt{Add}\,0\,1 \longrightarrow 1$.

The first solution seemed to have some unwanted properties, so we discarded it; the second one required a way to specify the new reduction rules. Adding the reduction rules for this example is simple and wouldn't change the proofs too much. However, we did not want to hard-code more constructs into the calculus: we wanted the constants to be as generic as possible. Allowing the addition of new reduction rules seemed like some kind of safe "plugins" rather than simply adding constants.

Finally, moving from STLC to System F reduced the utility of constants. More precisely, System F allows usage of Böhm-Berarducci encoding [4, 5], which allows to encode in a practical way many sets of constants which would need to be implemented as extensions in STLC.

Considering all this, we decided to concentrate instead on adapting the proofs to System F, and discarded the constants.

3.6 From STLC to System F

The main challenges when moving to System F are the two new typing rules T-TAbs and T-TApp. In particular, the fact that the shifting and substitution operations may appear in typing derivations means that computing the type of a term becomes non-trivial, e.g. assuming $\emptyset \vdash t : T$, we get

$$\emptyset \vdash (\Lambda.\ \Lambda.\ t)\ [S_1]\ [S_2] : T' \qquad \text{where}$$
$$T' := \downarrow_0^{-1} \left(\left[0 \mapsto \downarrow_0^{-1} (S_2)\right]\ \downarrow_0^{-1} \left(\left[0 \mapsto \downarrow_0^{-1} (S_1)\right]\ T\right)\right)$$

At first, we proved some theorems depending on our exact needs at the time, but ended up realizing that some general commutativity theorems on shifts and substitutions would be more efficient, as all computations are combinations of the two at low-level.

However, the commutativity of these operations is more convoluted than one may expect; more precisely, they highly depend on the ordering relation between the substituted index, the shift cutoff and the shift distance.

3.6.1 Proving through rewriting.

When looking at the reduction rules with type preservation in mind, it seemed quite natural for us to prove statements such as:

Lemma 3.8. *For all environment* $\Gamma \in \mathbf{\Gamma}$*, terms* $t, s \in \boldsymbol{\lambda}$*, types* $T, S \in \mathbf{\Lambda}$ *and integer* $j \in \mathbb{N}$ *such that*

$$\Gamma \vdash t : T \ \wedge \ \Gamma \vdash s : S \ \wedge \ (\Gamma)\,[j] = S$$

we have

$$\Gamma \vdash \ [j \mapsto s] \ t : T$$

Such statements can be proved by "rewriting" the typing derivation. This proceeds by induction on the original typing derivation, and modifies its subderivations as well as its conclusion so that the conclusion of the typing derivation becomes the expected typing relation (e.g. $\Gamma \vdash \ [j \mapsto s] \ t : T$), while keeping the whole derivation valid.

This approach is quite powerful, and allows to prove some less obvious properties such as the following:

Lemma 3.9. *For all environment* $\Gamma \in \mathbf{\Gamma}$*, term* $t \in \boldsymbol{\lambda}$ *and type* $T \in \mathbf{\Lambda}$*, we have*

$$\Gamma \vdash t : T \implies \downarrow_c^s (\Gamma) \vdash \downarrow_c^s (t) : \downarrow_c^s (T)$$

By using this approach to prove how typing changed after one single operation, and chaining many of them, we were able to prove preservation (theorem 2.3) for System F.

4 Conclusion

4.1 Rundown of our work

In this project, we:

- Implemented the basis of STLC (terms, types, reduction relation,…);
- Implemented some functions performing key operations on terms (`reduceCallByValue`, `deriveType`);
- Proved that these functions are correct;
- Proved the main theorems of this calculus;
- Expanded all of this to System F.

All theorems we mentioned in the present document were proved; table 1 lists all theorems, and indicates the name of the function in Stainless as well as the file it is implemented in.

4.2 Improvements and extensions

This project could obviously be expanded by making the calculus richer (e.g. add records, mutability,…) or adding typing features (existential types, System F_ω, …). We will however briefly discuss less obvious extensions:

- Generalize theorem 2.2 to full β-reduction;
- System F has one more[4] key property we didn't prove: *confluence* [6]. This property is however classically proved (as in [7]) using theorems on the transitive closure of the reduction relation and is non-constructive, which are two things which do not go well with Stainless. Thus, a new (or less common) proof should be used.

[4] In fact, it has yet one more: *termination*, which is lost when the fixpoint combinator is added.

Thm	Function Name
In file `SystemF.scala`	
3.4	`hasFreeVariablesInCompleteness`
3.5	`hasFreeVariablesInSoundness`
In file `Transformations.scala`	
3.6	`boundRangeShiftCutoff`
	`boundRangeShiftBelowCutoff`
3.7	`boundRangeSubstitutionLemma`*[1-4]*
In file `Reduction.scala`	
3.3	`reduceCallByValueSoundness`
In file `Typing.scala`	
2.1	`typeDerivationsUniqueness`
2.2	`callByValueProgress`[a]
2.3	`preservation`
3.1	`deriveTypeSoundness`
3.2	`deriveTypeCompleteness`
3.8	`preservationUnderSubst`
3.9	`shiftAllTypes`

[a] As the name suggests, we proved this theorem for the call-by-value reduction strategy, but not for full β-reduction.

Table 1. Proofs of mentioned theorems and lemmas

- Implement more efficient evaluation schemes. Our implementation was not done with efficiency in mind, and is thus excruciatingly slow. Now that our implementation is done and verified, it could be used as a specification for faster implementations.
  One could keep the same evaluation scheme, but e.g. use a Zipper [8] or do the substitutions lazily to improve performances. More involved evaluation methods using e.g. interaction nets [9] or abstract machines [10] would be more interesting, but most likely also more difficult to prove correct as well.

References

[1] Luca Cardelli and Peter Wegner. "On understanding types, data abstraction, and polymorphism". In: *ACM Comput. Surv.* 17 (1985), pp. 471–522.

[2] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.

[3] EPFL IC LARA. *Stainles: Formal Verification for Scala*. 2019. URL: https://stainless.epfl.ch/.

[4] Corrado Böhm and Alessandro Berarducci. "Automatic synthesis of typed Lambda-programs on term algebras". In: *Theoretical Computer Science* 39 (1985). Third Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 135–154. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(85)90135-5.

[5] Oleg Kiselyov. *Beyond Church encoding: Boehm-Berarducci isomorphism of algebraic data types and polymorphic lambda-terms*. URL: https://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html.

[6] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. DOI: 10.1017/CBO9781139172752.

[7] Ernesto Copello, Nora Szasz, and Álvaro Tasistro. "Machine-checked Proof of the Church-Rosser Theorem for the Lambda Calculus Using the Barendregt Variable Convention in Constructive Type Theory". In: *Electronic Notes in Theoretical Computer Science* 338 (Oct. 2018), pp. 79–95. DOI: 10.1016/j.entcs.2018.10.006.

[8] Gérard Huet. "Functional pearl: The Zipper". In: *Journal of Functional Programming* (Jan. 1995).

[9] Ian Mackie. "Efficient lambda Evaluation with Interaction Nets". In: June 2004. ISBN: 978-3-540-22153-1. DOI: 10.1007/978-3-540-25979-4_11.

[10] Beniamino Accattoli and Bruno Barras. "Environments and the Complexity of Abstract Machines". In: *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*. PPDP '17. Namur, Belgium: Association for Computing Machinery, 2017, pp. 4–16. ISBN: 9781450352918. DOI: 10.1145/3131851.3131855.

A De Brujin's indices in System F

In System F, using De Brujin's indices is way more subtle than in STLC, as integers are used to represent both term and type variables.

As mentioned, the context always allows to distinguish which is meant. However, this also means that a same integers can, at one place in the term, represent a term variable and, in an other place, represent a type variable even though they do not have anything in common (which can lead to confusions). For term variables, everything stays the same, meaning that the first integers represent variables bound by abstractions and the other ones represent free variables. Regarding type variables, it is quite similar, but they can either be bound by type abstractions (at the term level), or by universal types (at the type level).

For instance, in the type $\forall.\ 0 \to 1$, 0 is a bound variable whereas 1 is a free variable. Similarly, in the term $\Lambda.\ \lambda 0.\ 2, 0$ is the type variable bound by the type abstraction. Things become more intricate when we mix many of these structures. In the term $\Lambda.\ \lambda(\forall.\ (0 \to 1)).\ 2, 0$ is the type variable bound by the universal quantifier and 1 is the type variable bound by the type abstraction.

Even worse, in the term $\Lambda.\ \lambda(\forall.\ (0 \to 1)).\ 0\,1$, the first 0 and the first 1 are the same bound type variables as in the previous example, whereas the second 0 is the term variable bound by the abstraction and the second 1 is a free term variable!

B Shifting the environment

When we enter an abstraction, the term variable 0 refers to the bound variable and the free variables (those in the environment) start from 1. Therefore, we need to prepend an element to the environment in order for the free variables to start from 1. When we enter in a type abstraction, we need to run a similar trick. In fact, the type variable 0 will refer to the bound variable and the free type variables will start from 1. Therefore, the free type variables in the environment will now start from 1 instead of 0. That is why we need to shift them by 1, and therefore shift the entire environment by 1. Let's see an example that illustrates the solution. We know that the term $(\Lambda.\ (\lambda 0.\ 1))\ [T]$ reduces to $\lambda T.\ 1$ and intuitively has type $T \to 0$ under the environment $\{0\}$. However, if we do not shift the environment, we obtain the following type derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{(0\ ::\ 0)\ [1] = 0}{}\text{T-Var}}{0\ ::\ 0 \vdash 1 : 0}\text{T-Abs}}{0 \vdash \lambda 0.\ 1 : 0 \to 0}\text{T-TAbs}}{0 \vdash \Lambda.\ (\lambda 0.\ 1) : \forall.\ 0 \to 0}\text{T-TApp}}{0 \vdash (\Lambda.\ (\lambda 0.\ 1))\ [T] : T \to T}$$

We can formally check that this is false by reducing this term and checking that both types do not match which contradicts the preservation property:

$$\dfrac{\dfrac{(T\ ::\ 0)\ [1] = 0}{}\text{T-Var}}{T\ ::\ 0 \vdash 1 : 0}\text{T-Abs}}{0 \vdash \lambda T.\ 1 : T \to 0}$$

The issue is then solved as soon as we shift the environment when typing the type abstraction:

$$\dfrac{\dfrac{\dfrac{\dfrac{(0\ ::\ 1)\ [1] = 1}{}\text{T-Var}}{0\ ::\ 1 \vdash 1 : 1}\text{T-Abs}}{1 \vdash \lambda 0.\ 1 : 0 \to 1}\text{T-TAbs}}{0 \vdash \Lambda.\ (\lambda 0.\ 1) : \forall.\ 0 \to 1}\text{T-TApp}}{0 \vdash (\Lambda.\ (\lambda 0.\ 1))\ [T] : T \to 0}$$

C Negative shift semantics

One interesting question is "when is (or should) a negative shift be valid?". The first condition we came with is presented in fig. 8a.

The only thing which is checked is that, after performing the shift, the resulting term is a valid lambda term; this simply means that no variable ends up with a negative index.

This way of thinking is perfectly valid, however it has two main downsides:

1. There is no (simple) relationship between allowing a negative shift and (not) having free variables;
2. The obtained term might have (semantically) nothing in common with the original term.

The second point is better explained with an example:

$$\uparrow_0^{-1} (\lambda T.\ 1) = \lambda T.\ 0$$

the free variable in the body became bound after the shift. This suggests that this precondition is too weak and allows shifts which will never be of any use, as they "break the semantics" of the shifted term.

It is possible to rewrite the precondition while keeping our example in mind; this results in fig. 8b

We see that the shift presented before would be rejected, as $1 + (-1) < 1$. This new version loses some properties of the first one, however those properties were of no use for our implementation. It also becomes possible to prove a really handy propriety, which is that $\forall t \in \boldsymbol{\lambda}.\ \forall d > 0.\ \forall c \in \mathbb{N}$

```
t.negShiftValidity(-d, c)
==
!t.hasFreeVariablesIn(c, d)
```

In fact, this property holds trivially: the two definition are the same up to some mild changes in the `Var` case induced by the inverted sign.

We thus decided to completely get rid of the concept of `negShiftValidity`, and simply used `hasFreeVariablesIn` as a precondition for negative shifts. We still considered this reflection meaningful, as the simple precondition is unrelated to free variables, and the more thoughtful one can seem uselessly restrictive.

```
def negShiftValidity(t: Term, d: BigInt, c: BigInt):
Boolean = {
  require(d < 0)
  t match {
    case Var(k) => (k < c) || (k+d >= 0)
    case Abs(_, body) =>
        negativeShiftValidity(body, d, c+1)
    case App(t1, t2)  =>
        negativeShiftValidity(t1, d, c) &&
        negativeShiftValidity(t2, d, c)
  }
}
```

(a) "Syntactic" negative shift precondition

```
def negShiftValidity(t:Term,d:BigInt,c:BigInt):
Boolean = {
  require(d < 0)
  t match {
    case Var(k) => (k < c) || (k+d >= c)
    case Abs(_, body) =>
        negativeShiftValidity(body, d, c+1)
    case App(t1, t2) =>
        negativeShiftValidity(t1, d, c) &&
        negativeShiftValidity(t2, d, c)
  }
}
```

(b) "Semantic" negative shift precondition

In the following, $k, j, c \in \mathbb{N}$, $d \in \mathbb{Z}$, $b \in \texttt{char}^*$, $t, t_1, t_2, s \in \boldsymbol{\lambda}$, $T, T_1, T_2, S \in \mathbf{\Lambda}$ and $\emptyset, \Gamma \in \mathbf{\Gamma}$.

$$\begin{aligned}
\uparrow_c^d (k) &:= \begin{cases} k & \text{if } k < c \\ k + s & \text{if } k \geq c \end{cases} \\
\uparrow_c^d (\lambda T.\, t) &:= \lambda T.\, \uparrow_{c+1}^d (t) \\
\uparrow_c^d (t_1\, t_2) &:= \uparrow_c^d (t_1)\, \uparrow_c^d (t_2) \\
\uparrow_c^d (\textsc{Fix}\, t) &:= \textsc{Fix}\, \uparrow_c^d (t) \\
\uparrow_c^d (\Lambda.\, t) &:= \Lambda.\, \uparrow_c^d (t) \\
\uparrow_c^d (t\, [T]) &:= \uparrow_c^d (t)\, [T]
\end{aligned}$$

(a) Shift terms in terms

$$\begin{aligned}
[j \mapsto s]\ k &:= \begin{cases} s & \text{if } k = j \\ k & \text{if } k \neq j \end{cases} \\
[j \mapsto s]\ (\lambda T.\, t) &:= \lambda T.\, \left[j + 1 \mapsto \uparrow_0^1 (s)\right]\ t \\
[j \mapsto s]\ (t_1\, t_2) &:= [j \mapsto s]\ t_1\, [j \mapsto s]\ t_2 \\
[j \mapsto s]\ (\textsc{Fix}\, t) &:= \textsc{Fix}\, [j \mapsto s]\ t \\
[j \mapsto s]\ (\Lambda.\, t) &:= \Lambda.\, \left[j \mapsto \downarrow_0^1 (s)\right]\ t \\
[j \mapsto s]\ (t\, [T]) &:= ([j \mapsto s]\ t)\, [T]
\end{aligned}$$

(b) Substitute terms in terms

$$\begin{aligned}
\downarrow_c^d (k) &:= k \\
\downarrow_c^d (\lambda T.\, t) &:= \lambda \downarrow_c^d (T).\, \downarrow_c^d (t) \\
\downarrow_c^d (t_1\, t_2) &:= \downarrow_c^d (t_1)\, \downarrow_c^d (t_2) \\
\downarrow_c^d (\textsc{Fix}\, t) &:= \textsc{Fix}\, \downarrow_c^d (t) \\
\downarrow_c^d (\Lambda.\, t) &:= \Lambda.\, \downarrow_{c+1}^d (t) \\
\downarrow_c^d (t\, [T]) &:= \left(\downarrow_c^d (t)\right) \left[\downarrow_c^d (T)\right]
\end{aligned}$$

(c) Shift types in terms

$$\begin{aligned}
[j \mapsto S]\ k &:= k \\
[j \mapsto S]\ \lambda T.\, t &:= \lambda [j \mapsto S]\ T.\, [j \mapsto S]\ t \\
[j \mapsto S]\ t_1\, t_2 &:= [j \mapsto S]\ t_1\, [j \mapsto S]\ t_2 \\
[j \mapsto S]\ \textsc{Fix}\, t &:= \textsc{Fix}\, [j \mapsto S]\ t \\
[j \mapsto S]\ \Lambda.\, t &:= \Lambda.\, \left[j + 1 \mapsto \downarrow_0^1 (S)\right]\ t \\
[j \mapsto S]\ t\, [T] &:= ([j \mapsto S]\ t)\, [[j \mapsto S]\ T]
\end{aligned}$$

(d) Substitute types in terms

$$\begin{aligned}
\downarrow_c^d (k) &:= \begin{cases} k & \text{if } k < c \\ k + s & \text{if } k \geq c \end{cases} \\
\downarrow_c^d (b) &:= B \\
\downarrow_c^d (T_1 \to T_2) &:= \left(\downarrow_c^d (T_1)\right) \to \left(\downarrow_c^d (T_2)\right) \\
\downarrow_c^d (\forall.\, T) &:= \forall.\, \downarrow_{c+1}^d (T)
\end{aligned}$$

(e) Shift types in types

$$\begin{aligned}
[j \mapsto S]\ k &:= \begin{cases} S & \text{if } k = j \\ k & \text{if } k \neq j \end{cases} \\
[j \mapsto S]\ b &:= b \\
[j \mapsto S]\ (T_1 \to T_2) &:= \left([j \mapsto S]\ T_1\right) \to \left([j \mapsto S]\ T_2\right) \\
[j \mapsto S]\ (\forall.\, T) &:= \forall.\, \left[j + 1 \mapsto \downarrow_0^1 (S)\right]\ T
\end{aligned}$$

(f) Substitute types in types

$$\downarrow_c^d (\emptyset) := \emptyset \qquad \downarrow_c^d (T :: \Gamma) := \downarrow_c^d (T) :: \downarrow_c^d (t)$$

(g) Shift types in environments

Figure 9. Definition of shift and substitution