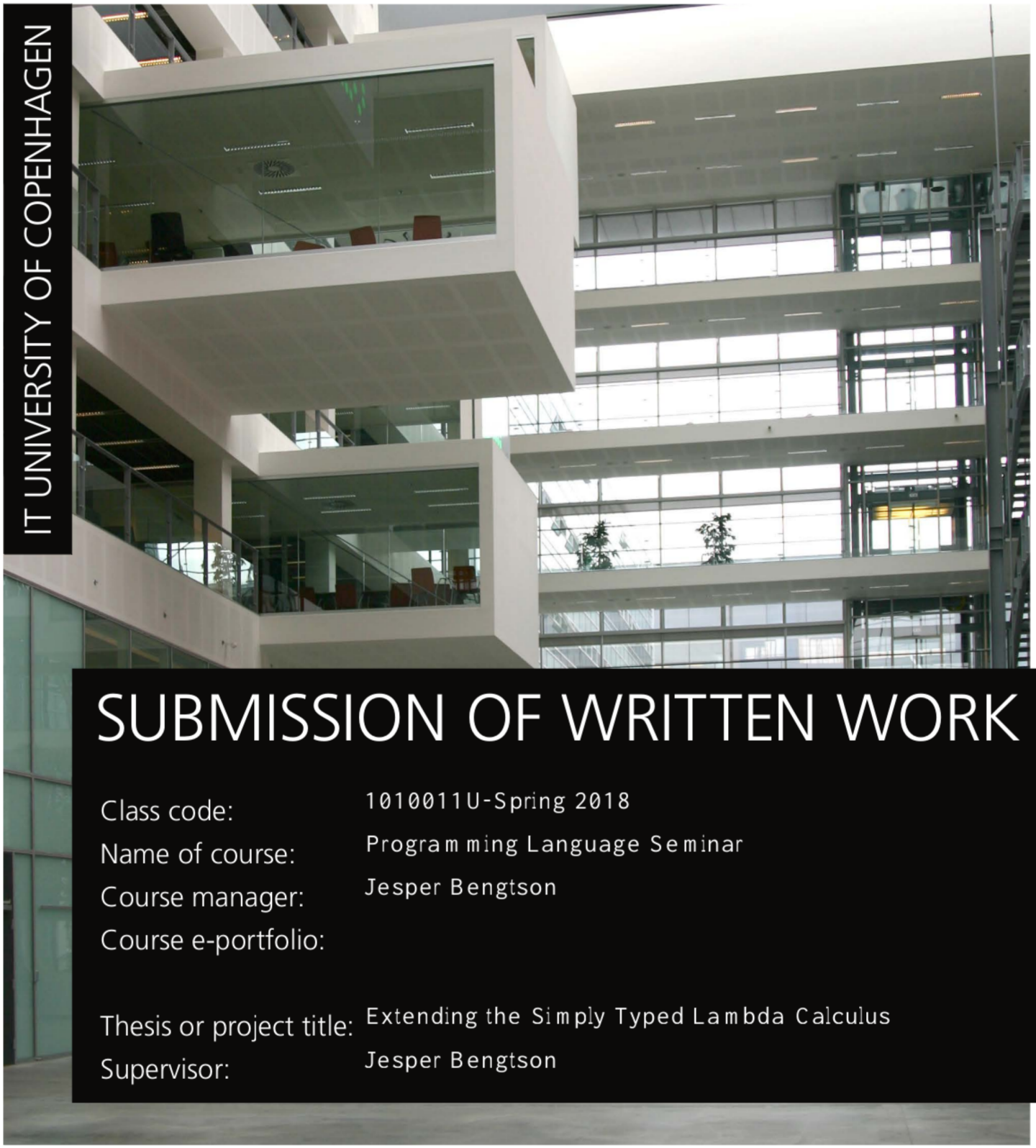

SUBMISSION OF WRITTEN WORK

Class code:	1010011U-Spring 2018
Name of course:	Programming Language Seminar
Course manager:	Jesper Bengtson
Course e-portfolio:	
Thesis or project title:	Extending the Simply Typed Lambda Calculus
Supervisor:	Jesper Bengtson

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Emil Christian Lynegaard	17/07-1994	ecly@itu.dk
2.	Michael Erik Vesterli	28/06-1995	miev@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk

Extending the Simply Typed Lambda Calculus

Emil Christian Lynegaard,
Michael Erik Vesterli

May 18, 2018

Contents

1 Introduction

Lambda Calculus forms the foundation on which most modern functional programming languages were built. The Untyped Lambda Calculus, introduced by Alonzo Church in the 1930s (Wikipedia, 2018a), provides a very small set of rules in the form of variables, abstraction and application, which together with its accompanying reduction operations form a universal model of computation that is Turing Complete. While the Untyped Lambda Calculus is capable of simulating a Turing machine, it fails to allow theorems concerning certain desirable properties to be proven about the calculus. As such, that Lambda Calculus has often been extended to typed versions, which for example allow theorems regarding *progress* to be proven, meaning that well-typed term are either values, or they can be further reduced.

By implementing an extension of the Typed Lambda Calculus in Coq, a formal proof management system, we are able to construct automatically checked proofs about the language using Coq's interactive proof assistant. This in turn allows us to formally verify the correctness of the extended Typed Lambda Calculus, and to prove the aforementioned *progress* theorem, as well a *preservation* theorem, stating that well-typed terms are similarly well-typed upon reduction, using Coq's formal language. In particular, we will be extending the Simply Typed Lambda Calculus defined in Pierce et al. (2018, Ch. Stlc), where 'Simply Typed' entails extensions such as natural numbers, sumbs, records and other non-polymorphic, non-dependent types(Wikipedia, 2018b). Implementing this extension of the Simply Typed Lambda Calculus with sums, recursion and more, in addition to formally proving the *progress* and *preservation* theorems for the language, will be the goal of this paper. Since the extension includes recursion, the operational semantics of the language will be implemented using small-step semantics, as the big-step semantic alternative fails to support our desired proofs in the presence of recursion(Pierce et al., 2018, Ch. Smallstep).

2 Language Definition

We have based our language on the Stlc language definition given in the Programming Language Foundations book (Pierce et al., 2018, Ch. Stlc). These chapters define a language that contains booleans, if statements, variables, arrow functions and function application. The following chapter also provides typing and reduction rules for various extensions. The associated proofs of our *progress* and *preservation* have however not been provided, and are therefore done by us. These extensions from the book include let bindings, pairs, sum types with

two variants, lists and general recursion. In addition, the Records chapter of the same book supplies a definition of arbitrarily sized records, together with an extension of the *preservation* proof.

We have extended the Stlc language with most of the books suggested extensions, except in cases where our extensions are more general. In particular, we support sum types with an arbitrary number of variants instead of only two. Our extension also does not include pairs, as we support the more general records instead. Natural numbers have also been added to the base types of the language.

Since we do not support recursive type declarations, it is not possible to define basic data structures, such as lists, by combining records and sum types. Therefore lists are also built in to the language. Our definition of lists differs slightly from the one given in the book, as we reuse the match term from sum types to also match on lists. As lists have no variants, the names that must be matched on are fixed in the language.

2.1 Terms

The language is built around the co-inductive definition of the types `term` and `case_list`. While most extensions to the language can be defined by simply adding additional terms, this is not the case for matching a sum type with a number of cases. This is because a `case_list` cannot exist as well-typed `term` on its own. Yet, it still needs to be well-formed and match the type of the sum types. This is most easily achieved by defining `term` and `case_list` separately. The resulting `term` constructors are listed below:

`t_var name`: Access the variable with the given name.

`t_app fun arg`: Apply the function with the given function.

`t_abs param_name type body`: Declare a function taking an argument of the given type. When called, the function body is evaluated with all occurrences of the parameter name substituted by the evaluated argument.

`t_true`: The true constant.

`t_false`: The false constant.

`t_if condition t1 t2`: Evaluate `t1` if the condition evaluates to `t_true`. Otherwise evaluate `t2`.

`t_let name t1 t2`: Evaluate `t1` and substitute all occurrences of the given name with the result when evaluating `t2`.

`t_rnil`: The empty record.

`t_rcons name value record`: Add a field with the given name to a record.

`t_proj record field`: Access the named field in a record.

`t_fix fun`: Make the given function able to call itself recursively.

`t_nat nat`: A natural number.

`t_nat_eq n1 n2`: Evaluate the equality of two natural numbers.

`t_succ nat`: The successor of a natural number.

`t_pred nat`: The predecessor of a natural number.

`t_mult n1 n2`: Multiply two natural numbers.

`t_sum variant term type`: Create an instance of the given variant of a sum type. Since many types can have the same variant names, a type annotation is required.

`t_lnil`: The empty list.

`t_lcons head tail`: Create a list by appending a term to an existing list.

`t_match term cases`: Handle each possible variant of a sum type or list using a distinct function, each with the same return type. If the `term` is a list, it acts as a sum type with a `"cons"` and `"nil"` variant. The `"cons"` variant contains a record which contains both the head and the tail of the list and the `"nil"` variant contains an empty record. The cases must be given in the same order as defined in the sum type.

The `case_list` constructors used by `t_match` are defined co-inductively as follows:

`t_case_one variant fun`: A list of cases that match a sum type with a single variant. We use this as the base case instead of an empty list to make it clearer, that a match on a sum without any variants cannot exist.

`t_case_cons variant fun cases`: Extend a list of cases with a single variant.

2.2 Typing rules

Each `term` listed in section 2.1 is in exactly one type. This can be either `TBool`, `TNat`, `TArrow T U`, `TRNil`, `TRCons x T U`, `TSNil`, `TSCons x T U` or `TList T`. `TArrow T U` is the type of a function which takes a single argument of type T and returns a value of type U. `TRCons x T U` is a record type that is build by appending a field `x` of type `T` to an existing record type `U`. Similarly, `TSCons x T U` builds a new sum type from an existing one by adding a new variant.

Terms are typed according to the following rules. Any `term` that cannot be typed using these rules is not well-typed. Most of the properties of the language do not apply to such terms.

$$\frac{\Gamma\ \texttt{x} = \texttt{Some T} \qquad \texttt{well\_formed\_ty T}}{\Gamma \vdash \texttt{t\_var x} \in \texttt{T}}\ \texttt{T\_Var}$$

$$\frac{\texttt{well\_formed\_ty T} \qquad \Gamma\ \texttt{\&}\ \{\ \texttt{x} \to \texttt{T}\ \}\ \vdash \texttt{t} \in \texttt{U}}{\Gamma \vdash \texttt{t\_abs x T t} \in \texttt{TArrow T U}}\ \texttt{T\_Abs}$$

$$\frac{\Gamma \vdash \texttt{t1} \in \texttt{TArrow T U} \qquad \Gamma \vdash \texttt{t2} \in T}{\Gamma \vdash \texttt{t\_app t1 t2} \in \texttt{U}}\ \texttt{T\_App}$$

$$\frac{}{\Gamma \vdash \texttt{t\_true} \in \texttt{TBool}}\ \texttt{T\_True}$$

$$\frac{}{\Gamma \vdash \texttt{t\_false} \in \texttt{TBool}}\ \texttt{T\_False}$$

$$\frac{\Gamma \vdash \texttt{t1} \in \texttt{TBool} \qquad \Gamma \vdash \texttt{t2} \in \texttt{T} \qquad \Gamma \vdash \texttt{t3} \in \texttt{T}}{\Gamma \vdash \texttt{t\_if t1 t2 t3} \in \texttt{T}}\ \texttt{T\_If}$$

$$\frac{\Gamma \vdash \texttt{t1} \in \texttt{U} \qquad \Gamma\ \texttt{\&}\ \{\ \texttt{x} \to \texttt{U}\ \} \vdash \texttt{t2} \in \texttt{T}}{\Gamma \vdash \texttt{t\_let x t1 t2} \in T}\ \texttt{T\_Let}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{TR} \qquad \texttt{TRlookup f TR = Some T}}{\Gamma \vdash \texttt{t\_proj t f} \in \texttt{T}}\ \texttt{T\_Proj}$$

$$\frac{}{\Gamma \vdash \texttt{t\_rnil} \in \texttt{TRNil}}\ \texttt{T\_RNil}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{T} \qquad \Gamma \vdash \texttt{tr} \in \texttt{Trest} \qquad \texttt{record\_ty Trest} \qquad \texttt{record\_term tr}}{\Gamma \vdash \texttt{t\_rcons f t tr} \in \texttt{TRCons f T Trest}}\ \texttt{T\_RCons}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{TArrow T T}}{\Gamma \vdash \texttt{t\_fix t} \in \texttt{T}}\ \texttt{T\_Fix}$$

$$\frac{}{\Gamma \vdash \texttt{t\_nat n} \in \texttt{TNat}}\ \texttt{T\_Nat}$$

$$\frac{\Gamma \vdash \texttt{n1} \in \texttt{TNat} \qquad \Gamma \vdash \texttt{n2} \in \texttt{TNat}}{\Gamma \vdash \texttt{t\_nat\_eq n1 n2} \in \texttt{TBool}}\ \texttt{T\_Nat\_Eq}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{TNat}}{\Gamma \vdash \texttt{t\_succ t} \in \texttt{TNat}}\ \texttt{T\_Succ}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{TNat}}{\Gamma \vdash \texttt{t\_pred t} \in \texttt{TNat}}\ \texttt{T\_Pred}$$

$$\frac{\Gamma \vdash \texttt{t1} \in \texttt{TNat} \qquad \Gamma \vdash \texttt{t2} \in \texttt{TNat}}{\Gamma \vdash \texttt{t\_mult t1 t2} \in \texttt{TNat}}\ \texttt{T\_Mult}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{T} \qquad \texttt{sum\_ty TR} \qquad \texttt{well\_formed\_ty TR}}{\Gamma \vdash \texttt{t\_sum x t (TSCons x T TR)} \in \texttt{TSCons x T TR}}\ \texttt{T\_SumEq}$$

$$\frac{\Gamma \vdash \texttt{t\_sum x t TR} \in \texttt{TR} \qquad \texttt{x} \neq \texttt{x'} \qquad \texttt{well\_formed\_ty T}}{\Gamma \vdash \texttt{t\_sum x t (TSCons x' T TR)} \in \texttt{TSCons x' T TR}}\ \texttt{T\_SumNeq}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{TSCons x T' TR} \qquad \texttt{match\_has\_type}\ \Gamma\ \texttt{(TSCons x T' TR)hs T}}{\Gamma \vdash \texttt{t\_match t hs} \in \texttt{T}}\ \texttt{T\_Match}$$

$$\frac{\texttt{well\_formed\_ty T}}{\Gamma \vdash \texttt{t\_lnil T} \in \texttt{T}}\ \texttt{T\_LNil}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{T} \qquad \Gamma \vdash \texttt{ts} \in \texttt{TList T}}{\Gamma \vdash \texttt{t\_lcons t ts} \in \texttt{TList T}}\ \texttt{T\_LCons}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{tList U} \qquad \texttt{match\_has\_type } \Gamma \texttt{ ListCases hs T}}{\Gamma \vdash \texttt{t\_match t hs} \in \texttt{T}} \texttt{T\_LMatch}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{TArrow T' T}}{\texttt{match\_has\_type } \Gamma \texttt{ (TSCons x T' TSNil)(t\_case\_one x t)T}} \texttt{T\_CaseOne}$$

$$\frac{\Gamma \vdash \texttt{t} \in \texttt{TArrow T' T} \qquad \texttt{match\_has\_type } \Gamma \texttt{ TR hs T}}{\texttt{match\_has\_type } \Gamma \texttt{ (TSCons x T' TR)(t\_case\_cons x t hs)T}} \texttt{T\_CaseCons}$$

These typing rules reference some other properties and functions from the language implementation. `well_formed_ty` simply states whether a type makes sense. This is only necessary because the lists that define record and sum types are defined directly in the typing system. Without it, there could be types that, for example, append a record field to a non-record type. This only appears in a few places, since `well_formed_ty T` is implied by $\Gamma \vdash \texttt{t} \in \texttt{T}$. `TRlookup` is simply a function that returns the type of a named field in a record type. `ListCases` is short for (`TSCons "cons" (TRCons "head" U (TRCons "tail" (TList U) TRNil)) (TSCons "nil" TRNil TSNil)`), which is the specific sum type that a `case_list` needs for match to be applicable to lists.

2.3 Semantics

The language has been defined using small-step semantics. This means that we define rules representing atomic steps of computation for reducing terms until they reach a value (Pierce et al., 2018)[Ch. Smallstep]. For describing these reductions, we will use the notation $t \Rightarrow t'$, meaning that t steps to t'. Most of the reduction rules are trivial, as they simply state that an inner `term` can take a step without the outer term changing structure. Operations on natural numbers are also generally trivial. Some of the more interesting definitions are given below.

$$\frac{\texttt{value v}}{\texttt{t\_app (t\_abs x T t)v} \Rightarrow \texttt{[x := v] t}} \texttt{ST\_AppAbs}$$

Here, a function `t_abs x T t` is applied with a completely evaluated argument `v`, which steps to evaluating the function body, with occurrences of the parameter name `x` substituted with the argument `v`. Let bindings are very similar, when the value being bound has been fully reduced.

$$\frac{\texttt{value r} \qquad \texttt{trlookup f r = Some v}}{\texttt{t\_proj r f} \Rightarrow \texttt{v}} \texttt{ST\_ProjValue}$$

Record projection is also relatively simple as it simply looks up a field in a fully reduced record type.

$$\frac{}{\texttt{t\_fix (t\_abs x T f)} \Rightarrow \texttt{[x := t\_fix(t\_abs x T f)] f}}\ \texttt{ST\_FixAbs}$$

Defining a function `f` as a fixpoint substitutes all occurrences of recursive calls in the function body with the fixpoint itself. This allows the function to be called recursively.

$$\frac{}{\texttt{t\_match (t\_sum x t T)(t\_case\_cons x t' hs)} \Rightarrow \texttt{t\_app t' t}}\ \texttt{ST\_MatchSumHead}$$

$$\frac{\texttt{x} \neq \texttt{x'} \qquad \texttt{t\_match (t\_sum x t TS)hs} \Rightarrow \texttt{t\_app t' t}}{\texttt{t\_match (t\_sum x t (TSCons x' T TS))}\ \texttt{(t\_case\_cons x' t'' hs)} \Rightarrow \texttt{t\_app t' t}}\ \texttt{ST\_MatchSumTail}$$

Match is also interesting, as we step to applying the case body with the value in the matched variant. This differs from the match term defined for lists in Pierce et al. (2018), as they immediately substitute the matched value into the case body. We chose this method instead, as it results in a smaller step and avoids requiring a fixed name in the case body. In a match expression, it might also not be the first variant that matches. This is handled by `ST_MatchSumTail`, which allows applying later matching variants.

In addition to these two rules, there is also a rule for matching on sum types with a single variant and separate rules for matching on lists. Matching on lists is very similar to sum types, except that the variants are fixed to `"nil"` and `"cons"`.

3 Examples

A few small programs that may be written using the extended Stlc language can be seen below. Note that we have not created notation and simply define them as they would have been in Coq. All the accompanying examples have been verified in Coq.

3.1 Factorial

```
Definition fact :=
  t_fix
    (t_abs "f" (TArrow TNat TNat)
      (t_abs "x" TNat
        (t_if
          (t_nat_eq (t_nat 0) (t_var "x"))
          (t_nat 1)
          (t_mult
```

```
            (t_var "x")
            (t_app (t_var "f") (t_pred (t_var "x")))))))).

Example fact_test:
  (t_app fact (t_nat 4)) ==>* (t_nat 24).
```

This demonstrates a recursive factorial function. It makes use of the `t_fix` constructor to define a classical recursive definition of the factorial, which recursively multiplies the current value with its predecessor until its predecessor is equal to `t_nat 0`. The notation on line 13 means that the factorial of 4 eventually steps to the value 24 through small-step semantics.

3.2 Either

```
Definition Either a b :=
  TSCons "Left" a (TSCons "Right" b TSNil).

Example either_test:
  t_match
    (t_sum "Right" t_true (Either TNat TBool))
    (t_case_cons "Left"
      (t_abs "x" TNat (t_var "x"))
      (t_case_one "Right" (t_abs "x" TBool (t_nat 0))))
        ==>* t_nat 0.
```

In the code above, an `Either` type constructor is defined using the arbitrary sum type. In the accompanying example, we create an instance of the `Either TNat TBool` type and match it. If the sum type is of the `"Right"` variant, `t_nat 0` is returned.

3.3 Product

```
Definition product :=
  t_fix
    (t_abs "f" (TArrow (TList TNat) (TNat))
      (t_abs "x" (TList TNat)
        (t_match (t_var "x")
          (t_case_cons "cons"
            (t_abs "l"
              (TRCons "head" TNat
                (TRCons "tail" (TList TNat) TRNil))
              (t_mult
                (t_proj (t_var "l") "head")
                (t_app
```

```
                  (t_var "f")
                  (t_proj (t_var "l") "tail"))))
          (t_case_one "nil" (t_abs "_" TRNil (t_nat 1)))))))
              .

Example product_test :
  t_app product (t_lcons (t_nat 3)
    (t_lcons (t_nat 7) (t_lnil TNat)))
      ==>* t_nat 21.
```

Here we define a function on lists of natural numbers that compute the product of all contained values. The product function is also recursively defined using `t_fix`. The function matches on the given list, multiplying the value of its head with the product of the tail of the list. This goes on until a `"nil"` case is met, denoting the end of the list.

4 Properties of Stlc

The main properties of the extended Stlc that we want to prove are *progress*, and *preservation*. *Progress* means that any `term` that is well-typed is either a value or will be able to take a step using small-step semantics. *Preservation* means that any `term` that is well-typed and may take a step, will also be well typed having taken that step. These two properties together ensure that a well-typed program cannot crash or otherwise terminate without producing a result.

To prove *preservation*, we additionally want to prove a lemma stating, that substituting a variable of some type with a value of said type, preserves typing of the expression.

Theorem 4.1 (Progress)**.** *For all well-typed terms t, either t is a value, or there exists some t', such that $t \Rightarrow t'$*

Proof. Proven by induction on the hypothesis that t is well-typed.
For the base types of t, the proof is trivial, as these are all values.
For the remaining parameterized types, the proofs are generally similar, and we will use conditionals as an example.

For a well-typed constructor *(t_ if t1 t2 t3)* it follows by construction that *t1*, *t2* and *t3* are also well-typed. Therefore we know that *t1* is of type *TBool*, and from the IH for *t1*, we get the following 2 cases:

- *t1* is a value. *t1* is either true or false and steps to either *t2* or *t3* accordingly, which follows from the definition of $\Rightarrow$.

- There exists a *t1'*, such that *t1* $\Rightarrow$ *t1'*, which is exactly what we need to prove.

Lemma 4.2 (Substitution preserves typing)**.** *For all terms t of type T, in a context where x is a variable of type U, substituting x in t with a value of type U, will preserve the typing of t.*

Proof. Proven by induction on *t*.
For base types it follows trivially from the definition of their typing rules.
For most abstract types it follows trivially from the IH.
For the remaining abstract types, we will use let bindings as an example of a proof.

For a constructor *(t_ let s t1 t2)* we need to prove that substituting x with a value of type U in *t1* and *t2* preserves typing.

- For *t1* this is true given the IH.
- For *t2* we have two cases.
  - $x = s$, where s shadows x in the context, in which case *t2* trivially preserves its type.
  - $x <> s$, which is proven by the induction hypothesis for *t2*

Theorem 4.3 (Preservation)**.** *For all well-typed terms t, where t* $\Rightarrow t'$*, t′ will also be well-typed.*

Proof. Proven by induction on the hypothesis that t is well-typed. Base types are trivial, as these cannot step. For abstract types, we again use let bindings as an example.

Knowing *(let x:=t1 in t2)* $\Rightarrow$ *t'*, we need to prove that *t'* is well-typed. From the definition of step we know that there are two cases for the let-expression to step.

- *t1* $\Rightarrow$ *t1'*. Which is proven by the IH.
- *t1* is a value. As the let-`term` is known to be well-typed, proven by 4.2.

4.1 Additional properties

Other than the previous *progress* and *preservation* proofs, our STLC implementation includes a few of other interesting theorems proven in Coq.

- *Type unique*, stating that a `term` that has one type, cannot have another.
- *Soundness*, stating that a well-typed `term` cannot step to a stuck state. Where stuck is defined as a `term` that is not a value and cannot step. Similar to *progress*.
- *Context invariance*, stating that a `term` t, well-typed in a context Γ, will remain well-typed if its context is swapped with a Γ', given that Γ and Γ' both bind all free variables in the same way.

5 Discussion

Throughout the implementation of the extended Stlc, Pierce et al. (2018) generally provided reasonable guidelines on how to add a certain features to its provided Stlc. For most extensions, we followed these guidelines. One addition in particular which deviated significantly from the scope of the book was that of sum types. For sum types, the book suggested a simple hard coded binary sum type, resulting in a classic *Either* type constructor.

Instead we have implemented arbitrary sum types capable of representing n-ary sums. Where the accompanying case-expression for the book's binary sum type had to handle two simple cases, in the form of *inl* and *inr*, representing *Left* or *Right* from a classical *Either*, we instead need to be able to match on arbitrary sums for them to be useful. This brought forth the need for a co-inductive definition of `term` and `case_list`, as we would otherwise be able to create invalid terms without typing rules to prevent it. For example, had `t_case_cons` been in `term`, we could have constructed `t_case_cons "Right" t_true t_false`, which would make no sense.

When formulating proofs using induction on co-inductive types in Coq, the proof assistant is no longer able to automatically generate an induction hypothesis. As a result, we have to provide our own induction hypotheses for `term` and `case_list` respectively, when inductively proving properties concerning these. When proving *progress* in Coq, this was done as follows:

```
induction Ht using has_type_ind_rec with
  (P := fun Gamma t T Ht => Gamma = empty
    -> value t  \/ (exists t', t ==> t'))
  (P0 := fun Gamma TS hs T Ht => Gamma = empty
    -> forall x t, Gamma |- t_sum x t TS \in TS
    -> match_has_type Gamma TS hs T
    -> exists t', t_match (t_sum x t TS) hs ==> t_app t' t);
```

Recall the proof for *progress* (Theorem 4.1). Here we see that P is equal to the induction hypothesis used for proofs regarding `term`, which is what Coq would have used automatically, had `term` not been co-inductive. *P0* then defines the induction hypothesis for `case_lists`. *P0* was generally much more difficult to define for the various relevant proofs. This is because we have to ensure that the assumptions can be proven when the hypothesis is applied and that the assumptions are sufficient to show the induction hypothesis for all `case_lists`.

6 Conclusion

We have extended the Stlc language definition provided by Pierce et al. (2018) to also include numbers, conditionals, lists, records, sums, let bindings and recursion. This includes defining typing and reduction rules for all of them. While the book has suggested all of these extensions, our implementation of sum types is more powerful than the suggested version.

By defining the language in Coq, we are able to formulate theorems about the language and have our proofs automatically verified by Coq. Using this, we have proven that our extended Stlc language satisfies both the *progress* and the *preservation* properties, as well as some other properties. With the final language implementation, we are also able to define small programs and verify that these examples correctly reduce to their expected values.

References

B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, A. Tolmach, and B. Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. 2018. URL `https://softwarefoundations.cis.upenn.edu/plf-current/index.html`.

Wikipedia. Lambda calculus — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Lambda%20calculus&oldid=841398022`, 2018a.

Wikipedia. Simply typed lambda calculus — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Simply%20typed%20lambda%20calculus&oldid=823647944`, 2018b.