A beginners guide to programming in Beluga

Brigitte Pientka[*] and Ryan Kavanagh[†]

May 27, 2014

Beluga is a type-safe functional language. This means, we have a theoretical foundation describing the type system, including the operational semantics, and we have a safety proof (see [**?**]). Beluga is implemented in OCaml and is under active development at McGill.

We begin with some simple examples—similar to the programs one would write in other functional languages such as SML or OCaml. Compared to such more established functional languages, there are several major differences.

1 Preliminary: Beluga is pure

Beluga is a pure functional language; it has no predefined data types such as integers, reals, strings or even booleans. One reason is that Beluga is meant to be a prototyping environment for formal systems and their proofs and it is not intended as a general purpose programming language. Beluga also does not provide input/output facilities, references, nor exceptions.

2 Defining data-types and functions

Beluga supports defining your own data via data-types. For example, we can define the natural numbers as follows:

```
nat : type.
z : nat.
succ : nat -> nat.
```

`nat` is introduced as a new type using the key word **type**. The type `nat` has two constructors: `z` and `succ`. They can be used to construct data objects such as `z`, `succ z`, `succ (succ z)`, etc. and we have modelled the natural numbers.

We can also model the booleans: we call the type of booleans here `yes_or_no`. The type `yes_or_no` has two constructors: the constructor `yes` and the constructor `no`. These constructors constitute the values of the type `yes_or_no`.

```
yes_or_no : type.
yes : yes_or_no.
no : yes_or_no.
```

[*]School of Computer Science, McGill University
[†]School of Computing, Queen's University

The next question is, how do we write programs about this data? The first program we write is a function **andalso** which behaves like a conjunction on data of type `yes_or_no`. Let's look at the program before discussing it.

```
rec andalso : [ |- yes_or_no ] -> [ |- yes_or_no ] -> [ |- yes_or_no ] =
fn x => fn y => case x of
| [ |- no ] => [ |- no ]
| [ |- yes ] => y
;
```

We declare functions (which can be recursive) by using the keyword **rec** followed by the name of the function, in this case **andalso**. Unlike languages such as SML or OCaml, we must specify the type of the function; in this example the type is `[ |- yes_or_no ] -> [ |- yes_or_no ] -> [ |- yes_or_no ]`. The [$\vdash$] describes the fact that we are manipulating data which is closed. Subsequently, we will see more uses of `[ |- ]`. For now, it is important to know that *data is embedded into computations via [|-]* and boxing data implies that this data is a closed value—in particular, there are no computations embedded in them. We will come back to this shortly.

The body of the function is built by function abstraction and pattern matching. Pattern matching allows us to inspect, analyze and take apart values of a given type. In the example program above, `x` will have type `yes_or_no` and hence it can have two possible values: either `x` will evaluate to `[ |- yes ]`, then the branch `[ |- yes ] => [ |- no ]` will be chosen or `x` will evaluate to `[ |- no ]` and the branch `[ |- no ] => y` will be chosen.

Since data does not live on the same level as computations, we will need to "unbox" it by writing `[ |- yes ]` and `[ |- no ]` in contrast to simply writing `yes` and `no`.

If `x` evaluates to `[ |- yes ]`, we will return whatever value `y` has. If it also evaluated to `[ |- yes ]`, we will return `[ |- yes ]` and otherwise we will return `[ |- no ]`, correctly modelling logical conjunction.

Let's look at a recursive program; we want to add two numbers.

```
rec add : [ |- nat ] -> [ |- nat ] -> [ |- nat ] = fn x => fn y => case x of
| [ |- z ] => y
| [ |- succ N ] => let [ |- R ] = add [ |- N ] y in
    [ |- succ R ]
;
```

Hence, we define a recursive function `add` which has type `[ |- nat ] -> [ |- nat ] -> [ |- nat ]`. It means `add` expects two natural numbers as input and returns a natural number. We write this function using pattern matching. The base case is straightforward. If `x` evaluates to `[ |- z ]`, we simply return `y`. But if `x` evaluates to `succ N`, then we need to call `add` again. We will call it on `[ |- N ]` and `y`. Note that before calling `add` we need "box" the number `N`. *Boxing is necessary to ensure we are working with pure data; unboxing is necessary for inspecting what the data looks like.* We then bind the result of the recursive call `add [ |- N] y` to the data-variable `R`, and rebuild a proper natural number `succ R`. To return it though, we need to box it. Note, that the following is illegal:

```
...
| [ |- succ N ] =>
   [ |- succ (add [ |- N ] y) ]
```

Under a box (written as `[ |- ]`) we are only allowed to have data never computation which is not yet a data-value. Hence, we are always forced to "stage" our computation, i.e. first evaluate the recursive call, bind its result (which is a pure data object) to a data-variable and continue building new data-objects.

Different syntax for different variables We hope we have driven home the point that we are working with two levels: data and computations. Beluga forces the user to use two different variables depending on what he wants to denote: capitalized letters such as `N` denote data-variables; lower-case letters such as `x`, `y`, `add` denote computation or program variables. Data-variables **always** occur inside a box. Program variables **never** occur inside a box; they stand for any computation.

Computation-level types in Beluga For now, we have very very simple types: either we have a user-defined base type such as `nat`, `yes_or_no` wrapped in a box, i.e. `[ |- nat ]` or `[ |- yes_or_no ]`, or we have functions. So, in general we have the following grammar for computation-level types.

Data-level Types	P	::=	nat | yes_or_no | ...
Computation-level Types	T, S	::=	[|- P] | T $\rightarrow$ S

We note there are no tuples in Beluga; this is fine, since we can always write function in a curried form.

Programming with lists We can also easily define lists in Beluga by defining a type `list` with two constructors `nil` and `cons`. We will only create lists of natural numbers [1].

```
list : type.
nil : list.
cons : nat -> list -> list.
```

We can then write a simple function which computes the length of a list by pattern matching on the structure of lists. As previously, we will stage our computation in such a way that we force the evaluation of the recursive call and bind its result to the value `[ |- R ]`.

```
rec length : [ |- list ] -> [ |- nat ] = fn l => case l of
| [ |- nil ] => [ |- z ]
| [ |- cons H T ] => let [ |- N ] = length ([ |- T ]) in [ |- succ N ]
;
```

Finally, we show the implementation of a well-known higher-order function `map`.

```
rec map : ([ |- nat ] -> [ |- nat ]) -> [ |- list ] -> [ |- list ] =
fn f => fn l => case l of
| [ |- nil ] => [ |- nil ]
| [ |- cons H T ] => let [ |- R ] = f [ |- H] in
                   let [ |- T' ] = map f [ |- T ] in
                       [ |- cons R T' ]
;
```

[1]Beluga does not support polymorphism; in fact polymorphism is incompatible with the foundation Beluga rests on and would lead to an inconsistent system. We will discuss the reasons later.

The `map` function takes a function `f` of type `[ |- nat ] -> [ |- nat ]` as a first argument and a list `l` as a second argument. We then apply `f` to each element of the list `l`. The only small difficulty arises from the separation of data and computation; for example, when using `f` we need to box the data `H` before passing it to the computation `f`.

3 Programming with dependent types

So far we have only written simply typed programs. However, simply typed programs do not check very strong guarantees and we need to catch error cases separately. So, for example the type of the `map` function does enforce that the input list and the list we return will have the same length. Let's see how we have to modify our definition of lists and our `map` function to guarantee that given a list of length `N` applying `map` will produce a list of the same length.

3.1 Introduction to programming with dependent types

Lists indexed by their length In general, dependent types allow us to index types by objects; there are in fact a full range of dependently typed languages out there and they are gaining popularity. Among them are for example: Agda, which supports "full" dependent types; Haskell which only allows us to index types by other types; and Beluga, which allows us to index types by terms. But there are several others.

To carry the length of a list as part of the list data-type, we define a *type family*. Here we consider lists containing elements of type `bool` where `bool` has two constructors, `true` and `false`.

```
% Types
bool: type.
true: bool.
false: bool.

nat: type.

z: nat.
succ: nat -> nat.

list : nat -> type.
nil : list z.
cons : bool -> list N -> list (succ N).
```

We can now create some lists as follows:

```
let l1 = [ |- cons true nil ] ;
let l2 = [ |- cons true (cons false nil) ] ;
```

Some remarks about type reconstruction for data Let's look at what the type reconstruction engine will produce as an output. For the first list `l1` it for example infers the type `list (succ z)`. This is of course correct, since indeed we have constructed a list of length 1 and the list we have constructed only had one element, namely `nat`. However, let us inspect the result of type reconstruction more closely.

```
let l1 : [ |- (list (succ z))] = [ |- cons z nat nil] : [ |- (list (succ z))]
 ===> [ |- cons z nat nil]
```

We notice that Beluga turned `cons nat nil` into `cons z nat nil`! To understand what just happened, let us look at the definition of `cons` again.

```
cons : tp -> list N -> list (succ N).
```

The type of `cons` takes in fact not only the element `tp` and another list of type `list N`, but it will also take in the actual `N`!

When processing the type of `cons`, type reconstruction will infer the type of the free variables `N` and abstract over it. This will result in

```
cons : {N:nat} tp -> list N -> list (succ N).
```

which states for every `N:nat`, given an object `T` of type `tp` and a list `L` of type `list N`, `cons N T L` is a list of length `list (succ N)`. In type theory, the universal quantifier is usually written as Π `N:nat.tp` $\longrightarrow$ `list N` $\longrightarrow$ `list (succ N)`. In our concrete syntax, we write

```
{N:nat} tp -> list N -> list (succ N).
```

We refer to `N` as the *implicit* argument, because the user does not need to write it explicitly, but type reconstruction will be to reconstruct it. Any free variable occurring in a type is an implicit argument.

The general recipe of type reconstruction can be described as follows: *If an argument was implicit when the type of a constructor was declared, then we must omit passing it to the constructor.*

Writing the map function Let us now look at the program for the map function:

```
rec map : ([ |- bool ] -> [ |- bool]) -> [ |- list N ] -> [ |- list N ] =
fn f => fn l => case l of
| [ |- nil ] => [ |- nil ]
| [ |- cons H T ] => let [ |- H' ] = f [ |- H ] in
                    let [ |- T' ] = map f [ |- T ] in
                        [ |- cons H' T' ]
;
```

Not much has in fact changed—except the type of lists used in the definition. We now say `map` takes a function of type `[ |- bool ] -> [ |- bool ]` and a list of type `[ |- (list N)]`, and produces a list of type `[ |- (list N)]`. This is quite astounding. Without changing the actual program, we suddenly track a much richer property. While what the user wrote did not change, the program the compiler reconstructed is quite different from the one it reconstructed earlier.

Excursion: type reconstruction for Beluga functions Intuitively, what happens can be explained in two phases: reconstruction of the type and reconstruction of the program. When we look at the declared type of `map`, we notice again that `N` is free. Hence, type reconstruction will infer its type and abstract over it.

```
rec map : {N :: [ |- nat]} ([ |- bool] -> [ |- bool]) -> [ |- (list N)] -> [ |-
    (list N)]
```

The type now states that for all `N`, given a function of type `([ |- tp ] -> [ |- tp ])` and a list of type `[ |- list N ]`, we produce a list of type `[ |- list N ]`. Since `N` was free when the type of `map` was declared, we must omit passing it whenever we use `map`.

And indeed, when we make a recursive call to `map`, we only pass `f` and the smaller list `[ |- T ]`. The exact `N` will be inferred and filled in by the type reconstruction engine.

The function `map` is then elaborated as follows: we first insert abstractions, called `mlam` for the implicit arguments, and then proceed to reconstruct the body of the function. The most interesting things happen in the case-expression: we split `l` which has type `list N` into two cases: `l = [ |- nil ]` and `l = [ |- cons H T ]`. Unlike pattern matching in the simply typed setting, where the pattern `nil` will have the same type as the scrutinee `l` (namely a list), we actually learn something in the dependently typed case: we learn that `N` must be `z`. So, pattern matching on dependently-typed objects refines the types!

Similarly, when pattern matching on `[ |- cons H T ]`, we learn that `N = succ N1`, since the type of `cons H T` is in fact `list (succ N1)`.

Hence, each branch will carry a refinement substitution, which will tell us how the types are to be refined. The reconstructed program fragment of the map function then looks as follows:

```
mlam N => fn f => fn l => case l of
| [[ |- nil] : . z = N ] =>
     [ |- nil] : [ |- (list z)]

| {Y1 :: [ |- (term nat)]} {Z :: [ |- tp]} {X :: [ |- (list Y1)]}
  [[ |- cons Y1 Z X] : . succ Y1 = N ] =>
```

In the second branch, `{Y1 :: [ |- (term nat)]} {Z :: [ |- tp]} {X :: [ |- (list Y1)]}` denote the variables occurring in the pattern `cons Y1 Z X`. We also note that we inferred the implicit argument `Y1` which denotes the length of the list and is passed as an extra argument to `cons`.

We will not go into more details about how types are reconstructed; this is an interesting problem which is actively investigated.

Head and tail One benefit of dependently typed programming is that we can naturally rule out exceptional cases; for example, we'd like to state that the function which returns the head of a list should only be called on a function which is not `nil`; or we may say that calling the tail on the empty list is not meaningful. This leads to the two little functions below:

```
rec head : [ |- list (succ N) ] -> [ |- bool ] =
fn l => let [ |- cons H _ ] = l in [ |- H ]
;
rec tail : [ |- list (succ N) ] -> [ |- list N ] =
fn l => let [ |- cons _ T ] = l in [ |- T]
;
```

The types will enforce that that the function `head` can only be called on a list with length greater than 0; similarly, the `tail` function will expect a list greater than 0 and returns a list whose length decreased by 1. Note that the use of dependent types eliminates the use of option types which allow us to express "yes there is a result" and "no there is no meaningful result". Now, we are ready to write lots of such functions, such as zipping two lists together.

Type preserving evaluation An interesting use of dependent types is when we implement an evaluator. Since we don't know that our expressions are always well-typed, we need to explicitly check for certain cases. Assume for example an implementation of a big-step semantics, where we have a rule for a `switch: term bool -> term T -> term T -> term T` expression, which evaluates the first branch if T1 is true and the second branch if T1 is false:

```
| [ |- switch T1 T2 T3 ] =>
  (case eval ([ |- T1 ]) of
    | [ |- result true ] => eval [ |- T2 ]
    | [ |- result false ] => eval [ |- T3 ]
    | [ |- result _ ] => [ |- no_result ]
    | [ |- _ ] => [ |- no_result ]
  )
```

If we only work with well-typed expressions, we know that the case `[ |- result _ ]` , i.e. evaluating `T1` produced a value which is not a boolean, is impossible. *In fact, we know that if we have a well-typed expression, then evaluation must succeed and return a value.* Recall that if an expression is well-typed then either it is a value or we can take another step and evaluate it to a value. The safety theorem implies that well-typed terms cannot get stuck!

We are rewriting our evaluator into a type-preserving one as follows. We distinguish between well-typed terms and well-typed values.

```
% Types                             % Typed expressions
tp: type.                           term: tp -> type.
bool: tp.
nat : tp.                           true : term bool.
                                    false : term bool.
% Values                            switch: term bool -> term T -> term T
value: tp -> type.                       -> term T.
v_zero: value nat.                  z : term nat.
v_succ: value nat -> value nat.     succ : term nat -> term nat.
v_true: value bool.                 pred : term nat -> term nat.
v_false: value bool.                iszero: term nat -> term bool.
```

Then we can implement the big-step semantics *without any explicit error handling for stuck expressions!* We are statically guaranteed that we can never encounter an ill-typed expression which would lead to a stuck state.

```
rec eval : [ |- term T ] -> [ |- value T ] = fn m => case m of
| [ |- true ] => [ |- v_true ]
| [ |- false ] => [ |- v_false ]
| [ |- switch T1 T2 T3 ] => (case eval [ |- T1 ] of
    | [ |- v_true ] => eval [ |- T2 ]
    | [ |- v_false ] => eval [ |- T3 ]
  )
| [ |- z ] => [ |- v_zero ]
| [ |- succ N ] => let [ |- V ] = eval [ |- N ] in [ |- v_succ V ]
| [ |- pred N ] => (case eval [ |- N ] of
   | [ |- v_zero ] => [ |- v_zero ]
   | [ |- v_succ V ] => [ |- V ]
  )
;
```

This function is type-preserving and will always produce a proper value! Hence, it is a much better (and much shorter and more efficient!) implementation of the big-step semantics. Because, we distinguish between values and expressions and in addition make sure that types are preserved, pattern matching on the result of a recursive call eliminates any need to handle non-meaningful results or stuck states.

3.2 Existentials

When you gain more experience with dependent types, you quickly realize that we often want to have more expressive invariants than simply that the length of a list is preserved. For example, let's write a filter function: filter will keep all elements form a list which satisfy a given property and will drop all elements which do not satisfy the property. Consequently, the length of the list we return is less or equal to the length of the list we started with. We begin by implementing a function, which simply says that given a list of length `N`, filter will produce some list of length `M` without stating the relationship between `N` and `M`. We'd like to say something like this:

```
rec filter: [ |- list N ] -> ([ |- tp ] -> [ |- yes_or_no ])
        -> exists M::[ |- nat ] . [ |- list M ]
```

However, supporting existentials is known to lead to many difficulties; luckily, there is an easy way to work around it: every existentially quantified type can be in fact turned in a universally quantified type! This is a convenient trick to know. We hence first define a type

```
exists_smaller_or_equal_list: type.

smaller_or_equal_list : list M
                     -> exists_smaller_or_equal_list.
```

The constructor `exists_small_or_equal_list` is a new type which we introduce using the key work **type**. It has one constructor, called `smaller_or_equal_list` which takes in one argument: a list (and implicitly also its length).

Then we can write the filter function as follows, keeping `yes_or_no` as above:

```
rec filter: [ |- list N ] -> ([ |- tp ] -> [ |- yes_or_no ])
          -> [ |- exists_smaller_or_equal_list ] =
fn l => fn f => case l of
| [ |- nil ] => [ |- smaller_or_equal_list nil ]
| [ |- cons H T ] => let [ |- smaller_or_equal_list T' ] = filter [ |- T ] f in
                    (case f ([ |- H ]) of
                    | [ |- yes ] => [ |- smaller_or_equal_list (cons H T') ]
                    | [ |- no ] => [ |- smaller_or_equal_list T' ]
                   )
;
```

Note the actual size of the list we return is an implicit argument to `smaller_or_equal_list` and we don't really see it. If we would like to see it, we can give the constructor `smaller_or_equal_list` the type: `{N:nat}list N -> exists_smaller_or_equal_list .` Here we state that we pass to `smaller_or_equal_list` two arguments, the first argument `N` denoting the length and the second the actual list of length `N`.

3.3 Programming with proofs

The filter function, we just wrote, does not quite express what we want: namely that the list we produce will be smaller or equal to the list we have started with. So, how can we achieve to state this additional information?

Once we have dependent types, we can use it to express when a natural number is smaller or equal to another natural number. This information can be directly represented as follows:

```
leq: term nat -> term nat -> type.
le_z : leq z N.
le_s : leq N M -> leq (succ N) (succ M).
```

Here, `leq` is a type which is indexed by two natural numbers. We use the keyword **type** to declare it. There are two constructors of this type called `le_z` and `le_s`. Intuitively, the type of these two constructors describe the relationship of the two natural numbers. We can think of these constructors as encoding two inference rules describing when a natural number `N` is smaller or equal to `M`.

$$\frac{}{\texttt{leq z N}}\ \texttt{le\_z} \qquad \frac{\texttt{leq M N}}{\texttt{leq (succ M)(succ N)}}\ \texttt{le\_s}$$

So, the constructors `le_z` and `le_s` allow us to represent derivations or proofs that a number is indeed smaller than another number. For example the derivation below

$$\cfrac{\cfrac{\cfrac{}{\texttt{leq z (succ z)}}\ \texttt{le\_z}}{\texttt{leq (succ z)(succ (succ z))}}\ \texttt{le\_s}}{\texttt{leq (succ (succ z))(succ (succ (succ z)))}}\ \texttt{le\_s}$$

can be simply represented as an object `le_s (le_s le_z)`. Let's look at the types:

A little lemma One useful lemma is that if `[ |- leq M N ]`, then `[ |- leq M (succ N)]`. The proof is an induction on the first derivation: `[ |- leq M N ]`. There are two cases we need to consider: 1) We used have `le_z : leq z N`, then we of course also have a proof of `leq z (succ N)` using `leq_z`. 2) We consider the case where we have `le_s D : leq (succ M)(succ N)` where `D : leq M N`. By the i.h., we know that `E:leq M (succ N)`. And therefore, `le_s E : leq (succ M)(succ (succ N))`.

This little proof can be directly translated into a recursive program.

```
rec lemma_leq : [ |- leq M N ] -> [ |- leq M (succ N) ] = fn d => case d of
| [ |- le_z ] => [ |- le_z ]
| [ |- le_s D ] => let [ |- E ] = lemma_leq [ |- D ] in [ |- le_s E ]
;
```

The case-analysis in the informal proof corresponds directly to the case-analysis in the program; the appeal to the induction hypothesis corresponds directly to the recursive call.

We have implemented our first proof! Using this little lemma, we can now attack the problem of writing a filter function which will produce a list which is proven to be equal or smaller than the list we have started with.

```
exists_smaller_or_equal_list: term nat -> type.
smaller_or_equal_list: list M -> leq M N
        -> exists_smaller_or_equal_list N.

rec filter : [ |- list N ] -> ([ |- tp ] -> [ |- yes_or_no ])
             -> [ |- exists_smaller_or_equal_list N ] =
fn l => fn f => case l of
| [ |- nil ] => [ |- smaller_or_equal_list nil le_z ]
| [ |- cons H T ] => let [ |- smaller_or_equal_list T' Prf ] = filter [ |- T ] f
    in
   (case f [ |- H ] of
    | [ |- yes ] => [ |- smaller_or_equal_list (cons H T') (le_s Prf) ]
    | [ |- no ] =>
    % Prf : leq M N from which we want to conclude that leq M (succ N)
    let [ |- Prf' ] = lemma_leq [ |- Prf ] in
        [ |- smaller_or_equal_list T' Prf' ]
   )
;
```

The idea of explicitly constructing proofs about certain properties of the program is called *certified programming*. It has many applications: besides certifying that filter indeed produces a list which is provably smaller than the input list, we can use certified programming to certify all kinds of programs: a certified evaluator will not only return a result but also a proof of how this result can be obtained; a certified type checker will not only say "yes" or "no" but return an actual typing derivation as evidence that the given term indeed has a given type.

3.4 Certified programming

Certifying software is important in many settings, because the certificate (=proof) presents independent evidence that something is true. This means we do not need to trust a complicated evaluator or type checker or inference engine, all we need to trust is a simple proof checker which can verify that the certificate (=proof) is valid. In Beluga, proof checking corresponds to type checking. Moreover, the certificate may act as a trail of why a certain action was taken; for example, we can track authorization policies using dependent types, and then the certificate shows why we have granted access to a given resource to someone. A server which grants access may want to store these certificates, since we can then subsequently perform an audit and see why something was allowed (or not).

Certified programming is a form of proof-carrying code where we attach to the output of a program an actual proof certificate which shows how this result has been derived.

We will discuss here a certifying evaluator, i.e., an evaluator which will not only produce a value but also a derivation demonstrating how this value can be derived using the rules of the big-step semantics.

First, we implement the big-step semantics rules using dependent types. For this we define a type family `eval: term T -> value T ->` **type**. This will in fact encode the relationship between a term of type `T` and its value of type `T`. Although we track this typing invariant, our inference rules themselves directly corresponds to our on-paper rules. We only show here the rules for evaluating booleans and the switch-expression.

```
eval : term T -> value T -> type.
ev_true : eval true v_true.
ev_false : eval false v_false.
ev_switch_true: eval M v_true -> eval M1 V
              -> eval (switch M M1 M2) V.
ev_switch_false: eval M v_false -> eval M2 V
               -> eval (switch M M1 M2) V.
```

To certify the evaluation we define a data-type `cert_eval: term T -> type`. It has one constructor: `cert_val: {V : value T} eval M V -> cert_eval M.`

We then write a function `cert_eval` which has type `{M :: [ |- term T ]} [ |- cert_eval M ]`. This states that all terms `M` of type `term T` there exists a certified evaluation for `M`, i.e. there exists a value `V:value T` and a proof `eval M V`. Recall that we use again the trick from Section 3.2. We write the universal quantifier as `{ M :: [ |- term T ]}`. The corresponding language construct in Beluga which introduces `M` is `mlam`-abstraction.

```
rec cert_eval : {M:[ |- term T ]} [ |- cert_eval M ] =
mlam M => case ([ |- M ]) of
| [ |- true ] => [ |- cert_val v_true ev_true ]
| [ |- false ] => [ |- cert_val v_false ev_false ]
| [ |- switch T1 T2 T3 ] => (case cert_eval < . T1> of
  | [ |- cert_val v_true C ] =>
    let [ |- cert_val V C2 ] = cert_eval < . T2 > in
      [ |- cert_val V (ev_switch_true C C2) ]
  | [ |- cert_val v_false C ] =>
    let [ |- cert_val V C3 ] = cert_eval < . T3> in
      [ |- cert_val V (ev_switch_false C C3) ]
  )
;
```

The term `true` evaluates to the value `v_true` and the corresponding evidence for `eval z z` is given by `ev_true`. The case for `false` is similar. When we evaluate a switch-expression, we evaluate `T1`. To use an expression of a universally quantified type, we apply the concrete value by writing `< . T1 >`. The result of `cert_eval < . T1>` is a certified value `cert_val V1 C`. The first case says, `V1` is in fact `v_true`. In this case, we evaluate `T2`. This will produce a certified value `cert_val V C2` and we can now return the value `V` together with the derivation `ev_switch C C2` which corresponds to a proof of `eval (switch T1 T2 T3)V`. The case where `V1` is in fact `v_false` is similar.

4 Proofs, proofs and more proofs

We have seen already how to implement a proof as a recursive program (see Section 3.3). The general idea of translating proofs into programs can be captured as follows:

Proof	Program
Apply I.H.	Make recursive call
Case analysis	Case analysis
Inversion	Let-expression

To interpret proofs as programs has its origin in the Curry-Howard correspondence. Proofs-as-programs correspondence and formulae-as-types correspondence.

Let us revisit this idea for the proof that the small-step semantics is deterministic. First, we represent the small-step rules below:

$$\frac{}{\mathsf{if\ true\ then\ } t_1 \mathsf{\ else\ } t_2 \longrightarrow t_1}\ \text{E-IF-TRUE} \qquad \frac{}{\mathsf{if\ false\ then\ } t_1 \mathsf{\ else\ } t_2 \longrightarrow t_2}\ \text{E-IF-FALSE}$$

$$\frac{t \longrightarrow t'}{\mathsf{if\ } t \mathsf{\ then\ } t_1 \mathsf{\ else\ } t_2 \longrightarrow \mathsf{if\ } t' \mathsf{\ then\ } t_1 \mathsf{\ else\ } t_2}\ \text{E-IF}$$

Our judgment $M \longrightarrow N$ is represented as a type. Every axiom and inference rule will be represented as a constant.

```
step: term T1 -> term T2 -> type.

e_switch_true : step (switch true T2 T3) T2.

e_switch_false: step (switch false T2 T3) T3.

e_switch : step T1 T1'
             -> step (switch T1 T2 T3) (switch T1' T2 T3).
```

Next, we will consider the proof that the small-step semantics is deterministic. Let us recall the proof; we state the theorem referring to the encoding of the step relation to make the translation of it to a Beluga program more direct. Since Beluga does not support any built-in notion of equality, we first define it as follows:

```
eq: term T1 -> term T2 -> type.
ref: eq T T.
```

The `eq` type family characterizes our equality judgment. There is only one way to prove that two terms are equal, namely if the two terms are identical.

Theorem 1 (Determinacy)**.** *If* $\mathcal{S}_1$*:*`step M N1` *and* $\mathcal{S}_2$*:*`step M N2` *then* `eq N1 N2`.

Proof. Induction on $\mathcal{S}_1$.

Case $\mathcal{S}_1 = \dfrac{}{\texttt{step (switch true T1 T2)T1}}\,\texttt{e\_switch\_true}$

Sub-Case: $\mathcal{S}_2 = \dfrac{}{\texttt{step (switch true T1 T2)T1}}\,\texttt{e\_switch\_true}$

`eq T1 T1` by `ref`

Sub-Case: $\mathcal{S}_2 = \dfrac{\begin{array}{c}\mathcal{S}_2' \\ \texttt{step true T'}\end{array}}{\texttt{step (switch true T1 T2)(switch T' T1 T2)}}\,\texttt{e\_switch}$

but there is no rule which could have derived `step true T'`; hence this is impossible

Case $\mathcal{S}_1 = \dfrac{}{\texttt{step (switch false T1 T2)T2}}\,\texttt{e\_switch\_false}$

Similar to previous case.

Case $\mathcal{S}_1 = \dfrac{\begin{array}{c}\mathcal{S}_1' \\ \texttt{step T T'}\end{array}}{\texttt{step (switch T T1 T2)(switch T' T1 T2)}}\,\texttt{e\_switch}$

Sub-case: $\mathcal{S}_2 = \dfrac{}{\texttt{step (switch true T1 T2)T1}}\,\texttt{e\_switch\_true}$

But this now means that `T = true`, and hence $\mathcal{S}_1'$:`step true T'`. But there is no rule which could have derived $\mathcal{S}_1'$; hence this is impossible.

Sub-case: $\mathcal{S}_2 = \dfrac{\begin{array}{c}\mathcal{S}_2' \\ \texttt{step T T''}\end{array}}{\texttt{step (switch T T1 T2)(switch T'' T1 T2)}}\,\texttt{e\_switch}$

$\mathcal{E}$:`eq T' T''` by i.h. on $\mathcal{S}_1'$ and $\mathcal{S}_2'$
`T' = T''` by inversion on $\mathcal{E}$ using `ref`.
`eq (switch T' T1 T2)(switch T' T1 T2)` (recall that `T' = T''`) by using `ref`.

□

This proof above can be directly translated into a program in Beluga; we translate the case-analysis on the small-step rules in the proof into a case-expression which pattern matches on objects of type `step T T1` and subsequently might pattern match on objects of type `step T T2`. The appeal to the induction hypothesis corresponds to making a recursive call. In the proof, we also needed to say that for example there is no possible derivation for `step true T1`; to put it differently, it is impossible to give a derivation which would end in `step true T1`. Since we model derivations as dependently-typed objects, checking whether there is a derivation for `step true T1` corresponds to checking whether there exists an object of type `step true T1`; but there is none. The type `step true T1` is empty, and hence we have shown that it is impossible to ever construct an object of this type. We employ the keyword `impossible e` **in** `[ ]` to check whether indeed the type of the expression `e` is empty.

```
rec det : [ |- step T T1 ] -> [ |- step T T2 ] -> [ |- eq T1 T2 ] =
fn s1 => fn s2 => case s1 of
| [ |- e_switch_true ] =>
    (case s2 of [ |- e_switch_true ] => [ |- ref ]
              | [ |- e_switch S1’ ] => impossible [ |- S1’ ] in [ ]
    )

| [ |- e_switch_false ] =>
  (case s2 of [ |- e_switch_false ] => [ |- ref ]
             | [ |- e_switch S2’ ] => impossible [ |- S2’ ] in [ ]
  )

| [ |- e_switch S1’ ] =>
  (case s2 of
   | [ |- e_switch S2’ ] => let [ |- ref ] = det [ |- S1’ ] [ |- S2’ ] in [ |-
      ref ]
   | [ |- e_switch_false ] => impossible [ |- S1’ ] in [ ]
   | [ |- e_switch_true ] => impossible [ |- S1’ ] in [ ])
;
```

The theorem translates directly into a type in Beluga and we can check whether we have covered all cases using Beluga’s coverage checker (use the flag `+coverage`).

This concludes our brief discussion of programming proofs for now.

5 Lambda Calculus

We now proceed to prove a classic theorems about Church-style simply-typed lambda calculus, type preservation. We begin by defining our types. We let `tp` denote our types and `tm` denote our terms. Then a function (arrow) has the type `tp -> tp -> tp`, and lambda terms have types `tm -> tm -> tm` and `tp -> (tm -> tm)-> tm` for application and abstraction respectively. Finally, values have the type `value`.

```
tp : type.
tm : type.

% Types
arr : tp -> tp -> tp.
```

```
% Terms
app : tm -> tm -> tm.
lam : tp -> (tm -> tm) -> tm.

% Values
value : tm -> type.
```

We must now encode the lambda-calculus' operational semantics. The small-step semantics is

$$\frac{M \longrightarrow M'}{MP \longrightarrow M'P}\ (\text{s\_app1}) \qquad \frac{V\ \text{value} \qquad P \longrightarrow P'}{VP \longrightarrow VP'}\ (\text{s\_app2}) \qquad \frac{V : T\ \text{value}}{(\lambda x : T.Mx)V \longrightarrow MV}\ (\text{s\_app3})$$

To encode this in Beluga, we must introduce a `step` relation, standing in for the $\longrightarrow$ in our operational semantics. We can then straightforwardly implement the above semantics as follows:

```
step: tm -> tm -> type.

s_app1 : step E1 E1' ->
      % -------------------------
        step (app E1 E2) (app E1' E2).

s_app2 : value E1 ->
         step E2 E2' ->
      % -------------------------
        step (app E1 E2) (app E1 E2').

s_app3 : value E2 ->
      % -------------------------
        step (app (lam T (\x. E1 x)) E2) (E1 E2).
```

The preservation theorem states that if $\Gamma \vdash t : T$ and $t \Rightarrow t'$, then $\Gamma \vdash t' : T'$. What's required to encode this theorem in Beluga? First of all, we need a means of representating type relations, i.e. a way of representing $t : T$. We do this by means of a `has_type` predicate. We must also encode our type inference rules,

$$\frac{\Gamma \vdash M : T_1 \rightarrow T_2 \qquad \Gamma \vdash P : T1}{\Gamma \vdash MP : T_2}\ (\text{is\_app}) \qquad \frac{\Gamma, x : T_1 \vdash E : T_2}{\Gamma \vdash (\lambda x : T_1.E) : T_1 \rightarrow T_2}\ (\text{is\_lam})$$

which can be done as follows:

```
% Typing
has_type : tm -> tp -> type.

is_app : has_type E1 (arr T1 T2) ->
         has_type E2 T1 ->
       % -------------------
         has_type (app E1 E2) T2.

is_lam : ({x:tm} has_type x T1 -> has_type (E x) T2) ->
       % ---------------------------------
         has_type (lam T1 (\x. E x)) (arr T1 T2).
```

In `is_lam`, we again used Beluga's concrete syntax for a Π-term. As for proving our theorem, proceed by case-analysis, as in the standard on-paper proof:

```
% Preservation
rec pres : [ |- has_type E T] -> [ |- step E E'] -> [ |- has_type E' T] =
fn d => fn s =>
case s of
  [ |- s_app1 S1] =>
    let [ |- is_app D1 D2] = d in
    let [ |- D1'] = pres [ |- D1] [ |- S1] in
      [ |- is_app D1' D2]

 | [ |- s_app2 V S2] =>
    let [ |- is_app D1 D2] = d in
    let [ |- D2'] = pres [ |- D2] [ |- S2] in
      [ |- is_app D1 D2']

 | [ |- s_app3 V] =>
   let [ |- is_app (is_lam (\x. (\d. (D1 x d)))) D2] = d in
     [ |- (D1 _ D2)]
;
```