Explicitly typed polymorphic languages, such as the version of MinHS introduced with parametric polymorphism, are very awkward to use in practice. The user must make explicit type abstractions and applications. Ideally, we would like to leave these type annotations \textit{implicit}, and have the compiler infer types for us.

Considering the following expression:

\begin{verbatim}
recfun f x = fst x + 1
\end{verbatim}

We could give a number of possible types to this expression:

\begin{itemize}
  \item \((\text{Int} \times \text{Int}) \rightarrow \text{Int}\)
  \item \((\text{Int} \times \text{Bool}) \rightarrow \text{Int}\)
  \item \((\text{Int} \times 0) \rightarrow \text{Int}\)
\end{itemize}

The exact type inferred must depend on the surrounding context; that is, the argument to which this function is applied. If the function is applied to many different arguments, then we would need to generalize the type to \(\forall a. (\text{Int} \times a) \rightarrow \text{Int}\).

In order to make type annotations implicit, we will start with polymorphic MinHS but remove the following features:

\begin{itemize}
  \item type signatures from \textit{recfun}, \textit{let}, etc.
  \item explicit \textit{type} abstractions, and type applications (the \(@@\) operator).
  \item recursive types, because there is no unique most general type \textit{(principal type)} for a given term if we have general recursive types.
\end{itemize}

Our types may still contain type variables quantified by the \(\forall\) operator, however now the compiler, not the user, determines when to generalize and specialize types.

\section{Implicitly-typed MinHS}

The basic constructs of implicitly-typed MinHS are identical to explicitly-typed MinHS:

\begin{align*}
  \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} & \quad \text{VAR} & \quad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2} & \quad \text{APP} \\
  \frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{if} \ e_1 \ e_2 \ e_3) : \tau} & \quad \text{IF}
\end{align*}

\*Minor edits.
For simplicity, however, we will treat constructors and primitive operations as functions, whose
types are available in the environment. Uses of these operations and constructors are then just
function applications:

\[ (+) : \text{Int} \to \text{Int} \to \text{Int} \]

\[ \Gamma \vdash (\text{App} (\text{App} (+) (\text{Num} 2)) (\text{Num} 1)) : \text{Int} \]

Other functions are defined as usual with `recfun`, but now types are not mentioned in the term:

\[ x : \tau_1, f : \tau_1 \to \tau_2, \Gamma \vdash e : \tau_2 \]

\[ \Gamma \vdash (\text{Recfun} (f.x. e)) : \tau_1 \to \tau_2 \]

The two constructs for polymorphism, type abstraction (`type`) and application (the `@` operator),
have now been removed. But, we still have the typing rules that allow us to specialise a polymorphic
type (replacing `@`):

\[ \Gamma \vdash e : \forall a. \tau \]

\[ \Gamma \vdash e : \tau[a := \rho] \]

And to quantify over any variable that has not already been used (replacing `type`):

\[ \Gamma \vdash e : \tau \quad a \notin TV(\Gamma) \]

\[ \Gamma \vdash e : \forall a. \tau \]

2 An Algorithm

We want a fully deterministic algorithm for type inference, which has a clear input and output.
We could imagine interpreting our existing rules as an algorithm, where the context and expression
are the input and the type is the output:

\[ \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1 \]

\[ \Gamma \vdash (\text{Recfun} (f.x. e)) : \tau_1 \to \tau_2 \]

\[ \text{infer} :: \text{Context} \to \text{Expr} \to \text{Type} \]

However this causes problems when we examine the rules for polymorphism (`AllE` and `AllI`).
Neither the rule to introduce nor the rule to eliminate `∀` quantifiers is syntax directed. They can
be applied at any time. For example, our `AllI` rule:

\[ \Gamma \vdash e : \forall a. \tau \quad a \notin TV(\Gamma) \]

\[ \Gamma \vdash e : \forall a. \tau \]

Because this rule works on any expression and context, we have an infinite number of possible
types for every possible expression. `Num 5` could be of type `Int` or `∀a. Int` or `∀a. ∀b. Int` etc.

In order to have an algorithmic set of rules, we need to fix not just when these rules are applied
but also how they are applied. For example, the rule to specialise a polymorphic type replaces a
quantified type variable with any type `ρ`, where this type is not able to be determined from the
input context and expression:

\[ \Gamma \vdash e : \forall a. \tau \]

\[ \Gamma \vdash e : \tau[a := \rho] \]

If the compiler makes the wrong decision when applying this rule, it can lead to typing errors even
for well-typed programs:

\[ \Gamma \vdash \text{fst} : \forall a. \forall b. (a \times b) \to a \]

\[ \Gamma \vdash \text{fst} : (\text{Bool} \times \text{Bool}) \to \text{Bool} \]

\[ \Gamma \vdash (\text{Pair} 1 \text{True}) : (\text{Int} \times \text{Bool}) \]

\[ \Gamma \vdash (\text{Apply} \text{fst} (\text{Pair} 1 \text{True})) : ??? \]

In the above example, we instantiated the type variable `a` to `Bool`, even though the provided pair
is actually `Int × Bool`.

\[ ^1 \text{Where } TV(\Gamma) \text{ here is all type variables occurring free in the types of variables in } \Gamma \]
The Solution

To start with, we will make two decisions:

1. ∀ quantified type variables will be instantiated to particular types as soon as a polymorphic type is found in the context for a particular term variable. That is, we shall merge the \( \text{All}_E \) and \( \text{Var} \) rules, and not have a separate \( \text{All}_E \) rule.

2. ∀ quantifiers will only be introduced for the types of variables bound in \texttt{let} expressions. So, we will not have a separate \( \text{All}_I \) rule either.

Leaving the second decision aside for a moment, we still have a problem with the first. We have fixed \( \text{when} \) the rule is applied but not \( \text{how} \): If we instantiate each \( \forall \)-quantified variable to a particular type as soon as possible, we will not (yet) know what type to instantiate it to. For example, looking up the type of \( \texttt{fst} \) in the context gives us a type \( \forall a. \forall b. (a \times b) \to a \), but we do not know at that point what \( a \) and \( b \) should be replaced with.

To resolve this, we allow types to include unknowns, also known as unification variables or schematic variables. These are placeholders for types that we haven’t worked out yet. We shall use \( \alpha, \beta \) etc. for these variables. For example, \( (\texttt{Int} \times \alpha) \to \beta \) is the type of a function from tuples where the left side of the tuple is \( \texttt{Int} \), but no other details of the type have been determined yet.

As we encounter situations where two types should be equal, we unify the two types to determine what the unknown variables should be, producing a substitution to these unknowns.

\[
\Gamma \vdash \texttt{fst} : \forall a. \forall b. (a \times b) \to a \\
\Gamma \vdash (\texttt{Pair} \, 1 \, \texttt{True}) : (\texttt{Int} \times \texttt{Bool})
\]

In the above example, we instantiated the quantified variables \( a \) and \( b \) in the type of \( \texttt{fst} \) to \( \alpha \) and \( \beta \), and used a placeholder \( \gamma \) to refer to the return type of the overall function application. Once we inferred the type of the argument as \( (\texttt{Int} \times \texttt{Bool}) \), we must now unify the type of the function we inferred \( (\alpha \times \beta) \to \alpha \) and the type of the function we expect based on the type of the argument we inferred \( (\texttt{Int} \times \texttt{Bool}) \to \gamma \):

\[
(\alpha \times \beta) \to \alpha \sim (\texttt{Int} \times \texttt{Bool}) \to \gamma
\]

Once we unify these two types, we get the unifier substitution:

\[
[a := \texttt{Int}, \beta := \texttt{Bool}, \gamma := \texttt{Int}]
\]

Observe that if this substitution is applied to the two types above, they become the same.

Unifiers

A substitution \( S \) to unification variables is a unifier of two types \( \tau \) and \( \rho \) iff \( S\tau = S\rho \).

Furthermore, it is the most general unifier, or mgu, of \( \tau \) and \( \rho \) if there is no other unifier \( S' \) where \( S\tau \subseteq S'\tau \).

We write \( \tau \sim U \rho \) if \( U \) is the mgu of \( \tau \) and \( \rho \).

Sometimes two types do not have a unifier. A clear example is \( \texttt{Int} \) and \( \texttt{String} \) — both types are concrete, and no amount of substitution to unknown variables will make them the same.

We can compute unifiers by structurally matching them. Our unify function would have a type like below, where the \texttt{Type} arguments do not include any \( \forall \) quantifiers and the \texttt{Unifier} returned is the mgu:

\[
\text{unify} :: \text{Type} \to \text{Type} \to \text{Maybe Unifier}
\]
We shall discuss cases for \texttt{unify} $\tau_1 \tau_2$:

1. Both are type variables: $\tau_1 = v_1$ and $\tau_2 = v_2$:
   \begin{itemize}
   \item $v_1 = v_2 \Rightarrow$ empty unifier
   \item $v_1 \neq v_2 \Rightarrow [v_1 := v_2]$
   \end{itemize}

2. Both are primitive type constructors: $\tau_1 = C_1$ and $\tau_2 = C_2$:
   \begin{itemize}
   \item $C_1 = C_2 \Rightarrow$ empty unifier
   \item $C_1 \neq C_2 \Rightarrow$ no unifier
   \end{itemize}

3. Both are product types $\tau_1 = \tau_{11} \times \tau_{12}$ and $\tau_2 = \tau_{21} \times \tau_{22}$.
   \begin{itemize}
   \item (a) Compute the mgu $S$ of $\tau_{11}$ and $\tau_{21}$.
   \item (b) Compute the mgu $S'$ of $S\tau_{12}$ and $S\tau_{22}$.
   \item (c) Return $S \cup S'$
   (same for sum, function types)
   \end{itemize}

4. One is a type variable $v$, the other is just any term $t$.
   \begin{itemize}
   \item $v$ occurs in $t \Rightarrow$ no unifier
   \item otherwise $\Rightarrow [v := t]$
   \end{itemize}

5. Any other case $\Rightarrow$ no unifier.

Try the algorithm out on the following examples:

1. $\alpha \times (\alpha \times \alpha) \sim \beta \times \gamma$ $[\beta := \alpha, \gamma := (\alpha \times \alpha)]$
2. $(\alpha \times \alpha) \times \beta \sim \beta \times \gamma$ $[\gamma := (\alpha \times \alpha), \beta := (\alpha \times \alpha)]$
3. $\text{Int} + \alpha \sim \alpha + \text{Bool}$ (no unifier)
4. $(\alpha \times \alpha) \times \alpha \sim \alpha \times (\alpha \times \alpha)$ (no unifier)

The last example is particularly interesting because if we ignore the “occurs check” in case 4 of the algorithm, and naively try to structurally match, we end up with a substitution:

$[\alpha := (\alpha \times \alpha)]$

But, applying this substitution to both sides of the original problem yields:

$((\alpha \times \alpha) \times (\alpha \times \alpha)) \times (\alpha \times \alpha) \sim (\alpha \times \alpha) \times ((\alpha \times \alpha) \times (\alpha \times \alpha))$

And both type terms are still not the same. Even worse, trying again yields the exact same substitution we started with. This is called an \textit{infinite type} error.
Type Inference Rules

We will decompose the typing judgement to allow for an additional output — a substitution that contains all the unifiers we have found about unknowns so far.

**Inputs** Expression, Context

**Outputs** Type, Substitution

We will write this as $ST \vdash e : \tau$, to make clear how the original typing judgement may be reconstructed.

Our new, combined variable and instantiation rule replaces all quantified variables with fresh unknown variables. Here “fresh” just indicates that the variable name has never been used before:

$$\left( x : \forall a_1. \forall a_2. \ldots \forall a_n. \tau \right) \in \Gamma$$

Observe that when the variable’s type is not polymorphic (i.e. no quantifiers), then the above rule simplifies to our previous Var rule.

For functions, we generate two placeholders for the type of the function and its argument, respectively. Then we unify the function’s type with the expected one based on the inferred return type $\tau$.

Let Generalisation

Earlier we decided to use let expressions as the syntactic point for $\forall$-generalisation. If we consider this example:

```plaintext
let f = (recfun f x = (x, x)) in (fst (f 4), fst (f True))
```

Just examining the inner recfun, we would compute a type like $\alpha \rightarrow (\alpha \times \alpha)$. The placeholder $\alpha$ would not be in use anywhere else — it would not be mentioned in the context outside of the recfun. We would expect the function $f$ in the context to have a type like $\forall a. a \rightarrow (a \times a)$.

Thus, we can define our generalisation operation to take all free placeholder variables in the type that are not still in use in our context, and $\forall$ quantify them. More formally, we define $Gen(\Gamma, \tau) = \forall (TV(\tau) \setminus TV(\Gamma))$, $\tau$

Then our rule for let expressions generalises the type before adding it to the context:

$$S(\Gamma, x : \alpha_1, f : \alpha_2) \vdash e : \tau \quad S\alpha_2 U (\alpha_1 \rightarrow \tau)$$

This means that let expressions are now not just sugar for a function application, they actually play a vital role in the language’s syntax, as a place for generalisation to occur.
Overall

We’ve specified Robin Milner’s algorithm $W$ for type inference, also called Damas-Milner type inference. Many other algorithms exist, for other kinds of type systems, including explicit constraint-based systems. This algorithm is restricted to the Hindley-Milner subset of decidable polymorphic instantiations, and requires that polymorphism is top-level — polymorphic functions are not first class.