Principals of Programming Languages: Revision Lecture

Sam Barrett

January 5, 2021

1 The Simply Typed λ-Calculus

1.1 Syntax

The syntax of the Simply Typed λ-Calculus can be defined as:

$$T ::= \mathbb{B} | T \to T$$
$$M ::= x | \lambda x : T.M | MM | \texttt{true} | \texttt{false} | \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M$$

You can see here that we use the Church style for typing whereby, variables in λ abstractions are annotated with types.

Values are atomic, i.e. they cannot be evaluated further and are of the form:

$$V ::= \lambda x : T.M | \texttt{true} | \texttt{false}$$

When we compute a term we are typically trying to reduce it to a value.

1.2 Evaluation Contexts

When we want to define the call-by-value small-step operational semantics of a language we use evaluation contexts. The Call-by-value evaluation contexts for the small-step operational semantics of λ-Calculus is defined as:

$$C ::= \bullet | CM | VC | \texttt{if}\ C\ \texttt{then}\ M\ \texttt{else}\ M$$

A context is a term with a *hole* ($\bullet$) in it.

You can tell that this is the call-by-value evaluation context as you can see that we always evaluate the arguments of a application before the application itself.

These contexts yield the following rules:

$$\frac{}{(\lambda x : T.M)V \to_v M[x\backslash V])}\beta$$

$$\frac{}{\texttt{if true then}\ M\ \texttt{else}\ N \to_v M}\texttt{IteT}$$

$$\frac{M \to_v N}{C[M] \to_v C[N]}\texttt{CTX}_C$$

$$\frac{}{\texttt{if false then}\ M\ \texttt{else}\ N \to_v N}\texttt{IteF}$$

1.3 Typing Rules

And to facilitate the typing of these expressions we use the following typing rules:

$$\frac{}{\Gamma, x : T \vdash x : T}\texttt{VAR}$$

$$\frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : T \to U}\texttt{ABS}$$

$$\frac{\Gamma \vdash M : T \to U \Gamma \vdash N : T}{\Gamma \vdash MN : U}\texttt{APP}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \mathbb{B}}\texttt{T}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \mathbb{B}}\texttt{F}$$

$$\frac{\Gamma \vdash M : \mathbb{B} \Gamma \vdash N : T \Gamma \vdash P : T}{\Gamma \vdash \texttt{if } M \texttt{ then } N \texttt{ else } P : T}\texttt{ITE}$$

1.4 Church-Numerals

We can define Church Numerals in the Simply Typed λ-Calculus as having the type $\texttt{Nat} = (\mathbb{B} \to \mathbb{B}) \to \mathbb{B} \to \mathbb{B}$.

$$\begin{aligned}
\underline{0} &= \lambda f : \mathbb{B} \to \mathbb{B}.\lambda x : \mathbb{B}.x \\
\underline{1} &= \lambda f : \mathbb{B} \to \mathbb{B}.\lambda x : \mathbb{B}.fx \\
\underline{2} &= \lambda f : \mathbb{B} \to \mathbb{B}.\lambda x : \mathbb{B}.f(fx) \\
\dots
\end{aligned}$$

Essentially, the number is a counter of how many applications of f appear.

We can now define a successor function, `succ` of type $\texttt{Nat} \to \texttt{Nat}$ and an `add` function of type $\texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$:

$$\begin{aligned}
\texttt{succ} &= \lambda a : \texttt{Nat}.\lambda f : \mathbb{B} \to \mathbb{B}.\lambda x : \mathbb{B}.f(afx) \\
\texttt{add} &= \lambda a : \texttt{Nat}.\lambda b : \texttt{Nat}.\lambda f : \mathbb{B} \to \mathbb{B}.\lambda x : \mathbb{B}.af(bfx)
\end{aligned}$$

Our `add` function essentially concatenates the fs in a with the fs in b and our `succ` function appends an f to the value a.

This encoding can be used to iterate over a function of type $\mathbb{B} \to \mathbb{B}$ by applying a function $f : \mathbb{B} \to \mathbb{B}$ to a base case $x : \mathbb{B}$ $n : \texttt{Nat}$ times.

$$nfx$$

However, if one wants to iterate over a function of another type, say, $T \to T$ one will need to define a new set of Church numerals over the type T, i.e. a type of the form $(T \to T) \to T \to T)$ (This is later solved through the use of System-F's paramerisation of types, similar to Monads)

1.4.1 Exercise 1

Can we iterate over a function of type $\texttt{Nat} \rightarrow \texttt{Nat}$ using this method? I.e. can we define $\texttt{add} = \lambda a : \texttt{Nat}.\lambda b : \texttt{Nat}.a \texttt{ succ } b$?

For this to be possible, a would need to have the type $(\texttt{Nat} \rightarrow \texttt{Nat}) \rightarrow (\texttt{Nat} \rightarrow \texttt{Nat})$. However, our value a is of the type $(\mathbb{B} \rightarrow \mathbb{B}) \rightarrow (\mathbb{B} \rightarrow \mathbb{B})$. These types are not compatible, if we were to redefine Nat to fit this function we would end up with a recursive type definition which is not allowed within the Simply Typed λ-Calculus.

Note: This is possible using System-F

1.5 System-F

We define the Church-style syntax of System-F as:

$$T ::= \alpha|\mathbb{B}|\mathbb{N}|T \rightarrow T|\forall\alpha.T$$
$$M ::= x|\lambda x : T.M|MM|\texttt{true}|\texttt{false}|\texttt{if } M \texttt{ then } M \texttt{ else } M|$$
$$\texttt{let } x = M \texttt{ in } M|\texttt{zero}|\texttt{succ}M|\texttt{pred}M|\texttt{iszero}M|\lambda\alpha.M|M\{T\}$$

Here we have both Boolean $\mathbb{B}$ and Nat $\mathbb{N}$ ground types. This leads to simpler examples.

System-F utilises a system of parameterised types. The general form of these types is $\forall\alpha.T$ This defines a family of types whereby for **any** type α. For example, given an expression $M : \forall\alpha.T$, we can construct a any type of the form $M : T[\alpha\backslash T']$ such as $M : T[\alpha\backslash\mathbb{B}]$ or $M : T[\alpha\backslash\mathbb{N}]$

We also have what are known as *type abstractions* in the form of $\lambda\alpha.M$ and *type applications*T of the form $M\{T\}$

Our Values take the form:

$$V ::= \lambda x : T.M|\texttt{true}|\texttt{false}|\texttt{zero}|\texttt{succ}M|\lambda\alpha.M$$

We define our Call-by-value evaluation contexts as:

$$C ::= \bullet|CM|VC|\texttt{if } C \texttt{ then } M \texttt{ else } M|\texttt{let } x = C \texttt{ in } M|\texttt{pred } C|\texttt{iszero } C|C\{T\}$$

Which is the same as before with the extension of let, pred, iszero and type application.

We also have an extended set of Call-by-value small-step operational semantics rules:

$$\frac{}{(\lambda x : T.M)V \rightarrow_v M[x\backslash V]}\ \beta$$

$$\frac{M \rightarrow_v N}{C[M] \rightarrow_v C[N]}\ \texttt{CTX}_C$$

$$\frac{}{\texttt{if true then } M \texttt{ else } N \rightarrow_v M}\ \texttt{IteT}$$

$$\frac{}{\texttt{if false then } M \texttt{ else } N \rightarrow_v N}\ \texttt{IteF}$$

$$\frac{}{\texttt{let}x = V\texttt{in}M \rightarrow_v M[x\backslash V]}\ \texttt{LetV}$$

$$\frac{}{\texttt{pred zero} \rightarrow_v \texttt{zero}}\ \texttt{PredZ}$$

$$\frac{}{\texttt{pred}(\texttt{succ}\ M) \rightarrow_v M}\ \texttt{PredS}$$

$$\frac{}{\texttt{iszero}\ \texttt{zero} \rightarrow_v \texttt{true}}\ \texttt{IsZZ}$$

$$\frac{}{\texttt{iszero}(\texttt{succ}M) \rightarrow_v \texttt{false}}\ \texttt{IsZS}$$

$$\frac{}{(\lambda\alpha.M)\{T\} \rightarrow_v M[\alpha\backslash T]}\ \texttt{T}\beta$$

And our typing rules are as follows:

$$\frac{}{\Gamma, x : T \vdash x : T}\ \texttt{VAR}$$

$$\frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : T \rightarrow U}\ \texttt{ABS}$$

$$\frac{\Gamma \vdash M : T \rightarrow U\ \Gamma \vdash N : T}{\Gamma \vdash MN : U}\ \texttt{APP}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \mathbb{B}}\ \texttt{T}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \mathbb{B}}\ \texttt{F}$$

$$\frac{\Gamma \vdash M : \mathbb{B}\ \Gamma \vdash N : T\ \Gamma \vdash P : T}{\Gamma \vdash \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P : T}\ \texttt{ITE}$$

$$\frac{}{\Gamma \vdash \texttt{zero} : \mathbb{N}}\ \texttt{Z}$$

$$\frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \texttt{succ}M : \mathbb{N}}\ \texttt{S}$$

$$\frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \texttt{pred}M : \mathbb{N}}\ \texttt{P}$$

$$\frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \texttt{iszero}M : \mathbb{B}}\ \texttt{I}$$

$$\frac{\Gamma \vdash M : T\ \Gamma, x : T \vdash N : U}{\Gamma \vdash \texttt{let}x = M\ \texttt{in}\ N : U}\ \texttt{LET}$$

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.T}\ \texttt{TABS}$$

Note TABS also requires that $\alpha \notin \texttt{FV}(\Gamma)$ is satisfied Where $\texttt{FV}(\Gamma)$ is the set of free variables in Γ i.e. α should be a *new* variable with respect to Γ in the hypothesis.

$$\frac{\Gamma \vdash M : \forall\alpha.T}{\Gamma \vdash M\{U\} : T[\alpha\backslash U]}\ \texttt{TAPP}$$

1.5.1 Examples

We previously saw that in the Simply typed λ-Calculus we could not construct a function, $\texttt{add}$ of type $\texttt{Nat} \to \texttt{Nat}$. Using System-F this is possible through an abstracted definition of Church Numerals:

$$
\begin{aligned}
\underline{0} &= \lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.x \\
\underline{1} &= \lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.fx \\
\underline{2} &= \lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(fx) \\
&\dots
\end{aligned}
$$

Again $\texttt{succ}$ has type $\texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$ and $\texttt{add}$ type $\texttt{Nat} \to \texttt{Nat}$:

$$\texttt{succ} = \lambda a : \texttt{Nat}.\lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(a\{\alpha\}fx) \tag{1}$$

$$\texttt{add} = \lambda a : \texttt{Nat}.\lambda b : \texttt{Nat}.\lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.a\{\alpha\}f(b\{\alpha\}fx) \tag{2}$$

Now, given any numeral $n : \texttt{Nat}$ we can iterate over a function F of type $T \to T$ as follows: $n\{T\}F$ and we can define $\texttt{add}$ more simply using $\texttt{succ}$ as:

$$\texttt{add} = \lambda a : \texttt{Nat}.\lambda b : \texttt{Nat}.a\{\texttt{Nat}\}\texttt{succ}b$$

Exemplar 1

Prove that $\texttt{add} = \lambda a : \texttt{Nat}.\lambda b : \texttt{Nat}.a\{\texttt{Nat}\}\texttt{succ}b$ is well typed under System-F. Where $\Gamma = a : \texttt{Nat}, b : \texttt{Nat}$ and assuming $\texttt{succ}$ has type $\texttt{Nat} \to \texttt{Nat}$

Remember: $\texttt{Nat}$ is defined as $\forall\alpha(\alpha \to \alpha) \to \alpha \to \alpha$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash a : \texttt{Nat}}\;\texttt{VAR}}{\Gamma \vdash a\{\texttt{Nat}\} : (\texttt{Nat} \to \texttt{Nat}) \to \texttt{Nat} \to \texttt{Nat}}\;\texttt{TAPP} \quad \dfrac{}{\Gamma \vdash \texttt{succ} : \texttt{Nat} \to \texttt{Nat}}}{\Gamma \vdash a\{\texttt{Nat}\}\texttt{succ} : \texttt{Nat} \to \texttt{Nat}}\;\texttt{APP} \quad \dfrac{}{\Gamma \vdash b : \texttt{Nat}}\;\texttt{VAR}}{\Gamma, b : \texttt{Nat} \vdash a\{\texttt{Nat}\}\texttt{succ}b : \texttt{Nat}}\;\texttt{APP}}{\Gamma, a : \texttt{Nat} \vdash \lambda b : \texttt{Nat}.a\{\texttt{Nat}\}\texttt{succ}b : \texttt{Nat} \to \texttt{Nat}}\;\texttt{ABS}}{\Gamma \vdash \lambda a : \texttt{Nat}.\lambda b : \texttt{Nat}.a\{\texttt{Nat}\}\texttt{succ}b : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}}\;\texttt{ABS}
$$

Remember: To proof something is well typed, construct a proof tree using the relevant typing rules and to prove something evaluates to a certain form use the context rules.

Exemplar 2

What does $\texttt{add}\ \underline{1}\ \underline{1}$ compute to?

$$
\dfrac{\dfrac{}{\texttt{add}\ \underline{1}\ \to_v \lambda b : \texttt{Nat}.\underline{1}\ \{\texttt{Nat}\}\texttt{succ}\ b}\;\beta}{\texttt{add}\ \underline{1}\,\underline{1}\ \to_v (\lambda b : \texttt{Nat}.\underline{1}\ \{\texttt{Nat}\}\texttt{succ}\ b)\underline{1}}\;\texttt{CTX}_{\bullet\ \underline{1}}
$$

$$
\dfrac{}{(\lambda b : \texttt{Nat}.\underline{1}\ \{\texttt{Nat}\}\texttt{succ}\ b)\underline{1}\ \to_v \underline{1}\ \{\texttt{Nat}\}\texttt{succ}\ \underline{1}}\;\beta
$$

$$\dfrac{\dfrac{}{\underline{1}\ \{\mathtt{Nat}\} \to_v \lambda f : \mathtt{Nat} \to \mathtt{Nat}.\lambda x : \mathtt{Nat}.fx}\ T\beta}{\underline{1}\ \{\mathtt{Nat}\}\mathtt{succ}\ \underline{1}\ \to_v (\lambda f : \mathtt{Nat} \to \mathtt{Nat}.\lambda x : \mathtt{Nat}.fx)\mathtt{succ}\ \underline{1}}\ \mathtt{CTX}_{\bullet\mathtt{succ}\ \underline{1}}$$

$$\dfrac{\dfrac{}{(\lambda f : \mathtt{Nat} \to \mathtt{Nat}.\lambda x : \mathtt{Nat}.fx)\mathtt{succ}\ \to_v \lambda x : \mathtt{Nat}.\mathtt{succ}\ x}\ \beta}{(\lambda f : \mathtt{Nat} \to \mathtt{Nat}.\lambda x : \mathtt{Nat}.fx)\mathtt{succ}\ \underline{1}\ \to_v (\lambda x : \mathtt{Nat}.\mathtt{succ}\ x)\underline{1}}\ \mathtt{CTX}_{\bullet\underline{1}}$$

$$\dfrac{}{(\lambda x : \mathtt{Nat}.\mathtt{succ}\ x)\underline{1}\ \to_v \mathtt{succ}\ \underline{1}}\ \beta$$

$$\dfrac{}{\mathtt{succ}\ \underline{1}\ \to_v \lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(\underline{1}\ \{\alpha\}fx)}\ \beta$$

This is our final stage. We cannot reduce a value any further. This value is not the same $\underline{2}$ that we defined earlier, but it is equivalent to it. I.e. we can define an equivalence relation between them.

1.5.2 Inductively defined data structures under System-F

We can define other data types in a similar way to how we have defined Church Numerals under System-F.

For example the list structure:

- $\mathtt{List} = \forall\alpha.(\mathbb{N} \to \alpha \to \alpha) \to \alpha \to \alpha$
- $\mathtt{nil} : \mathtt{List}$
- $\mathtt{nil} = \lambda\alpha.\lambda f : \mathbb{N} \to \alpha \to \alpha.\lambda x : \alpha.x$
- $\mathtt{cons} : \mathbb{N} \to \mathtt{List} \to \mathtt{List}$
- $\mathtt{cons} = \lambda n : \mathbb{N}.\lambda l : \mathtt{List}.\lambda\alpha.\lambda f : \mathbb{N} \to \alpha \to \alpha.\lambda x : \alpha.(fn)(l\{\alpha\}fx)$

Using this encoding, we can define a list of two elements N_1 and N_2 as:

$$\text{List}[N_1, N_2] : \lambda\alpha.\lambda f : \mathbb{N} \to \alpha \to \alpha.\lambda x : \alpha.(fN_1)(fN_2x)$$

1.6 Abstract Data Types

We can extend System-F to capture abstract data types through the use of existential types.

The syntax of this extension can be defined as:

$$\begin{aligned}
T &::= \alpha|\mathbb{B}|\mathbb{N}|T \to T|\forall\alpha.T|T \times T|\exists\alpha.T \\
M &::= x|\lambda x : T.M|MM|\mathtt{true}\ |\mathtt{false}\ |\mathtt{if}\ M\ \mathtt{then}\ M\ \mathtt{else}\ M \\
&\quad |\mathtt{let}\ x = M\ \mathtt{in}\ M|\mathtt{zero}|\mathtt{succ}\ M|\mathtt{pred}M|\mathtt{iszero}M \\
&\quad |\lambda\alpha.M|M\{T\} \\
&\quad |\langle M, M\rangle|\mathtt{fst}M|\mathtt{snd}M \\
&\quad |\mathtt{pack}\langle T, M\rangle asT|\mathtt{unpack}\alpha, x = MinM
\end{aligned}$$

Using existential ($\exists$) types we can *hide* the implementation of a type.

We can say that an abstract data type is comprised of:

- An abstract name
- A concrete representation type
- A concrete implementation
- An abstract interface

Here the α in the existential type facilitates the abstract name.

Packs create abstract data types where T is the concrete representation type, and M is the concrete implementation.

We define an abstract data type in the form:

$$\texttt{pack}\langle T, M\rangle \texttt{ as } \exists\alpha.U$$

We extend the definition of values to include pairs and packs:

$$V ::= \lambda x : T.M|\texttt{true}|\texttt{false}|\texttt{zero}|\texttt{succ}M|\lambda\alpha.M|\langle M, M\rangle|\texttt{pack}\langle T, M\rangle \texttt{ as } T$$

And our Call-by-value evaluation contexts are now:

$$\begin{aligned}C ::= \bullet|CM|VC|\texttt{if } C \texttt{ then } M \texttt{ else } M \\ |\texttt{let } x = C \texttt{ in } M|\texttt{pred } C|\texttt{iszero } C|C\{T\} \\ |\texttt{unpack } \alpha, x = C \texttt{ in } M|\texttt{fst } C|\texttt{snd}\end{aligned}$$

We also extend our operational semantics rules:

$$\frac{}{(\lambda x : T.M)V \to_v M[x\backslash V]}\ \beta$$

$$\frac{M \to_v N}{C[M] \to_v C[N]}\ \texttt{CTX}_C$$

$$\frac{}{\texttt{if true then } M \texttt{ else } N \to_v M}\ \texttt{IteT}$$

$$\frac{}{\texttt{if false then } M \texttt{ else } N \to_v N}\ \texttt{IteF}$$

$$\frac{}{\texttt{let}x = V\texttt{in}M \to_v M[x\backslash V]}\ \texttt{LetV}$$

$$\frac{}{\texttt{pred zero} \to_v \texttt{zero}}\ \texttt{PredZ}$$

$$\frac{}{\texttt{pred(succ } M) \to_v M}\ \texttt{PredS}$$

$$\frac{}{\texttt{iszero zero} \to_v \texttt{true}}\ \texttt{IsZZ}$$

$$\frac{}{\texttt{iszero(succ}M) \to_v \texttt{false}}\ \texttt{IsZS}$$

$$\frac{}{(\lambda\alpha.M)\{T\} \to_v M[\alpha\backslash T]}\ \texttt{T}\beta$$

$$\frac{}{\texttt{fst}\ \langle M,N\rangle \to_v M}\ \texttt{FstP}$$

$$\frac{}{\texttt{snd}\langle M,N\rangle \to_v N}\ \texttt{SndP}$$

$$\frac{}{\texttt{unpack}\ \alpha, x = \ \texttt{pack}\ \langle T,M\rangle\ \texttt{as}\ U\ \texttt{in}\ N \to_v N[\alpha\backslash T][x\backslash M]}\ \texttt{UnP}$$

We extend our typing rules as follows:

$$\frac{}{\Gamma, x : T \vdash x : T}\ \texttt{VAR}$$

$$\frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : T \to U}\ \texttt{ABS}$$

$$\frac{\Gamma \vdash M : T \to U\ \Gamma \vdash N : T}{\Gamma \vdash MN : U}\ \texttt{APP}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \mathbb{B}}\ \texttt{T}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \mathbb{B}}\ \texttt{F}$$

$$\frac{\Gamma \vdash M : \mathbb{B}\ \Gamma \vdash N : T\ \Gamma \vdash P : T}{\Gamma \vdash \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ P : T}\ \texttt{ITE}$$

$$\frac{}{\Gamma \vdash \texttt{zero} : \mathbb{N}}\ \texttt{Z}$$

$$\frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \texttt{succ}M : \mathbb{N}}\ \texttt{S}$$

$$\frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \texttt{pred}M : \mathbb{N}}\ \texttt{P}$$

$$\frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \texttt{iszero}M : \mathbb{B}}\ \texttt{I}$$

$$\frac{\Gamma \vdash M : T\ \Gamma, x : T \vdash N : U}{\Gamma \vdash \texttt{let}x = M\ \texttt{in}\ N : U}\ \texttt{LET}$$

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.T}\ \texttt{TABS}$$

Note TABS also requires that $\alpha \notin \texttt{FV}(\Gamma)$ is satisfied Where $\texttt{FV}(\Gamma)$ is the set of free variables in Γ i.e. α should be a *new* variable with respect to Γ in the hypothesis.

$$\frac{\Gamma \vdash M : \forall\alpha.T}{\Gamma \vdash M\{U\} : T[\alpha\backslash U]}\ \texttt{TAPP}$$

$$\frac{\Gamma \vdash M : T\ \Gamma \vdash N : U}{\Gamma \vdash \langle M,N\rangle : T \times U}\ \texttt{Pair}$$

$$\frac{\Gamma \vdash M : T \times U}{\Gamma \vdash \texttt{fst}\ M : T}\ \texttt{Fst}$$

$$\frac{\Gamma \vdash M : T \times U}{\Gamma \vdash \texttt{snd}\ \ M : U}\ \texttt{Snd}$$

$$\frac{\Gamma \vdash M : U[\alpha \backslash T]}{\Gamma \vdash \texttt{pack}\ \ \langle T, M\rangle\ \ \texttt{as}\ \ \exists\alpha.U : \exists\alpha.U}\ \texttt{Pack}$$

$$\frac{\Gamma \vdash M : \exists\alpha.T\ \Gamma, x : T \vdash N : U}{\Gamma \vdash \texttt{unpack}\ \ \alpha, x = MinN : U}\ \texttt{Unpack}$$

Note: unpack requires that the condition $\alpha \notin \texttt{FV}(U)$

1.6.1 Existential Types - Examples

$$\texttt{pack}\ \langle \mathbb{B}, \langle \texttt{true}\ \ , \lambda x : \mathbb{B}.\texttt{if}\ \ x\ \ \texttt{then false}\ \ \ \texttt{else true}\ \ \rangle\rangle\ \texttt{as}\ \exists\alpha.\alpha \times (\alpha \to \alpha)$$

For this example, the concrete representation type is Boolean ($\mathbb{B}$), we define two operations on this type. Firstly we define a constant of true and secondly we define an operation that takes a boolean and negates it. We pack this concrete implementation inside of the pack. Visible to the user is the name α and the abstract types of the operations contained within the pack.

Another example could be the List type we defined earlier.

$$\texttt{pack}\ \langle \texttt{List}, \langle \texttt{nil}, \texttt{cons}\rangle\rangle\ \texttt{as}\ \exists\alpha.\alpha \times (\mathbb{N} \to \alpha \to \alpha)$$

Here the user does not know about the List type and instead is allowed to construct an α using the first operation and to construct further structures using this initial α (nil) and the second operation that takes a $\mathbb{N}$ and a α.