Chapter 1

Extensions to the simply typed \(\lambda\)-calculus

This chapter presents three extensions to the simply typed \(\lambda\)-calculus: \textit{product types}, \textit{sum types}, and the \textit{fixed point construct}. Product types account for pairs, tuples, records, and units in SML. Sum types are sometimes (no pun intended!) called \textit{disjoint unions} and can be thought of as special cases of datatypes in SML. Like the fixed point \textit{combinator} for the untyped \(\lambda\)-calculus, the fixed point \textit{construct} enables us to encode recursive functions in the simply typed \(\lambda\)-calculus. Unlike the fixed point combinator, however, it is \textit{not} syntactic sugar: it cannot be written as another expression and its addition strictly increases the expressive power of the simply typed \(\lambda\)-calculus.

1.1 Product types

The idea behind product types is that a value of a product type \(A_1 \times A_2\) contains a value of type \(A_1\) and also a value of type \(A_2\). In order to create an expression of type \(A_1 \times A_2\), therefore, we need two expressions: one of type \(A_1\) and another of type \(A_2\); we use a \textit{pair} \((e_1, e_2)\) to pair up two expressions \(e_1\) and \(e_2\). Conversely, given an expression of type \(A_1 \times A_2\), we may need to extract its individual components. We use \textit{projections} \texttt{fst} \(e\) and \texttt{snd} \(e\) to retrieve the first and the second component of \(e\), respectively.

\[
\text{type} \quad A ::= \cdots | A \times A \\
\text{expression} \quad e ::= \cdots | (e_1, e_2) | \texttt{fst} e | \texttt{snd} e
\]

As with function types, a typing rule for product types is either an introduction rule or an elimination rule. Since there are two kinds of projections, we need two elimination rules (\(\times E_1\) and \(\times E_2\)):

\[
\begin{align*}
\Gamma &\vdash e_1 : A_1 & \Gamma &\vdash e_2 : A_2 \\
\Gamma &\vdash (e_1, e_2) : A_1 \times A_2 \\
\Gamma &\vdash e : A_1 \times A_2 \\
\Gamma &\vdash \text{fst } e : A_1 \\
\Gamma &\vdash \text{snd } e : A_2 \\
\Gamma &\vdash e : A_1 \times A_2 \\
\Gamma &\vdash \text{fst } e_1 : A_1 \\
\Gamma &\vdash \text{snd } e_2 : A_2 \\
\end{align*}
\]

As for reduction rules, there are two alternative strategies which differ in the definition of values of product types (just like there are two reduction strategies for function types). If we take an \textit{eager} approach, we do not regard \((e_1, e_2)\) as a value; only if both \(e_1\) and \(e_2\) are values do we regard it as a value, as stated in the following definition of values:

\[
\text{value} \quad v ::= \cdots | (v, v)
\]

Here the ellipsis \(\cdots\) denotes the previous definition of values which is irrelevant to the present discussion of product types. Then the eager reduction strategy is specified by the following reduction rules:

\[
\begin{align*}
\text{Pair} &\quad (e_1, e_2) \mapsto (e_1', e_2') \\
\text{Pair'} &\quad (v_1, e_2) \mapsto (v_1, e_2') \\
\end{align*}
\]
Exercise 1.1. Why is it a bad idea to reduce \( \text{fst} \)(\( e_1, e_2 \)) to \( e_1 \) (and similarly for \( \text{snd} \)(\( e_1, e_2 \))) under the eager reduction strategy?

In order to incorporate these reduction rules into the operational semantics, we extend the definition of \( FV(e) \) and \([e'/x]e\) accordingly:

\[
FV((e_1, e_2)) = FV(e_1) \cup FV(e_2) \quad [e'/x](e_1, e_2) = ([e'/x]e_1, [e'/x]e_2) \\
FV(\text{fst } e) = FV(e) \quad [e'/x]\text{fst } e = \text{fst } [e'/x]e \\
FV(\text{snd } e) = FV(e) \quad [e'/x]\text{snd } e = \text{snd } [e'/x]e
\]

1.2 General product types and unit type

Product types are easily generalized to \( n \)-ary cases \( A_1 \times A_2 \times \cdots \times A_n \). A tuple \( (e_1, e_2, \ldots, e_n) \) has a general product type \( A_1 \times A_2 \times \cdots \times A_n \) if \( e_i \) has type \( A_i \) for \( 1 \leq i \leq n \). A projection \( \text{proj}_i \) \( e \) now uses an index \( i \) to indicate which component to retrieve from \( e \).

| type | \( A \) := … | \( A_1 \times A_2 \times \cdots \times A_n \) |
| expression | \( e \) := … | \( (e_1, e_2, \ldots, e_n) \mid \text{proj}_i \ e \) |

\[
\Gamma \vdash e_i : A_i \quad 1 \leq i \leq n \\
\Gamma \vdash (e_1, e_2, \ldots, e_n) : A_1 \times A_2 \times \cdots \times A_n \\
\Gamma \vdash e : A_1 \times A_2 \times \cdots \times A_n \quad 1 \leq i \leq n \\
\Gamma \vdash \text{proj}_i \ e : A_i \quad \times E_i
\]

As in binary cases, eager and lazy reduction strategies are available for general product types. Below we give the specification of the eager reduction strategy; the lazy reduction strategy is left as an exercise.

| value | \( v \) := … | \( (v_1, v_2, \ldots, v_n) \) |

\[
\begin{align*}
e_i &\mapsto e_i' \\
(v_1, v_2, \ldots, v_{i-1}, e_i, \ldots, v_n) &\mapsto (v_1, v_2, \ldots, v_{i-1}, e_i', \ldots, v_n) & \text{Pair} \\
\text{proj}_i \ e &\mapsto \text{proj}_i \ e' & \text{Proj} \quad 1 \leq i \leq n \\
\text{proj}_i \ (v_1, v_2, \ldots, v_n) &\mapsto v_i & \text{Proj'}
\end{align*}
\]

Of particular importance is the special case \( n = 0 \) in a general product type \( A_1 \times A_2 \times \cdots \times A_n \). To better understand the ramifications of setting \( n \) to 0, let us interpret the rules \( \times 1 \) and \( \times E_i \) as follows:

- The rule \( \times 1 \) says that in order to build a value of type \( A_1 \times A_2 \times \cdots \times A_n \), we have to provide \( n \) different values of types \( A_1 \) through \( A_n \) in the premise.
- The rule \( \times E_i \) says that since we have already provided \( n \) different values of types \( A_1 \) through \( A_n \), we may retrieve any of these values individually in the conclusion.
Now let us see what happens when we set \( n \) to 0:

- In order to build a value of type \( A_1 \times A_2 \times \cdots \times A_n \), we have to provide \( n \) different values. That is, we do not have to provide any value in the premise at all!

- Since we have provided \( n \) different values, we cannot retrieve any value in the conclusion at all. That is, the rule \( \times E_i \) never applies if \( n = 0 \! /

The type \( \text{unit} \) is a general product type \( A_1 \times A_2 \times \cdots \times A_n \) with \( n = 0 \). It has an introduction rule with no premise (because we do not have to provide any value), but has no elimination rule (because there is no way to retrieve a value after providing no value). An expression \( () \) is called a \( \text{unit} \) and is the only value belonging to type \( \text{unit} \). The typing rule Unit below is the introduction rule for \( \text{unit} \):

\[
\text{type} \quad A ::= \cdots \mid \text{unit} \\
\text{expression} \quad e ::= \cdots \mid () \\
\text{value} \quad v ::= \cdots \mid () \\
\Gamma \vdash () : \text{Unit}
\]

The type \( \text{unit} \) is useful when we introduce computational effects such as input/output and mutable references. For example, a function returning a character typed by the user does not need an argument of particular meaning. Hence it may use \( \text{unit} \) as the type of its arguments.

### 1.3 Sum types

The idea behind sum types is that a value of a sum type \( A_1 + A_2 \) contains a value of type \( A_1 \) or \( \text{else} \) a value of type \( A_2 \), but not both. Therefore there are two ways to create an expression of type \( A_1 + A_2 \): using an expression \( e_1 \) of type \( A_1 \) and using an expression \( e_2 \) of type \( A_2 \). In the first case, we use a \( \text{left injection} \), or \( \text{inleft} \) for short, \( \text{inl}_{A_1} e_1 \); in the second case, we use a \( \text{right injection} \), or \( \text{inright} \) for short, \( \text{inr}_{A_1} e_2 \).

Then how do we extract back a value from an expression of type \( A_1 + A_2 \)? In general, it is unknown which of the two types \( A_1 \) and \( A_2 \) has been used in creating a value of type \( A_1 + A_2 \). For example, in the body \( e \) of a \( \lambda \)-abstraction \( \lambda x : A_1 + A_2. e \), nothing is known about variable \( x \) except that its value can be created from a value of either type \( A_1 \) or type \( A_2 \). In order to examine the value associated with an expression of type \( A_1 + A_2 \), therefore, we have to provide for two possibilities: when a left injection has been used and when a right injection has been used. We use a \( \text{case expression} \) \( \text{case} e \mid \text{inl}_{A_1} e_1 | \text{inr}_{A_1} e_2 \) to perform a case analysis on expression \( e \) which must have a sum type \( A_1 + A_2 \). Informally speaking, if \( e \) has been created with a value \( v_1 \) of type \( A_1 \), the case expression takes the first branch, reducing \( e_1 \) after binding \( x_1 \) to \( v_1 \); otherwise it takes the second branch, reducing \( e_2 \) in an analogous way.

\[
\text{type} \quad A ::= \cdots \mid A + A \\
\text{expression} \quad e ::= \cdots \mid \text{inl}_A e \mid \text{inr}_A e \mid \text{case} e \mid \text{inl}_A x. e \mid \text{inr}_A x. e \\
\]

As is the case with function types and product types, a typing rule for sum types is either an introduction rule or an elimination rule. Since there are two ways to create an expression of type \( A_1 + A_2 \), there are two introduction rules (\( \text{+}_L \) for \( \text{inl}_A e \) and \( \text{+}_R \) for \( \text{inr}_A e \)):

\[
\begin{align*}
\Gamma \vdash e : A_1 \\
\Gamma \vdash \text{inl}_{A_1} e : A_1 + A_2 & \quad \text{+}_L \\
\Gamma \vdash e : A_2 \\
\Gamma \vdash \text{inr}_{A_1} e : A_1 + A_2 & \quad \text{+}_R \\
\Gamma \vdash e : A_1 + A_2 & \quad \text{case} e \mid \text{inl}_A x. e_1 | \text{inr}_A x. e_2 : C \\
\end{align*}
\]

In the rule \( \text{+}_E \), expressions \( e_1 \) and \( e_2 \) must have the same type; otherwise we cannot statically determine the type of the whole case expression.

October 10, 2006
As with product types, reduction rules for sum types depend on the definition of values of sum types. An eager approach uses the following definition of values:

\[
\text{value } \ v \ ::= \ \cdots \ | \ \text{inl}_A\ v \ | \ \text{inr}_A\ v
\]

Then the eager reduction strategy is specified by the following reduction rules:

\[
\begin{align*}
\text{case } e & \rightarrow e' \\
\text{case } \text{inl}_A\ v & \rightarrow \text{case } e' \text{ of } \text{inl}_A\ v \\
\text{case } \text{inr}_A\ v & \rightarrow \text{case } e' \text{ of } \text{inr}_A\ v
\end{align*}
\]

A lazy approach regards \( \text{inl}_A\ e \) and \( \text{inr}_A\ e \) as values regardless of the form of expression \( e \):

\[
\text{value } \ v \ ::= \ \cdots \ | \ \text{inl}_A\ e \ | \ \text{inr}_A\ e
\]

Then the lazy reduction strategy is specified by the following reduction rules:

\[
\begin{align*}
\text{case } e & \rightarrow e' \\
\text{case } \text{inl}_A\ e & \rightarrow \text{case } e' \text{ of } \text{inl}_A\ e \\
\text{case } \text{inr}_A\ e & \rightarrow \text{case } e' \text{ of } \text{inr}_A\ e
\end{align*}
\]

In extending the definition of \( FV(e) \) and \( [e'/x]\ e \) for sum types, we have to be careful about case expressions. Intuitively \( x_1 \) and \( x_2 \) in \( \text{case } e \text{ of } \text{inl}_A\ e \ | \ \text{inr}_A\ e \) are bound variables, just like \( x \) in \( \lambda x : A. \ e \) is a bound variable. Thus \( x_1 \) and \( x_2 \) are not free variables in \( \text{case } e \text{ of } \text{inl}_A\ e \ | \ \text{inr}_A\ e \), and may have to be renamed to avoid variable captures in a substitution \( [e'/x]\ e \) case \( e \text{ of } \text{inl}_A\ e \ | \ \text{inr}_A\ e \).

\[
\begin{align*}
FV(\text{inl}_A\ e) & = FV(e) \\
FV(\text{inr}_A\ e) & = FV(e) \\
FV(\text{case } e \text{ of } \text{inl}_A\ e | \ \text{inr}_A\ e) & = FV(e) \cup (FV(e) - \{x_1\}) \cup (FV(e) - \{x_2\})
\end{align*}
\]

\[
\begin{align*}
[e'/x]\ \text{inl}_A\ e & = \text{inl}_A\ [e'/x]e \\
[e'/x]\ \text{inr}_A\ e & = \text{inr}_A\ [e'/x]e \\
\text{case } e' \text{ of } \text{inl}_A\ e | \ \text{inr}_A\ e & = \text{case } [e'/x]e \text{ of } \text{inl}_A\ e | \ \text{inr}_A\ e \\
\text{if } x \neq x_1, x_1 \notin FV(e'), x \neq x_2, x_2 \notin FV(e')
\end{align*}
\]

An example of using sum types, let us encode the type bool. The inherent capability of a boolean value is to choose one of two different options, as mentioned in Section ?? . Hence a sum type \text{unit}+\text{unit} is sufficient for encoding the type bool because the left unit corresponds to the first option and the right unit to the second option. Then true, false, and if \( e \) then \( e_1 \) else \( e_2 \) are encoded as follows, where \( x_1 \) and \( x_2 \) are dummy variables of no significance:

\[
\begin{align*}
\text{true} & = \text{inl}_{\text{unit}}\ () \\
\text{false} & = \text{inl}_{\text{unit}}\ () \\
\text{if } e \text{ then } e_1 \text{ else } e_2 & = \text{case } e \text{ of } \text{inl}_1\ e_1 | \ \text{inr}_2\ e_2
\end{align*}
\]

Sum types are easily generalized to \( n \)-ary cases \( A_1+A_2+\cdots+A_n \). Here we discuss the special case \( n \) = 0.
Consider a general sum type \( A = A_1 + A_2 + \cdots + A_n \). We have \( n \) different ways of creating a value of type \( A \): by providing a value of type \( A_1 \), a value of type \( A_2 \), \ldots, and a value of type \( A_n \). Now what happens if \( n = 0 \)? We have 0 different ways of creating a value of type \( A \), which is tantamount to saying that there is no way to create a value of type \( A \). Therefore it is impossible to create a value of type \( A \)!

Next suppose that an expression \( e \) has type \( A = A_1 + A_2 + \cdots + A_n \). In order to examine the value associated with \( e \) and obtain an expression of another type \( C \), we have to consider \( n \) different possibilities. (See the rule \( +E \) for the case \( n = 2 \).) Now what happens if \( n = 0 \)? We have to consider 0 different possibilities, which is tantamount to saying that we do not have to consider anything at all. Therefore we can obtain an expression of an arbitrary type \( C \) for free!

The type \( \text{void} \) is a general sum type \( A_1 + A_2 + \cdots + A_n \) with \( n = 0 \). It has no introduction rule because a value of type \( \text{void} \) is impossible to create. Consequently there is no value belonging to type \( \text{void} \). The typing rule \( \text{Abort} \) below is the elimination rule for \( \text{void} \); \( \text{abort}_A e \) is called an \textit{abort} expression.

\[
\begin{align*}
\text{type} & \quad A ::= \cdots \mid \text{void} \\
\text{expression} & \quad e ::= \cdots \mid \text{abort}_A e \\
\end{align*}
\]

\[
\Gamma \vdash e : \text{void} \\
\Gamma \vdash \text{abort}_C e : C \quad \text{Abort}
\]

There is no reduction rule for an abort expression \( \text{abort}_A e \): if we keep reducing expression \( e \), we will eventually obtain a value of type \( \text{void} \), which must never happen because there is no value of type \( \text{void} \). So we stop!

The rule \( \text{Abort} \) may be a bit disquieting because its premise appears to contradict the fact that there is no value of type \( \text{void} \). That is, if there is no value of type \( \text{void} \), how can we possibly create an expression of type \( \text{void} \)? The answer is that we can never create a value of type \( \text{void} \), but we may still “assume” that there is a value of type \( \text{void} \). For example, \( \lambda x : \text{void} . \text{abort}_A x \) is a well-typed expression of type \( \text{void} \rightarrow A \) in which we “assume” that variable \( x \) has type \( \text{void} \). In essence, there is nothing wrong with making an assumption that something impossible has actually happened.

### 1.4 Fixed point construct

In the untyped \( \lambda \)-calculus, the fixed point combinator is syntactic sugar which is just a particular expression. We may hope, then, that encoding recursive functions in the simply typed \( \lambda \)-calculus boils down to finding a type for the fixed point combinator from the untyped \( \lambda \)-calculus. Unfortunately the fixed point combinator is untypable in the sense that we cannot assign a type to it by annotating all bound variables in it with suitable types. Thus thus the fixed point combinator cannot be an expression in the simply typed \( \lambda \)-calculus.

It is not difficult to see why the fixed point combinator is untypable. Consider the fixed point combinator for the call-by-value strategy in the untyped \( \lambda \)-calculus:

\[
\lambda F. (\lambda f. F (\lambda x. f f x)) (\lambda f. F (\lambda x. f f x))
\]

Let us assign a type \( A \) to variable \( f \):

\[
\lambda F. (\lambda f : A . F (\lambda x. f f x)) (\lambda f : A. F (\lambda x. f f x))
\]

Since \( f \) in \( f f x \) is applied to \( f \) itself which is an expression of type \( A \), it must have a type \( A \rightarrow B \) for some type \( B \). Since \( f \) can have only a single unique type, \( A \) and \( A \rightarrow B \) must be identical, which is impossible.

Thus we are led to introduce a fixed point construct \( \text{fix } x : A . e \) as a primitive construct (as opposed to syntactic sugar) which cannot be rewritten as an existing expression in the simply typed \( \lambda \)-calculus:

\[
\begin{align*}
\text{expression} & \quad e ::= \cdots \mid \text{fix } x : A . e \\
\end{align*}
\]
fix $x : A. e$ is intended to find a fixed point of a $\lambda$-abstraction $\lambda x : A. e$. The typing rule $\textit{Fix}$ states that a fixed point is defined on a function of type $A \rightarrow A$ only, in which case it has also type $A$:

$$
\Gamma, x : A \vdash e : A \\
\Gamma \vdash \text{fix} x : A. e : A \\
\text{Fix}
$$

Since $\text{fix} x : A. e$ is intended as a fixed point of a $\lambda$-abstraction $\lambda x : A. e$, the definition of fixed point justifies the following (informal) equation:

$$
\text{fix} x : A. e = (\lambda x : A. e) \text{fix} x : A. e
$$

As $(\lambda x : A. e) \text{fix} x : A. e$ reduces to $(\text{fix} x : A. e/x)e$ by the $\beta$-reduction, we obtain the following reduction rule for the fixed point construct:

$$
\frac{\text{fix} x : A. e \rightarrow (\text{fix} x : A. e/x)e}{\text{Fix}}
$$

In extending the definition of $FV(e)$ and $(e/x)e$, we take into account the fact that $y$ in $\text{fix} y : A. e$ is a bound variable:

$$
FV(\text{fix} x : A. e) = FV(e) - \{x\} \\
[e/x]FV(\text{fix} y : A. e) = \text{fix} y : A. (e/x)e \\
\text{if } x \neq y, y \notin FV(e')
$$

In the case of the call-by-name strategy, the rule $\text{Fix}$ poses no particular problem. In the case of the call-by-value strategy, however, a reduction by the rule $\text{Fix}$ may fall into an infinite loop because $\text{fix} x : A. e/x)e$ needs to be further reduced unless $e$ is already a value:

$$
\text{fix} x : A. e \mapsto (\text{fix} x : A. e/x)e \mapsto \cdots
$$

For this reason, a typical functional language based on the call-by-value strategy requires that $e$ in $\text{fix} x : A. e$ be a $\lambda$-abstraction (among all those values including integers, booleans, $\lambda$-abstractions, and so on). Hence it allows the fixed point construct of the form $\text{fix} f : A \rightarrow B. \lambda x : A. e$ only, which implies that it uses the fixed point construct only to define recursive functions. For example, $\text{fix} f : A \rightarrow B. \lambda x : A. e$ may be thought of as a recursive function $f$ of type $A \rightarrow B$ whose formal argument is $x$ and whose body is $e$. Note that its reduction immediately returns a value:

$$
\text{fix} f : A \rightarrow B. \lambda x : A. e \mapsto \lambda x : A. [\text{fix} f : A \rightarrow B. \lambda x : A. e/f]e
$$

One important question remains unanswered: how do we encode mutually recursive functions? For example, how do we encode two mutually recursive functions $f_1$ of type $A_1 \rightarrow B_1$ and $f_2$ of type $A_2 \rightarrow B_2$? The trick is to find a fixed point of a product type $(A_1 \rightarrow B_1) \times (A_2 \rightarrow B_2)$:

$$
\text{fix} f_{12} : (A_1 \rightarrow B_1) \times (A_2 \rightarrow B_2). (\lambda x_1 : A_1, e_1, \lambda x_2 : A_2, e_2)
$$

In expressions $e_1$ and $e_2$, we use $\text{fst} f_{12}$ and $\text{snd} f_{12}$ to refer to $f_1$ and $f_2$, respectively. To be precise, therefore, $e$ in $\text{fix} x : A. e$ can be not only a $\lambda$-abstraction but also a pair/tuple of $\lambda$-abstractions.

### 1.5 Type inhabitation

We say that a type $A$ is inhabited if there exists an expression of type $A$. For example, type $A \rightarrow A$ is inhabited for any type $A$ because $\lambda x : A. x$ is an example of such an expression. Interestingly not every type is inhabited in the simply typed $\lambda$-calculus without the fixed point construct. For example, there is no expression of type $((A \rightarrow B) \rightarrow A)$ since every type is now inhabited: $\text{fix} x : A. x$ has type $A!$ In this regard, the fixed point construct is not a welcome guest to type theory.

---

1In logic, $((A \rightarrow B) \rightarrow A)$ is called Peirce’s Law. Note that it is not Pierce’s Law!
Type safety

Proof of Theorem ??.

Proof of Theorem ??.